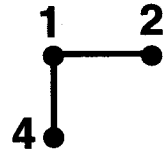
М. М. ГЛИБОВЕЦЬ

ОСНОВИ КОМП'ЮТЕРНИХ АЛГОРИТМІВ

М. М. ГЛИБОВЕЦЬ

ОСНОВИ КОМП'ЮТЕРНИХ АЛГОРИТМІВ



Київ
Видавничий дім
«КМ Академія»
2003

ББК 32.973.202-018
Г54

Монографію присвячено розгляду питань конструктивної алгоритміки в інформатиці. У книзі розглядаються основні методи побудови алгоритмів: пошук на графах, «розділяй і пануй», жадібний підхід, динамічне програмування, бектрекінг, гілок і границь, символьні обчислення, використання евристик.

Особливістю книги є цілісність викладення матеріалу. Воно *починається* з теоретичного огляду (постановка задачі, історія досліджень, коротке введення в потрібний математичний апарат дослідження задачі), опису алгоритму розв'язку задачі, характеристики та аналізу особливостей алгоритмів розв'язку задачі і *завершується* створенням програм реалізації відповідних алгоритмів на Паскалі.

Книгу адресовано студентам і аспірантам вищих навчальних закладів України, спеціалістам і науковцям спеціалізацій «прикладна математика» та «комп'ютерні науки».

Рецензенти:

Перевозчикова О. Л. – доктор фіз.-мат. наук, професор, завідуюча відділом автоматизації програмування Інституту кібернетики НАНУ

Бєлов Ю. А. – доктор фіз.-мат. наук, професор, завідувач кафедри теоретичної кібернетики факультету кібернетики Київського національного університету ім. Тараса Шевченка

Рекомендовано до друку рішенням Вченої ради НаУКМА.
Протокол № 13 від 5 березня 2002 р.

ISBN 966-518-193-9

Передмова

Сучасний рівень і темпи розвитку науки і техніки висувають перед студентом університету, випускником, молодим спеціалістом і фахівцем основну вимогу – вони повинні вільно оперувати знаннями і науковим потенціалом у вибраній ними галузі діяльності й мобільно налаштовуватись на нові знання, наукові відкриття, технології. Особливо це стосується комп'ютерних наук. Динамізм розвитку комп'ютерного обладнання, мов і систем програмування, різних спеціалізованих комп'ютерних систем обробки, аналізу і передачі інформації вимагає: забезпечення цілісності шляхом інтеграції вивчення суміжних дисциплін на загальній для них фундаментальній основі; модернізацію матеріалу вивчення, який би виокремив сучасний стан наукового знання; наявність навчальної літератури нового типу. Ці фактори мають створювати фундамент, на якому освічена людина не буде змушена прикладати максимум зусиль для опанування новітніх технологій інформатики та алгоритміки.

Широке застосування ЕОМ для розв'язання наукових і виробничих задач, проникнення комп'ютерів майже в усі сфери діяльності людини викликало потребу в новому жанрі навчальної літератури в галузі комп'ютерних наук. У книгах цього напряму викладення матеріалу починається з теоретичного огляду (постановка задачі, історія розв'язання, потрібний математичний апарат), характеристики та аналізу особливостей алгоритмів розгляду і завершується записом алгоритмів в одній із поширених мов програмування – створенням програм реалізації відповідних алгоритмів. Як зазначав А. П. Єршов, перші публікації такого типу описували традиційні розділи обчислювальної математики, але подальший вихід на моделі дискретної математики, алгоритміки зумовив відродження інтересу й отримання нових результатів у теорії графів, комбінаториці, аналізі алгоритмів, символьних і наближених обчисленнях. Ці розділи математики перетворились у дієвий інструмент розв'язання великого різноманіття сучасних задач.

У науковій літературі можна виділити такі найвідоміші й визнані монографії: Д. Кнут. «Искусство программирования для ЭВМ»: т. 1 «Основные алгоритмы», т. 2 «Получисленные алгоритмы», т. 3 «Сортировка и поиск»; А. П. Ершов. «Введение в теоретическое программирование»; А. Ахо, Дж. Хопкрофт, Дж. Ульман. «Построение и анализ вычислительных алгоритмов»; E. Horovitz, S. Sahi. «Fundamentals of data structures»; Н. Вирт. «Алгоритм + структура данных = программа»; А. В. Анисимов. «Рекурсивные преобразователи информации»; В. Липский. «Комбинаторика для программистов»; Г. Е. Цейтлин. «Введение в алгоритмику». Ці книги в

Україні або вже стали бібліографічною рідкістю, або майже не доступні широкому загалові читачів, до того ж бажано було б мати подібну україномовну працю.

Нині важко назвати розділ теоретичної інформатики, в якому упродовж останніх років були б досягнуті більші успіхи, ніж в конструюванні та аналізі алгоритмів. Знайдено багато нових ефективніших алгоритмів розв'язання задач (комбінаторні задачі) та конструктивного доведення окремих теоретичних результатів за допомогою ЕОМ (розфарбування планарного графа чотирма фарбами). Поряд з цим отримано нові теоретичні результати на підтвердження того, що для широкого кола проблем не існує достатньо ефективних алгоритмів розв'язання. Прикладом можуть слугувати праці М. Гері, Д. Джонсона, В. Ліпського.

Ще одним стимулом написання цієї монографії було прагнення установлення, поширення україномовної термінології у відповідній галузі інформатики. Автору не відоме жодне україномовне видання, яке б досить повно висвітлювало вищезгадану проблематику.

Тому в книзі зроблено спробу за допомогою конструктивного алгоритмічного підходу заповнити порожні ніші.

Зупинимося на проблемі подання матеріалу. Математика зробилася еталоном мислення завдяки строгості формальних доведень. Враховуючи те, що в книзі наголошується висвітлення алгоритмічних особливостей та їх реалізація в Паскалі, а також значний обсяг матеріалу, автор вибрав таку концепцію опису матеріалу: постановка задачі, короткий екскурс в історію розв'язання, короткий розгляд теоретичних досягнень, алгоритм(и) розв'язання, дослідження алгоритму (коректність, часові характеристики), порівняльний аналіз алгоритмів, програмна реалізація в Паскалі. В огляді теоретичних матеріалів наведено переважно тільки основні результати (теореми, твердження) без доведення. Для ознайомлення з доведенням подано посилання на необхідну літературу.

Вагомий внесок у розвиток алгоритміки зроблено київською школою науковців: В. М. Глушковим, О. А. Летічевським, Ю. В. Капітоновою, К. Л. Ющенко, А. В. Анісімовим, Г. Є. Цейтліним, С. С. Гороховським, В. С. Проценко. Всі вони тією чи іншою мірою вплинули на формування наукового світогляду автора, вироблення концепції написання цієї монографії. Авторові пощастило слухати курси лекцій багатьох з них. Професор А. В. Анісімов (перший доктор фізико-математичних наук у галузі інформатики в Україні) був науковим керівником значної частини наукових розробок автора, мудрим вчителем і дорадником. У становленні автора як програміста і педагога значну роль відіграли викладачі факультету кібернетики Київського національного університету ім. Тараса Шевченка, зокрема доцент В. С. Проценко.

Понад 20 років автор проводить наукові дослідження в галузі прикладної теорії алгоритмів. Наукові інтереси автора трансформувалися від досліджень в галузі верифікації програм і складних систем обробки інформації до розробки ефективних алгоритмів розв'язання задач на

графах, пов'язаних з роботою паралельних комплексів і мереж, створення інтерактивних систем дистанційної освіти. Завжди слушними у цій роботі були зауваження доцента С. С. Гороховського.

Всі ці роки, за винятком періоду стажування за кордоном (Тамперський технологічний університет, Фінляндія; Бостонський університет, США), автор поєднував наукову роботу з викладацькою діяльністю на факультеті кібернетики Київського національного університету ім. Тараса Шевченка (кафедра математичної інформатики) та факультеті інформатики Національного університету «Києво-Могилянська академія» (кафедра інформатики). Працюючи постійно в прямому контакті зі студентами, створюючи концепцію конкурентоспроможних навчальних програм, а також спираючись на певний досвід роботи за кордоном, автор по-новому зрозумів значущість алгоритміки і базових методів (алгоритмів) розв'язання задач за допомогою ЕОМ, їх фундаментальну роль щодо закладення основ вивчення всього спектра комп'ютерних наук.

Накопичені знання і досвід викладання відповідних дисциплін у галузі побудови й аналізу алгоритмів втілено в цій монографії. Книга може бути корисною студентам університетів, що вивчають комп'ютерні дисципліни, аспірантам, інженерам, науковим працівникам і широкому загалові спеціалістів, інтереси яких пов'язані з розробкою і дослідженням прикладних алгоритмів або вивченням програмування.

Ідея написання цієї книги знайшла підтримку керівництва Національного університету «Києво-Могилянська академія» та особисто президента університету В. С. Брюховецького. Публікації книги Видавничим домом «КМ Академія» автор зобов'язаний рекомендаціям віце-президента з розвитку Н. Р. Шумкової, а також директору видавництва В. Й. Соловйовій. Під час наукової редакції автором враховано цінні зауваження і поради.

Авторові надано технічну допомогу під час написання книги С. С. Гороховським, О. В. Олецьким, В. В. Омельченком, А. М. Глибовцем, Г. Г. Глибовець.

Матеріал книги апробований автором під час лекцій з основ комп'ютерних алгоритмів, програмування, що складають фундамент циклу профілюючих дисциплін бакалаврату спеціалізації «Комп'ютерні науки» факультету інформатики університету «Києво-Могилянська академія» та спеціалізації «прикладна математика» на факультеті кібернетики Київського національного університету ім. Тараса Шевченка. Студентів цих вузів можна вважати основними рецензентами нашої книги. Вони першими читали початкові версії (електронний варіант), робили зауваження і пропозиції, які було враховано в остаточному варіанті. Особлива подяка В. М. Федорченку, О. С. Книш, С. В. Комліченку, Р. М. Зозулі, О. В. Тавровському, В. М. Глобі, О. І. Пешко.

Книгу подано на сервері НаУКМА www.ukma.kiev.ua/win/univ/fac/dcss/publications.

Книга присвячується батькам і сину в знак подяки за їхню терпимість, любов і підтримку.

Вступ

Мир алгоритмики и сложен и богат,
он алгеброй и логикой поверен.
И стать в нем маршалом ты завтра будешь рад,
Но нынче будь ему солдатом верным!
Г. Е. Цейтлин. Введение в алгоритмику

Повсякденне життя людини, її досягнення тісно пов'язані з її здатністю до складання алгоритмів розв'язання різноманітних задач. Увесь процес розвитку людства передбачав необхідність розробки доступних засобів опису, аналізу та реалізації алгоритму. Уже первісна людина мала за допомогою якихось засобів передавати свої знання з утворення, зберігання і передачі вогню. Проте актуальності наука про алгоритми набула в останній період існування людства – період глобальної комп'ютеризації і автоматизації суспільства, проникнення сервісів мережних технологій в найрізноманітніші сфери задоволення попитів людини. Отже, ми приходимо до потреби наукового висвітлення матеріалу, орієнтованого на розв'язання досить нетривіальних проблем формалізації опису, обґрунтування коректності, оптимізації алгоритмів [98].

У книзі зроблено спробу дослідження повного технологічного ланцюжка побудови – реалізації алгоритмів: короткий огляд основних моделей обчислень, структур даних та їх використання у разі побудови ефективних алгоритмів; у більшості випадків проаналізовано коректність і часову оцінку алгоритму розгляду та особливості його програмної реалізації в Паскалі. Мову Паскаль вибрано як одну з найпоширеніших простих навчальних мов програмування.

Книга складається з 12 розділів.

Розділ 1 присвячено загальній характеристиці алгоритмів і обчислень. У ньому наведено прийняту класифікацію обчислювальних машин, коротку характеристику сучасних комп'ютерів, подано основи архітектури обчислювальних пристроїв і основні формальності задання алгоритму: автомати, блок-схеми, програми. Для розуміння загальної проблематики теорії обчислень описано клас функцій, обчислюваних за Тьюрингом, наведено тези Чорча і Тьюринга, моделі обчислення РАМ і РАСП. Подано визначення часових оцінок, задач типу P, NP. У кінці міститься набір контрольних завдань для закріплення теоретичного матеріалу.

Зрозуміло, що ефективність програми залежить від двох основних чинників: якості алгоритму і зручних структур даних програмної реалізації. Тому в **Розділі 2** розглянуто основні типи класичних структур даних, особливості їх застосування. За основу розгляду взято підхід Н. Вірта [20]. Жодна мова програмування не може містити такий набір стандартних типів даних, який би давав змогу кваліфікованому програмісту оптимально реалізувати більшість алгоритмів. Більшість сучасних мов програмування має апарат конструювання потрібних типів (структур) даних. Тому приділено увагу висвітленню як стандартних, так і класичних нестандартних типів даних: змінні, масиви, записи, файли, списки, стеки, черги, дерева, графи. Враховуючи подальше використання мови Паскаль, крім висвітлення теоретичних особливостей застосувань таких даних, подано і їх реалізацію в Паскалі. Наведено класичні приклади програмування алгоритмів, що висвітлюють нюанси роботи з масивами, файлами, списками. Особливу увагу звернено на опис техніки роботи з динамічними змінними, переваги використання рекурсивних типів даних. Описано ефективну реалізацію основних операцій роботи з множинами.

Прикладами отримання ефективних алгоритмів із застосуванням спеціалізованих структур даних є розглянуті задачі реалізації польського інверсного оберненого запису (стек), швидкого впорядкування (купа), ізоморфізм дерев (дерева).

Розділ завершує різноманітний набір контрольних завдань, достатній для закріплення вивченого матеріалу. Набір задач поділено на такі теми: системи числення та арифметичні задачі; складність алгоритмів; списки, дерева, графи.

У **Розділі 3** описано алгоритми пошуку на графах, їх реалізацію та застосування під час вирішення багатьох теоретичних і практичних задач. Серед різноманітних методів, за допомогою яких можна організувати пошук у графі потрібної вершини або шляху до неї, прийнято виділяти дві основні стратегії: пошук у ширину і пошук у глибину. Тут їх розглянуто особливо прискіпливо.

Застосування алгоритмів пошуку в задачах з графами наведено на прикладах розв'язання таких задач знаходження: остових дерев, фундаментальної множини циклів, компоненти двозв'язності, ейлерового шляху. Важливе місце в теорії графів посідає задача знаходження мінімальних шляхів. Вона має великий теоретичний інтерес і значне практичне застосування. У розділі наведено її розв'язання для різних типів графів і в особливих постановках: найкоротші шляхи між усіма парами вершин, найкоротші шляхи до всіх вершин від однієї вершини (джерела). У цьому разі використовують найефективніші алгоритми Дейкстри, Флойда, Уоршалла.

Особливе місце в теорії графів за широтою практичних застосувань посідає спеціальний тип графа – мережа і задача знаходження максимального потоку в мережі. У монографії наведено один з найкращих алгоритмів розв'язання цієї задачі – алгоритм Дініца та його реалізацію.

Розділ закінчується набором контрольних завдань.

Тепер тяжко назвати розділ теоретичної інформатики, який би тією чи іншою мірою не використовував елементи конструювання й аналізу комбінаторних алгоритмів. У **Розділі 4** розглянуто деякі найбільш вживані алгоритми комбінаторики. Особливу увагу приділено конструктивному підходу, наведено алгоритми генерування підмножин множини, розбиття чисел, генерування перестановок.

Типовою для багатьох задач, що обчислюють на ЕОМ, є ситуація, коли для розв'язання задачі не вистачає ресурсів системи обчислення. Наприклад, не вистачає пам'яті для розміщення вхідних даних. Природним виходом із цієї проблеми є розбиття задачі на підзадачі та пошук загальної проблеми на них. Потім шукають алгоритм, який би дав змогу знайти загальне розв'язання задачі, комбінуючи якимось чином розв'язання підзадач. У теорії обчислень такий підхід до розв'язання задач дістав назву методу «розділяй і пануй».

За допомогою методу «розділяй і пануй» розв'язують дуже багато прикладних задач. Він зручний у програмуванні; для побудови часової оцінки таких алгоритмів використовують методику розв'язання рекурентних рівнянь, що будують конструктивно, згідно із загальною схемою. Висвітленню цих питань присвячено **Розділ 5.** Теоретичні особливості методу «розділяй і пануй» продемонстровано в процесі розв'язання конкретних задач, таких як бінарний пошук, знаходження мінімального та максимального елементу в масиві, обмінне впорядкування і сортування злиттям.

У **Розділі 6** розглянуто розв'язання задач оптимізації, описано частину теорії алгоритмів, яка пов'язана з проблемою знаходження екстремуму. На важливість розгляду задач оптимізації вказував ще Гільберт. Він плекав надію, що у XX ст. математики знайдуть методи розв'язання таких задач, незважаючи на те, що обчислювальні характеристики більшості задач оптимізації відносять їх до класу NP-повних задач.

Серед основних ідей знаходження екстремуму функцій і функціоналів виділяють класичні (лінійне програмування, принцип максимуму Понтрягіна, теорія локальних екстремумів) та ідеї послідовного аналізу варіантів. Більшість з них описано в розділі. Наведено також загальну класифікацію задач оптимізації за Рейнгольдом [88]; розв'язання задачі лінійного програмування симплекс-методом (табличним і за допомогою оберненої матриці); жадібний метод.

Особливу увагу приділено жадібним алгоритмам. Розглянемо класичний приклад. Нехай потрібно знайти найкоротший шлях між двома точками. Якщо за допомогою жорстких обмежень можна звузити коридор пошуку (допустиму область руху), то задача оптимізації стає досить тривіальною: потрібно прямувати тільки визначеним коридором і довільний шлях в ньому буде практично оптимальним. Зрозуміло, що одним із базових методів оптимізації є метод послідовних локальних покращень. Алгоритми, побудовані на основі методу послідовних локальних покращень, називають жадібними алгоритмами в широкому розумінні, оскільки вони на кожному кроці намагаються наблизитися до мети. Жадібним алгоритмом у вузькому розумінні називають алгоритм, що намагається на кожному кроці підійти до мети якнайближче (градієнтний метод, метод найшвидшого спуску). У розділі наведено методологію використання жадібного підходу під час розгляду конкретних задач: про рюкзак, про оптимальне розміщення програм у пам'яті, оптимізації злиття файлів, задача Дейкстри, знаходження остового дерева найменшої вартості.

У разі використання жадібних алгоритмів актуальним є питання, коли вигідно бути жадібним, тобто коли локальний мінімум збігається з глобальним? На це питання дає відповідь теорія матроїдів. Основні її результати викладено тут в інтерпретації В. Ліпського [66].

У **Розділі 7** розглянуто основні аспекти динамічного програмування, в основу якого покладено використання принципу оптимальності Р. Беллмана. Перетворити цей загальний підхід на систему формальних процедур дуже важко. Проте вже багато зроблено в цьому напрямку. Перші ідеї такого підходу належать А. А. Маркову, розвивались у працях американських науковців Д. Вальда, Р. Айзекса, Р. Беллмана [11]. Значний внесок у створення загального формалізму послідовного аналізу варіантів (схема формалізації Михалевича, метод «київський вінник») належить київській школі математиків на чолі з В. С. Михалевичем [73, 74].

Методи динамічної оптимізації значною мірою використовують глибоке знання сутності задач розв'язання. Аналогічно методові «розділяй і пануй» задачу розбивають на підзадачі. Останні розв'язують, і з отриманих підрозв'язків будують загальний розв'язок. Підзадачі є залежними між собою, тобто у них є загальні підпідзадачі. Алгоритми динамічного програмування вирішуватимуть кожну із підзадач тільки один раз, запам'ятовуючи отримані розв'язки у спеціальних таблицях.

У динамічному програмуванні оптимальної послідовності рішень досягають унаслідок явного звернення до *принципу оптимальності*. Цей принцип стверджує: *оптимальна послідовність рішень має таку властивість, що, незважаючи на початковий стан та рішення, серед рішень, що залишились, завжди міститься оптимальна послідовність рішень щодо стану, який утворився після прийняття першого рішення.* Отже, істотна різниця між жадібним методом і динамічним програмуванням полягає в тому, що жадібний метод завжди генерує одну послідовність рішень. У динамічному програмуванні може утворюватись декілька послідовностей рішень.

Особливості використання методу продемонстровано в алгоритмах розв'язання задач знаходження найкоротшого шляху, шляхів у багаторівневих графах, оптимальної послідовності множення матриць. Для більшості задач рекурсивна природа задання принципу оптимальності приводить до рекурентного типу відношення. Алгоритми динамічного програмування розв'язують ці рекурентні відношення, щоб знайти оптимальний розв'язок заданої задачі. Існує окрема теорія розв'язання рекурентних рівнянь [62, 111, 135, 145]. Формулювати рекурентні відношення динамічного програмування можна за допомогою використання двох підходів перегляду варіантів розв'язань: прямого та оберненого. У розглянутих алгоритмах акцентовано увагу на цих аспектах.

У **Розділі 8** наведено загальну методологію використання бектрекінгу, а також особливості її застосування для розв'язання конкретних задач: про *n*-ферзів, про суму підмножин, знаходження гамільтонового циклу. Основну увагу приділено технології побудови експліцитних та імпліцитних обме-

жень. Також розглянуто побудову простору розв'язань у вигляді дерева (динамічного чи статичного). Для оцінки ефективності бектрекінгових алгоритмів наведено ідею оцінювання Кнута [131, 149].

Логічним завершенням основних методів розв'язання оптимізаційних задач і методик обмежень повного перебору є метод гілок і границь, що розглянуто в **Розділі 9**. Цей метод дає змогу серед елементів деякої множини можливих розв'язань задачі знайти найкращий розв'язок за умови, що цю множину можна розбити на неперетинні підмножини і для будь-якої підмножини можна визначити оцінку найкращого (найоптимальнішого) можливого розв'язку. Тобто має бути можливість визначити «оцінку» розв'язку, краще якого не можна досягти, якщо розв'язок шукати тільки у межах відповідної підмножини (але насправді такої оптимальності на цій підмножині може і не бути). У розділі наведено загальну характеристику методу, продемонстровано особливості застосування на прикладі розв'язання задач про рюкзак і комівояжера.

Кожній людині у повсякденному житті часто трапляються поняття «гра», «ігрова задача», проте на прохання дати формальне визначення цих понять отримаємо велику кількість визначень описового типу. Спеціалісти розрізняють математичну теорію ігор та ігрові програми. Перша належить до математичного програмування і числових методів задач пошуку екстремуму або екстремумів. Базою цієї теорії слугує результат Льюїса і Райфа про існування алгоритму визначення оптимального ходу на будь-якій стадії гри n-осіб, коли кожен учасник має повну інформацію і число можливих ходів скінченне. Інші займаються побудовою ефективних наглядних ігрових програм, реалізованих на комп'ютері з широким використанням комп'ютерної графіки.

Відомо, що антагоністичні ігри описують конфлікти часткового характеру. Для переважної більшості реальних конфліктів антагоністичні ігри або зовсім не можна вважати прийнятними, адекватними моделями, або, у кращому випадку, можна розглядати як перші грубі наближення. Тому тут коротко описано і безкоаліційні та кооперативні ігри.

У **Розділі 10** подано загальну характеристику ігрових задач, класифікацію ігор (матричні ігри). Природним узагальненням матричних ігор є нескінченні антагоністичні ігри, у яких хоча б один із гравців має нескінченну кількість можливих стратегій. Ми розглядатимемо *ігри двох гравців, що роблять по одному ходу, і після цього відбувається розподіл виграшів*. Формалізуючи реальну ситуацію з нескінченною кількістю варіантів вибору, можна кожну стратегію зіставити з визначеним числом з одиничного інтервалу, тому що завжди можна простим перетворенням будь-який інтервал перевести в одиничний і навпаки.

У розділі також велику увагу приділено розгляду другого напряму в теорії ігор – методики розробки алгоритмів ігрових програм. Тут розглянуто загальну стратегію побудови ігрових програм, проаналізовано методи задання даних, дерева ігор, функції оцінювання позиції і процедури вибору ходу, алгоритми відтинання, основні підходи до самонавчання, проведено огляд сучасних програм гри в шахи. У повному обсязі, включаючи і практичну реалізацію в Паскалі, наведено програму ***Crest*** (програма 10.1) – програмна реалізація всім добре відомої гри «хрестики-нулики». Гра проходить в діалоговому режимі «користувач–комп'ютер». У цій програмі реалізовано вищезазначені принципи побудови ігрових алгоритмів та використано певні евристики, наприклад «правило боротьби за центр» та ін.

У **Розділі 11** наведено методи швидкої і точної реалізації операцій з числами та багаточленами. Для цього слід мати формальний апарат, що давав би змогу зручно оперувати математичними виразами [113, 134]. Так, наприклад, під час роботи з поліномами бажано розглядати дві моделі: *щільну* – для задання поліномів, що мають більшість коефіцієнтів, відмінних від нуля; *розріджену* – для задання поліномів, що мають більшість коефіцієнтів, рівних нулю. Остання – особливо корисна для задання поліномів від багатьох змінних. Їх і досліджують тут.

Для розв'язання багатьох задач, пов'язаних з обробкою поліномів, також корисним є використання різних спеціалізованих перетворень. Прикладом може слугувати перетворення Фур'є (швидке перетворення Фур'є). Його вживають для побудови багатьох ефективних алгоритмів. Отже, слід мати швидку реалізацію перетворення Фур'є. Прикладом його використання може бути обчислення добутку двох поліномів. Ці питання детально досліджено у розділі.

Зрозуміло, що для роботи з багаточленами потрібно, по-перше, вибрати спосіб задання (опису) і, по-друге, написати процедури, кожна з яких представляє певну операцію. Систему, що дає змогу оперувати математичними виразами (зазвичай під математичними виразами розуміють цілі числа довільної точності, багаточлени та раціональні функції), називають *математичною системою з операціями над символами* [127]. Тому в розділі значну увагу приділено розгляду проблематики, пов'язаної з арифметичними операціями над цілими числами і поліномами.

Світ алгебричних алгоритмів настільки різноманітний, що автор лише спробував розкрити декілька найцікавіших тем. Звернуто увагу на проблеми цілих та модульну арифметику, яку можна представити як схему перетворень, що є корисною для прискорення арифметичних операцій з цілими. Розглянуто найвідоміші алгоритми для v-точкового обчислення та інтерполяції, обернену операцію інтерполяції в n точках. У висвітленні відповідного матеріалу наведено зіставлення результатів для цілих чисел з відповідними результатами для поліномів. Серед алгоритмів розгляду віддано перевагу алгоритмам, що є асимптотично найефективнішими серед відомих. За основу висвітлення матеріалу покладено підходи з праць [9, 127].

У **Розділі 12** розглянуто методологію побудови евристичних алгоритмів. Евристичні підходи до розв'язання задач людина використовувала з давніх часів. Короткий оксфордський словник англійської мови трактує це слово так: «Евристика – мистецтво знаходження істини». У давньогрецькій мові слово евристика (heuristic) має ті ж джерела що й слово

еврика. Прикладом ілюстрації цього методу в загальних рисах може слугувати класична задача про мавпу і банани.

Під час розгляду задач, що належать класу NP, важливе місце посідає знаходження варіантів розв'язання цих задач для певних наборів даних (а не для всіх даних) за поліноміальний час. Для цього дослідникові бажано підказати шлях, який позбавляє необхідності розгляду повного перебору, базуючись на певних особливостях задачі розгляду. Ці особливості називають евристиками, а правила їх використання – евристичними правилами. Евристичні правила можуть бути як дуже простими, так і досить складними.

Оскільки евристичні правила мають рекомендаційний характер, то евристичні методи не завжди приводять до бажаних результатів розв'язання задачі. Отже, слід мати якомога більший набір евристик. Для більш глибокого розуміння евристичного підходу розглянуто два типи задач: *добре* визначені задачі і *погано* визначені задачі [104].

Згідно із загальновідомим визначенням евристичного методу потрібне вірне адміністрування побудови розв'язання. Під останнім розуміють метод, який, згідно з деяким евристичним критерієм, серед декількох можливих підходів до розв'язання вибирає найоптимальніший на даному кроці. Інакше кажучи, адміністративна частина алгоритму має бути в змозі дати оцінки складності підзадач та їх корисності, а також використовувати ці оцінки для планування процесу розв'язання задач. Отже, евристичний метод є корисним у разі встановлення порядку виконання дій розв'язання задачі, який є не обов'язково оптимальним. Цей порядок, з великою вірогідністю, має виявитися набагато кращим за випадково вибраний порядок. Висвітлення цих питань займає значне місце в цьому розділі. Тут також пояснюється основна евристика, евристика-навчання Мінського і Селфріджа [140].

Наприкінці 1950-х та на початку 1960-х років Ньюел, Саймон і Шоу опублікували піонерські праці, які заклали початок класичного евристичного підходу до вирішення інтелектуальних задач; у цьому разі основний акцент було зроблено на планування цілеспрямованих дій. Першою інтелектуальною програмою стала розроблена ними програма *«Логік–Теоретик»*, за допомогою якої можна було доводити теореми математичної логіки. Ідеологія нового підходу полягала в породженні різноманітних здогадок та припущень з подальшою перевіркою їх справедливості. Розвиток цього напряму був пов'язаний з програмою *«Загальний вирішувач задач»* (GPS – General Problem Solver), яку ще іноді називають *«Універсальний розв'язник задач»*. Тому ці питання також розглянуто у розділі.

Багато практичних задач можна розв'язати на основі так званого евристичного пошуку. Його часто розглядають як *пошук шляху на дереві або графі.* Справді, розглянемо типову схему розв'язання задачі, згідно з якою на кожному кроці можна вибирати одну з можливих дій. Тоді можна говорити про породження *дерева можливостей*, вузли якого відповідають ситуаціям проблемної області, а дуги – можливим діям. У цьому разі задача переходу від початкової ситуації до бажаної зводиться до пошуку шляху на дереві. Аналогічно виникає і графова інтерпретація.

Застосування описаної методології використання евристик продемонстровано на прикладах розв'язання класичних задач складання розкладу виконання робіт, розфарбовування графа та визначення ізоморфізму графів.

Нарешті, враховуючи різнопланову орієнтованість викладеного матеріалу, можна запропонувати таку схему читання книги. Початківці або люди, що хочуть детальніше вивчити мову Паскаль, можуть обмежитися розглядом перших чотирьох розділів. Більш підготовлені читачі можуть починати читати книгу з п'ятого розділу.

Автор буде дуже вдячний, якщо читачі надішлють слушні пропозиції, зауваження щодо викладеного матеріалу та методології його подання в електронному вигляді на адресу glib@ukma.kiev.ua.

Розділ 1

АЛГОРИТМИ І ОБЧИСЛЕННЯ

1.1. Алгоритми

Потреби суспільства в узагальненні багатьох підходів до розв'язання різноманітних задач зумовили створення теорії алгоритмів. Бурхливий розвиток науки і техніки в останні століття, спроби філософів найточніше описати різноманітні аспекти людського буття дали світові науку наук – логіку розвитку, яка породжувала задачі, що стимулювали появу нових ідей і методів їх розв'язання. Складність і великий об'єм обчислень зумовили створення спеціалізованих обчислювальних пристроїв. Вони пройшли складний шлях розвитку. Серед них можна виділити один із найпотужніших класів – це електронно-обчислювальні машини (ЕОМ). Проникнення ЕОМ у різні сфери практичної діяльності людей значно вплинуло на еволюцію теорії алгоритмів, на вибір нових напрямів її застосування. Вона перетворилась у певну школу мислення й аналізу для широкого кола програмістів, дала основу мові алгоритмів, яка об'єднує різні напрями досліджень у програмуванні, полегшує розповсюдження ідей. Теорія алгоритмів є важливою частиною прикладної математики, а поняття алгоритму та форм його задання посідає чільне місце в теорії алгоритмів.

Як відомо, слово *алгоритм* прийшло з Персії, запропонував його автор книги з математики Abu Jafar Mohammed ibn Musa al Khowarizmi. Він визначав його як деякий спеціальний метод вирішення поставленої проблеми.

Поняття алгоритму належить до найскладніших концепцій природознавства. Воно пройшло складний історичний шлях розвитку, починаючи з інтуїтивного розуміння і стихійного застосування на перших кроках розвитку людського суспільства та закінчуючи усвідомленням закономірностей природних явищ, застосуванням у сучасних ЕОМ. Багато авторів визначають розвиток науки як безперервний процес уточнення алгоритмів і побудови відповідних моделей [4, 9, 20, 21, 41, 49, 59–61, 69].

Умовно можна виділити три основні етапи розвитку теорії алгоритмів.

На першому етапі відбувалося накопичення фактів людиною, що засвоювала знання про природу і стихійно формувала та закріплювала власні алгоритми поведінки.

Другий етап становлення алгоритмічного апарату пов'язаний з розвитком математичних наук. Для нього є характерним чітке задання математичних алгоритмів розв'язання різних прикладних задач і опис значних алгоритмічних проблем, що не піддавались розв'язанню традиційними методами, наприклад трисекція кута.

Третій період бере відлік з початку XX ст. Було побудовано формальну класичну теорію алгоритмів, яка уточнювала можливості теоретичного обчислення для практичного застосування в кібернетиці й програмуванні. Теорію алгоритмів можна поділити на класичну і прикладну. У класичній теорії алгоритмів існує безліч формальностей для опису алгоритмів. Серед них можна виділити найзначніші: арифметичне числення предикатів Гьоделя, машини Поста і Тьюринга, автомати Маркова, схеми Янова, блок-схеми [4, 41, 65, 91, 106].

Під алгоритмом розуміють сукупність правил функціонування, що описують поведінку деякої системи, керуючись якими вона досягає кінцевого результату.

Алгоритм має скінченну множину кроків, кожен з яких може виконати одну або більше операцій. *Операції мають бути однозначними та ефективними.* Кожен крок слід виконувати за *скінченний*, у межах розумного, час. Алгоритм повинен мати хоча б один вихід і нуль або більше зовнішніх входів та закінчуватись після скінченного числа операцій.

Відомо, що всі визначення алгоритму еквівалентні щодо можливості моделювання обчислень. Форма задання алгоритму особливої ролі не відіграє, важливим є тільки вибір мінімальних засобів для задання і перетворення інформації.

Прикладна теорія алгоритмів спрямована на дослідження моделей алгоритмів. Тут під алгоритмом розуміють довільне перетворення обчислювальної інформації, яке може бути виражене за допомогою деякої алгоритмічної мови. Алгоритмічна мова буває різного типу складності. У зв'язку з обчислюванням алгоритмів на комп'ютерах виберемо як форму запису алгоритму мову Паскаль [51].

Розглянемо такий приклад. Нехай потрібно дослідити розв'язання повного квадратного рівняння $ax^2 + bx + c = 0,\ a \neq 0,\ b \neq 0,\ c \neq 0.$

Алгоритм дослідження можна описати українською мовою за допомогою такої послідовності кроків:

0. Ввести значення a, b, c.

1. Обчислити дискримінант $D = b^2 - 4ac.$

2. Якщо $D = 0$, *тоді надрукувати повідомлення «рівняння має один розв'язок:* $x = -\frac{b}{2a}$».

В іншому разі, якщо $D > 0$, *тоді надрукувати повідомлення «рівняння має два корені:* $x_1 = \frac{-b + \sqrt{D}}{2a}$ *і* $x_2 = \frac{-b - \sqrt{D}}{2a}$».

Інакше (якщо $D < 0$) надрукувати повідомлення «рівняння не має дійсних коренів».

3. Кінець алгоритму.

Якщо уважно подивитися на таке задання алгоритму, то можна виділити основні недоліки опису: громіздкість та неоднозначність трактувань інструкцій. Для їх усунення використовують спеціалізовані формалізми: блок-схеми, мови програмування тощо.

Наш алгоритм можна описати блок-схемою, яку зображено на рис. 1.1.

Блок-схема є зв'язним орієнтованим графом, вузли якого можуть бути блоками чотирьох типів: *блок ініціалізації* (ввести значення *a*, *b*, *c*), *функціональні блоки* (обчислити $D = b^2 - 4ac$, вивести...), *предикатні блоки* ($D = 0$, $D > 0$) і *блок закінчення обробки інформації* (кінець). Кожен блок належить певному шляху з початкового блоку в кінцевий. Передачу керування між блоками визначають напрямленими стрілками. Предикатний блок використовують для розгалуження керування (умовної передачі керування). Він функціонує так: якщо предикат, що стоїть в середині ромба, справджується (набуває значення «істина», позначається «*i*»), то керування передається одному блоку, якщо ж предикат не справджується (набуває значення «хибність», позначається «*x*»), тоді керування передається іншому блоку.

Для дослідження цієї задачі на ЕОМ слід описати алгоритм дослідження за допомогою якоїсь мови програмування. Таке задання називають програмою.

Структура програми більшості мов програмування типова і має загальний характер. Розглянемо запис програми 1.1 дослідження розв'язання повного квадратного рівняння в мові програмування Паскаль.

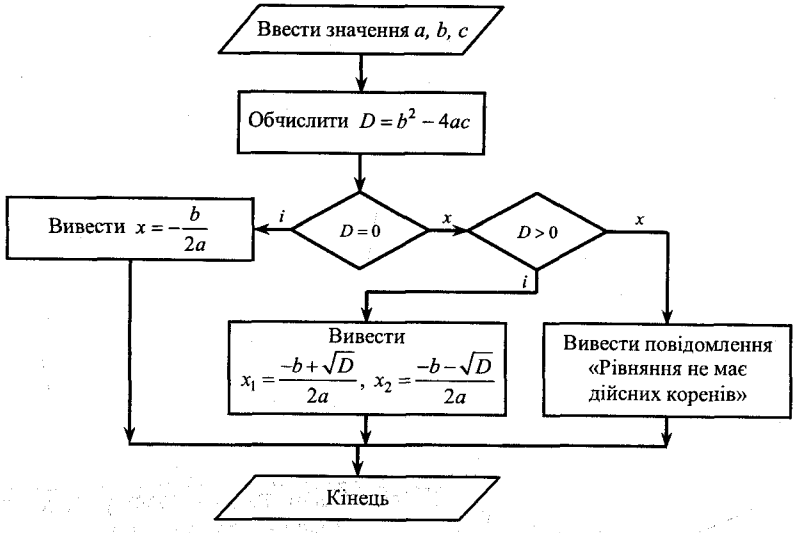


Рис. 1.1. Блок-схема дослідження розв'язків рівняння


```
program EX1_1;
var A, B, C, D: real;
begin
  read(A, B, C);
  D := B*B - 4*A*C;
  if D = 0 then
    writeln('Рівняння має один розв'язок x = ', -B/(2*A))
    else
      if D > 0 then
        writeln('Рівняння має два корені x1 = ', (-B + D)/(2*A),
        ' і x2 = ', (-B - D)/(2*A))
        else
          writeln('Рівняння не має дійсних коренів');
end.
```

Програма 1.1. Дослідження розв'язку повного квадратного рівняння

Знання англійської мови дає змогу зрозуміти в загальних рисах призначення кожної інструкції (відділяється крапкою з комою) у програмі.

Програми Паскаль, як і програми більшості мов програмування, складаються з таких основних частин: *заголовка* ***(program)***, *блока опису констант* ***(const)***, *блока конструювання типів* ***(type)***, *блока опису змінних* ***(var)***, *блока опису процедур та функцій і блока виконань* ***(begin–end)***.

Заголовок програми, в нашому прикладі іменується **EX1_1,** використовується для ідентифікації програми і може містити опис пристроїв введення-виведення інформації.

Блок опису змінних використовують для визначення розмірів ділянок пам'яті, де процесор оброблятиме інформацію для отримання кінцевого результату. У цьому випадку змінні *A*, *B*, *C*, *D* виступають у ролі засобів доступу до ділянок пам'яті (імен), а опис **real** визначає розмір пам'яті, що виділяється для змінної зазначеного типу у разі отримання даної програми на виконання процесором.

Блок опису констант використовують для визначення даних, що не змінюють своїх значень за час виконання програми.

За допомогою конструктора типів будують структури даних, зручні для побудови ефективних алгоритмів, яких немає серед стандартних типів даних вибраної мови програмування.

Блок виконань містить інструкції, що описують послідовність дій, які процесор здійснюватиме над пам'яттю для отримання кінцевого результату алгоритму.

Спробуйте описати алгоритм дослідження квадратного рівняння $x^2 + px + q = 0$, $p \neq 0$, $q \neq 0$ за допомогою блок-схеми, а потім переведіть блок-схему в програму на Паскалі.

Традиційно програми можна поділити на два класи: прикладні та системні. Прикладні програми використовують для розв'язання конкретних прикладних задач. Системні програми полегшують роботу користувача на комп'ютері. До системних програм належить і набір програм, який дістав назву *операційна система* (ОС).

ОС забезпечує взаємодію між користувачем і комп'ютером, а також між програмою і різними компонентами апаратури комп'ютера. Загальне керування комп'ютером здійснюють за допомогою мови директив. У кожній ОС є набір службових програм (утиліт), які використовують для полегшення роботи у разі написання програм. Одна з типових конфігурацій ОС для комп'ютера має такі складові компоненти: файлову систему; драйвери зовнішніх пристроїв; процесор командної мови.

1.2. Обчислювальні машини

Обчислювальна машина – це *автоматизована* лічильна машина, створена для підвищення швидкості обробки арифметичних та інших математичних операцій. Обчислювальні машини є апаратною реалізацією скінченних автоматів. Вони можуть бути цифровими або аналоговими [41].

Цифрова машина – це система, яка працює на *числовій дискретній* основі так, що число формально задається в середині системи.

Будь-яка аналогова система має істотно неперервну основу, числа задаються фізичними величинами (промінь світла, струм тощо).

Для ілюстрації основної відмінності між цифровим і аналоговим принципом обробки інформації розглянемо класичний приклад з музичними інструментами. Баян можна назвати дискретною системою, тому що в ній довільну фіксовану ноту можна або грати, або ні, бо кожна нота (клавіша) дискретно відділена від наступної. Скрипка ж буде аналоговою системою, оскільки на скрипці висота тону може змінюватися безперервно.

Відомо, що довільну аналогову машину теоретично можна апроксимувати деякою цифровою. Проте на практиці виникають труднощі реалізації. Наприклад, спроба імітації роботи людського мозку приводить до природних обмежень, пов'язаних з пам'яттю та часом.

Цифрові обчислювальні машини (ЦОМ) можна поділити на певні класи відповідно до організації загальних принципів обчислень. Наприклад, можна виділити одноадресні, двоадресні, *n*-адресні обчислювальні машини. Ця відмінність ґрунтується на різниці в числі адрес у командних словах обчислювальної машини. Іншим прикладом поділу є два великі класи ЦОМ: послідовні та паралельні. Цей поділ використовує такий критерій: ЦОМ можуть виконувати одночасно одну або кілька операцій.

Загальну схему ЦОМ ілюструє рис. 1.2.

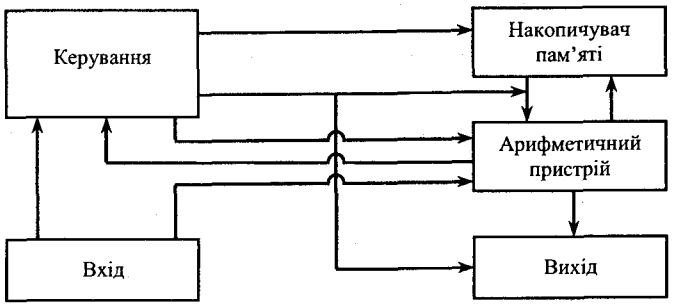


Рис. 1.2. Загальна функціональна схема ЦОМ

Слід зазначити, що не всі блоки, зображені на рисунку, присутні в усіх типах ЦОМ, але ця схема є найзагальнішою.

Пам'ять обчислювальної машини зберігає список команд (програму) обробки даних і самі дані (набір чисел), які оброблятимуться цими операціями (командами).

Керуючий пристрій визначає, які ж операції пересилатимуть числа в арифметичний пристрій для виконання необхідних операцій. Після виконання цих операцій дані повертаються в запам'ятовуючий пристрій.

Команди слід записувати у пристрій запам'ятовування у вигляді машинних «слів», що задають в числовому вигляді. Числа, які машина обробляє, мають бути заданими у вигляді машинних слів. Інакше кажучи, машинні слова задають команди і дані.

Проведіть аналогії загальної схеми функціонування ЕОМ із загальною схемою функціонування деякого виробничого комплексу.

Різні обчислювальні машини можуть використовувати різні коди для задання команд: десятковий, двійковий, вісімковий, шістнадцятковий тощо. Детально на цих кодах не зупинятимемося, тільки зазначимо, що довільне слово, число можна закодувати послідовністю двійкового коду 0 і 1.

Для того щоб реалізувати певні дії, які слід застосовувати до даних, що знаходяться в пристрої запам'ятовування, ми повинні мати можливість вказувати процесору, які ж конкретні команди він має виконати над цими даними.

Як ілюстрацію такого принципу розглянемо умовну триадресну обчислювальну машину, тобто машину, в якій командне слово містить три адреси: адреси двох чисел, що обробляються, і адресу, куди заносять результат обробки. Припустимо, ми вибрали таке кодування арифметичних операцій для триадресної машини: *додавання (01), віднімання (02), множення (03), ділення (04), умовний перехід (05), завантаження (06), друк (07).*

Розглянемо таку просту арифметичну задачу. Нехай потрібно перемножити числа 2 і 3, потім відняти 1 та надрукувати результат. Перше, що

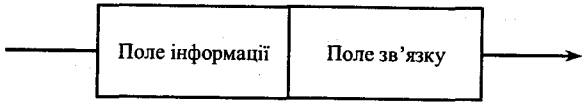


Рис. 1.3. Базовий елемент списку

слід зробити,– розмістити числа 2, 3, 1 у вільних регістрах пам'яті. Нехай такими будуть регістри з номерами 500, 501, 502. Для позначення цієї дії використаємо запис 06/500(2), 06/501(3), 06/502(1). Для триадресної машини команди мають таку структуру:

$$I/A1/A2/A3,$$

де I – код команди, $A1$, $A2$, $A3$ – адреси регістрів.

Тоді для реалізації нашої задачі потрібно написати програму:

03/500/501/500
02/500/502/503
07/503

Одноадресна і двохадресна машини працюють за іншим принципом. У них використовується пристрій накопичувального типу, який називають суматором. У таких машинах команди заносять усі числа обробки в суматор. Є спеціальні команди, які повертають числа із суматора у загальну пам'ять.

Для кращого зберігання операндів і операцій в пам'яті обчислювальної машини Ньюел у 1961 р. запропонував використовувати списки [93]. Базовим елементом списку є елемент, зображений на рис. 1.3.

Список можна використати для «зв'язування» різних елементів пам'яті, які «фізично» містяться в непослідовних фрагментах пам'яті ЦОМ. Так вираз $x + y$ можна описати списком, зображеним на рис. 1.4.

Є ще один тип обчислювальних пристроїв, який стоїть трішки виокремлено – це нейрокомп'ютер. З початку створення обчислювальних пристроїв спеціалісти намагалися створити такий пристрій, який би моделював роботу мозку людини. Оскільки в ньому головну роль виконує нейрон, можна знайти ключ до моделювання розумової діяльності людини в моделюванні роботи нейрону мозку. Для багатьох цілей нейрон можна розглядати як елемент з певним критичним значенням. Це означає, що він або ж дає на виході деяку постійну величину, якщо сума його входів досягає певного значення, або ж залишається пасивним.

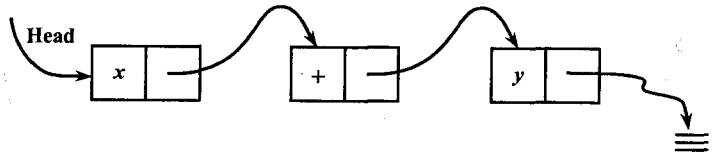


Рис. 1.4. Список для виразу $x + y$

Мак-Каллок і Піттс довели, що будь-яку обчислювальну функцію можна реалізувати за допомогою спеціально організованої мережі ідеальних нейронів, логічні властивості яких з високою вірогідністю можна приписати реальному нейрону [138]. Ця мережа матиме наступні вади. По-перше, проблема полягає в тому, чи можна знайти якийсь розумний принцип реорганізації мережі, який давав би змогу випадково об'єднаній спочатку групі ідеальних нейронів самоорганізовуватись в «обчислювальний пристрій», здатний розв'язувати довільну задачу. По-друге, потрібно використовувати велику кількість нейронів. Так, модель роботи мозку мурашки потребує використання близько 20 000 нейронів, що на практиці реалізувати неможливо.

Нейрокомп'ютер – програмно-технічна система (спеціалізована ЕОМ), що реалізує деяку формальну модель природної мережі нейронів. Про важливість розвитку цього напряму свідчить і те, що в основу побудови обчислювальних пристроїв п'ятого покоління покладено ідею паралельної обробки інформації в нейроноподібних системах. Незважаючи на те, що один електричний процесор працює в тисячі разів швидше, ніж його нейронний еквівалент у мозку, мережі нейронів розв'язують багато задач (особливо нечислових) в тисячі разів швидше, ніж електронний процесор.

Причина цього в наступному:

1) характер взаємозв'язків між нейронами дає змогу робити розв'язання багатьох задач, а також реалізацію різних функцій – у паралельний спосіб;

2) у нейронній мережі пам'ять не локалізована в одному місці (як в послідовних машинах), а розподілена по всій структурі; у біологічних системах пам'ять реалізується підсиленням або послабленням зв'язків між нейронами, а не зберіганням двійкових символів;

3) біологічні мережі реагують не на всі, а тільки на визначені зовнішні подразнення; кожен нейрон виступає як елемент прийняття рішення і елемент зберігання інформації; перевага такої структури – «життєздатність» (вихід з ладу декількох нейронів не спричинює значної зміни даних, що зберігаються, або руйнування всієї системи);

4) можливість адресації за вмістом є ще однією важливою характеристикою систем з розподіленою пам'яттю (кожен елемент відшукують за його вмістом, а не зберігають в комірці пам'яті з визначеним номером).

В основу зв'язків у нейрокомп'ютерах покладено асоціативний принцип. Асоціативні зв'язки пронизують все мислення людини. Наприклад, результат операції 5×2 у нас є в пам'яті, і ми кожного разу його не обчислюємо, на противагу комп'ютеру. Існує думка, що процеси мислення є не що інше як розповсюдження певного збудження, як деяка ланцюгова реакція. Навіть найпримітивніші процеси навчання принципово залежать від послідовності подій у часі. Це й закладено в природу нейронних систем. Тому їм притаманне реагування тільки на жорстко

визначене зовнішнє подразнення. Наприклад, домашні тварини «навчаються» ігнорувати повторні несуттєві зовнішні подразнення («цокання» годинника), але посилюють сприйняття подразнень, які можуть мати серйозні наслідки (звук гальм автомобіля).

Тому майбутнє – за комп'ютерами, які ґрунтуються на аналізі зв'язків, а не обробці символів. М. Мінський говорив, якщо комп'ютер має діяти подібно до мозку, тоді і його конструкція має бути також подібною до мозку.

Моделі нейронних мереж і схеми з адресацією за змістом мають і недоліки. Внаслідок нефіксованої організації вони можуть плутати різні об'єкти. Це аналогічно звиканню, посиленню чуттєвості до асоціацій, які лежать в основі психологічних особливостей людини. Інакше кажучи, довільний комп'ютер, що претендує на «розумність», повинен мати такі особливості.

Як було зазначено раніше, існує безліч мов програмування, які полегшують запис алгоритму розв'язання задачі. Для того щоб відповідні програми було виконано на конкретній машині, їх слід перевести в мову автокоду цієї машини. Для цього використовують інтерпретатори, асемблери і компілятори.

Робота інтерпретатора ґрунтується на тому, що для задання кожної операції в машині використовують особистий символ, так що увесь набір операцій програми можна задати простими символами. Вони інтерпретуються і послідовно обробляються машиною.

Асемблер застосовують до мов програмування, які дуже близькі до мови автокоду. Програму повністю перекладають з асемблерної мови в мову автокоду, а потім засилають на виконання.

Компілятори відрізняються від асемблерів тільки тим, що одне слово може позначати набір операцій мови машинних команд.

Слово *комп'ютер* походить від англійського слова *to compute*, яке означає «здійснювати обчислення». Інша назва комп'ютерів – електронні обчислювальні машини. Вона підкреслює два ключових аспекти. По-перше, комп'ютер – це сукупність апаратних засобів, призначених для обчислень і перетворення інформації. По-друге, основними елементами комп'ютерів є електронні прилади.

Комп'ютер здійснює обчислення та інші інформаційні перетворення за допомогою програм, тобто певних послідовностей інструкцій. Можна вважати, що за допомогою програм здійснюється керування апаратними засобами комп'ютера.

Найпростіші засоби для виконання простих арифметичних операцій були відомі людям у глибоку давнину. Перші механічні пристрої (арифмометри) створено в середині XVII ст. Паскалем і Лейбніцом. У XIX ст. Чарльз Беббідж уперше сформулював ідею створення універсальної обчислювальної машини. Він розробив детальний проект, який не був закінчений через відсутність необхідної технічної бази.

Значний внесок у розробку принципів функціонування комп'ютерів зробив Алан Тьюринг, який у 1937 р. описав гіпотетичну машину з універсальними можливостями. Ця машина Тьюринга, яку можна розглядати як одну з можливих формалізацій поняття «алгоритм», стала теоретичною основою сучасних комп'ютерів. Перша електронно-обчислювальна машина ENIAC створена у 1943–46 рр. Дж. У. Моклі та Дж. П. Екертом з Пенсільванського університету. Величезний внесок у розвиток обчислювальної техніки зроблено працею Дж. фон Неймана [138]. Початок серійного виробництва ЕОМ пов'язують зі створенням у 1951 р. ЕОМ UNIVAC. Роботи над створенням першої вітчизняної ЕОМ МЕЛМ (малої електронної лічильної машини) були розпочаті в Інституті електротехніки АН УРСР у 1947–1948 рр. групою науковців під керівництвом С. О. Лебєдєва. У 1951 р. машину було прийнято до експлуатації.

Як подальші віхи в розвитку обчислювальної техніки можна відзначити такі: поява операційних систем; виникнення мов програмування високого рівня; розвиток мережних технологій і масове розповсюдження персональних комп'ютерів.

Комп'ютери можна класифікувати за різними критеріями. Основним з них є *ступінь універсальності*. Комп'ютери бувають спеціалізовані, призначені для виконання окремих задач, і універсальні. Універсальну машину можна визначити як машину, що в принципі може розв'язувати будь-яку задачу, для якої існує алгоритм виконання, або як машину, здатну реалізувати будь-який алгоритм. Надалі розглядатимемо лише цифрові універсальні машини, і якщо не буде явно вказано інше, під словом *комп'ютер* ми розумітимемо саме такі машини.

Універсальний цифровий комп'ютер може реалізувати будь-який алгоритм. Інформація, яку обробляє комп'ютер, зберігається в його пам'яті у вигляді послідовності нулів та одиниць. Пам'ять складається з електронних схем, кожна з яких може перебувати в одному з двох стійких станів, один з них беруть за 0, а інший – за 1. Мінімальною одиницею інформації є *біт*. Біт – один двійковий розряд, за допомогою якого можна закодувати одне з двох можливих значень: 0 або 1.

Зрозуміло, що для виконання обчислень цифровий комп'ютер повинен мати у своєму складі засоби, що дають змогу виконувати операції над двійковими розрядами, тобто реалізовувати булеві функції. Булевою називається функція, аргументи і результат якої можуть набувати одне з двох можливих значень: 0 або 1.

Цифрові універсальні комп'ютери можна класифікувати за потужністю. Виділяють такі основні групи цих комп'ютерів.

Суперкомп'ютери. Для них характерна наявність великої (десятки, сотні або тисячі) кількості процесорів, які сумісно розв'язують ту чи іншу задачу, тим чи іншим способом взаємодіючи між собою; у цьому разі забезпечується дуже висока сумарна потужність. Типове призначення – виконання надскладних розрахунків. Найпростіші суперкомп'ютери можна використовувати як корпоративні мережні сервери. У цьому разі їх називають майнфреймами. Як приклади суперкомп'ютерів можна навести IBM System/390, T/9000, RS/6000. Ще більшу потужність забезпечують суперкомп'ютери фірм Intel, Fujitsu, Hitachi, Cray-SGI. Одна із останніх

розробок Intel об'єднує 9200 процесорів Pentium Pro, займає площу 160 $м^2$ і має масу 40 т. Пристрій для охолодження важить 272 т. Оперативна пам'ять становить 573 ГБ, швидкодія – понад трильйон операцій за секунду.

ЕОМ класу міні займають проміжне положення між великими ЕОМ і персональними комп'ютерами. Мають невелику кількість процесорів (від 1 до 8–12) та порівняно невеликі розміри і невисоку ціну. Використовуються як професійні робочі станції або як корпоративні мережні сервери.

Персональні комп'ютери. Їх можна визначити як комп'ютери, призначені для індивідуального користування. Поділяють їх на настільні та портативні. Останнім часом з'явився новий клас – персональні помічники, такі як Psion виробництва компанії Sharp, Newton Messag Pad фірми Apple Computer, Omni Go фірми Hewlett-Packard. Це мобільні комп'ютери, мініатюрні за розміром (300–4000 г), з порівняно невеликою оперативною пам'яттю (1–2 МБ) та малою тактовою частотою (7–9 МГц).

1.3. Основи фон-нейманівської архітектури

Значна частина сучасних комп'ютерів функціонує за принципами, сформульованими в середині 1940-х рр. Дж. фон Нейманом [142]. Він виділив пристрої, які мають входити до складу комп'ютера: керуючий; арифметико-логічний; оперативну пам'ять; зовнішню пам'ять (зовнішні запам'ятовуючі пристрої, зовнішні носії інформації); введення; виведення.

Роботу універсальної фон-нейманівської ЕОМ у найзагальніших рисах можна описати так.

Програми та дані, що зберігаються на зовнішніх носіях, вводять за допомогою пристроїв введення і пересилають до оперативної пам'яті. Обчислення здійснюють арифметико-логічним пристроєм. Інформацію, що є в оперативній пам'яті, за потреби передають до арифметико-логічного пристрою для обробки; проміжні результати обчислень знову передають в оперативну пам'ять. Результати роботи обчислювальної машини виводять на зовнішні носії за допомогою пристроїв виведення. Всі операції в ЕОМ здійснюють під управлінням керуючого пристрою.

Можна також виділити такі характерні риси фон-нейманівської архітектури: двійкова система числення; лінійна організація пам'яті, тобто пам'ять фон-нейманівського комп'ютера складається з послідовності однотипних комірок; будь-яка програма має вільний доступ до пам'яті за адресою; програма є послідовністю елементарних команд.

Елементарна команда – це команда, яку розуміє процесор і яку він може виконати безпосередньо. Після реалізації певної команди він починає виконувати наступну. У сучасних комп'ютерах арифметико-логічний і керуючий пристрої об'єднані в один, що називають центральним процесором.

Фон-нейманівська архітектура має багато «вузьких місць». Уже наприкінці 1950-х рр. намітилися тенденції відходу від фон-нейманівської архітектури. Зокрема, ці тенденції були пов'язані з розвитком багатопроцесорних систем. Нині можна вважати, що класична фон-нейманівська архітектура в основному вичерпала свої можливості.

Типовий центральний процесор фон-нейманівського комп'ютера має таку структуру. Регістри процесора використовують для тимчасового зберігання даних. В одному регістрі можна зберігати 2–4 байти інформації. Кількість розрядів, або бітів, в одному регістрі називають його розрядністю. Регістр адреси пам'яті містить адресу комірки пам'яті, з якої або в яку слід переслати дані. Регістр даних пам'яті містить дані, що мають бути зчитані в оперативну пам'ять. Програмний лічильник містить адресу комірки пам'яті, в якій зберігається команда, що виконується. Регістр команд містить команду, яку слід виконувати.

Типовий цикл виконання команди складається з трьох кроків: *вибірки*, *декодування*, *безпосереднього виконання*. На початку вибору чергової команди в програмному лічильнику міститься її адреса. Цю адресу пересилають до регістра адреси, а керуючий пристрій надсилає до оперативної пам'яті сигнал зчитування. Інформацію, зчитану з оперативної пам'яті, пересилають з пам'яті в регістр даних пам'яті, а звідти – в регістр команд. Аналогічним способом зчитують операнди команди.

Декодування – це перетворення двійкових кодів команд та операндів на послідовність електричних імпульсів, необхідних для виконання команд апаратними засобами. Типовим є мікропрограмне декодування, у разі якого процес декодування визначається мікрокодом. *Мікрокод* – сукупність правил, що визначають, яка послідовність імпульсів потрібна для виконання тієї чи іншої команди процесора. Цей мікрокод зберігається безпосередньо в процесорі. Його можна розглядати як програму, але програму ще більш низького рівня, ніж машинний код. Він є недоступним ні для користувачів, ні для програмістів, і є прерогативою розробників процесорів. Нарешті, на етапі безпосереднього виконання декодована команда реалізується апаратними засобами.

Робота усіх складових частин процесора синхронізується за допомогою електричних тактових імпульсів, що з певною частотою генерує керуючий пристрій. Час між двома імпульсами називають тактом. Одну команду можна виконати за один або декілька тактів. Тактовою частотою називають кількість тактів, що генеруються за 1 с. Від цієї характеристики істотно залежить швидкодія процесора.

Обмін даними між процесором та оперативною пам'яттю здійснюється через шину пам'яті. Фізично шина складається з певної кількості провідників. Через один провідник можна одночасно передати один біт інформації. Кількість таких провідників називають розрядністю шини. Один провідник інколи називають лінією. Шина пам'яті складається з двох частин: шина даних, адресна шина. Шину даних використовують для передачі даних, адресну шину – для передачі адреси.

Важливим для сучасних процесорів є поняття переривання. Переривання – сигнал, який надходить від якогось пристрою та змушує центральний процесор призупинити виконання програми і зайнятися обробкою переривання.

1.4. Автоматний спосіб задання алгоритму

Як згадувалося раніше, основою обчислювальної частини комп'ютера мають бути перетворювачі, що сприймають деякі вхідні послідовності нулів та одиниць і перетворюють їх на вихідні послідовності, тобто здійснюють інформаційні перетворення, в тому числі арифметичні операції. Такі перетворювачі назвемо логічними схемами.

Логічні схеми можна розділити на два класи. Перший з них – комбінаційні схеми. Значення вихідних змінних комбінаційної схеми визначаються лише значеннями вхідних змінних і не залежать від поточного стану схеми. Інший клас логічних схем – схеми з пам'яттю. Будь-яка комбінаційна схема реалізує ту чи іншу булеву функцію від своїх вхідних змінних.

На відміну від комбінаційної схеми скінченний автомат є перетворювачем, вихід якого залежить не тільки від вхідних і вихідних сигналів, але й від поточного стану автомата, причому кількість вхідних і вихідних змінних, кількість можливих значень цих змінних, а також кількість можливих станів автомата скінченна. Поточний стан автомата зберігається в його пам'яті.

Скінченні автомати мають досить обмежені обчислювальні можливості. Більше можливостей мають автомати з нескінченною пам'яттю магазинного типу, але й вони є недостатньо універсальними. Найбільш універсальною моделлю комп'ютерних обчислень є машина Тьюринга.

Автомати можна розглядати як машини, що мають пам'ять у вигляді стрічки та пристрій для зчитування інформації зі стрічки. Стрічка розділена на квадрати, в яких розміщуються визначені символи. Автомат, проглядаючи ці квадрати по черзі, виконує окремі обчислення і розв'язує задачі [65].

Кажуть, що автомати – це пристрої скінченного розміру, яким притаманна така властивість: визначений вихідний сигнал є функцією обробки вхідного сигналу.

Формально, найпростіший з автоматів – скінченний автомат, його можна описати так:

$$FA = \langle Q, A, b, q_0, F \rangle,$$

де $Q = (q_1, q_2, \ldots, q_n)$ – скінченна множина станів керування; $A = (a_1, a_2, \ldots, a_m)$ – вхідний алфавіт; $b : Q \times A \to Q$ – функція переходів; q_0 – початковий стан; $F \subseteq Q$ – множина заключних станів керування.

Приклад 1.1. Нехай нам потрібно побудувати скінченний автомат-розпізнавач. Він мусить допускати тільки послідовності символів 0 і 1, причому в послідовності розпізнавання кількість одиниць має бути парною, а нулів – непарною.

Згідно з умовою задачі, A складатиметься з символів 0, 1, #, де # позначатиме довільний символ, відмінний від 0 і 1. Інакше кажучи, $A = \langle 0, 1, \# \rangle$.

Множину станів Q виберемо таку:

$$Q = \langle q_0, q_1, q_2, q_3, q_4, q_5 \rangle,$$

де q_0 – початковий стан; q_1 – стан помилки; q_2 – стан, який визначає, що в послідовності кількість нулів непарна, а одиниць – парна (нуль вважатимемо парним числом); q_3 – стан, який визначає, що в послідовності кількість нулів і одиниць непарна; q_4 – стан, який визначає, що в послідовності кількість нулів і одиниць парна; q_5 – стан, який визначає, що в послідовності кількість нулів парна, а одиниць – непарна. Згідно з умовою задачі, $F = \langle q_2 \rangle$.

Функцію переходів визначають так:

$q_0\# \to q_1$	$q_0 0 \to q_2$	$q_3 0 \to q_5$
$q_1\# \to q_1$	$q_0 1 \to q_5$	$q_3 1 \to q_2$
$q_2\# \to q_1$	$q_1 0 \to q_1$	$q_4 0 \to q_2$
$q_3\# \to q_1$	$q_1 1 \to q_1$	$q_4 1 \to q_5$
$q_4\# \to q_1$	$q_2 0 \to q_4$	$q_5 0 \to q_3$
$q_5\# \to q_1$	$q_2 1 \to q_3$	$q_5 1 \to q_4$

Автомати зручно задавати табличним або графічним способом. Один з варіантів табличного задання автомата наведено на рис. 1.5.

Рядки таблиці позначають станами множини Q, а стовпці – символами вхідного алфавіту. Символ ← позначає заключні допустимі стани.

Рис. 1.6 ілюструє графічний спосіб опису функціонування автомата, де вершини графа q_i та q_j з'єднуються спрямованою дугою,

	#	0	1	
q_0	q_1	q_2	q_5	
q_1	q_1	q_1	q_1	
q_2	q_1	q_4	q_3	←
q_3	q_1	q_5	q_2	
q_4	q_1	q_2	q_5	
q_5	q_1	q_3	q_4	

Рис. 1.5. Табличне задання автомата

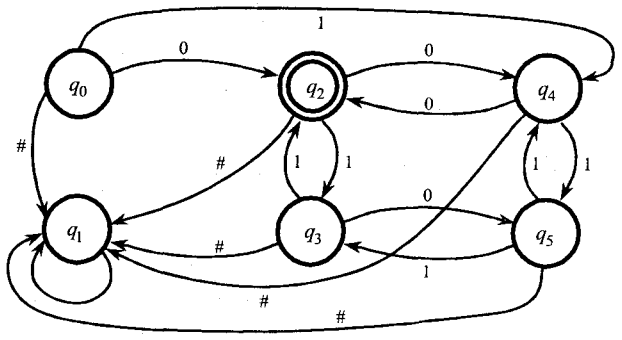


Рис. 1.6. Графічне задання автомата

позначеною символом a, якщо автомат має команду: $q_i a \to q_j$. Заключні допустимі вершини позначають подвійним кільцем.

Опишіть алгоритм розпізнавання зазначеного ланцюжка символів за допомогою мови програмування Паскаль.

Теорію автоматів можна використати і в нетрадиційних галузях, наприклад для вирішення проблеми безсмертя [41].

Цю задачу сформулюємо так. Нехай є машина, що здатна замінювати свої деталі в міру того, як вони починають виходити з ладу. Кожна деталь машини характеризується деяким середнім часом безвідмовної роботи. Тоді розв'язання проблеми безсмертя в постановці Е. Ф. Мура полягає в створенні машини, що мала б більший середній час безвідмовної роботи, ніж довільна з її деталей.

Відповідь виглядає так: якщо деяка машина складається не більш ніж з n деталей і ймовірність безвідмовної роботи кожної деталі не перевищує певної константи $k > 0$, тоді ймовірність одночасної відмови всіх деталей в якийсь момент часу дорівнює, в найгіршому разі, $1-(1-k)^n$. Тому настане час, коли всі деталі відмовлять одночасно.

Машина Тьюринга є спеціальним типом автомата. Вона функціонально складається зі стрічки, розділеної на квадратні комірки, та пристрою читання-запису. Останній може читати символ з комірки й записувати один символ в комірку, рухаючись уздовж стрічки в різні сторони. Є також програма роботи – набір четвірок чи п'ятірок, які визначають операції або обчислення машини як функцію від множини початкових символів на стрічці (входів). Наприкінці роботи отримують вихідний набір, який є сукупністю символів, що залишилися на стрічці, після того як було виконано всі операції, що задавались програмою.

Суттєвий факт, який вдалося довести за допомогою машини Тьюринга,– це існування машини Тьюринга, яка може за визначених вхідних

значень обчислити довільну, обчислювальну за Тьюрингом, функцію. Таку машину називають універсальною машиною Тьюринга.

Програму опису роботи машини Тьюринга M_1 можна розглядати як певну послідовність символів. Немає ніяких принципових перешкод для того, щоб розглядати цю програму як вхідні дані для деякої іншої машини M_y, що може моделювати машину M_1.

Універсальну машину Тьюринга можна неформально визначити як машину

$$\left\langle S^{(y)}, Q^{(y)}, \Pi^{(y)} \right\rangle,$$

що може сприймати програму Π для обчислення будь-якої функції, яку в принципі можна обчислити за допомогою спеціалізованої машини $\left\langle S^{(*)}, Q^{(*)}, \Pi^{(*)} \right\rangle$, і надалі працює як машина $\left\langle S^{(*)}, Q^{(*)}, \Pi^{(*)} \right\rangle$. Можна довести, що таку універсальну машину можна побудувати. Якщо у нас є така машина, то, згідно з тезою Тьюринга, для неї можна написати програму для обчислення будь-якої функції, обчислюваної в інтуїтивному розумінні, тобто запрограмувати будь-який інтуїтивний алгоритм.

1.5. Клас функцій, обчислюваних за Тьюрингом

Визначимо яким вимогам має задовольняти функція, для того щоб її можна було обчислити за Тьюрингом. Дамо деякі попередні визначення [71].

Базовими (вихідними) функціями називатимемо такі функції: *нуль-функція* – $Z(x)=0$ при будь-якому x; *додавання одиниці* – $N(x)=x+1$; *проектуючі функції* – $U_i^x(x_1,\ldots,x_n)=x_i$ при всіх $x_1,\ldots,x_n$.

Нові функції можна отримувати за допомогою таких правил:

1. *Підстановка.* Якщо $f(x_1,\ldots,x_n)=g\left(h_1(x_1,\ldots,x_n),\ldots,h_m(x_1,\ldots,x_n)\right)$, то кажуть, що функція сконструйована з функцій $g,h_1,\ldots,h_m$ за допомогою правила підстановки.

2. *Рекурсія.* Розглядають два випадки.

Якщо $n \neq 0$ і $f(x_1,\ldots,x_n,0)=g(x_1,\ldots,x_n)$; $f(x_1,\ldots,x_n,y+1)=h\big(x_1,\ldots,x_n, y, f(x_1,\ldots,x_n,y)\big)$, то кажуть, що функція сконструйована за допомогою рекурсії.

При $n=0$ визначення рекурсії таке: $f(0)=k$, де k – ціле невід'ємне число і $f(y+1)=h\big(y,f(y)\big)$.

3. μ-*Оператор.* Нехай функція $g(x_1,\ldots,x_n,y)$ така, що при будь-яких $x_1,\ldots,x_n$ існує хоча б одне значення y, за якого $g(x_1,\ldots,x_n,y)=0$.

Позначимо через $f(x_1,\ldots,x_n)=\mu\big(g(x_1,\ldots,x_n,y)=0\big)$ найменше значення y, за якого $g(x_1,\ldots,x_n)=0$. Тоді кажуть, що f сконструйована за допомогою μ-оператора.

Функцію f називають *примітивно-рекурсивною*, якщо її можна сконструювати з базових функцій за допомогою скінченного числа застосувань підстановок і рекурсій.

Функцію f називають *рекурсивною*, якщо її можна сконструювати з базових функцій за допомогою скінченного числа застосувань підстановок, рекурсій і μ-операторів.

Будь-яка примітивно-рекурсивна функція є водночас і рекурсивною. Клас примітивно-рекурсивних функцій дуже широкий. Навряд щоб вам доводилося мати справу з функціями, що є рекурсивними, але не примітивно-рекурсивними. Проте такі функції існують.

Частково рекурсивною функцією називатимемо функцію, що визначена не на всіх значеннях аргументу. Відома теорема:

Теорема 1.1. Клас функцій, частково обчислюваних за Тьюрингом, збігається з класом частково рекурсивних функцій.

Машини Тьюринга дають змогу обчислювати рекурсивні функції, і тільки їх. Однак залишається нез'ясованим ключове питання: чи не можна запропонувати інші формалізації поняття алгоритму і відповідні обчислювальні пристрої, що давали б змогу обчислювати нерекурсивні функції? А. Чорч висунув гіпотезу, яка отримала назву «тези Чорча».

Теза Чорча. Клас частково рекурсивних функцій збігається з класом частково обчислюваних функцій.

Теза Чорча еквівалентна тезі Тьюринга. Якщо ми визнаємо тезу Чорча, ми маємо визнати те, що, який би пристрій для здійснення обчислень ми не сконструювали, він не зможе робити більше, ніж машина Тьюринга. Він може здійснювати обчислення швидше, програмування для нього може бути легшим, але його обчислювальні можливості не вийдуть за межі рекурсивних функцій.

Тезу Чорча, як і рівнозначну їй тезу Тьюринга, довести неможливо, оскільки в її формулювання входить інтуїтивне поняття «обчислювана функція», яке неможливо точно сформулювати.

Тепер ми можемо дати точне визначення універсального комп'ютера. Універсальним називатимемо комп'ютер, на якому можна змоделювати роботу машини Тьюринга. Якщо ми маємо точне визначення поняття «алгоритм», ми можемо визначити, чи існує алгоритм для розв'язання тієї чи іншої задачі.

1.6. Моделі РАМ і РАСП

Можна зазначити, що машина Тьюринга є дуже незручною для програмування через перелічені нижче недоліки.

1. *Немає прямого доступу до пам'яті.* Якщо, наприклад, машина вказує на k комірку, а потрібно перейти до $k+n$, то вона має послідовно переглянути $k+1, k+2$ і т. д. комірки.

2. *Неструктурність записів на стрічці.* Заздалегідь невідомо, де закінчується одне число і починається інше; доводиться використовувати спеціальні роздільники.

3. *Дуже обмежений набір команд.* Відсутні, наприклад, основні арифметичні операції.

У праці [9] наведено дві моделі обчислювальних машин, які більш наближені до практичної реалізації і позбавлені вищезазначених недоліків: РАМ і РАСП.

РАМ є одноадресною машиною з довільним доступом до пам'яті. Вона складається з вхідної стрічки (з якої вона може тільки читати), вихідної стрічки (на яку вона може тільки записувати) і пам'яті. Вхідна і вихідна стрічки поділені на клітинки; у будь-якій з них може бути записане ціле число, можливо від'ємне. Пам'ять реалізується послідовністю регістрів; у будь-якому з них можна зберігати довільне ціле число. Структуру РАМ наведено на рис. 1.7.

Набір команд складається з одноадресних команд основних арифметичних операцій, читання та запису, переходу до потрібної команди та зупинки. Регістр r_0 має спеціальне призначення і називається акумулятором. Дію акумулятора прокоментуємо описом дії команди ADD a. Вона полягає у додаванні до акумулятора вмісту регістру з номером a; результат зберігається в акумуляторі.

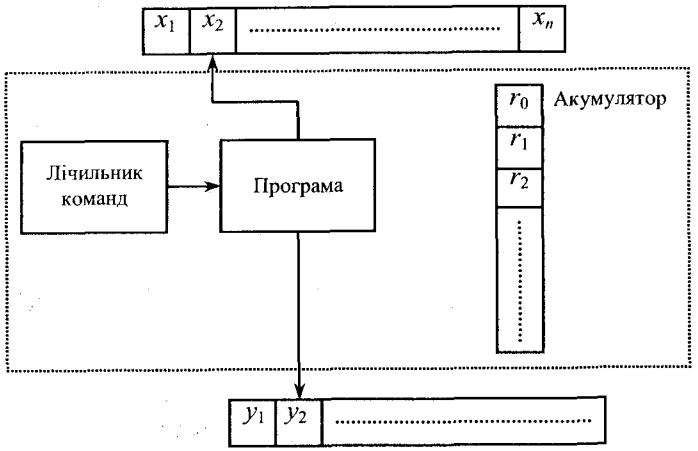


Рис. 1.7. Структура РАМ

Модель РАСП відрізняється від РАМ тим, що програма міститься в регістрах пам'яті й може змінювати сама себе. Чітко видно аналогії між РАСП і типовим фон-нейманівським процесором.

1.7. Складність алгоритмів

Якщо існує деяка проблема, то завжди намагаються знайти алгоритм її вирішення. Одну й ту саму задачу можна розв'язувати за допомогою різних підходів, різних алгоритмів. Постає питання, який з алгоритмів ліпший, ефективніший? Для цього потрібен спеціальний математичний апарат аналізу алгоритмів, їх складності [9, 39, 127]. Під складністю алгоритму розумітимемо часову оцінку роботи алгоритму або ж об'єм необхідної пам'яті для виконання алгоритму за допомогою деякої абстрактної обчислювальної машини. Визначення цих характеристик називатимемо *апріорним аналізом алгоритму*. Апріорний аналіз алгоритму ігнорує всі факти, що залежать від комп'ютера (обчислення здійснюють на РАМ-машині), мови запису алгоритму (мови програмування), і концентрує свої зусилля на визначенні або *типу* функції, яка характеризує час виконання алгоритму, або ж *типу* функції, що характеризує розмір потрібної пам'яті для виконання алгоритму.

Виконуючи повний аналіз часу обчислення або потрібної пам'яті, ми повинні проводити обов'язково дві фази: апріорний аналіз і тестування. Тестування дає змогу за допомогою експериментально отриманих статистик про час виконання алгоритму або необхідну пам'ять на різних даних визначити найтиповіші, найліпші та найгірші оцінки для різних випадків застосування алгоритму.

Алгоритми можна порівнювати і за допомогою інших критеріїв. Але визначення порядку зростання часової оцінки залежно від зростання розміру вхідних даних є проблемою, вирішення якої, з погляду комп'ютерного застосування, і в майбутньому буде найактуальнішим.

Із задачею асоціюватимемо *ціле* число, що характеризує розмір вхідних даних задачі. Наприклад, якщо ми розглядаємо задачу впорядкування, тоді таке ціле характеризує кількість елементів впорядкування, якщо задачу, пов'язану з обробкою матриці (обчислити визначник матриці), – тоді такою величиною буде розмірність матриці.

Час, необхідний алгоритму для вирішення проблеми, який є деякою функцією від розміру вхідних даних (розміру проблеми, задачі), *називають часовою складністю алгоритму*. Межову поведінку росту складності, залежно від розміру, називають *асимптотичною часовою складністю*. Аналогічно можна визначити складність алгоритму щодо пам'яті та асимптотичну складність алгоритму щодо пам'яті.

Можна було б вважати, що для сучасних комп'ютерів з величезною пам'яттю і швидкістю такі характеристики алгоритмів не актуальні. Однак це не так. Якщо часова оцінка алгоритму велика і досить великий розмір

задачі, то такий алгоритм навіть на сучасних суперкомп'ютерах за реальний час не можна виконати. Тому визначення типу часової оцінки алгоритму є досить важливою задачею.

Існує декілька спеціальних математичних позначень, які використовують під час аналізу алгоритму [9, 124].

Розглянемо запис $f(n)=O(g(n))$. Його читають так: функція f має порядок O від g від n тоді й тільки тоді, коли існують дві додатні константи c і n_0, такі що $|f(n)|\le c|g(n)|$ для всіх $n\ge n_0$.

Припустимо, ми визначили час обчислень $f(n)$ деякого алгоритму $f(n)=O(g(n))$.

Якщо кажуть, що алгоритм має час обчислень $O(g(n))$, мають на увазі таке: коли алгоритм буде оброблений комп'ютером на даних деякого типу, які мають розмір n, тоді результуючий час буде завжди меншим від добутку деякої константи на $|g(n)|$.

Під час визначення порядку $f(n)$ завжди намагатимемося отримати найменше $g(n)$, таке що $f(n)=O(g(n))$.

Оскільки нас цікавить межова поведінка часової складності алгоритму і те, що часова відмінність виконання базових операцій алгоритму (додавання, віднімання, множення, ділення, присвоєння) незначна, ми можемо взяти деяке одне середнє константне часове значення для характеристики виконання всіх базових операцій алгоритму. Що й зробимо.

Теорема 1.2. Якщо $A(n)=a_m n^m+\ldots+a_1 n+a_0$ є поліномом степеня m, тоді $A(n)=O(n^m)$.

Доведення.

$$|A(n)|\le|a_m|n^m+\ldots+|a_1|n+|a_0|\le\left(|a_m|+\frac{|a_{m-1}|}{n}+\ldots+\frac{|a_0|}{n^m}\right)n^m\le$$

$$\le\left(|a_m|+\ldots+|a_0|\right)n^m,\ \ n\ge 1.$$

Взявши $c=\left(|a_m|+\ldots+|a_0|\right)$ і $n_0=1$, отримаємо $A(n)=O(n^m)$.

Теорема 1.2 подає ідею знаходження часових характеристик для складних алгоритмів, що мають модульну структуру. Нехай алгоритм містить k блоків, часові оцінки яких відповідно мають вигляд $O(n^{m_1})$, $O(n^{m_2})$, ..., $O(n^{m_k})$, тоді часова оцінка всього алгоритму буде $O(n^{m_1})+\ldots+O(n^{m_k})$ і згідно з теоремою 1.2 є $O(n^m)$, де $m=\max\{m_i\}$, $i=1,\ldots,k$.

Розглянемо випадок, коли два алгоритми розв'язують одну задачу, вхід якої має довжину n. Перший має час обчислень $O(n)$, а другий – $O(n^2)$. Постає питання: якому з алгоритмів віддати перевагу? Очевидно, що для досить великих значень n час виконання другого алгоритму

буде значно більшим від часу виконання першого. Для невеликих n все залежить від значень константи c.

Наприклад, якщо дійсний час виконання цих алгоритмів є відповідно $2n$ і n^2, тоді перший алгоритм буде швидшим від другого для всіх $n > 2$. Якщо час виконань алгоритмів є $10^4 n$ і n^2, тоді другий алгоритм буде швидшим для всіх $n < 10^4$.

Тому ми не можемо твердити про перевагу одного алгоритму над іншим, доки не дізнаємося про характеристику константи і довжину входу.

Значення константи c на практиці залежить від багатьох факторів. У разі апріорного аналізу дослідження значень константи c не проводять, тому що для великих $n > n_0$ вони стають несуттєвими.

Найтиповішими часовими оцінками алгоритмів є $O(1)$ – *константна*, $O(\log n)$ – *логарифмічна*, $O(n)$ – *лінійна*, $O(n^m)$ – *поліноміальна* і $O(2^n)$ – *експоненційна*. Для досить великих n справедливе відношення

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n).$$

Зазначимо тільки, що часові оцінки $O(n)$ і $O(\log n)$ значно менші від інших. Для великих об'ємів даних алгоритми зі складністю, більшою за $O(n \log n)$, стають непрактичними. Алгоритми експоненційного типу можна використовувати на практиці тільки для дуже малих значень n і досить малого значення константи c.

Цікавим є й таке питання: чи існують коректно визначені математичні задачі, для яких немає алгоритму розв'язання? Тьюрингом доведено, що такі задачі є. Прикладом нерозв'язної задачі може слугувати проблема зупинки: для конкретної програми і вхідних даних визначити, чи зупиниться вона колись. Тьюринг довів, що не існує алгоритму, який би коректно вирішував всі часткові випадки цієї задачі.

Повернемося до визначення позначень.

$f(n) = \Omega(g(n))$ тоді й тільки тоді, коли існують додатні константи c і n_0, такі що $|f(n)| \geq c|g(n)|$ при $n > n_0$.

Існують випадки алгоритмів, для яких мають місце $f(n) = O(g(n))$ і $f(n) = \Omega(g(n))$.

Для цього випадку використаємо нотацію:

$f(n) = \Theta(g(n))$ тоді й тільки тоді, коли існують додатні константи c_1, c_2 і n_0, такі що для всіх $n > n_0$, $c_1|g(n)| \leq c_2|g(n)|$.

Якщо $f(n) = \Theta(g(n))$, тоді $g(n)$ є одночасно і нижньою, і верхньою межею $f(n)$. У цьому разі ліпший і гірший випадки залежать від значень констант, мають часову оцінку, яка лежить у межах від $\Omega(1)$ до $O(n)$.

Дамо більш строгу математичну нотацію:

$f(n) \sim O(g(n))$ (читається: «*f від n є асимптотою до $g(n)$*») тоді й тільки тоді, коли

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \to 1.$$

Тут $f(n)$ і $g(n)$ відрізняються тільки константним фактором. Наприклад, якщо $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$, тоді $f(n) = O(n^k)$ і $f(n) \sim O(a_k n^k)$.

Виділимо найбільш вживані суми оцінок. Для більшості алгоритмів сумарні оцінки мають вигляд $\sum_{g(n)\leq i\leq h(n)} f(i)$. У найпростіших випадках вони перетворюються на поліноми Бернуллі:

$$\sum_{i=1}^{n} 1, \sum_{i=1}^{n} i, \sum_{i=1}^{n} i^2 \ldots .$$

Зробимо оцінки цих сум:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2),$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3)$$

і в загальному випадку

$$\sum_{i=1}^{n} i^k = \frac{n^k}{k+1} + \frac{n^k}{2} + \text{lower order terms.}$$

Тому можна дійти висновку, що $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$, або точніше

$$\sum_{i=1}^{n} i^k \sim O\left(\frac{n^{k+1}}{k+1}\right).$$

Наведемо декілька прикладів побудови часових оцінок.

Розглянемо алгоритм, який визначає максимальний елемент з n елементів послідовності цілих чисел і описаний програмою 1.2.

```
program EX1_2;
  const n = 100;
  var max, i, m: integer;
  A: array [1..n] of integer;
begin
  readln(m);
  if m ≥ 1 then
    begin
      readln(A[1]);
      max := A[1];
      for i := 2 to m do
        begin
          readln(A[i]);
          if A[i] > max then max := A[i];
        end;
```

```
        writeln('Максимальний елемент = ', max);
      end
      else writeln('Порожня вхідна послідовність');
end.
```

Програма 1.2. Знаходження максимального елементу послідовності чисел

Часова оцінка цього алгоритму є одночасно і $O(m)$, і $\Omega(m)$, оскільки її визначає цикл **for**, який завжди потребує $m - 1$ ітерацій.

Розглянемо ще один алгоритм (програма 1.3) – алгоритм пошуку **search** елементу x в масиві A. Якщо знайдеться i, таке що $A[i] = x$, тоді перше таке i є результатом пошуку, в іншому разі результатом буде значення 0.

```
program EX1_3;
  const  n = 1000;
  type   int = 1..n;
         ar: array [int] of real;
  var x: real;
      j, k, l: int;
      a: ar;
  function  search (a: ar; m: int; x: real): int;
   begin
      i := 0; j := 1;
      while ((j <= m) or (i <> 0)) do
         if a[j] = x then i := j else j := j + 1;
      serch := i;
   end;
begin
   readln(x, k);
   for j := 1 to k read(a[j]);
   l := search (a, k, x);
   if l > 0 then writeln('x дорівнює елементу масиву з індексом i = ',l)
      else writeln('x не дорівнює жодному елементу масиву, i = 0')
end.
```

Програма 1.3. Пошук елементу в масиві

Алгоритм пошуку, реалізований функцією **search**, має часову оцінку, яка лежить в межах від $\Omega(1)$ до $O(k)$. У цьому алгоритмі потрібне значення можна знайти в найліпшому разі при першому ж порівнянні, а в найгіршому – переглянувши всі елементи масиву.

Модифікуйте алгоритм пошуку і проведіть аналіз його часової складності в разі, коли масив A впорядкований за зростанням.

1.8. Задачі класу *P* і *NP*

Терміни *задачі класу P* і *NP* з'явились на початку 1970-х рр. [39]. Вони стали символом великих труднощів, з якими борються розробники алгоритмів розв'язання задач постійно зростаючого розміру. Назвемо *P*-задачею таку, яку можна розв'язати на детермінованій машині Тьюринга за поліноміальний час $O(n^m)$, де n – розмір задачі. Інакше кажучи, для розв'язання *P*-задач існує поліноміальний алгоритм.

Під *детермінованою* машиною Тьюринга будемо розуміти машину, визначену в підрозділі 1.4. Детермінованість означає, що в будь-якій ситуації дія машини є чітко визначеною. На відміну від цього, недетермінованою машиною Тьюринга називають машину, команди якої мають вигляд $S_k q_i \to S'Q'G$. Це означає, що коли машина перебуває в стані q_i і читає символ S_k, вона може записати будь-який символ з множини S', перейти до будь-якого стану з множини Q' і зміститися вліво, вправо або залишитися на місці. Можна сказати, що на кожному кроці породжується певна кількість нових машин, і всі вони виконують ту чи іншу конкретну дію. Тепер можна дати визначення *NP*-задачі. *NP*-задачею називають таку, яку можна виконати за поліноміальний час на недетермінованій машині Тьюринга.

Багато задач математики, теоретичного програмування, дискретної математики, дослідження операцій належить до класу *NP*-**повних**, і список таких задач зростає з часом. Тому всім розробникам алгоритмів бажано розуміти сутність і значення цього поняття.

Для науковця характерним є прагнення розробляти ефективні алгоритми розв'язання якомога ширших класів задач. Історичний досвід розвитку математики, практика розв'язання багатьох проблем показали, що загальність методу і його ефективність перебувають у постійному антагонізмі: виграємо в одному – програємо в іншому. Під час розв'язання конкретних задач із застосуванням ЕОМ важливо знати, чи можна в ідеалі очікувати на створення достатньо загальних і ефективних методів, чи потрібно свідомо йти шляхом поділу задач на підзадачі і, користуючись їхньою специфікою, розробляти для них ефективні (можливо, евристичні) алгоритми, що даватимуть розв'язання підзадачі в реальний час.

Ці проблеми викликали потребу в аналізі складності задач. Проблематика аналізу актуальна тому, що такі задачі постійно виникають на практиці і їх вирішення на ЕОМ пов'язане з величезними затратами машинного часу і пам'яті.

Більшість дискретних і комбінаторних задач можна розв'язати за допомогою перебору варіантів. Наприклад, задачу про рюкзак типу $\sum_{i=1}^{n} a_i x_i = b,\ x_i = 0,1$ вирішують перебором варіантів булівських n-мірних векторів. Такі задачі називають перебірними. Число кроків перебору росте експоненційно залежно від розміру задачі. Для деяких задач із цього класу

вдається побудувати алгоритми, що виключають повний перебір варіантів (знаходження паросполук у графі). Проте число таких задач невелике.

Все це зумовило постановку центральної теоретико-методологічної проблеми всієї дискретної математики – чи можна виключити перебір під час розв'язання дискретних задач? Ця проблема має також велике пізнавальне значення. Під час пошуку ефективних точних методів розв'язання широкого класу дискретних задач слід враховувати можливість *відсутності таких методів* і, відповідно, вміти вчасно обмежувати своє прагнення, визнавши, що існують *складнорозв'язувані задачі*. Ця проблема залишається і нині [9].

Феномен складнорозв'язуваних задач відомий давно. Відразу ж після уточнення поняття алгоритму було виявлено значну кількість алгоритмічно невирішуваних проблем. Для доведення нерозв'язності коректно визначених математичних задач використовують апарат машин Тьюринга і правило звідності. Типовим прикладом нерозв'язної задачі може бути *проблема зупинки*. Тьюринг довів, що не існує алгоритму, який би коректно вирішував всі часткові випадки цієї задачі. Можна знайти деякі евристичні способи знаходження окремих нескінченних циклів за програмою і вхідними даними, але ніколи всіх можливих варіантів ми не врахуємо. Можна було б запустити програму на якомусь обчислювальному пристрої і чекати, доки вона не зупиниться. У разі, коли вона досягала при зупинці оператора закінчення програми, ми могли б дати відповідь на питання про зупинку. Однак така схема не буде алгоритмом, бо немає гарантії, що вона сама зупиниться.

У більшості перебірних задач є скінченна множина варіантів, серед яких потрібно знайти розв'язання. Зі збільшенням розміру входу задачі число варіантів швидко зростає і задача стає складнорозв'язуваною, інакше кажучи, практично нерозв'язною. Тому в остаточному разі аналогом алгоритмічної нерозв'язності є перебір експоненційного числа варіантів, а аналогом розв'язності – існування алгоритму поліноміального типу.

Для визначення типу задачі використовують поняття звідності задач з математичної логіки. Задача Π_1 зводиться до задачі Π_2, якщо метод розв'язання задачі Π_2 можна трансформувати в метод розв'язання задачі Π_1. Звідність називають поліноміальною, якщо подібне перетворення можна зробити за поліноміальний час. Тому можна виділити два основних класи задач: клас всіх перебірних *NP*-задач і клас перебірних *P*-задач, розв'язок яких можна отримати за поліноміальний час на машині Тьюринга. Зрозуміло, що $P \subseteq NP$. Головною проблемою теорії звідності є питання, чи збігаються класи *P* і *NP*. Так математично формулюють проблему елімінації (уникнення) перебору.

У класі *NP* є *NP*-повні задачі. Задачу називають *NP*-повною (універсальною), якщо до неї поліноміально зводиться довільна задача з класу *NP*. Їх використовують як еталони складності.

Сутність таких задач характеризує теорема Кука. Вона стверджує, що будь-яка *NP*-задача поліноміально зводиться до задачі про виконуваність булевого виразу. Цю задачу формулюють так: для будь-якого булевого

виразу знайти хоча б один набір значень змінних, який перетворює цей вираз на 1.

Якби вдалося довести, що якась *NP*-повна задача належить класу *P*, тоді було б доведено, що $P = NP$, і можна було б чекати на побудову ефективних алгоритмів для різних класів дискретних задач. Якщо ж класи *P* і *NP* відмінні, тоді потрібно розробляти ефективні алгоритми для більш вузьких класів задач. Більшість сучасних математиків вважають, що гіпотезу $P = NP$ неможливо ні довести, ні заперечити. Недоведеність відмінності між класами *P* і *NP* можна пояснити так. Ці класи визначають за допомогою часу роботи обчислювального пристрою з потенційно необмеженою пам'яттю. Добре відомо, що час виконання алгоритму на машині погано піддається опису та аналізу загальновідомими математичними засобами.

Задачі та вправи до розділу 1

1. Напишіть програму роботи скінченного автомата, що розпізнає мову з вхідним алфавітом $\{a, b\}$, яка складається з усіх ланцюжків з однаковим входженням *a* і *b*.
2. Доведіть, що всю стрічку скінченного автомата машини Тьюринга можна повністю закодувати одним цілим числом.
3. Напишіть програми роботи скінченного автомата для задач:
   а) обчислити $n!$ для вхідного n;
   б) прочитати n додатних чисел, за якими йде кінцевий маркер 0, а потім надрукувати їх за зростанням;
4. Проаналізуйте часову складність ваших програм із вправи 3.
5. Побудуйте машину Тьюринга, яка за двома даними двійковими цілими числами друкує їх суму.
6. Для кожної пари функцій $f(n)$ та $g(n)$ порівняйте порядок їх росту:
   а) $f(n) = n^{10}$, $g(n) = 2^{n/2}$;
   б) $f(n) = n^{3/2}$, $g(n) = n\lg^2(n)$;
   в) $f(n) = (n)^3$, $g(n) = \lg(n)$;
   г) $f(n) = \lg(3^n)$, $g(n) = \lg(2^n)$;
   д) $f(n) = 2n$, $g(n) = 2^{n/2}$;
   е) $f(n) = n^2$, $g(n) = \left(\frac{n}{2}\right)^2$.
7. Розташуйте такі функції за зростанням. Якщо деякі функції мають однаковий порядок, зазначте це.

$(\lg(n))^2$	$(2^n)^n$	$(\lg(n))!$	2^{n^n}
$\lg(2^n)$	$\lg(n!)$	$n^{\lg(n)}$	$n\lg(n)$
$n!$	$\lg(n)2^n$	n^2	$2^{\lg(n)}$
2^n	$\lg(n^2)$	$n(2^n)$	$\lg(\lg(n))$

8. Наступні дві функції, описані напівпаскалевською мовою, обчислюють k^n для цілих чисел k та n. Знайдіть часову складність цих функцій:

а)
```
function exp1(k, n);
begin power := 1;
    for i := 1 to n do
        begin newpower := 0;
            for j := 1 to k do
                newpower := newpower + power;
            power := newpower
        end;
    return(power)
end.
```

б)
```
function exp2(k, n);
begin power := 1;
    for i := 1 to n do
        power := power * k;
    return(power)
end.
```

9. Для кожного з наведених алгоритмів знайдіть його часову оцінку для найгіршого випадку.

```
{Ці функції обчислюють X у степені N, де N – додатне ціле число}
function power1(X, N);
begin if N = 1 then return(X)
    else return(X * power1(X, N–1))
end;
function power2(X, N);
begin if N = 1 then return(X)
    else
        begin HALF := N/2;
        HALFPOWER := power2(X, HALF)
        if 2*HALF = N                                    {N is even}
            then return(HALFPOWER * HALFPOWER)
            else return(HALFPOWER * HALFPOWER * X)   {N is odd}
    end
end;

function power3(X, N);
begin if N = 1 then return(X)
    else
        begin HALF := N/2;
            if 2 * HALF = N                              {N is even}
                then return(power3(X, HALF) * power3(X, HALF))
                else return(power3(X, HALF) * power3(X, HALF) * X) {N is odd}
        end
end.
```

10. Нехай є масив чисел розміру n. Слід вибрати з нього k найменших чисел та відсортувати їх. Вважати, що k набагато менше за n та набагато більше за 1.
А. Як мають бути змінені відомі алгоритми сортування для розв'язання цієї задачі?
Б. Знайдіть часові оцінки отриманих алгоритмів, які залежатимуть від k та n.

11. Вважатимемо, що масив A завдовжки n є майже відсортованим з похибкою k, $k < n$, якщо для довільних i, j, $j - i > k$, то $A[j] \geq A[i]$, тобто масив не є впорядкованим, але довільні два елементи, що не стоять на своїх місцях, не можуть знаходитися один від одного на відстані, більшій за k. Наприклад, масив 5, 8, 1, 6, 15, 12, 11, 20, 19, 25, 30, 35, 32 є майже впорядкованим з похибкою 2. $A[3] = 1$ менший за $A[1] = 5$, а $3 - 1 = 2$. Напишіть алгоритм сортування масиву з похибкою, меншою за k. Знайдіть часову оцінку наведеного алгоритму відносно k, вважаючи, що k набагато менше за n.

12. Для зберігання масиву S, елементи якого відсортовані, використовують структуру з двома полями:
а) число N – кількість елементів у S;
б) масив $A[1..M]$, перші N елементів якого є елементами S у порядку зростання (M вибирають більшим за довільне число N, яке може трапитися серед елементів масиву S); наприклад, для $M = 8$ множину $\{a, c, f, g\}$ задають парою $\{4, [a, c, f, g, –, –, –, –]\}$.
Напишіть наступні функції з наведеною часовою оцінкою:

Функція	*Часова оцінка*
MEMBER(x, S)	$O(S)$
ADD(x, S)	$O(S)$
DELETE(x, S)	$O(S)$
MIN(S)	$O(1)$
MAX(S)	$O(1)$
SUBSET($S1$, $S2$)	$O(\min(S2, S1 * \lg(S2)))$.

13. Опишіть довільну структуру даних, яка підтримує зберігання списку S цілих чисел. Для цієї структури напишіть наступні функції з відповідними часовими оцінками:

Функція	*Часова оцінка*
ADD(x, S)	$O(\lg S)$
DELETE(x, S)	$O(\lg S)$
MEMBER(x, S)	$O(\lg S)$
MIN(S)	$O(1)$
MAX(S)	$O(1)$
NEXT(x, S)	$O(\lg S)$. За заданим елементом x множини S знайти наступний елемент, більший за x.
PRED(x, S)	$O(\lg S)$. За заданим елементом x множини S знайти попередній елемент, менший за x.
COUNT(x, y, S)	$O(\lg S)$. Повернути кількість елементів в S, які лежать між x та y (включно).

Розділ 2

ПРОГРАМА = АЛГОРИТМ + СТРУКТУРА ДАНИХ

2.1. Об'єкти, імена та значення

Програми проводять свої дії над об'єктами даних, яким поставлено у відповідність їх імена.

Одна із сучасних концепцій програмування полягає в написанні програм мовами високого рівня процедурного типу. Програма містить певну послідовність команд, що мають виконуватися одна за одною. Кожна команда так чи інакше змінює вміст оперативної пам'яті або може здійснювати читання з диску чи запис на нього.

Дані, які розміщуються в оперативній пам'яті, можна розділити на: *неіменовані константи* – ділянки пам'яті, вміст яких змінюватися програмою не може; вони не мають назви та доступні тільки стандартним програмам; *іменовані константи* – мають, на відміну від неіменованих, імена та доступні всім програмам; *іменовані змінні* позначають ділянки пам'яті, що мають імена і можуть змінюватись. Класичним методом, за яким програма високого рівня отримує доступ до певної ділянки пам'яті, є доступ за іменем. Під час компіляції визначається, якому імені відповідає певна адреса ділянки пам'яті. Втім, практично всі мови високого рівня припускають і безпосереднє звертання до тих чи інших ділянок пам'яті за адресою. У найзагальнішому вигляді, команди задають виконання таких операцій: *присвоєння* – надання змінній нового значення; *введення змінної з файла* – присвоєння змінній початкового значення; *виведення змінної до файлу або на екран* – перенесення значення змінної на відносно довгострокове зберігання або для візуального огляду. Виділяють також такі інструкції керування: *складна команда*, що є засобом групування певної послідовності команд; *перехід до іншої команди* (*послідовний* – після закінчення чергової команди, за винятком команди завершення програми, керування передається наступній команді в програмі; *безумовний перехід* – управління передається на команду, позначену міткою, що знаходиться в контексті команди безумовного переходу; *умовний перехід* – «якщо вірна умова *A*, виконати команду *X*, якщо не вірна – виконати команду *У*»); *цикл* – повторити команди певне число разів або до виконання певної умови. Ці інструкції є значною мірою взаємозамінюваними. Команди в термінології мов високого рівня називають *операторами.*

Н. Вірт наголошував, що «програма = алгоритм + структура даних» [20]. Мови високого рівня традиційно містять різноманітні набори стандартних типів даних. Зрозуміло, що жодна мова програмування не може зосередити в собі набір стандартних типів даних, який би задовольняв всі вибагливі смаки програмістів. Тому слід знати можливість, особливості та переваги використання як стандартних типів даних, так і нестандартних. Більшість сучасних мов програмування має зручний апарат конструювання потрібних типів даних для ефективної реалізації алгоритму розв'язання задачі. Зупинимось на визначенні базових понять.

2.1.1. *Імена і покажчики*

Серед програмістів часто можна почути фрази типу «*X* має значення 2», «*X* має значення *Y*». Та насправді ми мали б говорити: «*X* – це місце в пам'яті, де в даний момент зберігається число 2», «*X* – це місце в пам'яті, де в даний момент зберігається число, назване *Y*».

Мішанина між «*де міститься*» і «*що міститься*» виникає тому, що в обох випадках ми використовуємо одне ім'я. Наприклад, під час розгляду оператора присвоєння $i := i + 1$, i лівої частини визначає, де міститься значення, а i правої – саме це значення.

Для усунення цих колізій у разі формалізації змінної вводять два об'єкти: покажчик (визначає місце в пам'яті) і значення (об'єкт збереження, який обробляється програмою) [21]. Змінну задають ім'ям. Взаємозв'язок між ними характеризує рис. 2.1.

Покажчик можна трактувати як деякий різновид імені, оскільки за його допомогою можна отримати значення, але він відомий тільки компілятору. Для перетворення покажчика в значення використовують функцію «*зміст*». Корисним використанням покажчиків є наявність можливості того, що вираз може також давати покажчик, а не значення.

Отже, обчислення виразу можна описати у такий спосіб. Вираз будують з імен, констант і операцій. Спочатку імена заміщують покажчиками. Потім у певному порядку виконують операції, і якщо в операції, яка вимагає як операнд значення, трапляється покажчик, застосовують операцію *зміст.*

Покажчик описує деяку область пам'яті, достатню для збереження об'єкта, і може відповідати набору адрес фізичної пам'яті. Відображення покажчиків на фізичні адреси може змінюватися в процесі виконання програми, якщо використовується динамічний механізм розподілу пам'яті. Область пам'яті може змінюватись, але покажчик завжди знаходиться на одному місці. Обчислення «*адреси*» можливе тільки в мовах високого рівня, якщо простір адрес є впорядкованим за покажчиками.

Рис. 2.1. Взаємозв'язок між об'єктом і значенням

2.1.2. *Атрибути даних*

Встановлення значення як об'єкта збереження приводить до того, що об'єктами збереження можуть бути числа, логічні значення, символи, символьні ряди і покажчики. Покажчики дають змогу оперувати зі змінними, значеннями яких є інші змінні *(неявна адресація)*, що використовують для роботи зі *списками.*

Атрибути об'єкта відбивають його тип. Вони можуть бути явними (описаними) або неявними (виявляються в процесі певного аналізу).

Більшість сучасних комп'ютерів здійснює обробку інформації, використовуючи її задання в двійковій системі числення. Очевидно, що за допомогою певної послідовності 0 та 1 можна закодувати довільну інформацію. У таких системах основною структурою пам'яті, що вважають неподільною, є *біт*. У ньому може бути записано 0 або 1. Послідовність з 8 бітів називають *байтом*. В одному байті можна зберегти один символ. Дані різних типів займають різну кількість пам'яті.

2.1.3. *Концепція типу даних*

У програмі кожна константа, змінна, вираз або функція завжди мають бути певного типу. Тип визначає множину значень, з якими може працювати програма. Найбільш вживаним підходом для визначення типу є використання його явного задавання в описі констант, змінних і функцій в тексті програми. Транслятор, відповідно до цього опису, обирає задання об'єкта в пам'яті ЕОМ.

Для встановлення концепцій типу даних використовуватимемо підхід Вірта: довільний тип даних визначає множину значень, до якої належить константа, які може набувати змінна (вираз) або виробляти операція чи функція; тип значення можна визначити або за виглядом відповідної константи, змінної, функції, або за їх описом без необхідності виконань якихось обчислень; кожна операція або функція вимагає аргументів фіксованого типу і виробляє результат фіксованого типу, але тип результату можна визначити за спеціальними правилами мови програмування [21].

Отже, транслятор може використовувати інформацію про типи для перевірки вірогідності обчислень і правильності різних конструкцій мови.

Число різних значень, що належать до типу T, називають кардинальним числом T. Воно визначає розмір потрібної пам'яті для розміщення змінної x типу T і позначається $x : T$.

Зрозуміло, що для ієрархічної побудови типів повинні бути базові, атомарні типи. Один із методів опису простого типу – це перелік. Наприклад, в Паскалі всі неструктуровані типи впорядковані, і у разі, коли значення явно перелічуються, вважають, що вони впорядковані за порядком переліку.

Серед статичних типів, які будують згідно із зазначеним принципом, можна виділити: масив, запис, множину, файл. Складніші структури

типу списків, дерев, графів будують динамічно, під час виконання програми.

Найбільш вживаними операціями під час визначення типів є явні операції перевірки рівності і присвоєння та неявна операція перетворення. Останні перетворюють одні типи даних в інші. До них належать операції конструювання і селектування, які відображають типи компонент в складні типи і навпаки.

2.1.4. *Прості типи даних*

Для роботи з даними у невеликому діапазоні значень вводять простий неструктурований тип T переліком цих значень $c_1, c_2, ..., c_n$:

$$\textbf{type } T = (c_1, c_2, ..., c_n).$$

Кардинальним числом T буде $\textbf{card}(T) = n$. Перевагою такого визначення порівняно з числовими значеннями є наочність.

Наприклад:

```
type фрукти = (яблуко, груша, вишня, черешня);
var X: фрукти;
```

Тоді можливе X := груша.

Якщо тип вважають впорядкованим, тоді визначають дві функції **pred**(X) і **succ**(X), які видають попереднє і наступне, відповідно, значення від свого аргументу. Впорядкованість значень типу T визначають правилом:

$$(c_i < c_j), \text{ коли } (i < j).$$

2.1.5. *Стандартні прості типи*

До стандартних простих типів відносять цілий (**integer**), дійсний (**real**), логічний (**boolean**) і символьний (**char**) типи.

Тип **integer** використовують для задання підмножини цілих чисел. Розмір цієї підмножини залежить від типу ЕОМ. Всі стандартні операції додавання (+), віднімання (–), ділення (**div**, **mod**), множення (*) дають точний результат. Якщо значення виходить за межі підмножини, видається повідомлення про помилку.

Тип **real** використовують для задання підмножини дійсних чисел. Враховуючи відомий математичний факт про щільність множини дійсних чисел (між довільними двома дійсними числами є безліч дійсних чисел), можна стверджувати, що дійсні числа в ЕОМ можна відображати тільки з певною точністю. Тому за реалізації арифметичних дій над ними можлива неточність у межах похибок заокруглень, тип **real** не є зліченним.

Для реалізації операцій булевої логіки: заперечення (**not**), диз'юнкції (**or**), кон'юнкції (**and**) – використовують стандартний тип **boolean**. Він має два значення: **true** (істина) і **false** (хибність).

Стандартний тип **char** задає множину символів, яка реалізована в комп'ютері. Найбільш вживаними наборами символів є набори: **ISO** – код міжнародної організації зі стандартизації; **ASCII** – американський стандартний код. Для більшості кодів характерним є дотримання таких домовленостей:

1. Тип **char** містить 26 латинських букв, 10 арабських цифр, певну кількість інших графічних символів (знаки, розподільники).
2. Підмножини букв і цифр впорядковані й зв'язані:

$$(\text{'A'} \le x) \,\&\, (x \le \text{'Z'}),\ \text{де } x - \text{буква};$$
$$(\text{'0'} \le x) \,\&\, (x \le \text{'9'}),\ \text{де } x - \text{цифра}.$$

3. Тип **char** містить недрукований пустий символ (_).

Для перетворень між типами **char** і **integer** сучасні мови програмування високого рівня традиційно мають дві вмонтовані функції **ord**(c) і **char**(i):

ord(c) – порядковий номер символу c в множині **char**;

char(i) – i-й символ множини **char**.

Ці функції також можна використовувати для переведення внутрішнього задання чисел у послідовності цифр і навпаки.

2.1.6. *Обмежені типи*

Якщо змінна набуває значення деякого типу тільки у визначеному інтервалі від **min** до **max**, тоді її можна визначити як змінну обмеженого типу:

type T = min..max.

2.1.7. *Масиви*

Масивом називають регулярну структуру, всі компоненти якої однотипні (базовий тип) і доступ до компонент має випадковий характер (всі компоненти можна вибирати довільно і вони є рівнодоступними).

Ідентифікацію компонент забезпечує *індекс* масиву. Він повинен мати значення типу *індексів масиву*. Тому опис масиву має вигляд:

type T = **array** [I] **of** T_0,

де T_0 – базовий тип, а I – тип індексу.

Наприклад, тип вектора цілих чисел розмірності 10 можна описати у такий спосіб:

type vector = **array** [1..10] **of integer**;

Якщо ми опишемо змінну a як **var** a: *vector*;, тоді доступ до компонент цієї змінної забезпечуватиме операція селектор, що має вигляд $a[i]$, де $1 \le i \le 10$.

Індекси масиву мають бути визначеного скалярного типу. Замість індексної константи можна використовувати індексний вираз. За використання останнього програмісту слід перевіряти, чи отримане значення індексного виразу не виходить за межі визначення типу індексу.

Компоненти масиву можуть бути також складними. Змінну-масив, компоненти якої є масивами, називають матрицею. Наприклад, опис A: ***array*** [1..3] ***of*** *vector* визначатиме матрицю 3×10 з цілими компонентами. Для доступу до компонент матриці використовують селектор такого вигляду: $A[i,j]$.

Змінна $A[i,j]$ визначає j-ту компоненту i-го рядка матриці. Найбільш використовуваним скороченням конструктора типу масив є таке:

type Matrix = **array** [1..3, 1..10] **of integer**; .

2.1.8. *Записи*

Складні типи можуть утворюватись із довільних типів шляхом їх об'єднання. Кардинальне число такого типу визначатиметься добутком кардинальних чисел всіх типів компонент. Для реалізації «*об'єднання*» використовуватимемо мнемоніку **record** … **end.** Тоді складний тип у загальному вигляді можна визначити у такий спосіб:

```
type T = record
           s1: T1;
           s2: T2;
           ...
           sn: Tn;
         end;
```

а кардинальне число

$$\text{card}(T) = \text{card}(T_1) * \text{card}(T_2) * \ldots * \text{card}(T_n).$$

Якщо маємо r: T, тоді доступ до її i-ї компоненти визначатиметься як $r.s_i$. Наприклад, для побудови інформаційно-довідкової системи факультету необхідно мати узагальнену інформацію про абітурієнта, яка складається з прізвища, дати народження, місця народження, статі, національності, середнього бала атестата. Тоді можна задати визначення типу:

```
type  date = record
               day: 1..31;
               month: 1..12;
               year: integer;
             end;
```

```
archar = array [1..30] of char;
abit = record
          fio: archar;
          da: date;
          place: archar;
          nat: archar;
          ba: real;
       end; .
```

2.1.9. *Файли*

Будь-які файли, а також логічні пристрої стають доступними програмі тільки після виконання особливої процедури відкриття файла (логічного пристрою). Ця процедура полягає у зв'язуванні раніше оголошеної файлової змінної з ім'ям файла, що вже існує або що заново створюється, а також у визначенні напряму обміну інформацією: читання з файла чи запис в нього [51].

Файлову змінну пов'язують з ім'ям файла внаслідок звернення до стандартної процедури ASSIGN:

ASSIGN(<ф.з.>, <ім'я файла>);

де <ф.з.> – файлова змінна (правильний ідентифікатор, оголошений в програмі як змінна файлового типу); <ім'я файла> – текстовий вираз, що містить ім'я файла.

Ініціювати файл – означає вказати для цього файла напрям передачі даних. В Паскалі можно відкрити файл для читання, для запису інформації, а також для читання і запису одночасно.

Для читання файл ініціюють за допомогою стандартної процедури RESET:

RESET(<ф.з.>).

Файлову змінну раніше пов'язують за допомогою процедури ASSIGN з уже існуючим файлом.

Стандартна процедура REWRITE(<ф.з.>) ініціює запис інформації у файл, пов'язаний раніше з файловою змінною. Процедурою REWRITE не можна ініціювати запис інформації в уже існуючий дисковий файл: під час виконання цієї процедури старий файл знищується і ніяких повідомлень про це в програму не передається. Новий файл готується до прийняття інформації, і його покажчик набуває значення 0.

Стандартна процедура APPEND(<ф.з.>) ініціює запис у існуючий текстовий файл для його розширення, у цьому разі його покажчик встановлюється на його кінець. Процедуру APPEND застосовують тільки до текстових файлів, тобто їхня файлова змінна повинна мати тип TEXT. Процедурою APPEND не можна ініціювати запис до типізованого або нетипізованого файла. Якщо текстовий файл раніше вже був відкритий за допомогою RESET або REWRITE, використання процедури APPEND

зумовить закриття цього файла і відкриття його знову, але вже для поповнення записів.

Процедура CLOSE закриває файл, але зв'язок файлової змінної з ім'ям файла, встановлений раніше процедурою ASSIGN, зберігається. Формат її використання типовий: CLOSE(<ф.з.>).

При створенні нового або розширенні старого файла процедура забезпечує збереження у файлі всіх нових записів і реєстрацію файла в каталозі.

Розглянемо приклади роботи з файлами на конкретних задачах.

Приклад 2.1. Нехай є символьні файли *f* та *g*. Визначити, чи збігаються компоненти файла *f* з компонентами файла *g*. Якщо ні, то отримати номер першої компоненти, в якій файли *f* та *g* різняться. У разі, коли один з файлів має *n* компонент ($n \geq 0$) і повторює початок іншого (довшого) файла, відповіддю має бути $n + 1$.

Розв'язок задачі наведено у програмі 2.1.

```
program EX2_1;
uses crt ;
var
   f, g: text;                          {Дані файли}
   i, k, riv, minlen, pr: integer;
   a, b: string;
begin
   clrscr;
   k := 1;                              {Номер компоненти, в якій файли різняться}
   assign(f,'a:\494in1.txt');           {Відкриття та ініціація файлів}
   reset(f);
   assign(g,'a:\494in2.txt');
   reset(g);
   while not eof(f) do
      while not eof(g) do
         begin
            readln(f,a);                {Зчитуємо вміст файлів у змінні a та b}
            readln(g,b);
         end;
   if length(a) < length(b) then minlen := length(a)        {Порівнюємо кількість}
      else                                                  {компонент файлів}
         if length(a) > length(b) then minlen := length(b)  {і визначаємо меншу}
            else
               begin
                  minlen := length(a);
                  riv := 1;             {Якщо компонент порівну, то помічаємо}
               end;                     {це, встановивши «прапорець» riv в 1}
   for i := 1 to minlen do
      begin
         if a[i] = b[i] then            {Порівнюємо компоненти}
```

```
            begin
               if (riv = 1) and (i = minlen) then riv := 2; {Якщо кількість компонент}
               {однакова, поточні компоненти однакові, встановлюємо riv в 2}
               k := k + 1;          {Підраховуємо номер наступної компоненти}
            end
            else
               begin
                  pr := 1;          {Якщо компоненти відрізняються, встановимо}
                  break;            {«прапорець» pr в 1 і виходимо з циклу}
               end;
      end;
   if riv <> 2 then writeln('Number of different component in f and g is ',k)
         {Якщо кількість компонент не рівна, виводимо номер компоненти k}
      else if pr <> 1 then writeln('Files have no differences')
                  {Якщо кількість компонент рівна і вихід з циклу відбувся}
                  {не через різницю компонент (pr <> 1), файли не різняться}
           else writeln('Number of different component in f and g is ',k);
                  {Якщо вихід через різницю компонент, виводимо номер}
   close(g);
   close(f);
   repeat until keypressed;
end.
```

Програма 2.1. Перший приклад роботи з файлами

Приклад 2.2. Задані символьні файли *f* та *g*. Записати у файл *h* всі початкові компоненти файлів *f* та *g*, які є однаковими.

Розв'язок задачі наведено у програмі 2.2.

```
program Ex2_2;
uses crt;
var
   f,g,h: text;                                   {Дані файли}
   i,minlen: integer;
   a,b,c: string;
begin
   clrscr;
   assign(f,'a:\495in1.txt');          {Відкриття та ініціація}
   reset(f);                           {файлів, що містяться на дискеті}
   assign(g,'a:\495in2.txt');
   reset (g);
   assign(h,'a:\495out.txt');          {Вихідний файл також буде на дискеті}
   rewrite(h);
   while not eof(f) do
      while not eof(g) do
```



```
         begin
            readln(f,a);                 {Зчитуємо вміст файлів у змінні a та b}
            readln(g,b);
         end;
   if length(a) < length(b) then minlen := length(a)  {Порівнюємо кількість компонент}
      else minlen := length (b);                      {файлів і визначаємо меншу}
   for i := 1 to minlen do
      begin
         if a[i] = b[i] then                  {Порівнюємо компоненти}
            begin
               c[i] := a[i];
               c := c + c[i];          {Якщо компоненти рівні, формуємо рядок}
            end
            else break;                {Якщо ні, виходимо з циклу}
      end;
   write(h,c); {Записуємо у файл h сформований рядок з компонент, які є однаковими}
   writeln(c);                         {Виводимо цей рядок на екран}
   close(h);
   close(g);
   close(f);
   repeat until keypressed;
end.
```

Програма 2.2. Другий приклад роботи з файлами

Приклад 2.3. Є символьний файл *f*. Записати у файл *g* зі збереженням порядку символи файла *f*, яким у цьому файлі передує літера *a*.

Розв'язок задачі наведено у програмі 2.3.

```
program Ex2_3;
uses crt ;
var
   f,g: text;                                     {Дані файли}
   i,pr: integer;
   a,b: string;
begin
   clrscr;
   assign(f,'a:\496(a)in.txt');                   {Відкриття та ініціація файлів}
   reset(f);
   assign(g,'a:\496(a)out.txt');
   rewrite(g);
   while not eof(f) do readln(f,a);
   for i := 1 to length(a) do
      begin
         if a[i] = 'a' then pr := 1;     {Перевіряємо чи є дана літера літерою a, i,}
                                         {якщо є, встановлюємо «прапорець» pr в 1}
```

```
        if (pr = 1) and (a[i] = 'a') then          {Пропускаємо цю літеру a}
          else if pr = 1 then  {Перевіряємо «прапорець» (тобто чи трапилась}
            begin              {вже літера a) і, якщо він 1, формуємо рядок}
              b[i] := a[i];    {з подальших символів}
              b := b + b[i];
            end;
      end;
    write(g,b);                          {Записуємо сформований рядок у файл g}
    writeln(b);                          {Виводимо цей рядок на екран}
    close(g);
    close(f);
    repeat until keypressed;
end.
```

Програма 2.3. Третій приклад роботи з файлами

2.2. Лінійні списки

Однією із структур даних, що найбільше використовують у програмуванні, є лінійні списки.

Список – це скінченна послідовність однотипних елементів. Кількість елементів послідовності називають довжиною списку. Списки залежно від організації зв'язку між елементами бувають однозв'язними (лінійними), двозв'язними (лінійними), двозв'язними (циклічними) тощо.

Лінійний список L, що складається з елементів $l_1, l_2, \ldots, l_n$, запишемо у вигляді: $L = \langle l_1, l_2, \ldots, l_n \rangle$ і зобразимо графічно, як показано на рис. 2.2.

Вперше поняття списку введено у праці Ньюела, Саймона, Шоу [80] при розробці програми «Логік–Теоретик» і вперше впроваджено за реалізації операцій з пам'яттю в ЕОМ.

Під час роботи зі списками найбільше використовують такі операції: знаходження елементу в списку із заданою характеристикою, вилучення певного елементу із списку, внесення елементу у спеціально визначене місце списку, здійснення певних дій над елементами списку.

У більшості мов програмування, за винятком мов, подібних до Ліспу або Прологу, не існує стандартних механізмів для задання лінійних списків.

Можна виділити два основні методи задання лінійних списків: послідовне і зв'язне зберігання. Перше переважно реалізується за допомогою

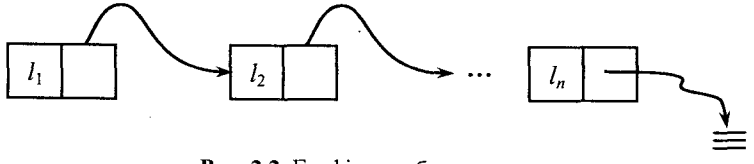


Рис. 2.2. Графічне зображення списку

масивів, а друге – за допомогою динамічних змінних. Під час вибору засобів зберігання списку слід враховувати, які операції і з якою послідовністю виконуватимуться над ним.

Розглянемо варіант послідовного зберігання списку і реалізацію основних операцій над ним у цьому разі. Припустимо, що ми маємо глобальний опис

```
...
const m = 100;
type  lisp = array [1..m] of integer;
var   L: lisp;
      n: 1..m;
...
```

для наступних процедур і функцій, що реалізують основні операції обробки списків.

Сформувати список можна за допомогою процедури:

```
procedure form(k: 1..m);
var j: 1..m;
begin
  for j := 1 to k do
    read('Формуєм ', j, '-й елемент списку', L[j]);
end;
```

У цьому разі процедура **printl** друкуватиме всі елементи списку завдовжки p.

```
procedure printl(p: 1..m);
var j: 1..m;
begin
  if p > 0 then
    for j := 1 to p do
      writeln('j-й елемент списку буде ', L[j])
      else writeln('список пустий')
end;
```

Замість друку повідомлення «список пустий» може стояти виклик довільної процедури обробки випадку, коли список не має жодного елементу.

Вважаємо, що для читача буде зрозумілим, що реалізація операцій видалення та поповнення елементів списку реалізовуватиметься за допомогою подібних операцій маніпулювання з індексами та елементами масиву.

У разі використання списку як базової структури для побудови складніших структур даних, а також для економії пам'яті, вживається зв'язне зберігання лінійних списків з використанням динамічних змінних. У цьому

разі кожен елемент списку складається з двох полів: поля значення **val** і поля зв'язку **next**. Опис такої структури даних матиме рекурсивний вигляд:

```
type  linln = ^ node;
      node = record
               val: integer;
               next: linln
             end;
```

Функція **forme** формує елемент списку зі значенням *g*, на голову якого вказує покажчик **forme**:

```
function forme(g: integer): linln;
begin
   new(forme);
   forme^.val := g;
   forme^.next := nil;
end;
```

де **new**(*p*) – вмонтована функція виділення пам'яті під структуру даних, на яку вказує покажчик *p*, а **nil** – вмонтована функція, яка визначає, що покажчик ні на що не вказує, і використовується для позначення закінчення списку (пустого списку).

Вибрані ідентифікатори для позначення елементів вузла списку не є фіксованими в мовах програмування. Тому їх мнемоніка вибирається відповідно до природи задачі розв'язання або уподобань програміста. Нехай, наприклад, потрібно розв'язати таку задачу. На вхід функції **element** подають ціле число і покажчик на список цілих чисел. Вона мусить визначити, чи містить список, на який вказує покажчик, це ціле число. Одним із варіантів вибору позначень змінних і розв'язання буде такий:

```
type  numlist = record
                  value: integer;
                  next: ^numlist
                end;

function element(X: integer; Q: ^numlist): boolean;   {Перевіряє чи X є елементом}
                                                      {зв'язного списку Q}
var found: boolean;                                   {Ознака виявлення, що X вже знайдено}
begin
   found := false;
   while (Q <> nil and not found) do
     begin
        found := Q^.value = X;
        Q := Q^.next;
```



```
     end;
   return(found)
end;
```

Після цього можна розв'язувати і складнішу задачу. Нехай на вхід функції **subset** подаються покажчики на два списки, що задають відповідно дві множини. Слід визначити, чи перша множина входить у другу множину.

```
function subset(L, M: ^numlist)          {Перевіряє, чи L є підмножиною M}
var success: boolean;                    {Ознака визначення, що L є підмножиною}
begin
   success := true;
   while (L <> nil) and success do
     begin
        success := element(L^.value, M);
        L := L^.next;
     end;
   return(success)
end;
```

Знайдіть найгіршу часову оцінку функції. В якому випадку її досягають?

Розглянемо ще один приклад використання списків під час вирішення задачі динамічного розподілу пам'яті. Алгоритмами динамічного розподілу пам'яті називають алгоритми, що дають змогу виділяти і звільняти різні за розміром блоки пам'яті, які беруть з однієї великої ділянки пам'яті.

Вважатимемо, що вся пам'ять задана списком вільних блоків. Кожен блок містить заголовок з розмірами блоку і покажчиком на наступний блок. Виділення пам'яті за запитом виконують або методом «першого, для якого справджується умова», або методом «оптимального».

Метод «оптимального» полягає в тому, що серед усіх блоків, що мають розмір, не менший від заданого, вибирають найменший.

Метод «першого, що підходить» полягає в тому, що виділенню підлягає перший у порядку перегляду елементів списку блок, розмір якого не менший від заданого.

Якщо блок, вибраний одним із методів, має розмір, що перебільшує вказаний при запиті, його розділяють на два: перший має заданий розмір і надається у відповідь на запит, а другий – залишається у списку вільної пам'яті. У разі звільнення пам'яті суміжні блоки об'єднують.

Приклад 2.4. Написати програму виділення блока вільної пам'яті заданого розміру (результат роботи програми має дорівнювати –1, якщо блок заданого розміру не може бути виділений) і процедуру для звіль-

нення – повторного включення до списку вільної пам'яті блока, що був виділений раніше.

Розв'язок задачі наведено у програмі 2.4.

Подана програма ілюструє одну із можливих реалізацій алгоритму розподілу пам'яті.

Після запуску програми на екрані монітора з'явиться меню 1:

Create block
Clear block
Exit
Your choice:

Ви маєте ввести цифри 1, 2 або 3, що відповідно означають «створити новий блок», «звільнити блок», «вийти з програми». Розглянемо кожен з цих режимів роботи.

Створити новий блок. При вході в цей режим перед вами з'явиться повідомлення:

Available memory = 1000000
What method? (1/2)

Перший рядок повідомляє, що всього доступно 1 000 000 одиниць пам'яті. (Програма працює з абстрактними величинами, бо Паскаль не дає прямого доступу до пам'яті.)

Другий рядок пропонує вам вибрати метод створення нового блока (1 – «перший, що задовольняє умові», 2 – «оптимальний»). Далі програма пропонує ввести розмір потрібного блока:

What size of block?

Якщо пам'ять виділено – програма надрукує список блоків пам'яті (наприклад, такий):

free memory
999889

---

busy memory
111

де **111**, введений вами розмір потрібного блока. Якщо ж блок не може бути створений, ви отримаєте повідомлення:

Memory is not given. Code –1

Після цього ви повертаєтесь у меню 1.

Звільнити блок. У цьому режимі ви можете звільнити блок, тобто надати йому статус вільної пам'яті. Програма пропонує вам ввести розмір потрібного блока:

What size of block?

Якщо потрібний блок існує, йому буде надано статус вільної пам'яті і, можливо, «злито» з наступним елементом, а також надруковано список блоків. В іншому разі буде видане повідомлення:

Block does not exist

і надруковано список блоків. Після цього програма повернеться в головне меню 1.

Вийти з програми. Вихід з програми.

```
program Ex2_4;
uses crt;                                   {Використані модулі}
const maxmem = 1000000;                     {Початковий розмір пам'яті}
type Tblock = ^block;                       {Опис типу даних}
   block = record
            busy: boolean;                  {Прапорець, що вказує на зайнятість}
                                            {або вільність блока}
            val: longint;                   {Розмір блока}
            next: Tblock;                   {Покажчик на наступний елемент списку}
           end;
var p,head,rab,vkaz,optima: Tblock; {Покажчики, що використовуються у програмі}
   s,msize: longint;                {Змінні, що означають розмір блока}
   choice,a: byte;                  {Змінні, що використовуються при виборі дії}
   is: integer;                     {Результат роботи процедури}
   exit: boolean;                   {Прапорець завершення роботи процедури}

procedure Insert1(msize: longint);  {Процедура виділення блока методом}
begin                               {«перший, що задовольняє умові»}
   exit := false; {Початкова ініціалізація прапорця закінчення роботи процедури}
   rab := head;  {Робочому покажчику присвоюється значення голови}
   p := head;
   while (rab <> nil) and (exit = false) do  {Поки робочий покажчик не вказує на nil}
      begin                                  {і прапорець виходу не дорівнює «істина»}
         if (rab^.val >= msize) and (rab^.busy = false)   {Якщо розмір блока >= }
            then                                          {заданого і блок вільний}
              begin
                rab^.val := rab^.val – msize;  {Зменшення розміру вільного блока}
                if rab^.next = nil {Перевіряється і запам'ятовується наступний блок}
                   then vkaz := nil
                   else vkaz := rab^.next;
                new(rab);                                 {Створюється новий блок}
                rab^.busy := true;      {Встановлюється прапорець зайнятості}
                p^.next := rab;         {Для попереднього елементу зберігається}
                                        {адреса його блока}
                rab^.val := msize;      {Встановлюється розмір блока}
                rab^.next := vkaz;      {Встановлюється покажчик на наступний}
```

```
            p := rab;                {Зберігання поточного елементу}
            vkaz := head;            {Виведення списку блоків на екран}
            while vkaz <> nil do
              begin
                if vkaz^.busy = true
                  then writeln('busy memory')
                  else writeln('free memory');
                writeln(vkaz^.val);
                writeln('————————');
                vkaz := vkaz^.next;
              end;
            repeat until keypressed;
            exit := true;            {Встановлення прапорця виходу в «істина»}
            is := 1;                 {Результат процедури = 1 (блок виділено)}
          end
        else rab := rab^.next;                {Перехід на наступний елемент}
    end;
  if is = -1 then writeln('Memory is not given. Code -1'); {Виведення результату}
                                                {процедури, якщо блок не виділено}
  repeat until keypressed;           {Утримання результатів на екрані}
end;

procedure Insert2(msize: longint);          {Процедура виділення блока методом}
begin                                       {«оптимального»}
  rab := head;              {Робочому покажчику присвоюється значення голови}
  p := head;
  optima := head;           {Оптимальному блоку присвоюється значення голови}
  if head^.val >= msize
    then s := head^.val
    else s := maxmem;  {Початкове встановлення розміру оптимального блока}
  while (rab <> nil) do       {Поки робочий покажчик не вказує на nil}
    begin
      if (rab^.val >= msize) and (rab^.busy = false)      {Якщо розмір блока >= }
        then                                              {заданого і блок вільний}
          begin
            if (rab^.val < s)  {Якщо розмір цього блока менший від попереднього}
              then optima := rab;                  {Оптимальний стає даним}
            is := 1;        {Елемент знайдено. Результат роботи процедури = 1}
          end;
      rab := rab^.next;     {Перехід на наступний елемент}
    end;
  if is = 1 then                                {Якщо оптимальний блок знайдено}
    begin
      optima^.val := optima^.val – msize;                {Зменшення розміру}
                                                         {оптимального блока}
```



```
      if optima^.next = nil    {Перевірка і запам'ятовування наступного блока}
        then vkaz := nil
        else vkaz := optima^.next;
      new(rab);                          {Створення нового блока}
      rab^.busy := true;                 {Встановлення прапорця зайнятості}
      optima^.next := rab;               {Для попереднього елементу зберігається}
                                         {адреса цього блока}
      rab^.val := msize;                 {Встановлення розміру блока}
      rab^.next := vkaz;                 {Встановлення покажчика на наступний}
      vkaz := head;                      {Виведення списку блоків на екран}
      while vkaz <> nil do
        begin
          if vkaz^.busy = true
            then writeln('busy memory')
            else writeln('free memory');
          writeln(vkaz^.val);
          writeln('————————');
          vkaz := vkaz^.next;
        end;
      repeat until keypressed;
    end;
  if is = -1 then writeln('Memory is not given. Code -1'); {Виведення результату}
                                                {процедури, якщо блок не виділено}
  repeat until keypressed;                {Утримування результатів на екрані}
end;

procedure Delete (msize: longint);              {Процедура звільнення блока}
begin
  rab := head;              {Робочому покажчику присвоюється значення голови}
  exit := false; {Початкова ініціалізація прапорця закінчення роботи процедури}
  p := head;
  while (rab <> nil) and (exit = false) do {Поки робочий покажчик не вказує на nil}
    begin                                  {і прапорець виходу не дорівнює «істина»}
  if (rab^.val = msize) and (rab^.busy = true)   {Якщо розмір блока = заданому}
      then                                       {і блок вільний}
        if (rab^.next^.busy = true) or (rab^.next = nil)  {Якщо наступний блок}
          then                                            {зайнято, або він nil}
            begin
              rab^.busy := false;     {Прапорець зайнятості = «хибність»}
              is := 1;                {Результат процедури = 1 (блок звільнено)}
              exit := true;           {Встановлення прапорця виходу в «істина»}
            end
          else
            begin
              p^.next := rab^.next;   {Перенесення покажчика з поточного}
                                      {блока на наступний}
```

```
rab^.next^.val := rab^.next^.val + rab^.val; {Збільшення розміру}
                                    {наступного блока на розмір поточного}
                  dispose(rab);     {Знищення поточного блока}
                  exit := true;     {Встановлення прапорця виходу в «істина»}
                  is := 1;          {Результат процедури = 1 (блок звільнено)}
                end;
      p := rab;
      rab := rab^.next;             {Встановлення покажчика на наступний}
    end;
  if is = –1 then writeln('Block does not exist'); {Виведення результату процедури,}
                                                   {якщо блок не звільнено}
  vkaz := head;                     {Виведення списку блоків на екран}
  while vkaz <> nil do
    begin
      if vkaz^.busy = true
        then writeln('busy memory')
        else writeln('free memory');
      writeln(vkaz^.val);
      writeln('————————');
      vkaz := vkaz^.next;
    end;
  repeat until keypressed;          {Утримування результатів на екрані}
end;

BEGIN                               {Основна програма}
  clrscr;                           {Очистка екрана}
  new(head);                        {Створення головного елементу}
  head^.val := maxmem; {Значенню головного елементу присвоюється вся пам'ять}
  head^.next := nil;                {Покажчик на наступний nil}
  head^.busy := false;              {Прапорець зайнятості = «істина»}
  repeat
    repeat
      clrscr;
      writeln('1. Create block');   {Список режимів роботи}
      writeln('2. Clear block');
      writeln('3. Exit');
      writeln;
      write('Your choice: ');       {Вибір режимів роботи}
      read(choice);
      until choice <= 3;            {Перевірка коректності режиму}
      case choice of
       1: begin                     {Режим роботи – створення нового блока}
            repeat
              clrscr;
              writeln('Available memory = ',maxmem); {Виведення доступної пам'яті}
              is := –1;  {Початкове встановлення результату роботи процедури}
```



```
            write('What method ? (1/2) ');   {Вибір методу створення блока}
            read(a);
            until a <= 2;
            write('What size of block ? ');
            read (msize);                    {Введення розміру потрібного блока}
            case a of                        {Виклик процедури створення блока}
              1: Insert1(msize); {Методом «першого, що задовольняє вимогам»}
              2: Insert2(msize); {Методом «оптимального»}
            end;
          end;
       2: begin                  {Режим роботи – звільнення блока}
            is := –1;   {Початкове встановлення результату роботи процедури}
            clrscr;
            writeln('Available memory = ',maxmem);  {Виведення доступної пам'яті}
            write('What size of block: ');
            read(msize);                     {Введення розміру потрібного блока}
            Delete(msize);          {Виклик процедури звільнення потрібного блока}
          end;
      end
    until choice = 3;               {Перевірка на закінчення роботи програми}
  repeat until keypressed;          {Утримування результатів на екрані}
END.
```

Програма 2.4. Алгоритм динамічного розподілу пам'яті

Якщо ми проводимо видалення та вставку елементів у список у спеціальний спосіб, то отримуємо загальноприйняті у програмуванні структури даних типу стек, черга тощо.

2.3. Стеки та черги

Стеком називають спеціально організований список, в якому занесення і видалення елементів можна проводити тільки через один початковий елемент, який називають вершиною стека [9, 20, 86, 91, 127]. Вершину стека іменуватимемо змінною **top**. Стек працює за принципом «останній зайшов – перший вийшов» (Last In First Out). Тому стек ще називають списком типу LIFO.

Чергою називають список, вставка елементу в який проходить в кінець списку (позначимо змінною **rear**), а всі видалення проходять з голови (початку) списку (позначимо змінною **front**). Ці операції проходять за принципом «перший зайшов – перший вийшов» (First In First Out), тому чергу ще називають списком типу FIFO.

Одним з найпоширеніших методів задання стека є використання одномірного масиву, наприклад, ***var*** *Stach: lisp*, тоді *m* (довжина масиву **lisp**) визначатиме максимальну розмірність стека.

Операції вставки та видалення в цьому разі можна реалізувати у такий спосіб:

```
procedure add(var St: lisp; item: integer; n, top: 1..m); {Вставити елемент item в стек}
begin                                                     {St максимального розміру n}
   if top > n then Stackfull
     else
        begin
           top := top + 1;
           St[top] := item
        end;
end;

procedure del(var St: Lisp; item: integer);               {Елемент з голови стека}
begin                                                     {засилається в item}
   if top < 1 then Stackempty
     else
        begin
           item := St[top];
           top := top – 1
        end;
end;
```

Кожне виконання процедур *add* і *del* потребує константного часу і не залежить від кількості елементів в стеку. Процедури *Stachfull* і *Stachempty* є типовими процедурами обробки аварійних ситуацій за пам'яттю, характер роботи яких визначає користувач.

У разі зв'язного задання стека за допомогою динамічних змінних процедури вставки і видалення набудуть вигляду:

```
procedure addd(var St: linln; item: integer);
  var t: linln;
begin
    new(t);
    t^.val := item;
    t^.next := St;
    St := t
end;
```

У такому разі спеціальний покажчик *St* використовують для адресації вершинного елементу стека. Пустий стек визначатиметься у такому разі дією St := **nil**.

Наприклад, якщо в стек *St* послідовно поступатимуть елементи *a*, *b*, *c*, тоді рис. 2.3 ілюструватиме послідовне наповнення стека.

```
procedure deld(var St: linln; item: integer);
begin
   if St = nil then Stackempty
     else
        begin
           item := St^.val;
           St := St^.next
        end;
end;
```

Для ефективного задання черги у разі використання структури даних типу масив бажано розглядати циклічний масив **Q**[0..n–1]. Елементи включаються в чергу за допомогою збільшення змінної **rear** – *кінець черги* до наступної вільної позиції. Коли **rear** = n–1, наступний елемент вводиться в **Q**[0] у разі, коли комірка вільна. *Початок черги* **front** завжди вказує на одну позицію проти годинникової стрілки від першого елементу черги. *Початок черги = кінцю черги* лише тоді і тільки тоді, коли черга порожня. На початку покладають: **front** = **rear** = 0.

Для того щоб вставити елемент, слід змістити *кінець черги* на одну позицію за годинниковою стрілкою. Це можна зробити у такий спосіб:

```
if rear = n – 1 then rear := 0
   else rear := rear + 1
```

або простіше, використавши ділення за модулем. До того як робити включення, потрібно збільшити покажчик кінця черги за допомогою виразу rear := (rear + 1) **mod** n. У такий спосіб потрібно зсувати *початок черги* на одну позицію за годинниковою стрілкою кожного разу, коли виконуються вилучення. Зрозуміло, що обробляючи масив циклічно, включення та вилучення для черг можна виконати за фіксований проміжок часу. Цікавим у цьому разі є й факт ідентичності перевірки повноти черги в процедурі включення нового елементу та перевірки на пустоту черги в процедурі видалення елементу з черги.

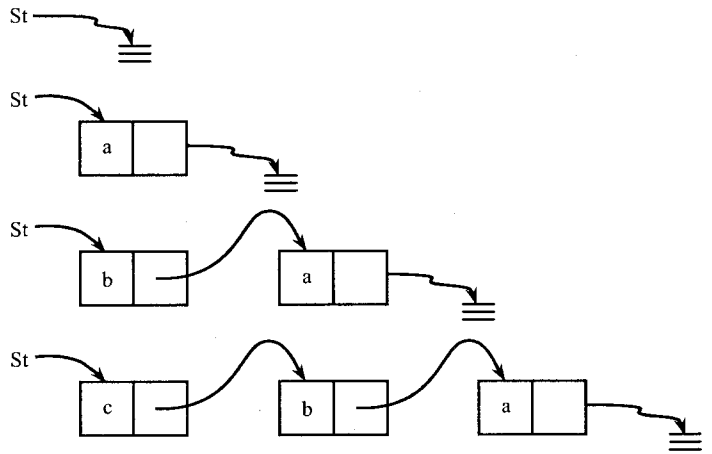


Рис. 2.3. Послідовне наповнення стека

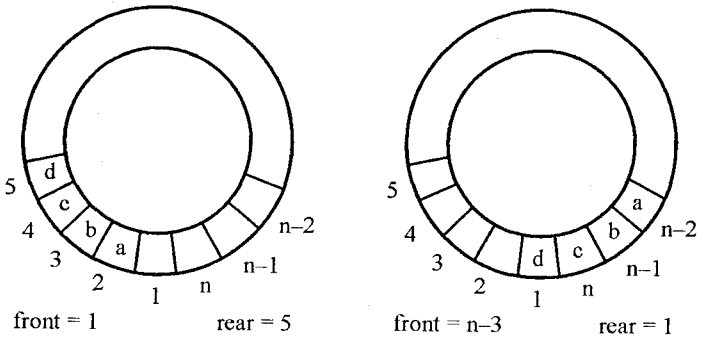


Рис. 2.4. Циклічна черга

Рис. 2.4 ілюструє можливе розміщення чотирьох елементів *a*, *b*, *c*, *d* у черзі *Q* завдовжки $n > 0$.

У цьому разі операції вставки і видалення елементу можна реалізувати за допомогою процедур *addq* і *delq*.

```
procedure addq(var Q: lisp; var n: 1..m; item: integer; var front, war: 1...m);
begin
    war := (war + 1) mod n;
    if front = war then Quenempty
      else Q[war] := item
end;

procedure delq(var Q: lisp; var n: 1..m; var item: integer; var front, war: 1..m);
begin
    if front = war then Quenempty
      else
        begin
          front := (front + 1) mod n
          item := Q[front];
        end;
end;
```

Подумайте, чому бажано було б мати ще додаткову змінну, наприклад ***tag****, яка б значенням* ***tag = 0*** *виявляла ситуацію, коли черга пуста.*

У разі задання черги за допомогою динамічних змінних техніка реалізації операцій вставки та видалення аналогічна до реалізації цих операцій над стеком.

Під час вирішення конкретних задач можуть виникати й інші різновиди зв'язного зберігання [86, 91, 127].

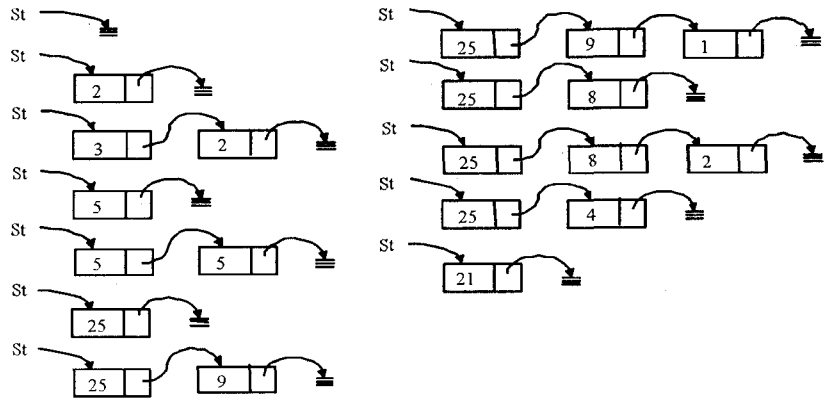


Рис. 2.5. Динаміка зміни стека при обчисленні виразу
2, 3, +, 5, *, 9, 1, –, 2, /, –

Наведемо приклад застосування описаних структур даних. Розглянемо задачу обчислення арифметичних виразів. Однією з форм задання виразів є бездужковий польський інверсний запис (БПІЗ), в якому порядок виконання операції визначається її контекстом та позицією у виразі. Наприклад, якщо елементи виразу розділити комою, тоді вираз (2 + 3) * 5 – (9–1) / 2 в польському інверсному записі набуде вигляду 2, 3, +, 5, *, 9, 1, –, 2, /, –. Привабливість останнього запису полягає в тому, що значення виразу можна обчислити за один його перегляд зліва направо, використовуючи стек.

Технологія використання стека така. Спочатку стек пустий. Якщо під час перегляду виразу трапляється операнд (дане), тоді його заносять у стек, а якщо з'являється символ операції, тоді цю операцію виконують над верхніми елементами стека із заміною їх результатом обчислень. Кількість елементів стека, які беруть участь у цьому обчисленні, визначається парністю операції. Динаміку зміни стека для нашого прикладу ілюструє рис. 2.5.

Продемонструємо програму реалізації роботи алгоритму в двох варіантах: стек описується за допомогою масива і динамічними змінними. У першому випадку при кожному занесенні даних у стек слід перевіряти можливість поповнення стека, в другому випадку ця проблема відпадає. Для простоти реалізації як дані використовуватимемо невід'ємні цілі числа, а як операції – операції додавання (+), віднімання (–), множення (*), ділення (/). Послідовність символів БПІЗ вводять з клавіатури. Кінець введення визначає кінець файла стандартного вводу (вмонтована функція **eof**). Відповідний алгоритм реалізується програмою 2.5.

```
program Ex2_5;
 const n = 1000;                    {Визначає максимальну глибину стека}
```

```
label exit, exitw;                    {Мітки виходу з програми і виходу з циклу}
type  int = 0..n;                     {Тип інтервалу}
      ar = array [int] of integer;    {Тип масиву}
var   c: char;   {Змінна для введення чергового символу зі стандартного файла}
      i: integer;                     {Індекс масиву}
      st: ar                          {Опис стека}
begin
   readln;
   i := 0;                                  {Характеризує пустий стек}
   while not (eof) do                       {Поки не кінець файла}
      begin
         read(c);
         if (i < 2) or (c = ',') then
            begin
               writeln('Помилка в БПІЗ');
               goto exit;
            end
            else
               case c of                                  {Реалізація операції}
                  '+': begin St[i – 1] := St[i] + St[i – 1]; i := i – 1 end;
                  '–': begin St[i – 1] := St[i] – St[i – 1]; i := i – 1 end;
                  '*': begin St[i – 1] := St[i] * St[i – 1]; i := i – 1 end;
                  '/': begin St[i – 1] := St[i] / St[i – 1]; i := i – 1 end;
               end
                  else
                     begin
                        i := i + 1;
                        if i > n then
                           begin
                              writeln('Стек переповнений. Змініть
                                                  розмірність стека, константу n');
                              goto exit;
                           end;
                        St[i] := 0;
                        while c <> ' ' do
                           begin                          {Формування цілого}
                              St[i] := St[i] * 10 + (ord(c) – ord('0'))
                              if not (eof) then
                                 if eoln then readln(c)
                                    else read(c)
                                 else goto exitw;
                           end;
                     end;
         readln;
      end;
   exitw: if i = 1 then writeln('Результат обчислення виразу = ', St[1])
          else writeln ('Помилка у виразі');
   exit:
end.
```


Програма 2.5. Реалізація БПІЗ, в якій стек описується за допомогою масиву

У разі, коли маємо сутужність з пам'яттю, можемо використати програму 2.6 обчислення БПІЗ, в якій стек реалізується за допомогою динамічних змінних.

```
program Ex2_6;
 label exitp, exitw;
 type  stnode = ^nod;
       nod = record                          {Опис структури вузла стека}
                info: integer;               {Поле даних}
                lin: stnode                  {Поле зв'язку}
             end;
 var   c: char;
       in: integer;
       st: stnode;
procedure Misst;
   begin
      writeln('Помилка у БПІЗ');
      goto exitp;
   end;
begin
   st := nil;
   readln;
   while not (eof) do
     begin
       while not (eoln) do
         begin
           read (c);
           if c = ' ' then Misst
             else
               case c of
                  '+': begin in := st^.info; st := st^.link; st^.info := st^.info + in; end;
                  '–': begin in := st^.info; st := st^.link; st^.info := st^.info + in; end;
                  '*': begin in := st^.info; st := st^.link; st^.info := st^.info + in; end;
                  '/': begin in := st^.info; st := st^.link; st^.info := st^.info + in; end;
               end
                 else
                   begin
                      in := 0;
                      while c <> ',' do
                        begin
                          in := in * 10 + (ord('c') – ord('0'));
```

```
                    if not (eof) then
                        if eoln then readln(c)
                            else read(c)
                        else goto exitw;
                    end;
                new (q);
                q^.info := in;
                q^.linln := st;
                st := q;
            end;
        readln;
    end;
exitw:
if St^.linln = nil then
    writeln('Результат обчислення виразу = ', St^.info)
    else writeln('Помилка у виразі');
exitp:
end.
```

Програма 2.6. Реалізація БПІЗ із динамічними змінними

2.4. Графи та дерева

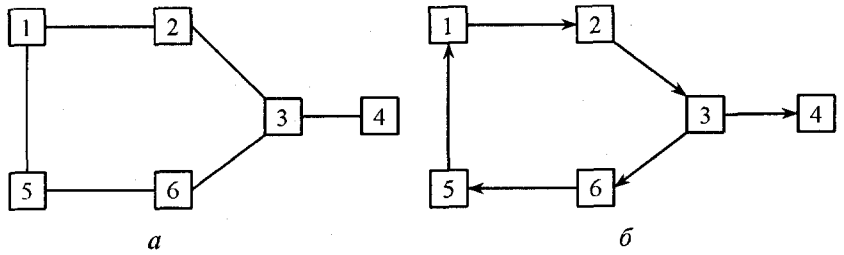
2.4.1. *Основні визначення*

Поняття «граф» вперше ввів до розгляду знаменитий математик Ейлер у 1736 р. Граф G містить дві множини $G = (V, E)$, де V – множина вершин, E – множина ребер. V є скінченною непустою множиною вершин (іноді їх називають вузлами), традиційно пронумерованою 1, 2, ..., n. E – скінченна множина пар вершин, E: $V \times V$. Коли пари впорядковані, тоді графи називають орієнтованими, в іншому разі – неорієнтованими. Впорядковане ребро позначатимемо $\langle i, j\rangle$, а невпорядковане – (i, j).

У більшості застосувань з множиною ребер графа пов'язують множину дійсних чисел, які називають вагами. Такий граф називають зваженим.

У неорієнтованому графі твердимо, що вершина i суміжна вершині j, коли існує ребро (i, j). Степенем вершини назвемо число суміжних їй вершин. В орієнтованому графі виділяють напівстепінь входу (число вхідних) та напівстепінь виходу (число вихідних ребер з вершини).

Шляхом з вершини v_p до вершини v_q є послідовність вершин v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, така що (v_p,v_{i1}), (v_{i1},v_{i2}), ..., (v_{in},v_q) є ребрами в $E(G)$. Довжиною шляху називають число ребер, які належать шляху. Простим шляхом називають шлях, в якому всі вершини, за винятком першої та останньої, є різними. Циклом називають простий шлях, в якому перша і остання



а *б*

Рис. 2.6. Неорієнтований (*а*) і орієнтований (*б*) графи

вершини збігаються. На рис. 2.6 маємо орієнтований і неорієнтований графи з 6 вершинами і 6 ребрами.

Неорієнтований граф називають зв'язним, коли для довільної пари вершин існує шлях між ними.

Є два основних підходи до задання графів: послідовний і зв'язний.

Послідовна форма використовує квадратну таблицю, яку називають матрицею суміжності. Для неорієнтованого графа матрицю суміжності Graf(1: n, 1: n) визначають так: Graf$(i, j) = 1$, якщо $(i, j) \in E$ і Graf$(i, j) = 0$ в іншому випадку. У разі навантаженого графа Graf(i, j) = *вартість ребра* (i, j), якщо ребро існує, та Graf$(i, j) = +\infty$ за відсутності ребра (i, j) в $G = (V,E)$. Аналогічно задають і орієнтований граф, але з урахуванням орієнтації ребер.

Рис. 2.7 ілюструє задання матрицями суміжності неорієнтованого і орієнтованого графів з рис. 2.6.

За такого задання нам потрібно зробити $O(n^2)$ операцій для заповнення таблиці. Тому всі алгоритми, які використовують послідовну форму задання графа, не можуть мати часових оцінок, ліпших за $O(n^2)$.

Введемо позначення: якщо множина V складається з n вершин, тоді $|V| = n$, аналогічно $|E| = m$. У теорії графів для задання графа використовують ще один спосіб задання графа – *матрицю інцидентності*. Вона має розмірність $n \times m$, де $|V| = n$, $|E| = m$. Для орієнтованого графа стовпець, що відповідає дузі $\langle u,v\rangle \in E$, містить 1 в рядку, що відповідає вершині u та 0

	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	0	0
3	0	1	0	1	0	1
4	0	0	1	0	0	0
5	1	0	0	0	0	1
6	0	0	1	0	1	0

а

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	1	0	0	0
3	0	0	0	1	0	1
4	0	0	0	0	0	0
5	1	0	0	0	0	0
6	0	0	0	0	1	0

б

Рис. 2.7. Матриці суміжності неорієнтованого (*а*) і орієнтованого (*б*) графів

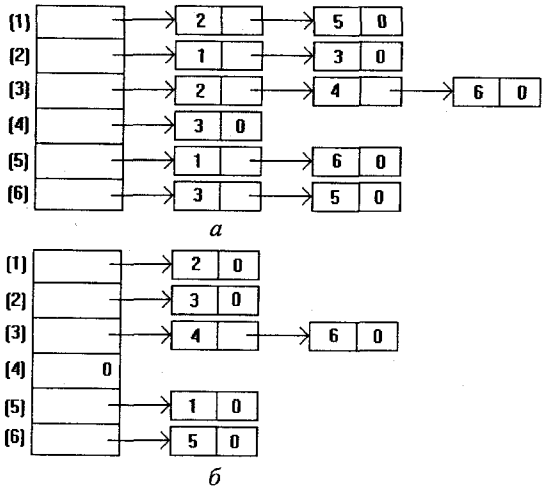


Рис. 2.8. Списки суміжності неорієнтованого (*а*) і орієнтованого (*б*) графів

в рядку v, і нулі в інших рядках. У разі неорієнтованого графа стовпець, що відповідає ребру (u,v), містить 1 в рядках, що відповідають u та v, і нулі в інших рядках. З погляду програмування такий спосіб задання графа потребує $n \times m$ комірок пам'яті, він незручний для організації доступу до інформації. Наприклад, для побудови відповіді на питання типу «*існує дуга* (u,v)?» у найгіршому випадку слід переглянути всі m стовпців.

Тому існує інша форма задання графа, яку називають списком зв'язності. Граф G з n вершинами задаватиметься n списками – по одному для кожної вершини **i**. Список для вершини **i** містить вершини, суміжні з **i**. Рис. 2.8 ілюструє списки суміжності для графів з рис. 2.6.

Списки суміжності можуть бути задані масивом VERTEX(1: p), де $p = e$, якщо граф орієнтований, і $p = 2e$ в неорієнтованому графі, а $e = |E|$. Head(i), $1 \le i \le n$ дає початкові вершини околу суміжності для вершини **i**. Якщо ми визначимо Head($n + 1$) = $p + 1$, тоді вершини списку суміжності для вершини **i** запам'ятовуються у VERTEX(j), де Head(i) $\le j <$ Head($i + 1$). Якщо список суміжності для вершини i пустий, тоді Head(i) = Head($i + 1$). Рис. 2.9 демонструє послідовне задання списків суміжностей із рис. 2.5.

Відзначимо, що в класичному списку суміжності для неорієнтованих графів кожне ребро (u,v) задається подвійно: через вершину u в списку **початок**[u] і через вершину v в списку **початок**[v]. Для багатьох алгоритмів на графах характерною є динамічна модифікація ребер, видалення, додавання ребер. Тому в цих випадках у вузлах списків суміжності слід мати додаткові поля, що забезпечуватимуть ефективність реалізації наведених операцій. Найчастіше вживаними є такі поля: покажчик на попередній елемент, вузол **початок**[v], що має серед своїх елементів вершину u, має покажчик на вузол **початок**[u].

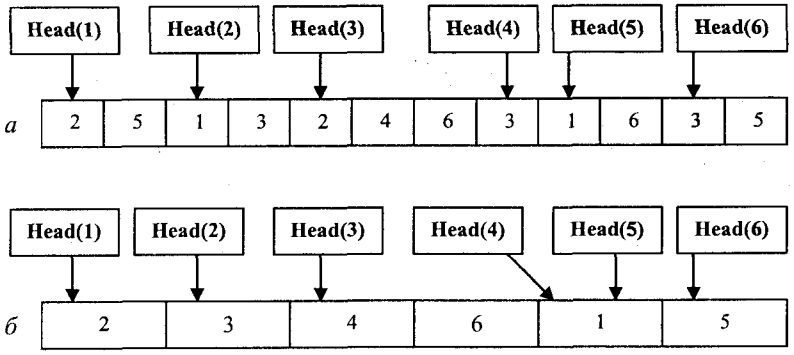


Рис. 2.9. Послідовне задання списків суміжностей із рис. 2.6

Одним із найуживаніших спеціальних типів графів є дерева. Орієнтований граф без циклів називають орієнтованим ациклічним графом.

Дерево – це орієнтований ациклічний граф, для якого виконуються такі умови:

1) у графі знайдеться одна вершина, в яку не входить жодне ребро; цю вершину називають коренем дерева;

2) у будь-яку вершину, крім кореня, входить тільки одне ребро;

3) із кореня можна знайти унікальний шлях до кожної вершини дерева.

Орієнтований граф, що складається з декількох дерев, називають *лісом*.

Нехай $F = (V, E)$ – граф, що є лісом. Якщо $\langle v, u \rangle \in E$, тоді v називають *батьком* вузла u, а u – *сином* вузла v. Якщо є шлях із v в u, тоді v називають *предком* вузла u, а u – *нащадком* вузла v. Вузол v та його нащадки разом утворюють піддерево лісу F, і вузол v називають коренем цього піддерева.

Глибина вузла v у дереві – це шлях із кореня у v. *Висота вузла v у дереві* – це довжина найдовшого шляху із v у деякий вузол. *Висотою дерева* називають висоту його кореня. *Рівень вузла v* у дереві дорівнює різниці висоти дерева і глибини вузла v.

Впорядкованим деревом називають дерево, в якому множина синів кожного вузла впорядкована. У разі зображення впорядкованих дерев вважають, що множина синів кожного вузла впорядкована зліва направо.

Двійковим (бінарним) деревом називають таке впорядковане дерево, для якого виконуються умови:

1) кожен син довільного вузла ідентифікується або як *лівий син*, або як *правий син*;

2) кожен вузол має *не більше одного* лівого сина і *не більше одного* правого сина.

Піддерево T_l, коренем якого є лівий син вузла v (якщо таке існує), називають *лівим піддеревом вузла v*. Аналогічно визначають *праве піддерево вузла* T_r. Відзначимо, що всі вузли T_l розташовані лівіше всіх вузлів T_r.

Максимальне число вузлів на рівні i бінарного дерева дорівнює $2^i - 1$. Також максимальне число вузлів у бінарному дереві завглибшки k дорівнює $2^k - 1$, де $k > 0$.

2.4.2. *Обходи дерев*

Для багатьох задач, пов'язаних з використанням дерев, виникає потреба переглянути всі вузли дерева в певному порядку. Розглянемо основні принципи проходження дерев на прикладі бінарних дерев. Алгоритми обходу мають рекурсивний характер:

I. *Пряме проходження:*

1) відвідати корінь;

2) відвідати, згідно з прямим проходженням, ліве піддерево кореня;

3) відвідати, згідно з прямим проходженням, праве піддерево кореня.

II. *Обернене проходження:*

1) відвідати, згідно з оберненим проходженням, ліве піддерево кореня;

2) відвідати корінь;

3) відвідати, згідно з оберненим проходженням, праве піддерево кореня.

III. *Кінцеве проходження:*

1) відвідати, згідно з кінцевим проходженням, праве піддерево кореня;

2) відвідати, згідно з кінцевим проходженням, ліве піддерево кореня;

3) відвідати корінь.

Візьмемо до розгляду дерево, зображене на рис. 2.10.

Тоді перегляд вузлів у прямому напрямі матиме таку послідовність: 1, 2, 4, 5, 7, 8, 10, 3, 6, 9, 11, 12; в оберненому проходженні – 4, 2, 7, 5, 10, 8, 1, 3, 11, 9, 12, 6 і в кінцевому порядку – 12, 11, 9, 6, 3, 10, 8, 7, 5, 4, 2, 1.

Структура даних дерева найчастіше використовується для реалізації різних довідників. Тому над деревами доводиться виконувати такі операції, як «*знайти вузол із певною властивістю*», «*визначити батька або синів заданого вузла*», «*вилучити вказаний вузол або частину дерева*», «*додати новий вузол*» тощо.

2.4.3. *Задання дерев у пам'яті*

Для дерев прийняте послідовне і зв'язане зберігання [86, 91, 127]. Послідовне створюється на основі одного з лінійних зображень дерева – у вигляді «рядка», зокрема, воно може бути рівневим або дужковим.

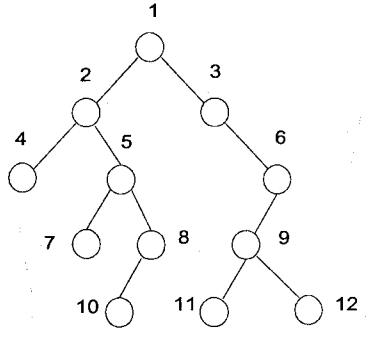


Рис. 2.10. Бінарне дерево

Рівневе задання. З кожним вузлом дерева v зв'язується номер рівня цього дерева k – ціле додатне число. Потім записують всі його вузли з номерами рівнів відповідно до якогось порядку проходження дерева. Так, рівневим заданням дерева, зображеного на рис. 2.10, згідно з прямим порядком, може бути 1, 1, 2, 2, 4, 3, 5, 3, 7, 4, 8, 4, 10, 5, 3, 2, 6, 3, 9, 4, 11, 5, 12, 5.

Дужкове задання. Дужковим зображенням ($g3$) дерева B з одного вузла є запис цього вузла. Якщо B складається з кореня W і піддерев B_1, B_2, ..., B_m, тоді дужкове зображення можна записати у вигляді:

$$W(g_3(B_1), g_3(B_2), ..., g_3(B_m)).$$

Дерево з рис. 2.9 в дужковому зображенні матиме вигляд:

$$1\ (2\ (4,\ 5\ (7,\ 8\ (10)),\ 3\ (6\ (9\ (11,\ 12))))).$$

Форми зв'язного зберігання ґрунтуються на використанні динамічних змінних. Вони залежать від опису відношень між «*предками*» та «*нащадками*» (батьками та синами).

Найбільш використовуваними є стандартна, обернена і розширена стандартна форми зв'язного зберігання. У стандартній формі фіксуються зв'язки від батька до всіх синів. Отже, якщо дерево має степінь n, тоді в кожному вузлі ми повинні мати n покажчиків на синів вузла. Якщо вузол має менше n синів, тоді непотрібні покажчики зв'язку онуляють за допомогою вмонтованої функції **nil**.

В оберненій формі зв'язного зберігання дерев вказуються зв'язки від синів до батьків. Розширена стандартна форма об'єднує стандартну та обернену форму зв'язного зберігання дерев.

Для дерева з рис. 2.10 зв'язне зберігання в стандартній формі наведено на рис. 2.11.

В описаних формах задання дерева один вузол задання відповідає одному вузлу дерева, і кількість зв'язків вузла залежить від степеня дерева. Коли дерево неоднорідне і має багато вузлів, степінь яких значно менший від степеня дерева, тоді обробка незадіяних зв'язків викликає зайвий клопіт. Тому розглянемо задання вузла з фіксованим розміром. Вузол матиме три поля TAG, DATA і LINK. Коли TAG = 1, DATA

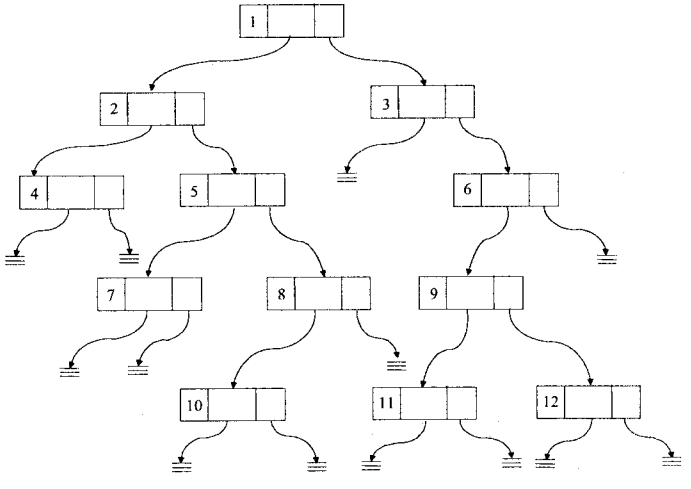


Рис. 2.11. Зберігання дерева в стандартній формі

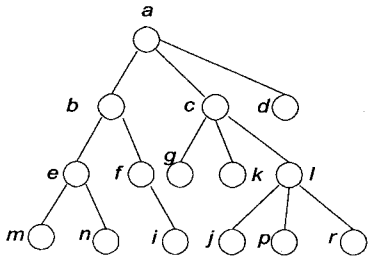


Рис. 2.12. Дерево

містить покажчик на список, елементи якого відповідають синам відповідного вузла; в іншому разі DATA містить елемент-листок. Поле LINK використовується як поле зв'язку списку піддерев кожного елементу дерева. Так дерево, зображене на рис. 2.12, відображатиметься у цьому разі списковою структурою із рис. 2.13.

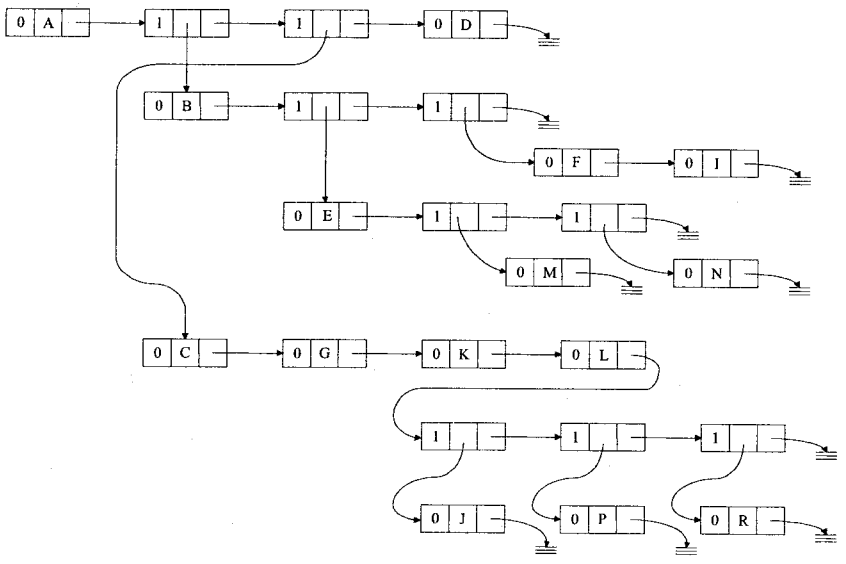


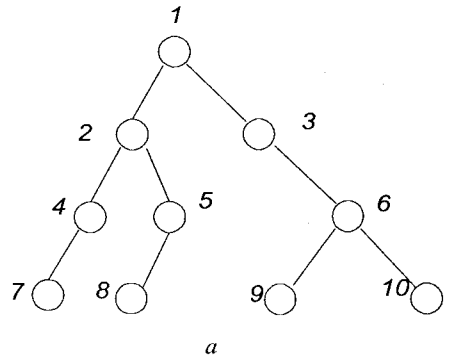
Рис. 2.13. Списокова структура задання дерева з рис. 2.12 у формі вузла фіксованого розміру

Задання повного бінарного дерева

Бінарне дерево називають повним, якщо для деякого цілого числа k кожен вузол завглибшки менше k має як лівого, так і правого сина і кожен вузол завглибшки k є листом. Повне бінарне дерево завглибшки k має 2^{k-1} вузлів.

У цьому разі повне бінарне дерево з n вузлами можна зручно задати послідовною нумерацією вершин, починаючи з кореня, по рівням зліва направо за допомогою масиву Tree(i). Тоді для будь-якого вузла з індексом i, $1 \le i \le n$, справджуватиметься:

1) батько вузла i, Parent(i), визначатиметься $\lfloor i/2 \rfloor$, якщо $i \ne 1$; коли $i = 1$, тоді цей вузол вже є коренем, і тому батька він не має;



а

	Lson	Rson
1	2	3
2	4	5
3	0	6
4	7	0
5	8	0
6	9	10
7	0	0
8	0	0
9	0	0
10	0	0

б

Рис. 2.14. Задання неповного бінарного дерева

2) лівий син вузла i, Lson(i), визначатиметься формулою Lson(i) = 2 * i, коли $2 * i \le n$; коли ж $2 * i > n$, тоді вузол i не має лівого сина;
3) правий син вузла i, Rson(i), визначатиметься формулою Rson(i) = 2 * i + + 1, коли $2 * i + 1 \le n$; коли ж $2 * i + 1 > n$, тоді вузол i не має правого сина.

Таке задання можливе для довільних бінарних дерев, але буде ефективним за пам'яттю тільки для повних бінарних дерев. Чому?

Наприклад, візьмемо бінарне дерево, зображене на рис. 2.14, *а*. Тоді на рис. 2.14, *б* відображене його задання за допомогою масивів Lson і Rson.

2.4.4. *Програмна реалізація задання дерев у пам'яті*

Розглянемо застосування запропонованих форм задання дерев на конкретних прикладах. Спочатку будемо аналізувати бінарні дерева. Для роботи з ними вони мають бути заданими в пам'яті, програмно або вручну за використання послідовного або зв'язного зберігання. Послідовне задається у вигляді «рядка» (рівневе або дужкове задання), зв'язне – за допомогою спискових структур.

В описаних формах задання дерева один вузол задання відповідає одному вузлу дерева, і кількість зв'язків вузла залежить від степеня дерева.

Для зображення дерева у таких заданнях можна використати програму 2.7.

```
program Ex2_7;
uses crt;
var f: Text;                {Змінна, за допомогою якої викликається текстовий файл}
const lvl: char = '1';      {Початковий рівень вузла}
const s: string = '';       {Змінна, в яку записується дерево в рівневому заданні}
const s1: string = '';      {Змінна, в яку записується дерево в дужковому заданні}
```

```
type  Pnode = ^Tnode;
      Tnode = record                {Структура вузла дерева}
                Value: char;        {Значення вузла}
                Ls, Rs: Pnode;      {Значення лівого та правого синів}
              end;

{Процедура InputTree вводить із зовнішнього файла дерево згідно з прямим обходом
і зберігає його у вигляді зв'язного списку}
procedure InputTree(var T: Pnode);
begin
    getmem(T, sizeof(Tnode));          {Резервування пам'яті розміром Tnode}
    T^.Ls := nil;
    T^.Rs := nil;
    Readln(f, T^.Value);
    if T^.Value = '0' then exit; {Якщо в батька синів немає, то виходимо з процедури}
    InputTree(T^.Ls); {Рекурсивно викликаємо процедуру для задання нащадків вузла}
    InputTree(T^.Rs);
end;

{Для рівневого задання дерева використовуватимемо процедуру LevelTree, яка
пов'язує кожен вузол дерева з його рівнем та задається в пам'яті рядком, в якому
послідовно задані через кому вузли та їх рівні}
procedure LevelTree(T: Pnode);
begin
    if T = nil then exit;              {Якщо дерево пусте, то виходимо з процедури}
    if T^.Value <> '0' then
     begin
       s := s + T^.Value + ',' + lvl + ',';     {Задає значення та рівень вузла у рядку}
       Inc(lvl);                                {і переходить на наступний рівень}
       LevelTree(T^.Ls);          {Рекурсивно викликається процедура для перевірки}
       LevelTree(T^.Rs);          {синів батька, і виходять на попередній рівень}
       Dec(lvl);
     end;
end;

{Процедуру BrackTree використаємо для дужкового задання дерева, яка перед
кожним новим рівнем ставить дужку і закриває її при виході на попередній
рівень}
procedure BrackTree(T: Pnode);
var written: Boolean;          {Змінна, яка набуває значення true, якщо дужка була}
begin                          {відкрита, і значення false в іншому разі}
    written := False;
    if T^.Value = '0' then exit;
    s1 := s1 + T^.Value;
    if ((T^.Ls^.Value <> '0') or (T^.Rs^.Value <> '0')) then
```



```
      begin                    {Якщо існує хоча б один син, то відкриваємо дужку}
        written := True;
        s1 := s1 + '(';
      end;
    if T^.Ls <> nil then brack (T^.Ls);              {Обробляємо лівого сина}
    if T^.Rs <> nil then
      begin
        if T^.Rs^.Value <> '0' then s1 := s1 + ','; {Якщо існує правий син, то між}
        brack(T^.Rs);                                {синами ставимо кому}
      end;
    if written then s1 := s1 + ')';
end;

{Початок основної програми}
var root: Pnode;
begin
    assign (f, 'a:\practice\tree.tr');
    Reset (f);

{Текстовий файл програми представлений у такому вигляді:
2 – корінь дерева
8 – лівий син кореня (2)
4 – лівий син 8
0 – лівий син 4 (відсутній, отже, переходимо на правого сина)
0 – правий син 4 (відсутній, отже, 4 є листом)
3 – правий син 8
5 – лівий син 3
0 – лівий син 5
0 – правий син 5 (відсутній, отже, 5 є листом)
9 – далі аналогічно…
0
0
7
1
0
0
0 …кінець обходу та повернення на корінь}

    InputTree(Root);
    LevelTree(Root);
    delete(s,length (s),1);
    writeln(s);
    BrackTree(Root);
    writeln(s1);
    readkey
end.
```

Програма 2.7. Задання дерев у пам'яті

На закінчення програма задає дерево в пам'яті у такому вигляді:

2, 1, 8, 2, 4, 3, 3, 3, 5, 4, 9, 4, 7, 2, 1, 3 – рівневе задання;
2 (8 (4, 3 (5, 9)), 7 (1)) – дужкове задання.

2.4.5. *Програми оберненого обходу бінарного дерева*

Повернемося до задачі проходження дерев. Запропоновані вище алгоритми розв'язання цієї задачі можна реалізувати або рекурсивно, або ж послідовно за допомогою стека, залежно від форми задання дерева в пам'яті. Зупинимось на послідовному зберіганні дерев, використовуючи нумерацію вершин, починаючи з кореня, по рівням зліва направо (Lson[I] – визначає лівого сина вузла *i*, Rson[I] – правого сина вузла *i*), як наводилось раніше на рис. 2.14, тільки вузли дерева позначаються не обов'язково послідовними цілими числами. Для простоти реалізації розглянемо випадок, коли вони не містять у своїх вузлах однакових чисел. Нехай корінь дерева позначає змінна Root. Розглянемо дерево з рис. 2.15. Тоді Root = 5, Lson [5] = 7, Rson [5] = 2, Lson [7] = 17, Rson [7] = 25, Lson [2] = 3, Rson [2] = 6, Lson [17] = 8, Rson [17] = 19, Lson [25] = 17, Rson [25] = 0, Lson [3] = 0, Rson [3] = 13, Lson [6] = 1, Rson [6] = 15, Lson [8] = 0, Rson [8] = 0, Lson [19] = 0, Rson [19] = 0, Lson [10] = 0, Rson [10] = 0, Lson [13] = 0, Rson [13] = 0, Lson [1] = 0, Rson [1] = 0, Lson [15] = 0, Rson [15] = 0.

Для обходу дерева в оберненому порядку з послідовним друком вершин відвідування використовуватимемо процедуру **Inorder (Root)**. Програми 2.8 і 2.9 демонструють реалізацію відповідно рекурсивного і стекового варіанта процедури **Inorder.**

```
program Ex2_8;
  const n = 1000;                    {Максимальна кількість вершин у дереві}
  type int = 1..n;                   {Тип інтервалу}
       ar = array [int] of integer;  {Тип масиву}
  var  i, k: int;                    {Реальна кількість вузлів дерева}
```

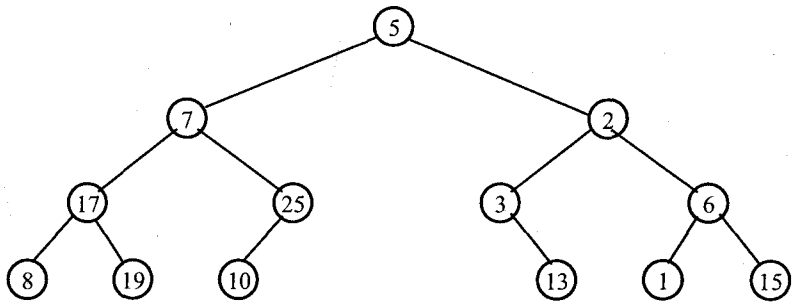


Рис. 2.15. Бінарне дерево глибини 4


```
       v, root: integer;              {Ім'я вершин дерева}
       Lson, Rson: ar;                {Масиви лівих і правих синів вузлів}

procedure Inorder(var node: integer);
  begin
    if Lson[node] <> 0 then Inorder(Lson[node]);
    write(' ', node);
    if Rson[node] <> 0 then Inorder(Rson[node]);
  end;

begin
    writeln('Введіть кількість k вершин дерева розгляду');
    readln(k);
    writeln('Введіть значення масиву лівих Lson і правих Rson синів вузлів дерева
            відповідно');
    readln;
    readln(root, Lson[root], Rson[root]);          {root – корінь дерева}
    for i := 1 to k – 1 do
      readln(v, Lson[v], Rson[v]);
      inorder(root);
end.
```

Програма 2.8. Реалізація рекурсивного варіанта оберненого проходження бінарного дерева

Визначте часову оцінку алгоритму.

Для реалізації нерекурсивної версії алгоритму проходження дерева за оберненим порядком і друку інформаційних частин вузлів можна використати стек STACK. Операції вставки елементу і видалення елементу зі стека здійснюватимемо за допомогою раніше описаних процедур **add** і **del** з підрозділу 2.3.

```
program Ex2_9;
  const m = 100;
  type int = 1..m;
       lisp = array [int] of integer;
  var  stack, Lson, Rson: lisp;
       v, root: integer;
       top, k: int;                          {Покажчик вершини стека}

procedure add(var St: Lisp; item: integer);  {Вставити елемент item у стек St}
begin                                        {максимального розміру m}
    if top >= m then writeln('стек повний')
      else
```

```
        begin
            top := top + 1;
            St[top] := item
        end;
end;

procedure del(var St: Lisp; item: integer);        {Елемент з голови стека}
begin                                              {засилається в item}
    if top < 1 then writeln('стек пустий')
      else
        begin
            item := St[stop];
            top := top – 1
        end;
end;
procedure Inorder1(var node: integer);
 label ll, lc;
 var ver: integer;
begin
    ver := node;
    top := 0;
    ll: while Lson[ver] <> 0 do
          begin
              add(stack, ver);                          {Заноситься ver в стек}
              ver := Lson[ver]
          end;
    lc: write (' ', ver);
       if Rson[ver] <> 0 then
          begin
              ver := Rson[ver];
              goto ll;
          end;
      if top <> 0 then          {Якщо стек не пустий, тоді ver присвоїти значення}
          begin                 {вершинного елементу стека}
              del(stack, item);
              ver := item;
              goto lc;
          end;
end;

begin
    writeln('Введіть кількість k вершин дерева розгляду');
    readln(k);
    writeln('Введіть значення масиву лівих Lson і правих Rson синів
                                                    вузлів дерева відповідно');
```



```
    readln;
    readln(root, Lson[root], Rson[root]);                {root – корінь дерева}
    for i := 1 to k – 1 do
      readln (v, Lson[v], Rson[v]);
    inorder(root);
end.
```

Програма 2.9. Реалізація нерекурсивного варіанту оберненого проходження бінарного дерева

Визначте часову складність алгоритму. Які зміни потрібно внести до наведених програм, щоб реалізувати інші типи обходів дерева?

2.5. Множини

Для багатьох задач основну проблематику можна сформулювати на мові множин, і тоді алгоритм розв'язання задачі можна викласти в термінах основних операцій над множинами (елементами множин). У цьому розділі розглянемо основні структури даних для задання множин і проаналізуємо доцільність їх використання за реалізації різних операцій над елементами множин.

2.5.1. *Основні операції роботи з множинами*

Можна виділити такі операції над множинами:

1. *Належність* (a, S): встановлює належність a множині S; якщо $a \in S$, тоді друкується «*так*», в іншому разі – «*ні*».
2. *Вставити* (a, S): замінює S на $S \cup \{a\}$.
3. *Видалити* (a, S): замінює S на $S - \{a\}$.
4. *Об'єднати* (S_1, S_2, S_3): $S_3 = S_1 \cup S_2$.
5. *Знайти* (a): друкує ім'я тієї множини, якій на цей момент належить a. Якщо a належить більш ніж одній множині, тоді значення операції – *невизначеність*.
6. *Розділити* (a, S): тут вважатимемо, що на множині S визначений лінійний порядок <= . Операція розділяє S на дві множини

$$S_1 = \{b \mid b <= a, b \in S\} \text{ та } S_2 = \{b \mid b > a, b \in S\}.$$

7. *Min* (S): видає найменший (відносно $\leq$) елемент множини S.

Багато задач можна переформулювати як послідовність основних команд на деякій базі даних (універсальній задачі). Розглядатимемо таку послідовність J команд, кожна з яких є однією з вищезазначених операцій 1–7.

Наприклад, розглянемо задачу підтримки визначеної структури даних для множини, яка постійно змінюється. Нові елементи додають до S,

а старі видаляють з S. Час від часу слід також відповідати на питання «*Чи належить x множині S?*». Цю задачу природно моделює словник; нам потрібна структура даних, що дасть змогу зручно виконувати послідовності, які складаються із операцій *належить*, *вставити* і *видалити*.

Наприклад, транслятор контролює «таблицю символів» всіх ідентифікаторів, на які він потрапляє у програмі. Для більшості мов програмування множина всіх ідентифікаторів дуже велика. Тому нереально будувати таблицю символів масивом з одним входом для кожного можливого ідентифікатора, незалежно від того, з'являється він в програмі чи ні.

Операції, які проводить транслятор з таблицею символів, можна поділити на два типи. Перший – поповнення таблиці новими ідентифікаторами (в міру появи) з виділенням у ній комірки для запам'ятовування ідентифікатора. Другий – з часом транслятору потрібна інформація про якийсь з ідентифікаторів, наприклад про його тип.

2.5.2. *Хешування*

Розглянемо спосіб «*розстановки*» (хешування), який дає змогу вдало реалізувати операції *вставити* і *видалити*. Спочатку розглянемо основну ідею [9].

Функція розміщення h відображає елементи універсальної множини S (всі можливі ідентифікатори) у множину цілих чисел від 0 до $m-1$. Вважатимемо, що для всіх елементів a, $a \in S$, значення $h(a)$ можна встановити за константний час. Компонентами масиву $A[0..m-1]$ є покажчики на списки елементів множини. Список, на який вказує $A[i]$, складається із всіх тих елементів $a \in S$, для яких $h(a) = i$. Схему розміщення показано на рис. 2.16.

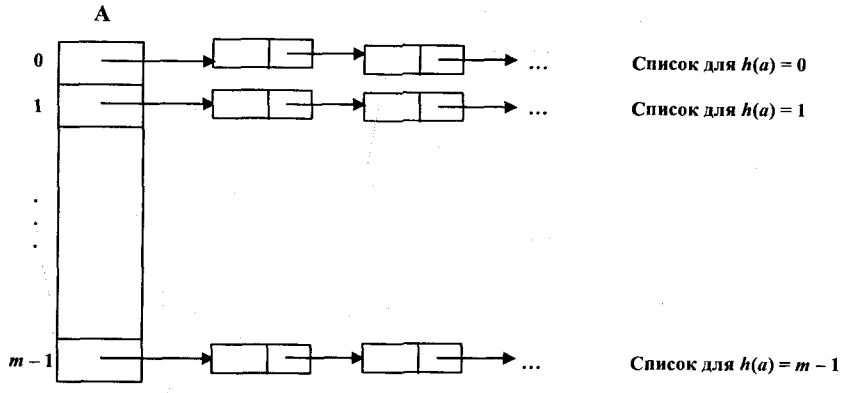


Рис. 2.16. Схема хешування

Для виконання операції *вставити*(a, S), слід обчислити $h(a)$ і потім проглянути список, на який вказує $A[h(a)]$. Якщо елементу a в цьому списку немає, його потрібно додати до кінця цього списку. Аналогічно реалізуються й інші дві операції.

Проаналізуємо обчислювальну складність такої схеми хешування [9]. Розглянемо найгірший випадок. Припустимо, що послідовність J містить n різних операцій *вставити*. Може так трапитися, що на всіх елементах, які потрібно вставити, h набуде одного значення. Тому всі елементи знаходитимуться в одному списку. У цій ситуації для виконання k-ї операції із J буде потрібен час, пропорційний k. Отже, для вставки n елементів до множини S буде необхідним час $O(n^2)$.

Коли ми розглядатимемо серединний випадок, цей процес матиме значно кращий час. Припустимо, що $h(a)$ з рівною вірогідністю може набувати значень, що лежать в проміжку між числами 0 і $m-1$. Вставляється $n \le m$ елементів. Тоді при вставці i-го елементу середня довжина списку, в який він вставляється, дорівнює $(i-1)/m$, що завжди менше або ж дорівнює 1. Тому середній час, необхідний для вставки n елементів, буде $O(n)$.

Аналогічні роздуми проведіть і для операцій ***видалити*** *та* ***належить****.*

Слід зазначити, що в нашому аналізі ми розглядали m (розмір таблиці хешування) не меншим від максимального розміру n множини S. Якщо це не так, тоді слід будувати послідовність таблиць хешування T_0, T_1, T_2, … .

Спочатку вибирають потрібне значення m для розміру початкової таблиці T_0. Після наповнення T_0 m елементами будують нову таблицю T_1 розміром уже $2m$. За допомогою нової функції хешування h, яка тепер набуває значення від 0 до $2m-1$, всі елементи, які знаходились раніше в T_0, переносяться в T_1, і таблицю T_0 більше не використовують. Якщо таблиця T_1 через деякий час знову переповниться, тоді починають аналогічно будувати таблицю T_2 розміром $4m$ з новою функцією хешування, щоб перемістити старі елементи із T_1 в T_2 і т. д. Кожного разу, коли в таблиці T_{k-1} стає $2^{k-1}m$ елементів, будують нову таблицю T_k розміром $2^k m$. Процес зупиняється, коли будуть вичерпані всі елементи.

Підрахуємо середній час вставки в таблицю хешування 2^k елементів за нашою схемою при $m = 1$. Очевидно, що цей процес можна описати рекурентним рівнянням

$$T(1) = 1,$$
$$T(2^k) = T(2^{k-1}) + 2^k,$$

розв'язком якого буде $T(2^k) = 2^{k+1} - 1$. Отже, за допомогою хешування в послідовності J із n елементів операції *вставити*, *належить*, *видалити* можна виконати за середній час $O(n)$ у разі вдалого вибору функції хешування h.

2.5.3. *Використання дерев для реалізації операцій об'єднати і знайти*

Нехай ми маємо деяку скінченну область з n елементів U, з якої конструюватимемо множини. Ці множини можуть бути пустими або складатись з будь-якої підмножини елементів U.

Загальним підходом до задання таких множин є використання характеристичного бітового вектора Set (n) довжини n, такого що:

$$Set(i) = \begin{cases} 1, & \text{якщо } i\text{-й елемент } U \text{ знаходиться в цій множині;} \\ 0 - & \text{в іншому випадку.} \end{cases}$$

Перевага такого задання полягає в швидкому визначенні належності елементу множині та у можливості використання для реалізації операцій об'єднання і розділення стандартних логічних операцій диз'юнкції та кон'юнкції. Але значення n у такій реалізації обмежується довжиною машинного слова вибраної ЕОМ. Тому альтернативним заданням множин є задання кожної множини списком її елементів. Проміжок часу для виконання об'єднання чи перетину пропорційний до n більш ніж до кількості елементів у двох множинах.

За певних несуттєвих обмежень щодо елементів множин, зручною формою задання множин є використання дерев [9, 61, 127].

Вважатимемо, що елементи множин – числа 1, 2, 3, ..., n. Ці числа могли б, практично, бути індексами в символьній таблиці, де зберігаються фактичні імена елементів. Будемо також вважати, що множини зберігання попарно не перетинаються; тобто, якщо S_i та S_j, де $i \neq j$,– дві множини, тоді не існує елементів, які належать і S_i, і S_j одночасно.

Операції *union* і *find*, які ми бажаємо виконати на цих наборах (*об'єднання множин*, *пошук*(i)), треба виконувати ефективно – часова оцінка повинна мати константний характер і не залежати від кількості елементів у множинах.

Припустимо, множини об'єднання S_1 і S_2 неперетинні, $S_1 = \{1, 5, 9, 20\}$, а $S_2 = \{2, 7, 13\}$. Тоді у вигляді дерев ці множини можна задавати так, як наведено на рис. 2.17.

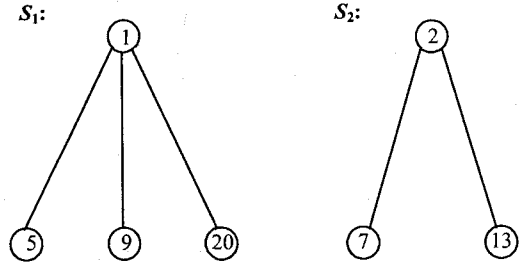


Рис. 2.17. Дерева множин

Таке задання зручне для реалізації об'єднання. У цьому разі одне з дерев стає піддеревом кореневого рівня іншого дерева. Реалізацію $S_1 \cup S_2$ подано на рис. 2.18.

Інакше кажучи, все, що нам треба зробити,– це встановити батьківський зв'язок одного кореня з іншим.

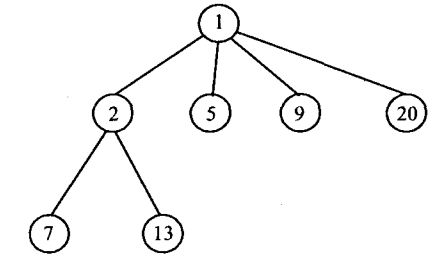


Рис. 2.18. Дерево об'єднання

В алгоритмах *union* та *find* реалізації відповідних операцій ми можемо ідентифікувати множини за індексом коренів дерев. Це спростить обговорення. Перехід до імені множини простий: ім'я множини – це індекс кореня. Тепер завданням операції *find*(i) стає визначення кореня дерева, в якому знаходиться елемент i. Операція *union*(i, j) об'єднує два дерева з коренями i та j. Вважатимемо, що вузли в деревах пронумеровані від 1 до n так, що індекс вузла відповідає індексу елементу. Тобто, елемент 5 зображений вузлом з індексом 5.

Тоді для задання таких дерев достатньо у вузлі мати лише одне поле зв'язку: *parent* – поле зв'язку зі своїм батьком. У разі вузла-кореня значення цього поля дорівнює 0. Для вузлів з рис. 2.18 parent[7] = 2, parent[9] = = 1, parent[1] = 0.

Нехай в головній програмі присутній опис

```
const n = 1000;                                  {n – визначає розмірність U}
type  int = 1..n;
      arr = array [int] of integer;   {Індекс визначає вузол дерева, а значення}
                                      {елементу масиву – реалізує зв'язок parent}
```

Тоді найпростіші алгоритми реалізації операцій *union* і *find* задаватимуться такими процедурами:

```
procedure union(i,j: int; var parent: arr);
begin
   parent[i] := j;
end;

function find(i: int; var parent: arr): integer;
 var j: int;
begin
   j := i;
   while parent[j] > 0 do                        {Поки j – не корінь}
      j := parent[j]
   find := j
end;
```

Рис. 2.19. Найгірший варіант дерева

Реалізація цих алгоритмів дуже проста, але іноді застосування може бути неефективним. Розглянемо такий випадок [124].

Наприклад, якщо ми почнемо роботу з n елементами, кожен з яких знаходиться у своїй власній множині, тобто $S_i = \{i\}$, де $1 \le i \le n$, тоді початкова конфігурація є лісом з n вузлами та PARENT$(i) = 0$, де $1 \le i \le n$. Тепер уявіть, що ми обробляємо наступну послідовність з *union–find* операцій:

$$U(1, 2), F(1), U(2, 3), F(1), U(3, 4),$$
$$F(1), U(4, 5), \ldots, F(1), U(n-1, n).$$

Результат цієї послідовності – вироджене дерево, зображене на рис. 2.19.

Щоб такого не траплялось, нам слід накласти додаткову умову, яка б примушувала будувати дерево. Один із варіантів – це застосування вагового правила (WR) для операції об'єднання $U(i, j)$: якщо число вузлів у дереві i менше від числа вузлів у дереві j, тоді візьмемо j як корінь для вузла i, інакше i стає коренем для j. Використавши правило WR для попередньої послідовності операцій об'єднання і знаходження, будемо мати дерево рівня два (рис. 2.20). Тому час для n операцій знаходження буде $O(n)$, оскільки найбільший рівень для будь-якого вузла дорівнює n. Можна також довести [127], що за використання правила WR для найгіршого випадку послідовності операцій *union* і *find*, максимальний рівень довільного вузла обмежуватиметься значенням $\lfloor \log n + 1 \rfloor$.

Реалізуємо вагове правило. Для цього нам необхідно знати кількість вузлів у деревах розгляду. У попередній структурі даних за реалізації дерева масивом parent для кореня i parent$[i] = 0$. Тепер замість нуля поставимо від'ємне число, яке і характеризуватиме своїм абсолютним зна-

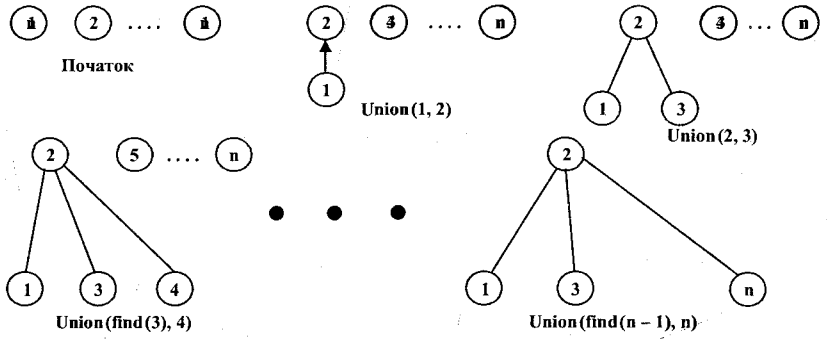


Рис. 2.20. Реалізація послідовності дій

ченням кількість вузлів у відповідному дереві. Якщо для дерева з рис. 2.18 раніше parent[1] = 0, то тепер parent[1] = –7. Матимемо такий алгоритм реалізації операції об'єднання:

```
procedure union(i,j: int; var parent: arr);
 var x: int;
begin
    x := parent[i] + parent[j];        {Визначили кількість вузлів у новому дереві }
    if parent[i] > parent[j]           {Не забудьте, що це порівнюються від'ємні числа}
      then
        begin
          parent[i] := j;
          parent[j] := x
        end;
      else
        begin
          parent[j] := i;
          parent[i] := x
        end
end;
```

Визначте часову оцінку алгоритму, описаного процедурою ***union****. Напишіть процедуру або функцію реалізації операції* ***знайти****.*

2.5.4. *Черги з пріоритетом*

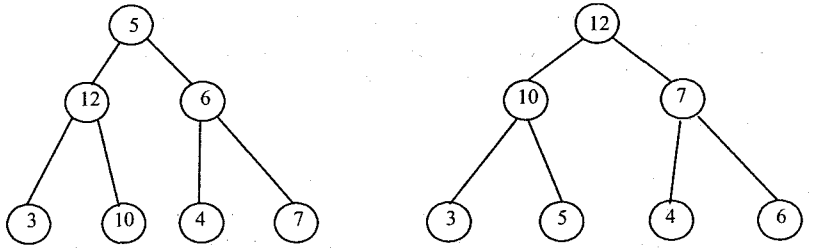
Розглянемо ще одну структуру даних, яка забезпечує одночасно ефективну реалізацію операцій вставки елементу в множину і знаходження найбільшого елементу в множині – чергу з пріоритетом. Таку структуру даних ще називають *купою*.

Купою (heap) називають повне бінарне дерево, для довільного вузла якого виконується умова: значення, що міститься у вузлі, більше або дорівнює значенням, які знаходяться в синах цього вузла (якщо вони існують).

Це визначення імплікує той факт, що в корені купи знаходитиметься найбільший елемент множини, якщо він у ній єдиний.

Якщо подивимось на дерево з рис. 2.21, *а*, тоді купа, отримана з нього, буде зображена на рис. 2.21, *б*.

Розглянемо загальне правило формування купи для n цілих чисел за допомогою масиву $A[1..n]$. Купу будують послідовною вставкою n елементів, починаючи з порожнього дерева. Черговий елемент вставки додають до «дна купи» (додають як сина вузла найвищого рівня, для якого таке доповнення не порушує властивості бінарності дерева). Далі порівнювати його і переставляти з батьком, дідом, прадідом і т. д. доти, доки його значення не стане меншим або рівним значенню, яке міститься в предку.



а *б*

Рис. 2.21. Дерево (*а*) і купа (*б*) з нього

Детальніше алгоритм побудови купи за допомогою процедури вставки *insert* наведено в програмі 2.10.

```
program Ex2_10;
 const m = 100;
 type  int = 1..m;
       arr = array [int] of integer;
 var   i, j: int;
       a: arr;

procedure insert(n: int; var aa: arr);
var k, l: int;
    item: integer;

begin
    k := n;
    l := n div 2;
    item := aa[n];
    while ((k > 0) and (aa[l] < item)) do
      begin
        aa[k] := aa[l];                    {Зсунути батька вниз}
        k := l;
        l := l div 2;                      {Батьком aa[l] є aa[l div 2]}
      end;
    aa[k] := item                          {Місце для aa[l] вже знайдено}
end;

begin
    readln(j);          {Визначає довжину масиву, з якого будуватиметься купа}
    for i := 1 to j do
      read(a[i]);
    for i := 2 to j do                     {Побудова купи}
      insert(i,a);
    for i := 1 to j do                     {Виведення до друку купи}
      write(a[i], ' ');
end.
```

Програма 2.10. Побудова купи за допомогою алгоритму вставки

В якому разі процедура ***insert*** *буде мати найгіршу часову оцінку?*

Побудова купи для множини з n елементів за допомогою процедури *insert* матиме найгірший час у разі, коли елементи, з яких будують купу (масив $A(n)$), знаходитимуться за порядком зростання. У цьому разі новий елемент вставки кожного разу переміщуватиметься аж у корінь уже існуючого дерева.

Якщо в повному бінарному дереві кількість вузлів рівня i дорівнює 2^{i-1} та відстань від будь-якого вузла до кореня дорівнює $(i-1)$, часову оцінку побудови купи визначатиме формула

$$\sum_{i=1}^{\lceil \log 2(n+1) \rceil} (i-1)2^{i-1} < \lceil \log 2(n+1) \rceil 2^{\lceil \log 2(n+1) \rceil} = O(n \log n).$$

Але тестування процедури на різних вхідних послідовностях [127] показує, що в серединному випадку вона потребує $O(n)$ часу, тому що нове значення піднімається кожен раз тільки на константне число рівнів у дереві.

Нехай множину задає масив, який описує повне бінарне дерево (дерево вже існує). Можна запропонувати ще один підхід до побудови купи. Він працює так. Лист дерева є купою, тоді для чергового вузла вищого рівня слід перевірити, щоб ліве і праве піддерева були купами, скоригувавши для цього значення вузла-батька і його синів. Процес зупиняється у разі досягнення кореня дерева. Назвемо цю процедуру *adjust*.

Процедура *heapify* буде викликати процедуру *adjust* для побудови купи з початкового дерева. Програма 2.11 демонструє деталізацію реалізації останнього алгоритму.

```
program Ex2_11;
 const m = 100;
 type int = 1..m;
      arr = array [int] of integer;
 var  i, j: int;
      a: arr;

procedure adjust(i, n: int;var aa: arr);  {Процедура побудови купи для масиву aa[1..n]}
  var j: int;
      item: integer;
 label exit;
begin
    j := 2 * i; item := aa[i];
    while j <= n do
     begin
      if (j < n) and (aa[j] < aa[j + 1])      {Порівняння лівого і правого синів}
        then j := j + 1;                      {j вказує на більшого сина}
```

```
    if item >= aa[j]
      then goto exit                    {Позицію для item вже знайдено}
      else
       begin
         aa[(j div 2)] := aa[j];
         j := 2 * j
       end
   end;
  exit: aa[(j div 2)] := item
end;

procedure heapify(n: int; var a: arr);
 var i: int;
begin
   for i := n div 2 downto 1 do
     adjust(i, n, a)
end;

begin
   readln(j);
   for i := 1 to j do
     read(a[i]);
   heapify(j, a);
   for i := 1 to j do
     write(a[i],' ');
end.
```

Програма 2.11. Побудова купи за допомогою процедури *adjust*

Проаналізуємо часову оцінку алгоритму *heapify* для найгіршого випадку [127]. Це випадок, коли нам потрібно зробити перестановки в усіх піддеревах. Нехай кількість елементів у купі n задовольняє умові $2^{k-1} \le n < 2^k$, де $k = \lceil \log_2(n+1) \rceil$ – кількість рівнів повного бінарного дерева з n-вузлами. Для кожного вузла рівня i найбільша кількість застосувань *adjust* буде $k - i$. Звідки загальний час визначатиметься формулою

$$\sum_{i=1}^{k} 2^{i-1}(k-1) = \sum_{i=1}^{k-1} i 2^{k-i-1} \le \sum_{i=1}^{k-1} \frac{i}{2^i} < n = O(n).$$

Час – кращий за попередній, але використання *adjust* потребує уже створеного дерева елементів, з яких будується купа.

Повернемось до використання купи для реалізації черги з пріоритетом. Основні операції у цьому разі – це вставка і видалення найбільшого елементу.

Для видалення елементу з черги слід видалити елемент з голови купи (елемент $A[1]$) і перемістити на його місце елемент $A[n]$. Для відновлення властивості купи слід викликати процедуру *adjust* відносно кореневого елементу. Вставку елементу в чергу можна реалізувати за допомогою процедури *insert*. Тому реалізацію операцій видалення і вставки в чергу з пріоритетом можна зробити за час $O(\log 2n)$.

2.5.5. *Сортування купою*

Зрозуміло, якщо ми послідовно виконаємо n разів операцію видалення із купи з подальшим зберіганням елементів видалення, тоді на виході матимемо впорядковану за зростанням чи спаданням (залежно від властивості формування купи: значення батька більше за значення синів, чи значення батька менше за значення синів) послідовність елементів. Отже, маємо алгоритм впорядкування: сформувати купу (використовуючи *insert*, можна зробити за час $O(n\log_2 n)$); знайти і видалити максимальний (мінімальний) елемент (можна зробити за час $O(n\log_2 n)$) n разів. Тому часова оцінка цього алгоритму буде $O(n\log_2 n)$. Програма 2.12 впорядковує послідовність елементів за допомогою купи.

```
program Ex2_12;
  const m = 100;
  type int = 1..m;
       arr = array [int] of integer;
  var  i, j: int;
       a: arr;

procedure adjust(i, n: int; var aa: arr);
  var  j: int;
       item: integer;
  label exit;
begin
    j := 2 * i; item := aa[i];
    while j <= n do
      begin
        if (j < n) and (aa[j] < aa[j + 1])
          then j := j + 1;
        if item >= aa[j]
          then goto exit
          else
            begin
              aa[(j div 2)] := aa[j];
              j := 2 * j
            end;
      end;
```

```
      exit: aa[(j div 2)] := item
end;

procedure heapify(n: int; var a: arr);
  var i: int;
begin
    for i := n div 2 downto 1 do
      adjust(i, n, a)
end;

procedure heapsort(n: int; var a: arr);
  var i, t: int;
begin
    heapify(n, a);                    {Створимо з масиву a[1..n] купу}
    for i := n downto 2 do
      begin
        t := a[i]; a[i] := a[1]; a[1] := t;   {Перемістимо черговий елемент з голови}
        adjust(1, i – 1, a)           {в кінець дерева і відновимо купу}
      end;
end;

begin
    readln(j);
    for i := 1 to j do
      read(a[i]);
    heapsort(j, a);
    for i := 1 to j do
      write(a[i], ' ');
end.
```

Програма 2.12. *Heap*-впорядкування

Проведіть тестування програми ***heap****-впорядкування на конкретних даних.*

2.6. Ізоморфізм дерев

Продемонструємо описану техніку роботи з деревами на прикладі задачі ізоморфізму дерев. Два дерева називають *ізоморфними*, якщо можна відобразити одне дерево в інше з точністю до перейменування синів вершин [9].

Нехай маємо два дерева T_1 і T_2. Потрібно визначити, ізоморфні ці дерева чи ні? Розглянемо алгоритм, який дає відповідь на поставлене запитання за лінійний час, пропорційний числу вершин.

1. Припишемо всім листкам дерев T_1 та T_2 ціле число 0.

2. Припустимо, що всі вершини дерев T_1 та T_2 рівня i мають приписані цілі числа. Нехай L_1 є списком вершин дерева T_1 рівня i, які впорядковані

за зростанням значень приписаних цілих чисел. L_2 є подібним списком для T_2.

3. Припишемо вузлам, що відмінні від листків дерева T_1 рівня $i - 1$, кортежі цілих, проглядаючи список L_1 зліва направо і розміщуючи в кортежі цілі числа (відмітки синів) за зростанням. Кортеж для деякої вершини v цього рівня матиме вигляд: $(i_1, i_2, ..., i_k)$. Побудуємо послідовність S_1 кортежів, створених для дерева T_1 на $(i - 1)$-му рівні.

4. Повторимо крок 3 для дерева T_2 на $(i - 1)$-му рівні, створивши, відповідно, послідовність кортежів S_2.

5. Впорядкувавши S_1 та S_2 згідно з лексично-графічним порядком, отримаємо відсортовані послідовності кортежів S_1' та S_2'.

Якщо S_1' та S_2' не ідентичні, тоді зупинитись: дерева не ізоморфні і кінець. Якщо ж вони ідентичні і ми досягли рівня кореня, тоді кінець – дерева ізоморфні. В іншому разі, по-перше, проведемо кодування нових кортежів: припишемо число $k + 1$ першому новому кортежу із S_1, який не кодували заздалегідь, припишемо число $k + 2$ іншому новому кортежу, що не кодували раніше, і так далі, поки не будуть перебрані всі нові кортежі із S_1 (перший кортеж кодується $k = 1$); по-друге, припишемо вершинам-нелисткам наступного рівня розгляду мітку, що відповідає кортежу її синів; по-третє, перейдемо на крок 3 алгоритму.

Рис. 2.22 ілюструє процес дослідження ізоморфності двох дерев.

Доведіть, що часова оцінка цього алгоритму визначення ізоморфізму двох ***n****-вершинних дерев має порядок* $\boldsymbol{O(n)}$.

На першому рівні після ініціалізації нулями листків дерева маємо таку початкову ситуацію для дерев T_1 та T_2:

$$L_1^0 = \langle 9,10,11,12,13,14,15,16 \rangle, \quad L_2^0 = \langle 9,10,11,12,13,14,15,16 \rangle,$$
$$S_1 = \langle (0,0),(0,0,0,0),(0),(0,0) \rangle, \quad S_2 = \langle (0),(0,0),(0,0),(0,0,0,0) \rangle.$$

Впорядкуємо S_1 і S_2:

$$S_1' = \langle (0),(0,0),(0,0),(0,0,0,0) \rangle, \quad S_2' = \langle (0),(0,0),(0,0),(0,0,0,0) \rangle.$$

На цьому рівні дерева ізоморфні ($S_1' = S_2'$), тому закодуємо кортежі: (0) – 0, (0,0) – 1, (0,0,0,0) – 2 і припишемо їх відповідним батькам в деревах T_1 та T_2:

$$L_1^1 = \langle 4,5,6,7,8 \rangle, \qquad L_2^1 = \langle 4,5,6,7,8 \rangle,$$
$$S_1 = \langle (0,1),(0),(0,0,2),(0,1) \rangle, \qquad S_2 = \langle (0,1),(0,0,2),(0,1),(0) \rangle,$$
$$S_1' = \langle (0),(0,0,2),(0,1),(0,1) \rangle, \qquad S_2' = \langle (0),(0,0,2),(0,1),(0,1) \rangle.$$

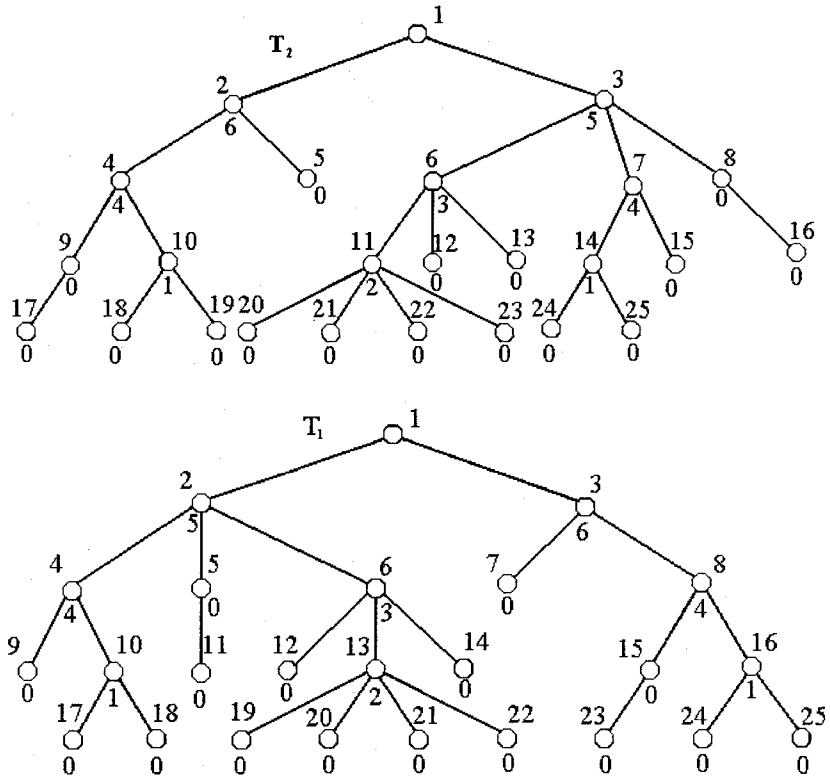


Рис. 2.22. Дерева перевірки

На цьому рівні дерева знову ізоморфні, тому що $S_1' = S_2'$. Закодуємо нові кортежі (0,0,2) – 3, (0,1) – 4 і перейдемо до наступного рівня:

$$L_1^2 = \langle 2,3 \rangle, \qquad L_2^2 = \langle 2,3 \rangle,$$
$$S_1 = \langle (0,3,4),(0,4) \rangle, \qquad S_2 = \langle (0,4),(0,3,4) \rangle,$$
$$S_1' = \langle (0,3,4),(0,4) \rangle, \qquad S_2' = \langle (0,3,4),(0,4) \rangle.$$

Дерева ізоморфні і на цьому рівні. Перейдемо до наступного рівня (це рівень кореня), попередньо закодувавши нові кортежі (0,3,4) – 5, (0,4) – 6:

$$L_1 = \langle 1 \rangle, \qquad L_2 = \langle 1 \rangle,$$
$$S_1 = \langle (5,6) \rangle, \qquad S_2 = \langle (5,6) \rangle,$$
$$S_1' = \langle (5,6) \rangle, \qquad S_2' = \langle (5,6) \rangle,$$

$S_1' = S_2'$ і рівень розгляду дерева – корінь. Тому дерева ізоморфні, і кінець роботи алгоритму.

Під час перевірки роботи алгоритму на аркуші паперу з олівцем в руках ніяких проблем з алгоритмом перевірки ізоморфізму дерев не виникає. Під час програмування цього алгоритму маємо розв'язати такі основні підзадачі:

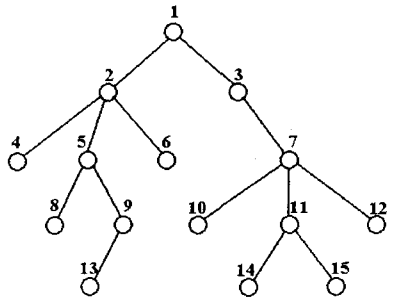


Рис. 2.23. Дерево

1) для введення дерев T_1 і T_2 у пам'ять ЕОМ слід вибрати якийсь принцип задання структури вузла дерева;

2) потрібна процедура, яка б аналізувала вхідну послідовність символів і будувала в пам'яті ЕОМ дерево;

3) структура вузла дерева має задовольняти таким основним вимогам: мати інформаційне поле для запису імені вершини (номера), поле для інформації про рівень вузла в дереві, орієнтуватись на довільне (максимум заздалегідь фіксований) число синів вузла; давати змогу простого розпізнавання: вузол дерева – лист чи ні, та мати зручні можливості встановлення для вузла як його нащадків, так і предків;

4) потрібна така процедура побудови списку кортежів, яка б розміщувала кортежі в список S у лексично-графічному порядку за лінійний час;

5) маємо потребу кодувати кортежі чисел і зіставляти на рівність списки кортежів.

Почнемо розробляти програму для реалізації алгоритму ізоморфізму двох довільних дерев з урахуванням вищенаведеного.

Підзадачі 1–2 можна розв'язати у такий спосіб. Для кодування дерева використовуватимемо алгоритм, який ґрунтується на прямому проходженні дерев. Піддерева вузла обходитимемо зліва направо, записуючи номери вузлів. У разі досягнення вершини-листа запишемо символ 0. Нуль також позначає обробку всіх піддерев вузла і сигналізує аналізуючій програмі підняття на один рівень уверх і побудову піддерев для вузла цього рівня. Дерево з рис. 2.23 описуватиметься такою послідовністю символів: 1, 2, 4, 0, 5, 8, 0, 9, 13, 0, 0, 0, 6, 0, 0, 3, 7, 10, 0, 11, 14, 0, 15, 0, 0, 12, 0. Опишемо детально процес побудови цієї послідовності.

Згідно з прямим обходом, отримуємо послідовність 1, 2, 4. Вершина 4 – лист, тому далі запишемо 0. Він сигналізуватиме про підйом до вершини-батька 2. Продовжуючи обхід, отримуємо послідовність 5, 8, 0, і далі 9, 13, 0. Останній 0 підніме тільки до вершини батька 9, а нам необхідно піднятись до вершини з номером 2, на що й вказує наявність двох наступних нулів. Далі буде 6, 0, і наступний 0 підніме нас до кореня дерева. Після цього аналогічно отримуємо 3, 7, 10, 0, 11, 14, 0, 15, 0, 0, 12, 0. Ознакою кінця вхідної стрічки вважатимемо ознаку кінця файла стандартного вводу.

Для розв'язання підзадачі 3 скористаємося таким описом:

```
const m = 10;                              {Визначає степінь дерева}
type  ar = array [1..m] of integer;
      node = ^nod;
      nod = record                         {Структура вузла дерева}
              inf: integer;                {Ім'я вузла}
              fct: ar;                     {Кортеж чисел кодування синів}
              fc: 0..m;                    {Позначка для вузла-листа}
              arl: array [1..m] of node;   {Масив покажчиків на синів вузла}
              val: 0..m;                   {Число синів вузла}
              bach: node;                  {Покажчик на батька вузла}
              lev: integer;                {Рівень вузла в дереві}
              longh: 0..m;                 {Довжина кортежу}
             end;
```

Для опису списку кодів кортежів скористаємося такою структурою вузла списку:

```
hnode = ^node;
hnod = record
        i: 0..m;                           {Довжина кортежу}
        hor: array [1..m] of integer;
        n: integer;                        {Код кортежу}
        lin: hnode                         {Поле зв'язку}
        end;
```

Скористаємося також ще одним списком – список вузлів поточного рівня аналізу дерева, список типу *stnod*.

```
stnod = ^stn;
stn = record
        inf: node;                 {Покажчик на вузол у дереві}
        lin: stnod;                {Поле зв'язку списку}
      end;
```

Для реалізації початкового 0-го кроку алгоритму нам потрібна процедура, яка б визначала висоту дерева. Її можна написати, використовуючи рекурсивну процедуру *ob* проходження дерева в оберненому порядку.

```
procedure ob(p: node; var le: integer);
        {p – покажчик на корінь дерева, що аналізується}
        {le – результуюча змінна висоти дерева. За початкового виклику le = 0}
  var k: 1..m;
begin
   if p^.fc <> 0                          {Якщо вузол розгляду не лист, тоді викликати}
     then                                 {процедуру обходу для всіх синів вузла}
```



```
        for h := 1 to p^.val do
            ob (p^.arl[h], le);
    if p^.lev > le then le := p^.lev
end;
```

Для вирішення підзадачі 2 використовуватимемо процедуру *btr*, яка аналізує вхідну послідовність символів і відтворює в пам'яті ЕОМ відповідне дерево з покажчиком на голову *head*.

```
procedure btr(var head: node)
  var  c: integer;
       p,q: node;
begin
  if eof then writeln('Немає вхідних даних')
    else
      begin
       readln(c);
       if c = 0 then writeln('Дерево пусте')
         else
           begin
             new(head);     {Створимо вузол дерева, на який вказує покажчик head}
             p := head;     {Робочий покажчик p також вказуватиме на цей же}
             p^.fc := 0;    {вузол, зробимо початкову ініціалізацію полів вузла}
             p^.val := 0;
             p^.lev := 0;
             p^.longh := 0;
             p^.inf := c;                  {В інформаційне поле засилається ім'я вузла}
             while (not (eof)) do          {Поки не кінець стандартного файла}
              begin
                while (not (eoln)) do      {Поки не кінець стрічки}
                 begin
                   read(c);                {Читати чергове ціле в c}
                   if c <> 0 then
                      begin                {Створити вузол}
                        new(q);            {Вузол p буде батьком для вузла q,}
                        p^.fc := 1;        {тому він не лист}
                        p^.val := p^.val + 1;   {Збільшимо кількість синів}
                        p^.arl [p^.val] := q; {Черговий покажчик масиву arl покажчиків}
                                              {на сини p встановимо на q}
                        q^.bach := p;      {Покажчик на батька вузла q встановимо}
                                           {на p і зробимо початкову ініціалізацію вузла q}
                        q^.fc := 0;
                        q^.val := 0;
                        q^.inf := c;
                        q^.longh := 0;
```

```
                q^.lev := p^.lev + 1;    {Висота вузла q на 1 більша за висоту p}
                                         {Потенційним вузлом-батьком на наступному}
                                         {кроці буде останній оброблений вузол}
              end
            else                         {Якщо поточний символ 0, слід піднятись}
              begin                      {на один рівень уверх}
                p := q^.bach;
                q := p
              end;
          end;                           {Кінець обробки стрічки}
          readln;
        end;                             {Кінець обробки вхідного файла}
    end;
  end;
end;                                     {Кінець процедури}
```

Для побудови списків вузлів типу L поточного рівня розгляду використовуватимемо процедуру *inc*, яка вставляє черговий вузол q у потрібне місце списку p згідно зі зростанням чисел – імен вузлів, не порушуючи впорядкування. Простоту циклу перебігу елементів списку забезпечує використання спеціального покажчика *bar* (бар'єр), який визначає кінець списку. Процедура використовує три робочих покажчика r, t, tn, де r вказує на новий елемент списку, в який занесено значення q і який треба вставити в потрібне місце. Його визначають покажчики t і tn: t вказує на попередній елемент, tn – на наступний елемент списку вставки. Маємо два варіанти вставки. Перший варіант – це вставка в пустий список або на початок списку, а другий – це вставка між деякими вузлами списку або в кінець списку.

```
procedure Inc(var p, bar: stnod; q: node);
        {p – покажчик на початок списку; bar – покажчик на кінець списку;}
        {q – покажчик в дереві на вузол, який слід вставити в список p}
  var r, t, tn: stnod;
begin
    new(r);                      {Породити структуру вузла}
    r^.inf := q;                 {Наповнити інформаційну частину}
    t := p;                      {Встановити робочий покажчик на початок списку}
    if ((p = bar) or (p^.inf^.fc < q^.fc))
      then
        begin                    {Реалізація першого варіанта вставки}
          p := r;
          r^.lin := t;
        end;
      else                       {Реалізація другого варіанта вставки}
```



```
        begin
          tn := t^.lin;
          while ((tn <> bar) and (tn^.inf^.fc < q^.fc)) do
            begin                {Цикл пошуку потрібного місця вставки}
              t := tn;
              tn := t^.lin;
            end;
          t^.lin := r;
          r^.lin := tn
        end
end;
```

Повну реалізацію алгоритму визначення ізоморфізму довільних двох дерев наведено в програмі 2.13.

```
program Ex2_13;
  label m1;
  const m = 10;                          {Максимальна кількість синів у вузлі}
  type
    ar = array [1..m] of char;
    int = 0..m;
    node = ^nod;
    nod = record                         {Вузол дерева}
            inf: integer;                {Ім'я вузла}
            fct: ar;                     {Кортеж кодів синів}
            fc: char;                    {Відмітка про листок}
            arl: array [1..m] of node;   {Покажчики на синів вузла}
            val: integer;                {Число синів вузла}
            back: node;                  {Покажчик на батька}
            lev: integer;                {Висота вузла}
            longk: 0..m                  {Довжина кортежу}
          end;
    knode = ^knod;
    knod = record                        {Вузол списку кодів кортежу}
            i: 0..m;                     {Довжина кортежу}
            kor: array [1..m] of char;   {Кортеж}
            lin: knode;
            n: char                      {Код кортежу}
           end;
    stnod = ^ stn;
    stn = record                         {Вузол списку вузлів поточного рівня}
            inf: node;                   {Покажчик на вузол дерева}
            lin: stnod;
          end;
  var head1, head2: node;
```

```
        hdk: knode;
        q, p: node;
        lisph1, lisph2, bar1, bar2: stnod;
        c: char;
        tn: char;
        llev1, llev2: integer;
        pris: boolean;

procedure ob(p: node; var le: integer);
{Знаходження висоти дерева}
{p – покажчик на голову дерева;}
{le – висота дерева}
  var k: 1..m;
begin
    if p^.fc <> '0' then
       for k := 1 to p^.val do
           ob(p^.arl[k], le);
    if (p^.lev > le) then le := p^.lev;
end;

procedure inc(var p, bar: stnod; q: node);
{Впорядковує вставку вузлів у список поточного рівня}
{p – покажчик на початок списку;}
{bar – покажчик на бар'єрний елемент кінця списку;}
{q – покажчик на вузол вставки}
  var r, t, tn: stnod;
begin
    new(r);
    r^.inf := q;
    t := p;
    if ((p = bar) or (p^.inf^.fc < q^.fc)) then
       begin
          p := r;
          r^.lin := t;
       end
      else
       begin
          tn := t^.lin;
          while ((tn <> bar) and (tn^.inf^.fc < q^.fc)) do
             begin
                t := tn;
                tn := t^.lin;
             end;
          t^.lin := r;
          r^.lin := tn;
```



```
       end
end;

function num(var head: knode; a: ar; k: int): char;
{Кодування кортежу}
{head – покажчик на початок списку;}
{a – масив кортежів кодування;}
{k – довжина кортежу}
  var mark, ma: boolean;
      t, q, qq: knode;
      i: 0..m;
begin
    if k = 1 then num := a[1]                    {Якщо вузол є списком}
      else
        begin
          q := head;
          mark := true;
          while (mark and (q <> nil)) do
            begin
              ma := true;
              if (q^.i = k) then
                begin
                  for i := 1 to k do
                      if a[i] <> q^.kor[i] then
                         ma := false;
                      if (ma) then
                        begin
                          num := q^.n;
                          mark := false;
                        end;
                end;
              qq := q;                           {Можна обійтися і без qq}
              q := qq^.lin;
            end;
          if mark then
            begin
              new(t);
              for i := 1 to k do
                  t^.kor[i] := a[i];
              t^.i := k;
              t^.n := tn;
              tn := chr(ord(tn) + 1);
              t^.lin := nil;
              qq^.lin := t;
              if head = nil then head := t;
```

```
            num := t^.n;
          end
        end;
end;

procedure blch(p: node; var bar, lisph: stnod; var h: integer);
{Побудова списку вузлів поточного рівня}
{p – покажчик на голову дерева;}
{bar – покажчик на кінець списку;}
{lisph – покажчик на бар'єрний елемент початку списку;}
{h – поточний рівень}
  var k: 1..m;
begin
    if p^.fc <> '0' then
      for k := 1 to p^.val do
          blch (p^.arl[k], bar, lisph, h);
    if (p^.lev = h) then
      begin
          if (p^.fc <> '0') then
              p^.fc := num(hdk, p^.fct, p^.longk);
          inc(lisph, bar, p);
      end
end;

procedure prl(var p, bar: stnod);                    {Друк списку}
  var r: stnod;
begin
    r := p;
    while(r <> bar) do
      begin
          write(r^.inf^.fc, ' ');
          r := r^.lin;
      end
end;

procedure fork(var p, bar: stnod);
                    {Побудова кортежу відміток вузлів для вузла рівня i із i + 1 рівня}
                    {p – покажчик на початок списку вузлів рівня i + 1;}
                    {bar – покажчик на кінець цього списку}
  var r: stnod;
      q, t: node;
      k: 1..m;
begin
    r := p;
    while (r <> bar) do
```



```
      begin
          q := r^.inf;
          t := q^.back;
          k := t^.longk + 1;
          t^.fct[k] := q^.fc;
          t^.longk := k;
          r := r^.lin;
      end
end;

function rescoml(var p1,br1, p2, br2: stnod): boolean;        {Порівняння двох списків}
  var r1, r2: stnod;
      res: boolean;
begin
    r1 := p1;
    r2 := p2;
    res := true;
    while ((r1 <> br1) and (r2 <> br2) and res) do
      begin
          if (r1^.inf^.fc <> r2^.inf^.fc) then res := false;
          r1 := r1^.lin;
          r2 := r2^.lin;
      end;
    if res then
      if ((r1 <> bar1) or (r2 <> bar2)) then
          res := false;
    rescoml := res;
end;

procedure btr (var head: node);      {Побудова дерева}
                                     {head – покажчик на корінь дерева}
  var c: integer;
      p,q: node;
begin
    read(c);
    if c <> 100 then
        begin
          if (c <> 0) then
              begin
                  new(head);
                  p := head;
                  p^.fc := '0';
                  p^.val := 0;
                  p^.lev := 0;
                  p^.longk := 0;
```

```
            p^.inf := c;
            read(c);
            while (c <> 100) do
               begin
                  if (c <> 0) then
                        begin
                           new(q);
                           p^.fc := '1';
                           p^.val := p^.val + 1;
                           p^.arl [p^.val] := q;
                           q^.back := p;
                           q^.fc := '0';
                           q^.val := 0;
                           q^.inf := c;
                           q^.longk := 0;
                           q^.lev := p^.lev + 1;
                           p := q;
                        end
                     else
                        begin
                           p := q^.back;
                           q := p;
                        end;
                  read(c);
               end;
         end
      else writeln('пусте дерево');
   end
else writeln('немає даних');
end;

begin
   head1 := nil;
   head2 := nil;
   hdk := nil;
   pris := true;
   tn := 'a';
   readln;
   writeln('введіть перше дерево: ');
   btr(head1);
   llev1 := 1;
   ob(head1,llev1);
   writeln;
   writeln('high1: ',llev1);
   writeln('введіть друге дерево: ');
```



```
   btr(head2);
   writeln;
   llev2 := 1;
   ob(head2,llev2);
   writeln('high2: ',llev2);
   if llev1 <> llev2 then
      begin
         writeln('дерева не ізоморфні');
         goto m1;
      end;
   while (pris) do
      begin
         new(bar1);
         new(q);
         bar1^.inf := q;
         q^.fc := '0';
         bar1^.lin := nil;
         lisph1 := bar1;
         new(bar2);
         new(p);
         bar2^.inf := p;
         p^.fc := '0';
         bar2^.lin := nil;
         lisph2 := bar2;
         blch(head1,bar1,lisph1,llev1);
         prl(lisph1,bar1);
         blch(head2,bar2,lisph2,llev1);
         prl(lisph2,bar2);
         if not (rescoml(lisph1,bar1,lisph2,bar2)) then
            begin
               writeln('дерева ізоморфні');
               goto m1;
            end
          else
            if llev1 = 1 then
               begin
                  writeln('дерева ізоморфні');
                  goto m1;
               end;
         fork(lisph1,bar1);
         fork(lisph2,bar2);
         llev1 := llev1 – 1;
      end;
   m1: ;
end.
```

Програма 2.13. Визначення ізоморфізму двох дерев

2.7. Елементи графіки в Паскалі

Для наглядної демонстрації результатів роботи програм іноді буде потреба використовувати графічні можливості Паскалю. Коротко зупинимось на їх застосуванні, наприклад, в Турбо Паскалі [51].

Стандартний запуск персонального комп'ютера з моменту ввімкнення і до моменту запуску програми з середовища Турбо Паскалю ініціює роботу дисплея в текстовому режимі. Тому будь-яка програма, що використовує графічні засоби комп'ютера, має певним чином ініціювати графічний режим роботи дисплейного адаптера.

У більшості пакетів графіки стандартна процедура INITGRAPH ініціює графічний режим роботи адаптера значеннями таких формальних параметрів:

Procedure InitGraph(var Driver,Mode: **Integer**; Path: **String**),

де Driver – змінна типу **Integer**, визначає тип графічного драйвера; Mode – змінна того ж типу, задає режим роботи графічного адаптера. Path – вираз типу **String**, який містить ім'я файла драйвера і, можливо, маршрут його пошуку.

На момент виклику процедури на одному з дискових носіїв інформації має бути файл, що містить потрібний графічний драйвер. Процедура завантажує цей драйвер до оперативної пам'яті і переводить адаптер до графічного режиму роботи. Тип драйвера має відповідати типу графічного адаптера. Для визначення типу драйвера в модулі Graph можна використати константу Detect = 0 (режим автовизначення типу).

Багато графічних процедур і функцій використовують покажчик поточної позиції на екрані, який на відміну від текстового курсора невидимий. Положення цього покажчика, як і взагалі будь-яка координата графічного екрана, задається відносно лівого верхнього кута, що має координати (0,0). У такий спосіб, горизонтальна координата екрана збільшується зліва направо, а вертикальна – збільшується зверху вниз.

Процедура PUTPIXEL виводить заданим кольором точку за вказаними координатами:

Procedure PutPixel(X,Y: **Integer**; Color: **Word**);

де X, Y – координати точки; Color – колір точки.

Процедура LINE накреслює лінію зі вказаними координатами початку і кінця:

Procedure Line(X1, Y1, X2, Y2: **Integer**);

де X1, Y1, X2, Y2 – координати початку (X1,Y1) і кінця (X2,Y2) лінії. Лінія накреслюється за встановленим стилем і кольором.

Процедура SETLINESTYLE встановлює новий стиль ліній:

Procedure SetLineStyle(Type, Pattern, Thick: **Word**);

де Type, Pattern, Thick – відповідно тип, зразок і товщина лінії. Тип лінії може бути заданий за допомогою однієї з констант:

SolidLn = 0; (Суцільна лінія)
DottedLn = 1; (Лінія з точок)
CenterLn = 2; (Штрих-пунктирна лінія)
DashedLn = 3; (Пунктирна лінія)
UserBitLn = 4; (Візерунок лінії визначає користувач).

Pattern враховують тільки для ліній, вигляд яких визначається користувачем (тобто у разі, коли Type = UserBitLn). Параметр Thick може набувати одне з двох значень:

NormWidth = 1; (Товщина в один піксель)

ThickWidth = 3; (Товщина в три пікселі).

Процедура SETCOLOR. Встановлює поточний колір для ліній та символів, що виводяться.

Procedure SetColor(Color: **Word**);

де Color – поточний колір.

Розглянемо на наступному прикладі застосування цих процедур.

Приклад 2.5. Задані натуральні числа X1, Y1, X2, Y2. Накреслити штрихову лінію між точками (X1,Y1) та (X2,Y2) такого вигляду: _ _ _ _ _ _ _ _ _ _ _ _ _.

Розв'язок задачі наведено в програмі 2.14.

Спосіб визначення координат точки, що лежить на відрізку з кінцями (X1,Y1) та (X2,Y2) і ділить відрізок у заданому відношенні, такий. Нехай дві точки задані своїми координатами (X1,Y1) та (X2,Y2). Пряму, що проходить через ці дві точки, можна описати наступними параметричними рівняннями:

$$X = X1 + (X2 - X1) * t \qquad Y = Y1 + (Y2 - Y1) * t.$$

При $0 < t < 1$ точка (X,Y) лежить всередині відрізка і ділить його у відношенні $t/(1-t)$, при $t = 0$ досягається кінець відрізка (X1, Y1), при $t = 1$ – кінець (X2,Y2). При $t > 1$ точка (X,Y) лежить на прямій поза відрізком з того ж боку від (X1,Y1), що й (X2,Y2); при $t < 0$ – з протилежного.

Суть алгоритму (програма 2.14) полягає в тому, що вираховують координати точки, яка ділить відрізок у відношенні 1/40 (тобто $t = 1/41$). На початок відрізка встановлюють проміжні координати X11 та Y11, вираховані координати будуть X22 та Y22. Між цими проміжними координатами малюють лінію. Вираховують різницю між X22 та X1 і між Y22 та Y1. Координати X11 та Y11 змінюються на X22 та Y22 відповідно. Різниці додають до X22 та Y22 відповідно і змінюють ці координати. Далі вже малюють невидиму лінію (для штрихів). Таким чином, по-різному вираховуючи, де встановлювати, видима лінія чи невидима, можна накреслити задану штрихову лінію.

```
Program Ex2_14;
  uses graph,crt;
  var grdriver,grmode,X1,X2,Y1,Y2,X11,X22,Y11,Y22,k,odx,ody: integer;
      t: real;
begin
  clrscr;
  grdriver := detect;
  writeln('enter X1,Y1');                    {Введення координат X1 та Y1 і X2 та Y2}
  readln(X1);
  readln(Y1);
  writeln('enter X2,Y2');
  readln(X2);
  readln(Y2);
  initgraph(grdriver,grmode,'e: \tp\tpu');   {Ініціалізація графічного режиму}
  setlinestyle(solidln,0,normwidth);         {Стиль ліній}
  X11 := X1; X22 := X2;                      {Початкові значення}
  Y11 := Y1; Y22 := Y2;                      {проміжних координат}
  t := 1/41;
  X22 := round(X1 + (X2 – X1) * t); Y22 := round(Y1 + (Y2 – Y1) * t);
  odx := X22 – X1; ody := Y22 – Y1;
  if X2 > X1 then
   while X22 <= X2 do
     begin
       k := k + 1;                {Змінна, яка визначає, видима чи невидима буде лінія}
       if k mod 2 <> 0 then
           setcolor(white)  {Якщо k – парне, лінія чорна (невидима), якщо k – непарне,}
         else setcolor(black);  {лінія біла (видима), тобто малюють штрихи}
       line(X11,Y11,X22,Y22);
       X11 := X22; Y11 := Y22;                {Зміна проміжних координат}
       X22 := X22 + odx; Y22 := Y22 + ody;
     end
   else
       while X22>= X2 do
         begin
           k := k + 1;
           if k mod 2 <> 0 then setcolor(white)
              else setcolor(black);
           line(X11,Y11,X22,Y22);
           X11 := X22; Y11 := Y22;
           X22 := X22 + odx; Y22 := Y22 + ody;
         end;
  repeat until keypressed;
  close graph;
end.
```

Програма 2.14. Застосування графічних можливостей

Задачі та вправи до розділу 2

Системи числення та арифметичні задачі

1. Нехай маємо число, задане у двійковій системі, довжина якого не перевищує 10000 двійкових розрядів. Визначте, чи ділиться воно на 15?
2. Нехай маємо число $a_n a_{n-1}...a_1 a_0$ у системі числення з основою k. Знайти остачу від ділення цього числа на число m. Числа k, n, m та остача від ділення задаються в десятковій системі числення.
3. У факторіальній системі числення довільне натуральне число N можна зобразити за допомогою деяких невід'ємних чисел $d[0],...,d[s]$, єдиним чином у вигляді $N = d[s](s+1)! + d[s-1]s! + ... + d[1]2! + d[0]$ за умови $0 \le d[i] \le i+1$, $i = 0, ..., s$, де $d[s] \neq 0$.

   Задано $s+1$ натуральне число $d[0],...,d[s]$, та натуральне $k, s < 200$, $d[i] < 65535$, $k < 65535$. Знайти остачу від ділення числа N на число k.
4. Числа Фіббоначі $U[1]$, $U[2]$, ... визначаються початковими значеннями $U[1] = 1$, $U[2] = 2$ та співвідношенням $U[N + 1] = U[N] + U[N–1]$.

   Розглянемо систему числення з двома цифрами 0 та 1, в якій, на відміну від двійкової системи, вагою є не степені двійки 1, 2, 4, 8, 16, ..., а числа Фіббоначі 1, 2, 3, 5, 8, 13, ... У цій системі числення кожне додатне ціле число єдиним чином задається у вигляді рядка з нулів та одиниць, який починається з 1 та в якому немає двох одиниць, що стоять поруч.

   Нехай є два рядки, що задають числа A та B. Знайти рядок, який зображує число $A + B$.

   Приклад. Вихідні рядки '10101' та '100' зображують числа 8 + 3 + 1 = 12 та 3. Відповіддю є рядок '100010', що зображує число 13 + 2 = 15 = 12 + 3.

   Зауваження. Рядки можуть бути настільки довгими, що числа A та B будуть більшими за максимальне у вашому комп'ютері ціле число.
5. Знайти кількість одиниць у двійковому запису числа i.
6. Послідовність 0112122012202001l2... будується так: спочатку записується 0, потім повторюється наступна дія: вже написану частину дописують справа, замінюючи 0 на 1, 1 на 2, 2 на 0, тобто 0 → 01 → 0112 → 01121220 → ... .

   Скласти алгоритм, який за введеним N, ($0 \le N \le 3\,000\,000\,000$) визначає, яке число стоїть на N-му місці в послідовності.
7. Задано масиви X (100) та Y (100). Написати алгоритм, який замінює послідовно місцями значення елементів $X(k)$ та $Y(k)$ для $k = 1, 2, ...,100$, не використовуючи проміжних змінних.
8. Точки з цілочисловими координатами з 1-го квадранта позначаються числами 0, 1, 2,... зліва направо та знизу вверх у такий спосіб, що черговій точці приписується мінімальне число, відсутнє по вертикалі та горизонталі, що проходять через точку. Першою відзначається точка (0,0).

   Написати програму, яка

   а) за заданими координатами x та y, $x \ge 0$, $y \ge 0$, x, y – цілі, визначатиме позначку точки;

   б) за заданою координатою x та позначкою точки y, $x \ge 0$, $y \ge 0$, x, y – цілі, визначатиме другу координату точки.

9. Знайти довжину періоду та сам період нескінченного дробу за основою P, який зображує раціональне число $\frac{N}{M}$ (для скінченних дробів вважати, що довжина періоду дорівнює 1). M, N, P – цілі десяткові числа, $0 < N < M$, $1 < P$.
10. Для введених дійсного числа $r > 0$ та натурального числа $q_{\max}$ слід знайти найкраще наближення r у вигляді раціонального дробу $\frac{p}{q}$, де $q \le q_{\max}$.
11. Визначимо множину $K[i]$ рекурентно. Нехай $K[0] = [0,1]$. Розділимо сегмент [0,1] на три частини точками $1/3$ і $2/3$ та видалимо з нього інтервал $(1/3, 2/3)$. Отримаємо множину $K[1]$, що складається із двох сегментів $[0, 1/3]$ та $[2/3, 1]$, які залишилися. Кожен з них розділимо на три частини (точками $1/9$ і $2/9$ для першого сегмента, і точками $7/9$ та $8/9$ – для другого) і видалимо середні інтервали $(1/9, 2/9)$ та $(7/9, 8/9)$. Так отримаємо множину $K[2]$, і так далі. Нехай ми побудуємо множину $K[i]$. Поділимо кожний сегмент, що залишився з $K[i]$, на 3 частини та видалимо з цих сегментів середні інтервали. У такий спосіб отримаємо з $K[i]$ множину $K[i + 1]$.

    Вводяться 3 цілих числа n, a, b. Слід визначити, чи належить точка з координатою a/b множині $K[n]$.
12. Число називають доконаним, якщо воно дорівнює сумі всіх своїх дільників, за винятком його самого. Довільне парне доконане число може бути задане у вигляді $2^{p-1} * (2^p - 1)$, де p – натуральне число. Знайти двійкове задання для максимального доконаного парного числа, меншого за введене N.
13. Нехай є N склянок, об'єми яких виражаються цілими числами літрів $V_1, ..., V_n$, порожня посудина та кран з водою. Чи можна за допомогою цих склянок налити в посудину рівно V літрів води?
14. Обчислити число e (основу натурального логарифма) з точністю до n десяткових цифр після коми.
15. За заданим числом n вивести на екран число 2^n, $n \le 10000$, n – натуральне.
16. За заданим числом N визначити кількість повторень кожної цифри 0,1,2,...,9 у числі N^N, $N \le 1000$.
17. Визначити, чи можна представити задане натуральне число у вигляді добутку чотирьох послідовних натуральних чисел. Довжина числа не перевищує 250 символів.
18. Знайти всі прості числа, не більші за введене число N.
19. Вводиться N. Слід знайти, скількома нулями закінчується число $N! = 1 * 2 * 3 * ... * N$.
20. На вхід програми подаються два числа N та P. Програма на виході має дати таке максимальне число M, що $N!$ ділиться на P^M, але не ділиться на P^{M+1}.
    Зауваження. Числа N і P великі, немає сенсу обчислювати значення $N!$. Числа N та P натуральні.
21. Натуральне число $N > 1$ представити у вигляді суми натуральних чисел так, щоб добуток цих доданків був максимальним.
22. Задається додатне дійсне число R. Знайти додатні дійсні $R_1, R_2, ..., R_n$, $R_i < 4$, $i = 1,...,n$, такі що $R = R_1 * R_2 * \cdots * R_n = R_1 + R_2 + ... + R_n$.
23. Задано цілі числа $A(0), A(1), ..., A(5)$. Знайти всі корені рівняння
$$A(5) * X^5 + A(4) * X^4 + ... + A(0) = 0,$$
якщо відомо, що всі корені – цілі числа, $A(0) \ne 0$.

24. Вивести за зростанням всі звичайні дроби, що не скорочуються і знаходяться в проміжку між 0 та 1, знаменники яких не більші за 15. Масиви не використовувати.
25. Поліном $P(x) = A[n] * x^n + A[n-1] * x^{n-1} + ... + A[1] * x + A[0]$ задається своїми коефіцієнтами $A[n]$, ..., $A[0]$. Знайти значення P в точці x.
26. Поліном N-го степеня $A(x) = \sum_{i=0}^{n} a_i * x^i$ задається своїми коефіцієнтами $a[i]$. Знайти коефіцієнти $b[i]$, $i = 0, ..., n * m$, m-го степеня полінома $A(x)$. Числа n, $m \le 40$.
27. Обчислити коефіцієнти $A[1]$, $A[2]$, ..., $A[n]$ многочлена
$$P(x) = x^n + A[1] * x^{n-1} + ... + A[n-1] * x + A[n]$$
із заданими дійсними коренями $X[1]$, $X[2]$, ..., $X[n]$.
28. Обчислити значення полінома $f(x) = ax^4 + bx^3 + cx^2 + dx + e$ для $x = 1, ...$ ..., 10000, використовуючи не більш ніж 51000 операцій $*$, $+$.

Змінні. Складність алгоритмів

29. Нехай є дві змінні a та b, значеннями яких є цілі числа. Написати програму, яка змінює місцями значення цих символів. Тобто значення змінної a присвоїти змінній b, а значення b – змінній a. Використовувати допоміжні змінні заборонено.
30. Нехай є натуральні числа a та n. Обчислити a^n. Не дозволяється змінювати значення змінних a та n. Часова складність алгоритму має дорівнювати $O(\log_2 n)$. Вважати, що $a^n < 2^{15}$.
31. Знайти добуток двох чисел a та b, якщо в програмі дозволяється використовувати тільки операції додавання одиниці, присвоєння та порівняння.
32. За натуральним n обчислити суму $1/0! + 1/1! + ... + 1/n!$. Складність алгоритму має дорівнювати $O(n)$.
33. Послідовність Фіббоначі визначається у такий спосіб:
$$a(0) = 1,\ a(1) = 1,\ a(k) = a(k-1) + a(k-2) \text{ при } k \ge 2.$$
    Нехай є n, обчислити $a(n)$. Вважати, що $a(n) < 2^{15}$. Складність алгоритму має дорівнювати $O(\log_2 n)$.
34. Написати програму, яка за заданим натуральним числом n, $n < 100$ друкує послідовність квадратів усіх натуральних чисел від 1 до n. Дозволяється використовувати тільки операції додавання та віднімання. Складність алгоритму має дорівнювати $O(n)$.
35. Обчислити функцію $f(x) = sqrt(x)$ (ціла частина від квадратного кореня). Використовувати змінні типу *real* забороняється.
36. За заданим n знайти довжину періоду десяткового дробу $1/n$.
37. Нехай є натуральні числа a та b. Знайти $d = \text{НСД}(a, b)$ та такі цілі числа x та y, що $d = a * x + b * y$.
38. Функція f задана у такий спосіб: $f(x) = x/2$, якщо x парне, $f(x) = 3 * x + 1$, якщо x непарне. Для заданого натурального n побудуємо послідовність $n_1, n_2, ..., n_k$, в якій $n = n_1$, $n_{i+1} = f(n_i)$, $1 < i < k$, $n_k = 1$, $n_{k-1} \ne 1$. Довжину цієї послідовності назвемо циклом числа n. Для заданих натуральних a та b, $a < b < 10000$ знайти число k з проміжку $[a, b]$ з максимальним циклом. Надрукувати число k та довжину його циклу.

39. Надрукувати всі послідовності довжини k, $k < 10$ з чисел $1..n$, n – натуральне.
40. Для заданого числа k надрукувати всі підмножини множини $\{1..k\}$.
41. Для заданого числа k надрукувати всі послідовності довжини k з чисел $1..k$, у яких i-й член не перевищує i, $1 \le i \le k$.
42. Надрукувати всі перестановки чисел $1..n$.
43. Надрукувати всі k-елементні підмножини множини $\{1..n\}$. Часова оцінка алгоритму повинна дорівнювати $O(n)$.
44. Надрукувати всі зростаючі послідовності довжини k з чисел $1..n$. Наприклад, при $n = 5$, $k = 2$ маємо: 12 13 14 15 23 24 25 34 35 45.
45. Надрукувати всі розділення натурального числа n на натуральні доданки. Розділення надрукувати у лексикографічному порядку (як неспадні послідовності). Наприклад, для $n = 4$ надрукувати 1 + 1 + 1 + 1, 1 + 1 + 2, 1 + 3, 2 + 2, 4.
46. Надрукувати всі послідовності довжини n з чисел $1..k$ за таким порядком, щоб кожна наступна послідовність відрізнялась від попередньої тільки в одній цифрі, у цьому разі не більше ніж на 1. Наприклад, при $n = 3$: 000 001 011 010 110 111 101 100.
47. Надрукувати всі перестановки чисел $1..n$ так, щоб кожна наступна відрізнялася від попередньої транспозицією двох сусідніх чисел. Наприклад, при $n = 3$: 321, 231, 213, 123, 132, 312.
48. Надрукувати всі послідовності довжини $2n$ з n нулів та n одиниць, в яких на довільному початковому відрізку сума цифр не є від'ємною.
49. Надрукувати всі розстановки дужок у добутку з n множників. Наприклад, при $n = 4$ таких розстановок 5: ((ab)c)d, (a(bc))d, (ab)(cd), a((bc)d), a(b(cd)).
50. Послідовніть з $2n$ цифр називають «щасливим» квитком, якщо сума перших n цифр дорівнює сумі останніх n цифр. Знайти кількість «щасливих» квитків заданої довжини n.

Сортування та послідовності

51. Нехай є n камінців масою A_1, A_2, ..., A_n. Необхідно розділити їх на дві купи так, щоб ваги куп відрізнялися не більше ніж у 2 рази. Якщо цього зробити не можна, то повідомити про це.
52. Нехай є дві цілочислові таблиці $A[1:10]$ та $B[1:15]$. Написати алгоритм та програму, яка перевіряє, чи є ці таблиці схожими. Дві таблиці називають схожими, якщо збігаються множини чисел, що трапляються в цих таблицях.
53. Нехай є словник. Знайти в ньому всі анаграми (слова, що складені з однакових літер).
54. На прямій пофарбували n відрізків. Відомі координата $L[i]$ лівого кінця відрізка та координата $R[i]$ правого кінця i-го відрізка для $i = 1$, ..., n. Знайти суму довжин всіх пофарбованих частин прямої.
55. Нехай є $2n$ чисел. Відомо, що їх можна розділити на пари так, що добутки чисел в парах рівні. Зробити розділення, якщо числа
    а) натуральні;
    б) цілі.
56. Нехай є числа $A_1, A_2, ..., A_n$ та $B_1, B_2, ..., B_n$. Скласти з них n пар (A_i,B_j) так, щоб сума добутків пар була максимальною (мінімальною). Кожне A_i та B_j в парах трапляються рівно один раз.
57. У музеї упродовж дня відбувається реєстрація приходу та виходу кожного відвідувача. Таким чином, за день отримані n пар значень, де перше значення в парі показує час приходу відвідувача та друге значення – час його виходу. Знайти проміжок часу, упродовж якого в музеї одночасно перебувало максимальне число відвідувачів.
58. Впорядкувати за спадним порядком 5 чисел за 7 операцій порівняння.
59. Нехай є число $n > 1$ – вимірність простору та розміри M n-вимірних паралелепіпедів $(a_{i1}, ..., a_{in})$, $i = 1, ..., M$. Паралелепіпед може розташовуватися в просторі довільно, але його ребра паралельні осям координат. Знайти максимальну послідовність вкладених один в один паралелепіпедів.
60. Нехай є цілі M, N та вектор дійсних чисел $X[1..N]$. Знайти ціле число $i(1 \le i \le N - M)$, для якого сума $x[i] + ... + x[i + M]$ найближча до нуля.
61. Нехай є два відсортованих за спадним порядком масиви $A[1,N]$ та $B[1,M]$. Отримати відсортований за спадним порядком масив $C[1,N + M]$, що складається з елементів масивів A та B (*«злити»* масиви A та B).
62. Нехай є масив $X[1..N]$. Слід циклічно зсунути його на k елементів управо (тобто елемент $X[i]$ після зсуву має стояти на місці $X[i + k]$; тут ми вважаємо, що за $X[N]$ йде $X[1]$). Вводити допоміжний масив заборонено.
63. Побудувати максимально можливу множину, яка складається з попарно незрівнюваних векторів **v**. Вектори **v** визначаються парами чисел, які вибирають із заданої послідовності чисел a_1, ..., a_n, $n \ge 1$. Два вектори $\mathbf{v} = (a,b)$ та $\mathbf{v}' = (a',b')$ називають зрівнюваними, якщо $a \le a'$ та $b \le b'$ або $a \ge a'$ та $b \ge b'$.

Списки, дерева, графи

64. Послідовність цілих чисел a_1, a_2, ..., a_n задана зв'язним списком та покажчиком на його перший елемент. Написати функцію, яка обертає список.
65. На вході задана послідовність цілих додатних чисел. Написати програму для запам'ятовування послідовності у вигляді зв'язного списку L та її впорядкування.
66. Написати функцію для копіювання черги, реалізованої у вигляді циклічного списку.
67. Відношенням r на множині M називають будь-яку підмножину декартового добутку $M \times M$, тобто $r \subseteq M \times M$. Відношення r називають антирефлексивним, якщо для кожного елементу $a \in M$ $(a,a) \notin r$; транзитивним, якщо для будь-яких елементів a,b,c множини M зі співвідношень $(a,b) \in r$, $(b,c) \in r$ випливає $(a,c) \in r$. Відношення, що одночасно антирефлексивне і транзитивне, називають відношенням часткового порядку. Для відношення часткового порядку r на множині M відношення $\gamma \subseteq r$ утворює базис r, якщо для кожного елементу $(x,y) \in r$ існують елементи $x_0,x_1,...,x_n \in M$, такі що $x_0 = x$, $x_n = y$ і $(x_{i-1},x_i) \in \gamma$, $1 \le i \le n$.

    Припустимо, що множина M скінченна. Лінійний список $F = \langle a_0, a_1, \ldots, a_n \rangle$ всіх елементів M без повторень називають результатом топологічного сортування множини M.

    Розглянемо одне з можливих формулювань задачі топологічного сортування. Множиною M є множина цілих чисел від 0 до $n - 1$. Базис відношення часткового порядку γ на M задається списком пар $\langle i_0, j_0 \rangle, \langle i_1, j_1 \rangle, ..., \langle i_m, j_m \rangle$ на вході. Скласти програму для введення пар, що задають відношення γ, і друку результату топологічного сортування на множині M.
68. Написати функцію для визначення кількості листків дерева степеня 3, заданого в розширеній стандартній формі.

69. Написати функцію для створення копії дерева степеня 3, заданого в розширеній стандартній формі.
70. Написати програму для введення дерева в дужковому поданні порядку 3 (вузли – цілі числа) та організації його зберігання в стандартній формі.
71. Написати функцію для визначення висоти бінарного дерева, яке подане в стандартній формі і кожен вузол якого містить один символ.
72. Написати функцію, яка за значеннями змінних x, y, z обчислює значення виразу, що містить цілі числа, змінні x, y, z та операції +, –, *,/. Вираз зображено бінарним деревом.
73. Написати функцію перевірки, чи є орієнтований граф зв'язним.
74. Написати функцію перевірки, чи є орієнтований граф ациклічним.
75. Нехай елементами таблиці хешування будуть послідовності із букв, і використовується така функція розстановки для таблиці розміру $m = 5$: просумувати «*значення*» букв, де A має значення 1, B – значення 2 і т. д.; розділити суму на 5 і взяти остачу. Написати програму вставки чергового елементу в таблицю.
76. Написати функцію для переведення множини n зовнішніх імен, які належать інтервалу 1 і r, в множину внутрішніх імен. Внутрішні імена – це цілі числа від 1 до n. Функція має забезпечувати перехід у дві сторони в посиланні, що r значно більше за n.
77. Знайдіть ефективну структуру даних для задання підмножини S цілих чисел, які знаходяться між 1 і n. На множині ми повинні виконувати наступні операції:
    а) знаходити і видаляти з неї один із елементів;
    б) додавати до неї ціле число j.
78. Написати функцію для реалізації операції *min* типу *min*(i), яка знаходить всі цілі числа, менші за i, вставлені до неї в множину, і які ще не були знайдені попередньою операцією *min*.
79. Розробіть алгоритм для вставки нового вузла в 2–3-дерево в посиланні, що множина листів впорядкована. Проведіть повний аналіз запропонованого алгоритму та напишіть його програмну реалізацію.
80. *АВЛ*-деревом називають таке бінарне дерево пошуку, в якому для кожного вузла **v** висота його лівого і правого піддерев відрізняється не більше як на одиницю. Якщо якогось піддерева немає, його висоту вважають рівною 1.

    Нехай T – це *АВЛ*-дерево. Напишіть програмну реалізацію алгоритму складності $O(\log_2 n)$ для операцій *вставити* і *видалити*, які зберігають властивість *АВЛ*-дерева.

Розділ 3

АЛГОРИТМИ ПОШУКУ НА ГРАФАХ

3.1. Простір задач і простір станів

Багато які прикладні задачі можна розв'язувати як задачі планування цілеспрямованих дій. Останні прийнято розподіляти на два класи [13, 31]: *планування у просторі станів (SS-проблема) і планування у просторі задач (PR-проблема).*

У разі планування у просторі станів заданим вважають деякий набір станів (ситуацій). Опис кожної ситуації складається з опису стану як зовнішнього світу, так і самої системи. Відомі дії, які може здійснювати система і які визначають перехід з одного стану до іншого. Проблема полягає у пошуку шляху від початкового стану до одного з остаточних. У такій постановці задачу планування цілеспрямованих дій можна уявляти собі як задачу пошуку шляху з однієї до іншої вершини в графі. Відповідно, після того як зведення задачі до формальної моделі здійснено, можна використовувати відомі алгоритми пошуку шляхів на графах (алгоритми Мура, Дейкстри, гілок та обмежень тощо).

Графом станів задачі називають орієнтований граф, вершини якого відповідають можливим станам предметної області, а дуги – методам переходу від стану до стану.

Дуги можуть мати позначки, які інтерпретуються як вартість або довжина відповідного переходу.

Тоді розв'язання задачі являє собою пошук шляху від початкового стану до цільового; у цьому разі типовою є вимога оптимізації цього рішення, тобто пошуку найкоротшого шляху.

Класичним прикладом планування у просторі станів можна вважати задачу пошуку мінімального шляху від одного міста до іншого. Можна навести й інші, менш очевидні приклади.

Планування у просторі задач передбачає декомпозицію вихідної задачі на підзадачі, поки не буде досягнута задача, для якої є готовий алгоритм розв'язання. Таку декомпозицію зручно уявляти собі у вигляді дерева типу *І/АБО*. Дійсно, для розв'язання більшості задач слід у жорсткій послідовності здійснювати такі кроки: *аналіз відмінностей між початковою та бажаною ситуаціями, з'ясування необхідних заходів для усунення цих відмінностей; генерування нових підзадач*. Якщо позначити початковий стан через A, мету – через B, а засіб досягнення мети – через C, то декомпозицію здійснюють за схемою $\langle A,B\rangle \to \langle A,C\rangle$, $\langle C,B\rangle$.

Називатимемо задачу *елементарною*, якщо для неї існує явний алгоритм розв'язання. Тоді метою декомпозиції будь-якої задачі є її зведення до сукупності елементарних задач.

І-АБО графом називають орієнтований граф, вершини якого відповідають задачам, а дуги – відношенням між задачами. Між дугами вводять відношення І/АБО.

Нехай задача А зв'язана дугами із задачами В та С (під час планування рішень це означає, що задачу А можна звести до задач В та С). Скажемо, що дуга АВ зв'язана з дугою АС відношенням І, якщо для розв'язання А необхідно розв'язати і В, і С. Якщо ж для розв'язання А достатньо розв'язати лише одну з цих задач, то будемо говорити, що відповідні дуги зв'язані відношенням АБО.

Будь-який *І-АБО* граф можна звести до певної нормальної форми, у якій з будь-якої вершини виходять або тільки *І*-дуги, або тільки *АБО*-дуги.

Вершину, з якої виходять лише І-дуги, називають І-вершиною.

Вершину, з якої виходять лише АБО-дуги, називають АБО-вершиною.

Одну з вершин графа, що відповідає початковій задачі, називають *початковою* вершиною.

Вершину називають розв'язною, якщо задача, що відповідає цій вершині, має розв'язок.

Метою пошуку, що здійснюють в *І-АБО*-графі, є встановлення того, чи є початкова вершина розв'язною, чи ні. Взагалі кажучи, можна дати такі рекурсивні визначення розв'язної вершини:

1) заключна вершина розв'язна, оскільки вона відповідає елементарній задачі;
2) незаключна *І*-вершина розв'язна, якщо розв'язними є всі вершини-нащадки;
3) незаключна *АБО*-вершина розв'язна, якщо розв'язною є хоча б одна з вершин-нащадків.

Сформулюйте самостійно аналогічне визначення нерозв'язної вершини.

Графом розв'язання називають підграф із розв'язних вершин, який показує, що початкова вершина є розв'язною і є І-АБО-деревом.

Це дерево *Т* визначають так:

- коренем дерева є початкова вершина *Р*;
- якщо *Р* є *АБО*-вершиною, то *Т* містить будь-який з її розв'язних нащадків з *І-АБО*-графа разом із власним деревом розв'язання;
- якщо *Р* є *І*-вершиною, то *Т* містить розв'язки всіх її нащадків з *І-АБО*-графа разом із власним деревом розв'язання.

3.2. Алгоритми пошуку в ширину і глибину

Як зазначалося, розв'язання багатьох задач пов'язане з діями над графами: орієнтованими і неорієнтованими, циклічними і ациклічними, навантаженими і ненавантаженими. Часто такі маніпуляції вимагають від дослідника визначення вершини (вузла) або множини вершин, ребра або множини ребер, які задовольняють якійсь певній умові. Наприклад, ними можуть бути такі: знаходження множини ребер графа, значення ваги яких менше за *X*; знаходження остового дерева графа, остового дерева мінімальної вартості, компонентів двозв'язності графа тощо.

Визначити такі множини можна лише шляхом систематичної перевірки всіх вершин або ребер заданого графа. Фактично задача зводиться до пошуку їх у графі. Тому важливою компонентою багатьох алгоритмів на графах є алгоритм обходу графа.

Технічні прийоми, які ми розглянемо, можна розподілити на дві категорії. До першої належать заходи реалізації класичних алгоритмів пошуку на графах методом *в глибину* або *в ширину*, до другої – заходи реалізації класичних алгоритмів на графах, в основі яких лежать обходи графа.

Процедура пошуку в ширину передбачає аналіз на кожному кроці вершин-сусідів усіх вершин, що проаналізовано спочатку (в інтерпретації прийняття рішень це означає *паралельну* перевірку всіх можливих альтернатив), пошуку в глибину – першочерговий аналіз нащадків вершин, що проаналізовано останніми, тобто всі альтернативи аналізують *послідовно*, одна за одною; аналіз деякої альтернативи завершують лише тоді, коли є змога остаточно встановити, приводить вона до успіху чи ні. Якщо ж альтернатива призводить до невдачі, відбувається *повернення* і розгляд інших альтернатив.

І пошук в ширину, і пошук в глибину за досить загальних умов мають, узагалі беручи, експоненційну оцінку часової складності. Можна навести низку прикладів, коли перебір в ширину дає змогу виграти час порівняно з перебором в глибину і навпаки. Як правило, пошук в глибину дозволяє зекономити *пам'ять*, оскільки за його реалізації немає необхідності запам'ятовувати все дерево, достатньо зберігати в пам'яті лише вершини, що мають відношення до поточної альтернативи.

На практиці процедура пошуку в глибину набула значно більшого поширення. Можна стверджувати, що перебір з поверненням (бектрекінг) став класичною загальноінтелектуальною процедурою, яку покладено в основу сучасних методик планування цілеспрямованих дій, програмування ігор, автоматизованого доведення теорем тощо.

Враховуючи широке застосування процедур пошуку, розглянемо їх практичну реалізацію, використовуючи нотацію Паскалю. Спочатку слід задати граф, на якому працюватимуть процедури пошуку. Граф можна задати, користуючись матрицею суміжності. Реалізацію матриці суміжності за допомогою масиву описує процедура **InitalizeGraph.** Вважаємо, що головна програма містить такий опис:

```
...
const u = 1000;                          {Максимальна кількість вершин у графі}
type arr2 = array [1..n,1..n] of integer;
...
procedure InitalizeGraph (var B: arr2);
  var  m: 1..n;                          {Кількість вершин у графі}
       i, j: 1..n;                       {Номери рядків і стовпців матриці}
begin
    writeln('Введіть кількість вершин у графі m');
    read(m);
    for i := 1 to m do
      for j := 1 to m do
         begin
            writeln('Введіть значення елементу матриці B[', i, ', ', j, '] = ');
            read(B[i, j]) ;
         end
end;
```

Під час запиту на введення значення $B[i,j]$ слід ввести 1, якщо в графі задання існує ребро, що йде із вершини з номером i у вершину з номером j, та 0 в іншому разі.

Задання графа за допомогою матриці суміжності виправдане, коли в алгоритмі пошуку превалює умова, яка обґрунтована питанням «*чи існує ребро між вершинами i та j?*»

Кращим варіантом задання графа для алгоритмів, які б мали часову оцінку роботи, меншу за $O(n^2)$, є використання списків суміжності. У цьому разі важливим є факт орієнтовності графа. За неорієнтованого графа до списку суміжності вершини заносять всі суміжні з нею вершини. Коли ж граф орієнтований, то суміжні вершини бажано розбивати на дві групи або два списки: попередників і наступників вершини. Якщо ми маємо ребро $\langle u,v\rangle$, таке що $u \to v$, тоді вершина u є вершиною-попередником вершини v, а вершина v є вершиною-наступником вершини u.

Вважатимемо, що в головній програмі присутній наступний опис, який дає змогу реалізувати такий список суміжності.

```
type
   CoreLink = ^CoreElementType;          {Покажчик на CoreElement}
   BindLink = ^BindElementType;          {Покажчик на BindElement}
   CoreElementType = record
                        Index: integer;      {Порядковий номер}
                        Value: integer;      {Поле для визначення відвідування}
                                             {вершин}
                        Prev: CoreLink;      {Покажчик на наступну вершину}
                        Next: CoreLink;      {Покажчик на попередню вершину}
                        Incomings: BindLink; {Покажчик на список суміжних}
                                             {вершин (ребра входять)}
                        Outcomings: BindLink;{Покажчик на список вершин}
                                             {(ребра виходять)}
                     end;
   BindElementType = record              {Опис елементу списку суміжності}
                        Value: integer;      {Порядковий номер}
                        Prev: BindLink;      {Покажчик на попередній елемент}
                                             {у списку}
                        Next: BindLink;      {Покажчик на наступний елемент}
                                             {у списку}
                     end;
var
   KnotNumber: integer;                  {Кількість вузлів у графі}
   OrientationAnswer: char;              {Орієнтовність графа 'y' or 'n'}
   Root: CoreLink;                       {Покажчик на початковий елемент графа}
```

Розглянемо два графи, зображені на рис. 3.1.

Тоді результатом задання будуть відповідно списки, наведені на рис. 3.2.

Якщо описати дві змінні:

```
var  CoreElement: CoreElementType;
     BindElement: BindElementType;
```

покажчик CoreElement↑.Next вказує на список вузлів-наступників вузла CoreElement↑.Index; CoreElement↑.Next – на наступний вузол графа; CoreElement↑.Prev – на попередній елемент головного списку; CoreElement↑.Incomings – на список суміжних вершин, ребра яких входять до вузла; CoreElement↑.Outcomings – на список суміжних вершин, ребра яких виходять з вузла; BindElement↑.Next – на наступний вузол списку суміжних вузлів; BindElement↑.Prev – на попередній.

Як іменами вершин графа скористаємося натуральними числами.

Опісля можна запропонувати процедуру InitializeGraph задання в пам'яті ЕОМ графа (неорієнтованого і орієнтованого) за допомогою динамічних змінних.

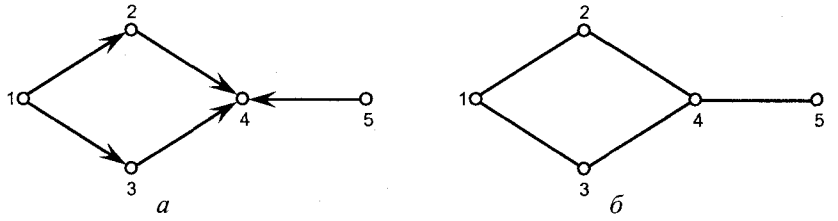
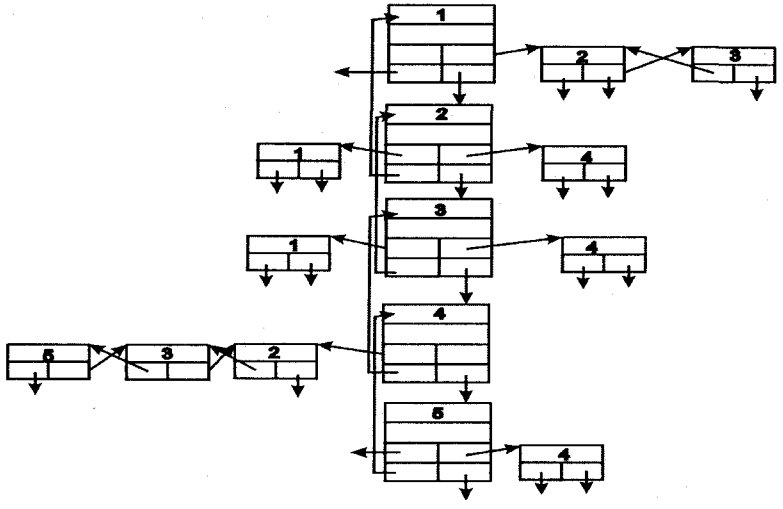
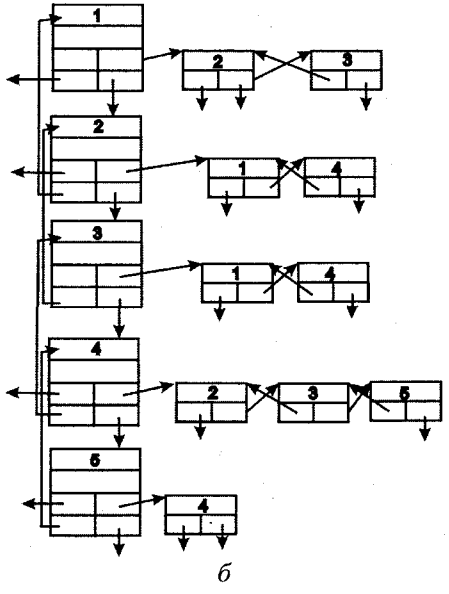


Рис. 3.1. Орієнтований (*а*) і неорієнтований (*б*) графи



а



б

Рис. 3.2. Списки задання орієнтованого (*а*) і неорієнтованого (*б*) графів

```
procedure InitializeGraph;                    {Головна процедура задання}
  var NewCoreElement, CurrentCoreElement: CoreLink;
begin
    CurrentCoreElement := Root;               {Root вже проініціалізований заздалегідь}
    repeat
      New(NewCoreElement);
```



```
      CurrentCoreElement^.Next := NewCoreElement;
      BuildBinds(CurrentCoreElement);         {Побудова списку суміжних}
      with NewCoreElement^ do                 {вершин для вузла}
        begin
          Index := CurrentCoreElement^.Index + 1;
          Value := 0;
          Prev := CurrentCoreElement;
        end;
      CurrentCoreElement := NewCoreElement;
    until CurrentCoreElement^.Index = KnotNumber;   {KnotNumber – кількість}
                                              {вершин у графі}
    CurrentCoreElement^.Next := nil;
    BuildBinds (CurrentCoreElement) ;   {Побудова списку суміжних вершин для вузла}
end;
```

Зосередимо вашу увагу на процедурі BuildBinds, яка створює списки суміжних з вузлом вершин і в якій безпосередньо відбувається розгалуження побудови орієнтовних і неорієнтовних графів. Процедуру BuildBinds викликає процедура InitializeGraph для побудови послідовностей Binds для кожного з вузлів.

```
procedure BuildBinds(Nucleus: CoreLink);
  var BindAnswer: char;
  procedure BuildBindChain (ChainType: char; var BindRoot: BindLink);
  begin
          {Тіло цієї процедури пропущено і описано нижче}
  end;

begin
    if OrientationAnswer = 'n'          {Глобальна змінна, отримує значення}
                                        {в головній програмі}
      then                              {Побудова списку для неорієнтовного графа}
        begin
          clrscr;
          writeln('Does not ', Nucleus^.Index, ' have ajacents?');
          repeat
            BindAnswer := ReadKey;
          until BindAnswer in ['y', 'n'];
          if BindAnswer = 'y' then
            begin
              BindAnswer := 'a';
              BuildBindChain (BindAnswer, Nucleus^.OutComings) ;
            end
            else
              Nucleus^.OutComings := nil;
```

```
      Nucleus^.Incomings := nil;
    end
  else                                  {Побудова списку для орієнтовного графа}
    begin
      clrscr;
      writeln('Does knot ', Nucleus^.Index, ' have incomings?');
      repeat
        BindAnswer := ReadKey;
      until BindAnswer in ['y', 'n'];
      if BindAnswer = 'y' then
        begin
          BindAnswer := 'i';
          BuildBindChain(BindAnswer, Nucleus^.Incomings);    {Побудова списку}
                                                             {попередників}
        end
        else
          Nucleus^.Incomings := nil;
      writeln('Does knot ', Nucleus^.Index, ' have outcomings?') ;
      repeat
        BindAnswer := ReadKey;
      until BindAnswer in ['y', 'n'];
      if BindAnswer = 'y' then
        begin
          BindAnswer := 'o';
          BuildBindChain (BindAnswer, Nucleus^.Outcomings);  {Побудова списку}
                                                             {наступників}
        end
        else
          Nucleus^.Outcomings := nil;
    end;
end;
```

Змінну BindAnswer вводять виключно заради інтерфейсу, і вона не виконує будь-якого функціонального навантаження. Підпрограма має два відділи, які працюють залежно від орієнтованості графа (if OrientationAnswer = 'n' then … else …):

- якщо граф неорієнтований, то всі суміжні елементи вважають вихідними, а послідовність вхідних елементів не будують;
- якщо граф орієнтований, то будують окремі послідовності для вхідних та вихідних вершин, хоча дані про вхідні вершини не використовують і вводять лише для можливості використання їх іншими алгоритмами обробки графа, що побудований за даною схемою.

Організація інтерфейсу для опитування користувача щодо наявності суміжних вершин є теж надлишковою та існує для можливості побудови незв'язних графів.

Така процедура викликає процедуру BuildBindChain, що описана в тілі процедури BuildBinds, для побудови послідовності BindChain, що починається з елементу, на який посилається конкретний покажчик Incomings або Outcomings елементу ядра.

Процедура BuildBindChain будує зв'язану послідовність за принципом побудови послідовності Core процедурою InitializeGraph.

```
Procedure BuildBindChain(ChainType: char; var BindRoot: BindLink);
  var
    i: integer;
    NewBindElement, CurrentBindElement: BindLink;
begin
    new(BindRoot);
    BindRoot^.Prev := nil;
    write('Enter indexes of knots ');
    case ChainType of
      'i': write('incoming into ');
      'o': write('outcoming from ');
      'a': write('ajacent to ');
    end;
    writeln('knot ', Nucleus^.Index, ' separating them with "Returns".');
    writeln('Enter 0 upon completion or if there are no any.');
    CurrentBindElement := BindRoot;
    repeat
      readln(i);
      CurrentBindElement^.Value := i;
      new(NewBindElement);
      CurrentBindElement^.Next := NewBindElement;
      NewBindElement^.Prev := CurrentBindElement;
      CurrentBindElement := NewBindElement;
    until i = 0;
    CurrentBindElement^.Prev^.Next := nil;
    dispose(CurrentBindElement);
end;
```

Змінну BindAnswer, передану батьківською процедурою, використовують для створення найпростішого розгалуження в інтерфейсі.

3.2.1. *Пошук в глибину*

Розглянемо детальніше реалізацію методу пошуку в глибину в неорієнтованому графі G.

Уточнимо загальну ідею методу. Процес перевірки деякої умови у вершині графа засоціюємо з процесом відвідування вершини. Відвідаємо вершину v_1. Потім виберемо вершину u із околу вершини v_1 (суміжну з v_1), яку не було ще відвідано, і повторимо процес знову. Нехай ми відвідали вершину v. Якщо існує нова, ще не відвідана суміжна вершина u з околу v,

тоді відвідаємо її і відзначимо, що вона перестала бути новою. Процес пошуку далі продовжують з вершини u. Якщо ж для v не існує жодної нової суміжної вершини, тоді відзначимо, що вершина v уже використана, і повернемось у вершину, з якої ми потрапили у v, і продовжуємо процес пошуку. Якщо вершина v була вершиною v_1, то процес пошуку закінчується.

Нехай процес відвідування вершини полягає в друкуванні інформаційної частини вузла задання графа – друку номера вершини, тоді алгоритм пошуку в глибину реалізовуватиме рекурсивна процедура DepthSearch.

```
procedure DepthSearch(TempLink: CoreLink);
  var CurrentBind: BindLink;
begin
    if TempLink^.Value = 1 then exit;
    write(TempLink^.Index, ' ');
    TempLink^.Value := 1;
    if TempLink^.Outcomings <> nil then
     begin
      CurrentBind := TempLink^.Outcomings;
      while CurrentBind^.Next <> nil do
        begin
          DepthSearch(PointFinder(CurrentBind^.Value));
          CurrentBind := CurrentBind^.Next;
        end;
      DepthSearch(PointFinder(CurrentBind^.Value));
     end;
    writeln;
end;
```

Необхідно описати незалежну функцію PointFinder, яка знаходить серед вузлів списку суміжності задання графа вузол з потрібним номером в інформаційній частині.

```
function PointFinder(n: integer): CoreLink;          {Знаходить вузол з номером n}
  var Current: CoreLink;
begin
    Current := Root;
    while Current^.Index <> n do
      Current := Current^.Next;
    PointFinder := Current;
end;
```

Очевидно, що для незв'язного графа слід реалізувати виклик процедури DepthSearch $v \in V$ з перевіркою, чи було вершину v відвідано заздалегідь.

Методика пошуку процедури DepthSearch така сама і для орієнтованого графа. У цьому разі суміжними для вершини v вершинами вважаються вершини, що виходять із неї.

Теорема 3.1. Для довільного графа $G(V,E)$, заданого списками суміжності, процедура DepthSearch коректна і потребує часу $O(n + m)$.

Доведення. Алгоритм починає пошук послідовно, від кожної ще не відвіданої вершини. Після відвідування вершини її позначають як відвідану. Отже, кожну вершину графа обов'язково відвідають не більше одного разу.

Для визначеності вважатимемо, що час, потрібний на відвідування вершини, має константний характер. Враховуючи, що для вибору чергової вершини відвідування слід переглянути всі суміжні вершини попередньої вершини відвідування (максимальну кількість суміжних елементів визначають як m, а кількість усіх вершин у графі – n), часова складність становитиме $O(n + m)$.

Наслідок. Зв'язні компоненти довільного графа G, заданого списками суміжності, можна обчислити за час $O(n + m)$.

Дійсно, якщо G – зв'язний неорієнтований граф, тоді всі вершини G відвідають у разі першого виклику процедури DepthSearch.

В іншому разі необхідно організовувати наступні виклики цієї процедури з відповідним запам'ятовуванням вершин, які відвідували за один виклик. Для цього DepthSearch можна модифікувати у такий спосіб, щоб всі нові вершини відвідування запам'ятовувались в окремому списку.

3.2.2. *Пошук в ширину*

Реалізацію пошуку в ширину можна отримати із алгоритму пошуку в глибину заміною стека чергою. Після відвідування вершини простежуються всі її суміжні вершини. Розглянемо процедуру ініціалізації черги, занесення в чергу та видалення елементу.

```
procedure InitializeQueue;                    {Створює один елемент у черзі}
begin
    new(QueueTail);
    QueueHead := QueueTail;
    QueueTail^.Next := nil;
end;

procedure AddToQueue(Val: integer);           {Додає вузол з номером Val у чергу}
begin
    new(QueueElement);
    QueueElement^.Value := Val;
    QueueTail^.Next := QueueElement;
    QueueTail := QueueElement;
    QueueTail^.Next := nil;
end;
```

```
procedure CutQueueHead;                    {Видаляє елемент з черги}
begin
   QueueElement := QueueHead^.Next;
   dispose(QueueHead);
   QueueHead := QueueElement;
end;

procedure DisplayAjacents(Bind: BindLink);   {Додає в чергу всі суміжні з вузлом}
                                             {вершини}
begin
   while Bind^.next <> nil do
     begin
       AddToQueue(Bind^.Value);
       Bind := Bind^.Next;
     end;
   AddToQueue(Bind^.Value);
end;
```

Для процедури WidthSearch можна навести аналогічні роздуми щодо часової оцінки, як і у разі пошуку в глибину.

```
procedure WidthSearch;
  var
    TempLink: CoreLink;
    CurrentQueue: QueueLink;
begin
  InitializeQueue;
  QueueTail^.Value := Root^.Index;
  repeat
    TempLink := PointFinder (QueueHead^.Value) ; {Шукає в головному списку вузол}
                                                 {з номером QueueHead^.Value}
    if TempLink^.Value = 1 then
       CutQueueHead
      else
        begin
          if TempLink^.Outcomings <> nil
            then DisplayAjacents (TempLink^.Outcomings) ;  {Додає в чергу список}
                          {суміжних вершин, починаючи з TempLink^.Outcomings}
          write (TempLink^.Index, ' ') ;
          TempLink^.Value := 1;
          CutQueueHead;
        end;
  until QueueHead = nil;
  writeln;
end;
```


Зрозуміло, що обидва зазначені типи обходу графа можна використовувати за певної модифікації для знаходження шляху між фіксованими вершинами u і v. Зрозуміло також, що процедуру WidthSearch можна використовувати і для пошуку в ширину в орієнтованому графі.

3.3. Застосування алгоритмів пошуку

3.3.1. *Остові дерева*

Розглянемо зв'язний неорієнтований граф $G = (V, E)$. Довільне дерево (V, T), де $T \subseteq E$ називають остовим деревом графа G. Зафіксуємо остове дерево (V, T). Тоді ребра такого дерева називають гілками, а інші ребра графа – хордами.

Процедури пошуку в глибину і в ширину можна використовувати для побудови остових дерев з такою модифікацією: за досягнення нової вершини u із v слід внести ребро $\{v, u\}$ у множину ребер T. Крім того, у разі застосування процедури пошуку в ширину за умови, що ребрам графа приписано довжини, що дорівнюють 1, можна довести, що шлях у (V, T) із довільної вершини v до кореня дерева r є найкоротшим шляхом із вершини v у вершину r у графі G.

Програма 3.1 будує остове дерево графа, який задають матрицею суміжності, застосовуючи алгоритм пошуку в глибину.

```
Program Ex3_1;
 uses crt;
 const  n = 6;                                                    {n <= 254}
        A: matr = ((0, 16, 0, 0, 19, 21),
                   (16, 0, 5, 6, 0, 11),
                   (0, 5, 0, 10, 0, 0),
                   (0, 6, 10, 0, 18, 14),
                   (19, 0, 0, 18, 0, 33),
                   (21, 11, 0, 14, 33, 0));
 type matr = array [1..n, 1..n] of byte;
      st = array [1..n] of byte;
 var  R: st;
      candidate: byte;
      tmp: byte;
      j: byte;
      B: matr;
procedure Ost(var A: matr; start: byte; R: st) ;
 var  Stack: st;
      c: boolean;
      candidate, top, current: byte;
      i: byte;
```

```
begin                                              {Ініціалізація масиву}
  for candidate := 1 to n do
    R[candidate] := 0;
  for i := 1 to n do
    for tmp := 1 to n do
      B[i, tmp] := 0;
  top := 1;                                        {top – покажчик на вершину стека}
  Stack[top] := start;                             {Заносимо початковий вузол до стека}
  R[start] := 255;                                 {Відзначаємо, що вузол обробляється}
  candidate := 0;
  repeat
    current := Stack[top];                         {Вибір вузла із стека}
    c := false;
    repeat
      candidate := candidate + 1;
      if candidate > n then c := true
        else
          if ((R[candidate] = 0) and (A[current,candidate] > 0)) then c := true
    until c;
    if candidate > n then                          {Крок назад}
      begin
        candidate := current;
        top := top-1;
      end
      else                                         {Крок вперед}
        begin
          top := top + 1;
          Stack[top] := candidate;                 {Заносимо в стек нового кандидата}
          R[candidate] := current;                 {Відзначаємо обробку кандидата}
          B[current,candidate] := A[current,candidate];   {і запам'ятовуємо}
          B[candidate,current] := A[current,candidate];   {його батька}
          candidate := 0;
        end;
  until top = 0;
end;

BEGIN
  Ost(A, 1, R);                                    {Пошук остового дерева з вершини 5}
  for j := 1 to n do
    begin
      for tmp := 1 to n do
      write(B[j, tmp], ', ');
      writeln;
    end;
  write('Press "ANY KEY" to continue . . .');
  repeat until KeyPressed;
END.
```

Програма 3.1. Знаходження остового дерева

3.3.2. *Фундаментальна множина циклів графа*

Розглянемо граф $G=(V,E)$ і його остове дерево (V,T). Якщо до остового дерева додати хорду $e \in E \setminus T$, тоді отримаємо рівно один елементарний цикл у новоутвореному графі. Позначимо його через C_e. Множину $\Re = \{C_e : e \in E \setminus T\}$ називають фундаментальною множиною циклів графа G відносно остового дерева (V,T), тобто кожен цикл графа G можна деяким природним способом отримати із множини $\Re$.

Симетричною різницею множин A і B (позначаємо $A \oplus B$) називають множину $A \oplus B = (A \cup B) \setminus (A \cap B)$.

Можна довести [66], що симетрична різниця множин $A_1, A_2, \ldots, A_k$ містить, незалежно від розміщення дужок, рівно ті елементи, які виникають у непарному числі множин A_i. Отже, в симетричній різниці множин $A_1, A_2, \ldots, A_k$ ми можемо не вживати дужки.

Множину C ребер графа називають псевдоциклом (пуста множина і довільний цикл графа), якщо кожна вершина графа (V,C) має парний степінь. Симетрична різниця довільного числа псевдоциклів є псевдоциклом. Відома також теорема [66]:

> ***Теорема 3.2.*** Нехай $G=(V,E)$ – зв'язний неорієнтований граф, (V,T) – його остове дерево. Довільний цикл графа G можна однозначно задати як симетричну різницю деякого числа фундаментальних циклів. У загальному випадку довільний псевдоцикл C графа G можна однозначно виразити як $C = \oplus C_e$, де $e = C \setminus T$.

Можна запропонувати ідею алгоритму знаходження фундаментальних циклів [66], який базований на алгоритмі пошуку в глибину та має структуру, аналогічну алгоритму знаходження остового дерева. Кожну нову вершину розміщують у стеці й видаляють із нього після використання аналізу. Зрозуміло, що стек завжди містить послідовність вершин від вершини v до кореня. Тому, якщо ребро аналізу $\{v,u\}$ замикає цикл, тоді вершину u, згідно з останньою теоремою, знаходять у стеці, і цикл, який замикає ребро $\{v,u\}$, задаватиметься верхньою групою елементів стека, які починаються з v і закінчуються вершиною u.

3.3.3. *Знаходження компонент двозв'язності*

Вершину a неорієнтованого графа $G=(V,E)$ називають *точкою з'єднання*, якщо видалення цієї вершини і всіх інцидентних ребер зумовлює збільшення числа компонент зв'язності графа.

Неорієнтований граф називають двозв'язним, якщо він зв'язний і не має точок з'єднання. Довільний максимальний двозв'язний підграф графа G називають компонентою двозв'язності або блоком цього графа.

Двозв'язність графа – бажана властивість для багатьох задач, наприклад для задачі маршрутизації в інформаційній системі. Розглянемо граф з рис. 3.3, *а*. Точками з'єднання у ньому будуть вершини v_5 і v_6. Його блоки наведено на рис. 3.3, *б*.

Якщо (V_1, B_1) і (V_2, B_2) два різні блоки графа G, тоді $V_1 \cap V_2 = \varnothing$ або $V_1 \cap V_2 = \{a\}$, де a точка з'єднання графа G.

Точки з'єднання блоків графа можна знайти, використавши методологію пошуку в глибину [145]. Ідея алгоритму обґрунтована наступною теоремою [66].

Теорема 3.3. Нехай $D = (V, T)$ – остове дерево з коренем r зв'язного графа $G = (V, E)$, побудоване процедурою пошуку в глибину. Вершина $v \in V$ є точкою з'єднання графа G тоді і тільки тоді, коли або $v = r$ і r має не менше двох синів в D, або $v \neq r$ та існує син w вершини v, такий що ні w, ні будь-який з його нащадків не зв'язані ребром з жодним предком v.

Отже, для знаходження компонент двозв'язності достатньо провести пошук в глибину з деякої вершини r, обчислюючи для кожної вершини v два параметри: $WGN[v]$ і $L[v]$. $WGN[v]$ визначає номер вершини v за порядком, в якому вершини відвідують у разі пошуку в глибину, починаючи з вершини r. $L[v]$ визначає найменшу величину $WGN[u]$, де $u = v$ або вершина u зв'язана хордою з вершиною v або її довільним нащадком в D, де $D = \langle V, T \rangle$ – дерево, яке відповідає нашому пошуку в глибину. Параметр $L[v]$ обчислюють індукцією відносно дерева D, якщо відомі $L[w]$ для всіх синів w вершини v. Позначивши

$$A = \min\{L[w]: w - \text{син вершини } v\},$$
$$B = \min\{WGN[u]: \{u, v\} \in E \backslash T\},$$

матимемо $L[v] = \min\{WGN[v], A, B\}$.

З теореми 3.3 випливає, що v буде точкою з'єднання або коренем тоді і тільки тоді, коли $L[w] \geq WGN[v]$ для деякого сина u вершини v (покладемо, що $n > 1$).

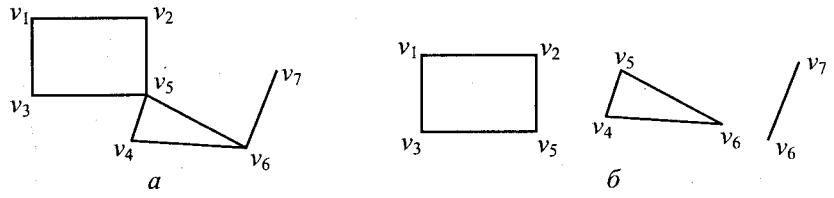


Рис. 3.3. Граф з точками з'єднання (*а*) і блоки цього графа (*б*)

3.3.4. *Ейлерові шляхи*

Ейлеровим шляхом називають у графі такий шлях, який проходить через кожне ребро графа тільки один раз, інакше кажучи, шлях $v_1, ..., v_{m+1}$ такий, що кожне ребро $e \in E$ з'являється в послідовності $v_1, ..., v_m$ тільки один раз, як $\{v_i, v_{i+1}\}$. Коли $v_1 = v_m$, тоді такий шлях називають ейлеровим циклом.

Задача існування ейлерового циклу має давню історію. Місто Кенігсберг мало багато каналів і мостів через них. Виникла природна задача: чи можна обійти всі мости, побувавши на кожному тільки один раз, і повернутись назад. Цю задачу було вирішено Ейлером у 1736 р. Він сформулював теорему (першу теорему теорії графів) про необхідні й достатні умови існування такого шляху.

Теорема 3.4. У графі існує ейлерів шлях тоді й тільки тоді, коли граф зв'язний і містить не більше двох вершин непарного степеня.

Доведення теореми можна знайти в [66].

Якщо у зв'язному графі немає вершин непарного степеня, тоді довільний ейлерів шлях буде циклом, тому що кінці ейлерового шляху, який не є циклом, будуть вершинами непарного степеня. Припустимо, що u і v – єдині вершини непарного степеня зв'язного графа $G = (V, E)$, і створимо граф G^*, додавши вершину t і ребра $\{u, t\}$ і $\{v, t\}$ (або просто $\{u, v\}$, якщо $\{u, v\} \notin E$). Тоді G^* – зв'язаний граф без вершин непарного степеня, а ейлерові шляхи в G знаходитимуться у взаємно однозначній відповідності з ейлеровими циклами в G^*. Тому далі можна розглядати тільки ейлерові цикли.

Основна ідея алгоритму така. Зафіксувавши будь-яку з вершин як початкову (наприклад, вершина з номером 1), будують шлях від неї, поки його можна подовжити, додавши нову вершину. Ребра цього шляху видаляють з графа. Процес має закінчитися за повернення в початкову вершину, тому що в іншому разі це означало б непарність вершини v_{m+1} вибраного шляху $v_1, ..., v_{m+1}$. Отже, з графа видалили цикл, для запам'ятовування вершин якого можна використати стек. Доведено, що після такого видалення степінь довільної вершини залишається парним. Вибрану початкову вершину переносять з першого стека у другий, і черговою вершиною аналогічного процесу побудови шляху стає верхній елемент першого стека. Знову ж через парність степенів всіх вершин графа знайдемо новий цикл. Його аналогічно додають до першого стека. Процес повторюють до моменту, коли перший стек стає пустим. Очевидно, що вершини, розміщені в другому стеці, утворюють для них шлях. Також відзначимо, що вершину переносять у другий стек тільки у разі, коли всі її ребра задані парами сусідніх вершин в одному із стеків. Тому після закінчення перебору другий стек міститиме ейлерів цикл. Процедура AlerLoop реалізує цей алгоритм на Паскалі. У ній замість занесення вершини в другий стек її виводять до друку.

```
{Допоміжні процедури AddStack та DelStack}
procedure AddStack(Val: integer; var Top: ^StackElementType);
                                        {Процедура додає номер вузла в стек}
  var StackElement: ^StackElementType;
begin
    new(StackElement) ;
    StackElement^.Prev := Top;
    StackElement^.Index := Val;
    Top := StackElement;
end;

procedure DelStack(var Top: ^StackElementType);        {Видаляє елемент зі стека}
begin
    Top := Top^.Prev;
end;

procedure AlerLoop(Root: ^CoreElementType);            {Головна процедура}
  var StackTop: ^ StackElementType;
      CoreElement: ^CoreElementType;
      BindElement: ^BindElementType;
      Index: integer;
begin
    StackTop := nil;
    AddStack(Root^.Index, StackTop);
    while (StackTop <> nil) do
      begin
        CoreElement := PointFinder(StackTop^.Index);
        Index := CoreElement^.Index;
        BindElement := CoreElement^.Outcomings;
        if (BindElement <> nil) then
          begin
            AddStack(BindElement^.Value, StackTop);
            CoreElement^.Outcomings := BindElement^.Next;
            BindElement^.Next^.Prev := CoreElement^.Outcomings;
            CoreElement := PointFinder(BindElement^.Value);
            BindElement := CoreElement^.Outcomings;
            while (BindElement^.Value <> Index) do
              BindElement := BindElement^.Next;
            if (BindElement^.Next = nil) then
                BindElement^.Prev^.Next := nil
              else
                begin
                  BindElement^.Prev^.Next := BindElement^.Next;
                  BindElement^.Next^. Prev := BindElement^.Prev;
                end
          end
```



```
        else
          begin
            writeln('Вершина ', StackTop^.Index);
            DelStack(StackTop);
          end;
end;
```

Для коректної роботи процедури необхідно, щоб у головній програмі був визначений тип StackElementType та потрібні змінні у такий спосіб:

```
type StackElementType = record
                          Prev: ^ StackElementType;
                          Index: integer;
                        end;
```

Теорема 3.5. Часова складність алгоритму, описаного процедурою AlerLoop становить $O(mn)$.

Доведення. Зрозуміло, що часову складність алгоритму визначає часова складність головного циклу while. Кожна ітерація цього циклу виконує одну з двох операцій: або розміщує вершину в стек і видаляє ребро із графа, або видаляє вершину із стека і друкує її. Часова складність цих операцій, за винятком знаходження вузла головного списку, що містить u, має константний характер. Знайти вузол головного списку з вершиною u можна за час $O(n)$. Тому часова оцінка алгоритму визначатиметься кількістю ітерацій циклу while (кількістю ребер у графі), помноженою на кількість ітерацій внутрішнього циклу while (кількість вершин у графі n), і становитиме $O(mn)$.

3.4. Знаходження найкоротших шляхів у графі

Важливе місце в теорії графів посідає задача знаходження шляхів між вершинами графа і особливо задача знаходження мінімальних шляхів. Вона має великий теоретичний інтерес і широке практичне застосування: для складання розкладу виконання робіт, розв'язання транспортних задач, для ефективного розподілу електроенергії тощо.

Розглянемо навантажені орієнтовані графи. Нагадаємо, що довжину шляху між довільними двома вершинами визначають як суму ваг ребер, що входять до цього шляху (вагу ребра (u,v) позначають $a(u,v)$, а якщо вершини u і v не мають спільного ребра, тоді $a(u,v) = \infty$). Якщо в довільному графі вага кожного ребра дорівнює одиниці, тоді отримаємо традиційне визначення довжини шляху через кількість ребер; довжина нульового шляху дорівнює нулю. Серед всіх шляхів між двома вершинами u, v найцікавішим є найкоротший шлях. Довжину цього найкоротшого шляху

позначають $d(u,v)$ і називають *відстанню* від u до v. Зрозуміло, що визначена таким чином відстань може бути і від'ємною. Якщо не існує жодного шляху з u до v, покладають $d(u,v) = \infty$.

Зазначимо [66], що якщо кожен цикл графа має додатну довжину, тоді найкоротший шлях буде завжди елементарним шляхом. Якщо ж у графі існує цикл від'ємної довжини, тоді відстань між деякими парами вершин стає невизначеною (обходячи цей контур потрібну кількість разів, можна визначити шлях між цими вершинами з довжиною, меншою за довільне дійсне число). У цьому разі також можна говорити про довжину найкоротшого елементарного шляху, але ця задача буде значно складнішою.

Багато алгоритмів розв'язання задач наведеного типу знаходять відстані між вершинами, але не шляхи. У разі додатної довжини всіх циклів за відстанню легко визначити найкоротші шляхи. Наведіть свою версію такого перетворення.

3.4.1. *Шляхи в ациклічному графі*

Для розв'язання багатьох задач корисно розглядати спеціальні типи графів, структура яких має широке застосування на практиці. Одним із них є ациклічні графи. В останніх виділяють ще один підклас: графи з однією виділеною вершиною – джерелом. Вершину орієнтованого графа, в яку не входить жодне ребро, називають *джерелом*. Іноді таку вершину вводять фіктивно. Справа полягає в тому, що для багатьох задач не існує алгоритмів, ліпших за алгоритми, що працюють з графами, в яких виділена вершина-джерело.

Основою більшості алгоритмів знаходження відстаней від джерела до всіх вершин в ациклічному непустому графі є такі факти [66]:

- вершини можна перенумерувати так, що кожне ребро матиме вигляд (v_i, v_j), де $i < j$;
- існує вершина, в яку не заходить жодне ребро.

Щоб пересвідчитись у цьому, вибирають довільну вершину w_1, потім вершину w_2, таку що $(w_2, w_1) \in E$, потім вершину w_3, таку що $(w_3, w_2) \in E$ і т. д. За скінченну кількість кроків ми маємо дійти до деякої вершини w_i, в яку не заходить жодне ребро, бо через ациклічність жодна вершина не може повторюватися в послідовності $w_1, w_2, w_3, \ldots$.

Ідея алгоритму запропонованої нумерації буде такою. Вершини, в які не заходить жодна дуга, можна зберегти в стеці. Після початкового заповнення стека такими вершинами починаємо аналізувати вершини стека. Вибираємо верхній елемент стека – вершину u. Вершині u присвоюють мінімальний номер із ще не використаних номерів. Цим ми гарантуємо, що всі дуги, які виходять із вершини u, приводитимуть до вершини з більшими номерами. Потім вершину u разом з дугами, що виходять з неї, видаляють з графа. Це зменшує на одиницю число дуг, що заходять у

Вхідні дані: орієнтований граф (V, E), $V = \{v_1,\ldots,v_n\}$, $\forall\langle v_i, v_j\rangle \in E : i < j$ (такий граф отримаємо після процедури перенумерації). Граф задано списками інцидентності ENTER(v_i) (перелік вхідних вершин).
Вихідні дані: відстані від вершини v_i до всіх інших DIST[v_j].

```
begin
   D[v_i] := 0;
   for j := 2 to n do DIST[v_j] := ∞;
   for j := 2 to n do
      for v_i ∈ ENTER[v_j] do DIST[v_j] := min(DIST[v_j], DIST[v_i] + a(v_i,v_j));
end.
```

Алгоритм 3.1. Підрахунок відстані від джерела до кожної вершини

кожну вершину v із списку інцидентності вершини u, тобто таких, що $(u, v) \in E$. Отже, визначимо поле count для кожної вершини графа, значення якого визначатиме напівстепінь входу відповідної вершини (кількість вхідних дуг вершини). Якщо для деякої з вершин значення count стало рівним нулю, тоді цю вершину додають до стека. Завдяки ациклічності графа повне спустошення стека, що призводить до зменшення роботи алгоритму, настане тільки після того, як всі вершини графа перенумерують.

Якщо після видалення з графа певної вершини і дуг значення поля count для якоїсь з вершин, суміжних з видаленою, стало рівним нулю, то її теж заносимо до стека і аналізуємо за описаним вище принципом. Повне спустошення стека означатиме, що всі вершини вже перенумеровані. Описаний процес реалізує процедура в основній програмі.

Підрахунок відстані від джерела до кожної вершини описує алгоритм 3.1.

Визначимо часову складність алгоритму. Припустимо, що максимальний напівстепінь входу вершин графа дорівнює d. Тоді цикл визначення напівстепенів входу витрачатиме $O(nd)$ кроків, де n – кількість вершин графа. Цикл початкового занесення до стека з нульовою кількістю попередників не перевищуватиме попередню оцінку.

Загальний цикл перенумерації має n кроків. В його тілі є процедура для зменшення поля count, що, в загальному випадку, може мати до n кроків. Тому часова складність циклу перенумерації $O(n^2)$ визначає і всю часову складність алгоритму нумерації.

Наведений вище алгоритм перенумерації вершин за своєю структурою нагадує алгоритм топологічного впорядкування [61].

Тоді процедура FindDistance відшукуватиме відстані від джерела v до всіх вершин в ациклічному графі. Граф $G = (V, E)$ задається списками інцидентності. Результатом будуть відстані від v_1 до всіх вершин графа:

$$\mathrm{DIST}[v_i] = \mathrm{d}(v_1, v_i), \quad i = 1, 2, \ldots, n.$$

```
Procedure FindDistance;
  var i, j: integer;
      now: nodeptr;
begin
    Rename;
    for i := 1 to dim do dist[i] := max_real;
    dist[org_new[start]] := 0;
    for i := 2 to dim do
      begin
        now := g_enter[new_org[i]];
        while (now <> nil) do
          begin
            if (dist[i] > dist[org_new[now^.num]] + now^.w)
              then dist[i] := dist[org_new[now^.num]] + now^.w;
            now := now^.next;
          end;
      end;
end;
```

Зрозуміло, що часову оцінку процедури FindDistance визначатиме $O(n^2)$.

Реалізація цього алгоритму знаходження мінімальних відстаней від джерела до всіх вершин в ациклічному графі наведено у програмі 3.2.

```
program Ex3_2;

{Вхід: орієнтований граф, заданий матрицею суміжностей, та початкова вершина
Вихід: відстані від початкової вершини до всіх інших}

const max_n = 50;                          {Максимальна кількість вершин}
      max_real = 10000;                    {Максимальна вага ребра}

type  nodeptr = ^node;                     {Для списку інцидентності}
      node = record
               num: integer;               {Вихідний номер вершини}
               w: integer;                 {Вага ребра до цієї вершини}
               next: nodeptr;              {Наступне ребро}
             end;

var graph: array [1..max_n] of nodeptr;    {Список інцидентності (вихідні ребра)}
    g_enter: array [1..max_n] of nodeptr;  {Список інцидентності (вхідні ребра)}
    org_new: array [1..max_n] of integer;  {Таблиця перенумерації}
    new_org: array [1..max_n] of integer;  {Таблиця оберненої перенумерації}
    dist: array [1..max_n] of integer;     {Відстані}
    dim, start: integer;          {dim – кількість вершин, start – початкова вершина}

{Процедура додає ребро до списку інцидентності (вихідні ребра)}
procedure AddNode(i, j: integer; d: integer);
  var now, ptr: nodeptr;
```



```
begin
    now := graph[i];
    new(ptr);
    ptr^.w := d;
    ptr^.num := j;
    ptr^.next := nil;
    if (now = nil) then
      begin
        graph [i] := ptr;
        exit;
      end;
    while (now^.next <> nil) do
      now := now^.next;
    now^.next := ptr;
end;

{Процедура додає ребро до списку інцидентності (вхідні ребра)}
procedure AddEnter(i, j: integer; d: integer) ;
  var now, ptr: nodeptr;
begin
    now := g_enter[j];
    new(ptr);
    ptr^.w := d;
    ptr^.num := i;
    ptr^.next := nil;
    if (now = nil) then
      begin
        g_enter[j] := ptr;
        exit;
      end;
    while (now^.next <> nil) do
      now := now^.next;
    now^.next := ptr;
end;

{Процедура читає граф з файла}
procedure ReadGraph(fname: string);
  var inf: text;
      i, j: integer;
      d: integer;
begin
    assign(inf, fname);
    reset(inf);
    readln(inf, dim);
    readln(inf, start);
    for i := 1 to dim do
```

```
        begin
            graph[i] := nil;
            for j := 1 to dim do
                begin
                    read(inf, d);
                    if (d > 0) then
                        begin
                            AddNode(i, j, d);
                            AddEnter(i, j, d);
                        end;
                end;
                readln(inf);
        end;
        close(inf);
end;

{Процедура перенумеровує вершини графа, щоб для ребра ⟨v[i],v[j]⟩  i < j}
procedure Rename;
  var  count: array [1..max_n] of integer;
       i, j, num, u: integer;
       now: nodeptr;
       stck: array [1..max_n + 1] of integer;
       s_top: integer;

procedure push(n: integer);
begin
    stck[s_top] := n; inc(s_top);
end;

function pop: integer;
begin
    if (s_top <= 1) then
      begin
          pop := -1;
          exit;
      end;
    dec(s_top); pop := stck[s_top];
end;

begin
    for i := 1 to dim do count[i] := 0;
    s_top := 1;
{Підрахування напівстепеня заходу для кожної вершини}
    for i := 1 to dim do
      begin
          now := graph[i];
```



```
          while (now <> nil) do
              begin
                  inc(count[now^.num]);
                  now := now^.next;
              end;
      end;
{Усі вершини з напівстепенем заходу 0 => стек}
    for i := 1 to dim do
      if count[i] = 0 then push(i);
{Перенумерація}
      num := 0;
      while (s_top > 1) do
          begin
              u := pop;
              inc(num);
              org_new[u] := num;
              now := graph[u];
              while (now <> nil) do
                  begin
                      dec(count[now^.num]);
                      if (count[now^.num] = 0) then push(now^.num);
                      now := now^.next;
                  end;
          end;
{Таблиця оберненої перенумерації}
      for i := 1 to dim do
          new_org[org_new[i]] := i;
end;

{Виведення результату}
procedure WriteOutput;
  var i: integer;
begin
    writeln;
    for i := 1 to dim do writeln('1 -> ', new_org[i], ' = ', dist[i]);
end;

procedure FindDistance;
  var  i, j: integer;
       now: nodeptr;
begin
    Rename;
    for i := 1 to dim do
      dist[i] := max_real;
    dist[org_new[start]] := 0;
```

```
    for i := 2 to dim do
      begin
        now := g_enter[new_org[i]];
        while (now <> nil) do
          begin
            if (dist[i] > dist[org_new[now^.num]] + now^.w) then
              dist[i] := dist[org_new[now^.num]] + now^.w;
            now := now^.next;
          end;
      end;
end;

{Головна програма}
begin
    ReadGraph('input.txt');                    {Вхідні дані читаємо з input.txt}
    FindDistance;
    WriteOutput;
end.
```

Програма 3.2. Знаходження мінімальних відстаней

3.4.2. *Найкоротші шляхи між усіма парами вершин*

Зрозуміло, що для знаходження найкоротших шляхів між усіма парами вершин у графі можна використати алгоритм їх знаходження від джерела до всіх вершин у графі з вищерозглянутого розділу, вибираючи як джерело нову вершину графа. Процес слід повторити стільки раз, скільки є вершин у графі. Однак часова складність такого алгоритму буде не найкращою.

Визначте часову складність алгоритму.

Розглянемо орієнтований граф $G = (V, E)$, де $V = \{v_1, v_2, ..., v_n\}$ з матрицею ваг $A = [a_{ij}] = [a(v_i, v_j)]$. Позначивши через d_{ij}^m довжину найкоротшого шляху із v_i і v_j, який містить не більше m дуг, отримують очевидні рівняння [66]:

$$d_{ij}^0 = \begin{cases} 0, & \text{якщо } i = j, \\ \infty, & \text{якщо } i \neq j, \end{cases}$$

$$d_{ij}^{m+1} = \min\left\{d_{ik}^m + a_{kj} : 1 \le k \le n\right\}.$$

Останнє рівняння за структурою нагадує визначення добутку двох квадратних матриць: операцію min асоціюють з «сумою», а операцію «+» – з «добутком». Позначимо такий «добуток» двох матриць A і B через $A * B$. Відзначимо, що одиничним елементом слугує матриця:

$$U = \begin{bmatrix} 0 & \infty & \infty & \dots & \infty \\ \infty & 0 & \infty & \dots & \infty \\ \infty & \infty & 0 & \dots & \infty \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \infty & \infty & \infty & \dots & 0 \end{bmatrix}.$$

Зрозуміло, що $\left[d_{ij}^0\right] = U$ і $d_{ij}^m = \underbrace{\left(\left(\cdots\left(\left(A * A\right) * A\right)\cdots\right) * A\right)}_{m}$, $m \ge 1$.

Можливий один із варіантів:

1) $d_{ij}^{n-1} = d_{ij}^n$, і в результаті $d_{ij}^m = d_{ij}^{n-1}$ для кожного $m \ge n$, тоді $d_{ij}^{n-1} = d(v_i, v_j)$.

2) $d_{ij}^{n-1} \neq d_{ij}^n$ – це свідчить, що граф має цикл від'ємної довжини.

Враховуючи, що добуток $A * B$ двох матриць розмірності $n \times n$ можна обчислити за час $O(n^3)$, матрицю $\left[d_{ij}^{n-1}\right]$, а отже, і відстані між усіма парами вершин, можна обчислити за час $O(n^4)$.

Часову оцінку можна знизити, скориставшись фактом, що у формулі визначення d_{ij}^m операція «*» асоціативна,– можемо обчислювати добуток визначення d_{ij}^m, покроково підносячи матрицю A до квадрата. Це приведе до заміни $n-1$ множень матриці на $\lceil \log n \rceil$ множень. Отже, знайти відстані між усіма парами вершин у графі без циклів (контурів) від'ємної довжини можна за час $O(n^3 \log n)$.

Реалізацію цього алгоритму знаходження мінімальних відстаней між усіма парами вершин наведено у програмі 3.3.

Ми ще розглянемо один із найкращих алгоритмів знаходження всіх шляхів у графі в розділі 7.

```
Program Ex3_3;
                {Вхід: файл з матрицею суміжності графа
                 Формат файла:
                      1 рядок – кількість вершин
                      наступні n рядків містять n чисел
                      (вага ребра між парою вершин або –1, якщо ребра не існує)
                Вихід: відстані між усіма парами вершин}
const maxgraph = 20;
      maxreal = 100000;
type  matr = record
               data: array [1..maxgraph, 1..maxgraph] of real;
               n: integer;
             end;
var m1, m2: matr;
```

```
{Процедура читає граф з файла}
procedure ReadInputData(fname: string; var res: matr);
  var inf: text;
      i, j: integer;
begin
    assign(inf, fname);
                        {$I-}
    reset(inf);
                        {$I+}
    readln(inf, res.n);
    for i := 1 to res.n do
      begin
        for j := 1 to res.n do
          begin
            read(inf, res.data[i, j]);
            if (res.data[i, j] = -1) then res.data[i, j] := maxreal;
          end;
        readln(inf);
      end;
    close(inf);
end;

{Процедура шукає відстані між усіма парами вершин}
procedure FindDist(org: matr; var res: matr);
  var i, j: integer;
      tmp, tmp2: matr;

{Функція проводить «множення» матриць та повертає відзнаку зміни}
function Mul(m1, m2: matr; var res: matr): boolean;
 var i, j, k: integer;
     val: real;
     flag: boolean;
begin
    res.n := m1.n;
    flag := false;
    for i := 1 to res.n do
      for j := 1 to res.n do
        begin
          val := maxreal;
          for k := 1 to res.n do
            if (m2.data[i, k] + m1.data[k, j]) < val then
              val := m2.data[i, k] + m1.data[k, j];
          if (m2.data[i, j] <> val) then flag := true;
          res.data[i, j] := val;
        end;
    Mul := flag;
end;
```



```
begin
                {Формуємо одиничний елемент}
    tmp.n := org.n;
    for i := 1 to org.n do
      for j := 1 to org.n do
        if (i = j) then tmp.data[i, j] := 0 else tmp.data[i, j] := maxreal;
                {Виконуємо максимум n ітерацій}
    for i := 1 to tmp.n do
      begin
                {Якщо матриця відстаней не змінюється – припинити ітерації}
        if (not Mul(org, tmp, tmp2)) then break;
        tmp := tmp2;
      end;
    res := tmp;
end;

procedure WriteOutputData(mm: matr);
  var i, j: integer;
begin
    writeln('Найкоротші шляхи між парами вершин: ');
    for i := 1 to mm.n do
      begin
        for j := 1 to mm.n do
        write(mm.data[i, j]: 0: 0, ' ');
        writeln;
      end;
end;

begin
    ReadInputData('input.txt', m1);
    FindDist(m1, m2);
    WriteOutputData(m2);
end.
```

Програма 3.3. Знаходження мінімальних відстаней між усіма парами вершин

3.4.4. *Максимальний потік у мережі*

Мережею називають пару $S = \langle G, c \rangle$, де $G = (V, E)$ – довільний орієнтований граф, а c: $E \to R$ – функція, що кожній дузі $\langle u, v \rangle$ ставить у відповідність невід'ємне дійсне число $c(u, v)$, яке називають пропускною здатністю цієї дуги, або вагою дуги. Множини V і E називають відповідно множиною вершин і множиною дуг мережі S.

Для довільної функції

$$f: E \to R \tag{3.1}$$

і довільної вершини $v \in S$ розглянемо величину

$$\mathrm{Div}_f(v) = \sum_{u:v\to u} f(u,v) - \sum_{v:u\to v} f(u,v).$$

$\mathrm{Div}_f(v)$ визначає «*кількість потоку*», який виходить із вершини v, за умови, що $f(u, v)$ – потік із u до v.

Виділимо в мережі S вершину-джерело s і вершину-стік $t (s \neq t)$. Під потоком із s в t мережі S розумітимемо довільну функцію типу (3.1), для якої дотримуються умови:

$$0 \le f(u, v) \le c(u, v) \text{ для кожної дуги } (u, v) \in E; \tag{3.2}$$

$$\mathrm{Div}_f(v) = 0 \text{ для кожної вершини } v \in V\setminus\{s,t\}. \tag{3.3}$$

Величину $W(f) = \mathrm{Div}_f(s)$ називають величиною потоку f

Розглядатимемо потік, який виникає тільки у вершині-джерелі s і не накопичується в жодній з вершин, за винятком вершини-стока t, та задовольняє умові: *через дугу* (u, v) *можна пропустити не більше ніж* $c(u, v)$ *одиниць потоку.*

Є багато задач, пов'язаних з потоками. Розглянемо одну з них – знаходження максимального потоку [66].

Під розрізом $P(A)$ мережі S, що відповідає підмножині $A \subseteq V$ ($A \neq \varnothing$, $A \neq V$), називають множину дуг $(u,v) \in E$, таких що $u \in A$ і $v \in V\setminus A$:

$$P(A) = E \cap (A \times (V \setminus A)).$$

Проходження довільного потоку f у мережі S через розріз $P(A)$ визначають так:

$$f(A, V \setminus A) = \sum_{e\in P(A)} f(e).$$

Можна також довести, що якщо $s \in A$ і $t \in V\setminus A$, тоді для довільного потоку f із s в t матимемо:

$$W(f) = f(A, V\setminus A) - f(V\setminus A, A), \tag{3.4}$$

і, поклавши $A = V\setminus\{t\}$, отримаємо

$$\mathrm{Div}_f(s) = W(f) = f(V\setminus\{t\},\{t\}) - f(\{t\}, V\setminus\{t\}) =$$
$$= -\left(f(\{t\}, V\setminus\{t\}) - f(V\setminus\{t\},\{t\})\right) = -\mathrm{Div}_f(t).$$

Формула (3.4) визначає можливість обчислення кількості потоку в довільному розрізі, що відокремлює s від t. А отримана рівність $\mathrm{Div}_f(s) = -\mathrm{Div}_f(t)$ відбиває факт, що у стік входить така кількість потоку, яка виходить із джерела.

Пропускну здатність розрізу $P(A)$ визначають так:

$$c(A, V\setminus A) = \sum_{e\in P(A)} c(e).$$

Розріз з мінімальною пропускною здатністю називають мінімальним розрізом. Фордом і Фалкерсоном [120] доведено, що величина кожного

потоку із s в t не перевищує пропускної здатності мінімального розрізу, що розділяє s і t, та що потік, який досягатиме цього значення, існує.

Усі відомі алгоритми знаходження максимального потоку обґрунтовані методом ланцюгів, що збільшуються (послідовне збільшення потоку).

Кажуть, що дуга e мережі S є допустимою дугою із u до v відносно потоку f, якщо

$$e = (u,v) \quad \text{і} \quad f(e) < c(e) \tag{3.5}$$

або

$$e = (v,u) \quad \text{і} \quad f(e) > 0. \tag{3.6}$$

Якщо дотримується умова (3.5), тоді дугу називають узгодженою, у разі виконання (3.6) – неузгодженою.

Ланцюгом збільшення довжини k для даного потоку f із s в t називають довільну знакозмінну послідовність (попарно різних) вершин і дуг

$$v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k, \tag{3.7}$$

таку що $v_0 = s$, $v_k = t$ і для кожного $i \le k$ дуга e_i є допустимою із v_{i-1} до v_i відносно потоку f

Знання ланцюга типу (3.7) дає змогу легко збільшити величину потоку f на

$$\delta = \min\{\Delta(e_i) : 1 \le i \le k\},$$

де

$$\Delta(e_i) = \begin{cases} c(e_i) - f(e_i), & \text{якщо дуга узгоджена,} \\ f(e_i), & \text{якщо дуга неузгоджена,} \end{cases}$$

шляхом збільшення на δ потоку по кожній узгодженій дузі e_i: $f'(e_i) = f(e_i) + \delta$ і зменшення потоку на δ по кожній неузгодженій дузі e_i: $f'(e_i) = f(e_i) - \delta$.

Якщо до такого визначення функції f' додати ще умову: $f'(e) = f(e)$ для дуг, що не належать ланцюгу, тоді функція f буде потоком.

Доведіть останнє твердження.

Тому

$$W(f') = \mathrm{Div}_f(s) = \mathrm{Div}_f(s) + \delta = W(f) + \delta.$$

Проілюструємо збільшення потоку в мережі з рис. 3.4.

Потік $f(e)$ вказано біля відповідних дуг (пропускну здатність кожної дуги у дужках). Збільшуваним ланцюгом може бути така послідовність:

$$v_1, [v_1, v_3], v_3, [v_3, v_2], v_2, [v_2, v_4], v_4, [v_4, v_5], v_5, [v_5, v_6], v_6, [v_6, v_7], v_7.$$

Використання цього ланцюга збільшує потік на $\delta = \Delta(v_4, v_5) = 1$, що ілюструє рис. 3.5.

Звідси видно, що мінімальним розрізом буде $A = \{v_1, v_2, v_3, v_4\}$ (його пропускна здатність $W(A) = c(v_2, v_5) + c(v_4, v_5) + c(v_4, v_6) = 11$).

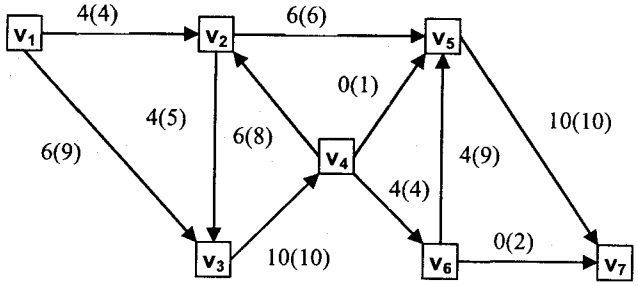


Рис. 3.4. Мережа

Більшість алгоритмів побудови максимального потоку обґрунтована наступною теоремою [66].

Теорема 3.6. Такі три умови еквівалентні:
1. Потік із s в t максимальний.
2. Не існує ланцюга, що збільшується, для f.
3. $W(f) = c(A, V \setminus A)$ для деякого $A \subseteq V$, такого що $s \in A$, $t \notin A$.

Із цих роздумів випливає ідея алгоритму знаходження максимального потоку і мінімального розрізу, якщо довести факт існування максимального потоку в довільній мережі. Починаємо з довільного потоку (наприклад, $f(e) = 0$ для кожної $e \in E$), шукаємо ланцюги, що збільшуються, і збільшуємо вздовж них максимальний потік. Виникає ще одне питання – чи закінчить алгоритм коли-небудь свою роботу. Л. Фордом і Д. Фалкерсоном [120] доведено, що можна так «спеціально» підбирати ланцюги, що процес ніколи не закінчиться. Відомі також теоретичні результати щодо скінченності побудови зростаючих ланцюгів. Наприклад, у праці [119] доведено, що коли збільшувати фактичний потік вздовж найкоротшого ланцюга зростання, тоді максимальний потік рахують за допомогою не більше $\frac{mn}{2}$ ланцюгів, де $n = |V|$, $m = |E|$.

Реалізувати останній підхід можна так. Почнемо з джерела s вести пошук в ширину в графі G_f, який складається із таких дуг (u,v), що в

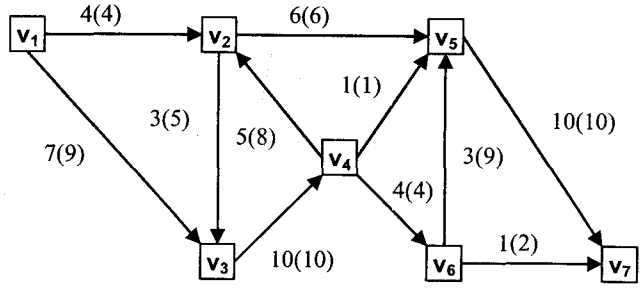


Рис. 3.5. Мережа зі збільшеним потоком

мережі існує допустима дуга від u до v відносно фактичного потоку f в напрямку досягнення стоку t. Знайдений найкоротший шлях із s в t відповідає найкоротшому ланцюгу зростання із s в t початкової мережі.

Доведіть, що часова складність такого алгоритму визначатиметься $O(mn(n + m))$.

Е. Дініцем [42] запропонований алгоритм знаходження максимального потоку за час $O(n^2)$. Він будує для початкової мережі додаткову ациклічну мережу S_f, структура якої точно відбиває всі найкоротші ланцюги збільшення із s в t відносно фактичного потоку f. Знову ж можна використати метод пошуку в ширину. Мережа S_f містить джерело s, стік t і дуги графа G_f типу (u,v), де u знаходиться на віддалі d, а v на віддалі $d + 1$ від джерела s, $0 \le d < l$, де l – довжина мережі S_f (відстань від s до t у графі G_f). Пропускну здатність $C_f(u,v)$ визначають так:

$$C_f(u,v) = \begin{cases} c(u,v) - f(u,v), & \text{якщо } (u,v) \text{ задає узгоджену дугу,} \\ f(u,v), & \text{якщо } (u,v) \text{ задає неузгоджену дугу,} \\ c(u,v) + f(u,v) & \text{в іншому разі.} \end{cases}$$

Якщо для задання початкової мережі S використати списки інцидентності $REC[v]$, $PRED[v]$, $v \in V$ та масиви пропускних здатностей дуг $C[u,v]$, $u, v \in V$ і величину фактичного потоку задавати матрицею $F[u,v]$, $u, v \in V$, тоді, згідно з працею [66], процедура *PSA* реалізовуватиме алгоритм побудови додаткової ациклічної мережі S_f. Допоміжну мережу реалізують за допомогою аналогічних структур в основній мережі. Її імена відрізняються від імен початкової мережі префіксом X. Наприклад, $xc(u,v) = c(u,v) - f(u,v)$ – величина, що відбиває пропускну здатність ребра в допоміжній мережі. Відстань від джерела до вершини зберігатиметься у масиві LENGHT.

```
procedure psa;
begin
  for u := 1 to t do                          {t – кількість вершин у мережі}
    begin
      lenght[u] := ∞;
      for v := 1 to t do
        begin
          xc[u,v] := 0;
          xf [u,v] := 0;
        end;
    end;
  begq := nil;                                {Занесення першої}
  endq := nil; addq(1);                       {вершини до черги}
  XV := [];
```

```
lenght[1] := 0;
while begq <> nil do
  begin
    u := outq; XV := XV + [u];
    for v := 1 to t do
      begin
        if c[u, v] > 0 then        {Визначення пропускної здатності ребер, що}
                                   {з'єднують наступні вершини, і занесення вершин до черги}
          if (lenght[u] <lenght[v]) and (lenght[v] <= lenght[t]) and (f[u, v] < c[u, v]) then
            begin
              if lenght[v] = ∞ then addq(v);
              lenght[v] := lenght[u] + 1;
              xc[u, v] := c[u, v] – f [u, v];
            end;
        if c[v, u] > 0 then
          if (lenght[u] < lenght[v]) and (lenght[v] <= lenght[t]) and f [v, u] > 0) then
            begin
              if lenght[v] = ∞ then addq(v);
              lenght[v] := lenght[u] + 1; xc[u, v] := xc[u, v] + f [v, u];
            end;
      end;
  end;
end;
```

Якщо після завершення роботи процедури отримана відстань від джерела до стоку (lenght[t]) в додатковій мережі дорівнює ∞, то це означає, що ми в нашому процесі не досягли стоку, тобто в мережі не існує ланцюга, що збільшується від джерела до стоку. Враховуючи теорему 3.6, фактичний потік в мережі S буде максимальним.

Головна ідея Е. Дініца обґрунтована розбиттям процесу збільшення потоку вздовж ланцюгів зростання на фази, що відповідають використанню найкоротших ланцюгів певної довжини. Фаза починається з побудови допоміжної мережі S_f та знаходження в ній псевдомаксимального потоку. Згідно з працею [66], псевдомаксимальним потоком у мережі S_f завдовжки k називають деякий потік в S_f, такий що в S_f не існує ланцюга зростання завдовжки k відносно потоку f^*. Псевдомаксимальний потік потім «переноситься» в основну мережу: потік $f^*(u, v)$ додається до $f(u,v)$, і якщо це викликає переповнення дуги, тоді цей надлишок ліквідують відповідним зменшенням потоку $f(v, u)$. Така модифікація потоку, проведена для всіх дуг допоміжної мережі, визначає новий потік f', такий що $W(f') = W(f) + W(f^*)$.

На цьому фазу вважають завершеною.

Метод Е. Дініца ґрунтується на ітераційному процесі виконання фаз, починаючи з нульового потоку до моменту, коли потік в мережі не стане максимальним.

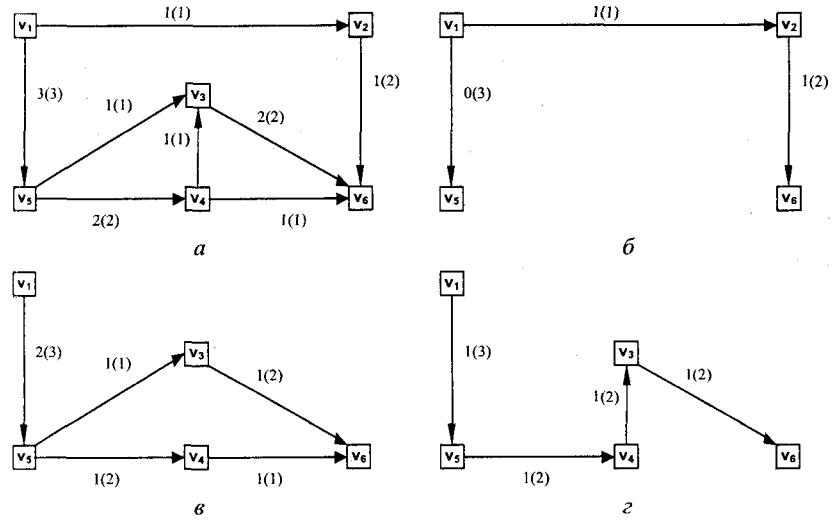


Рис. 3.6. Мережа і фази побудови псевдопотоків:

а – деяка мережа; *б* – фаза 1 з довжиною псевдомаксимального потоку 2; *в* – фаза 2 з довжиною псевдомаксимального потоку 3; *г* – фаза 3 з довжиною псевдомаксимального потоку 4

Процес побудови додаткових ациклічних мереж проілюструє рис. 3.6.

Видно, що отриманий потік (4) є максимальним, оскільки пропускна здатність розрізу $A = \{v_1\}$ $P(A) = 4$.

Нарешті, слід знайти ефективний алгоритм побудови псевдомаксимального потоку в допоміжній ациклічній мережі. Одним із таких алгоритмів є алгоритм, що використовує *поняття потенціалу вершини [136]*.

Потенціалом вершини v довільної мережі $S = (G,c)$ з джерелом s і стоком t називають величину

$$P(v) = \min\{P\_in(v), P\_out(v)\},$$

де $P\_in(v) = \sum_{u:u\to v} c(u,v)$, якщо $v \neq s$; $P\_in(v) = \infty$, якщо $v = s$; $P\_out(v) = \sum_{u:v\to u} c(u,v)$, якщо $v \neq t$; $P\_in(v) = \infty$, якщо $v = t$.

Наступна процедура на основі обчислення потенціалів всіх вершин будує псевдомаксимальний потік для допоміжної безконтурної мережі:

```
procedure maxpsa;
  var ...
begin
    stk := nil;
    for v := 1 to t do
      if v in XV then
```

```
    begin
      P_in[v] := 0; P_out[v] := 0;
          {Pr – потенціал вершин, мінімальне значення з P_in і P_out, сумарного
          потоку, що входить у вершину, і потоку, що виходить; вершини
          з нульовим потоком заносять у стек для подальшого вилучення}
      if v = 1 then P_in[v] := ∞
          else
              for u := 1 to t do
                  P_in[v] := P_in[v] + xc[u, v];
      if v = t then P_out[v] := ∞
          else
              for u := 1 to t do
                  P_out[v] := P_out[v] + xc[v, u];
      if P_in[v] < P_out[v] then Pr[v] := P_in[v]
          else Pr[v] := P_out[v];
      if Pr[v] = 0 then
          begin
              push(v);
              xc[v, u] := 0;
          end;
    end;
for v := 1 to t do
  if v in XV then Q[v] := 0;
XN := XV;
while stk <> nil do
 begin
  v := pop;
  XN := XN – [v];
  for u := 1 to t do
   begin                                              {*}
    if xc[u, v] > 0 then
      begin
        P_out[u] := P_out[u] – (xc[u, v] – xf [u, v]);
        xc[u, v] := 0;
        if Pr[u] <> 0 then
          begin
            if P_in[v] < P_out[v] then Pr[v] := P_in[v]
                else Pr[v] := P_out[v];
            if (Pr[v] = 0) and (v in XN) then push(v);
          end;
      end;
    if xc[v, u] > 0 then
      begin
        P_out[u] := P_out[u] – (xc[v, u] – xf [v, u]);
        xc[v, u] := 0;
        if Pr[u] <> 0 then
```



```
          begin
            if P_in[v] < P_out[v] then Pr[v] := P_in[v]
                else Pr[v] := P_out[v];
            if (Pr[v] = 0) and (v in XN) then push(v);
          end;
      end;
   end;                                                {*}
 end;
if XN <> [] then          {XN – множина вершин з ненульовим потенціалом}
 begin
  Pm := ∞;
  for v := 1 to t do
    if v in XN then
      if Pr[v] < Pm then
                          {Знаходження вершини з мінімальним потенціалом
                          і побудова від неї потоку в кінець і початок мережі}
        begin
          r := v;
          Pm := Pr[r];
        end;
  begq := nil;
  endq := nil;
  addq(r);
  Q[r] := Pm;
  repeat
    v := outq;
    P_in[v] := P_in[v] – Q[v];
    P_out[v] := P_out[v] – Q[v];
    Pr[v] := Pr[v] – Q[v];
    if v = t then Q[v] := Pm
      else
        begin
          for u := 1 to t do
            if xc[v, u] > 0 then
              while Q[v] > 0 do
                begin
                  if Q[u] = 0 then addq(u);
                  if Q[v] < (xc[v, u] – xf [v, u]) then delta := Q[v]
                      else delta := (xc[v, u] – xf [v, u]);
                  xf [v, u] := xf [v, u] + delta;
                  Q[v] := Q[v] – delta;
                  Q[u] := Q[u] + delta;
                  if xf [v, u] = xc[v, u] then xc[v, u] := 0;
                end;
        end;
  until (v = t) or (begq = nil);
end;
```

```
    endq := nil;
    begq := nil;
    addq(r) ;Q[r] := Pm;
    repeat
     v := outq;
     if v <> r then
      begin
        P_in[v] := P_in[v] – Q[v];
        P_out[v] := P_out[v] – Q[v];
        Pr[v] := Pr[v] – Q[v];
      end;
     if v = 1 then Q[v] := 0        {Побудова потоку в початок мережі}
      else
        begin
          for u := 1 to t do
            if xc[u, v] > 0 then
              while Q[v] > 0 do
                begin
                  if Q[u] = 0 then addq(u);
                  if Q[v] < (xc[u, v] – xf [u, v]) then delta := Q[v]
                      else delta := (xc[u, v] – xf [u, v]);
                  xf [u, v] := xf [u, v] + delta;
                  Q[u] := Q[u] + delta;
                  Q[v] := Q[v] – delta;
                  if xf [u, v] = xc[u, v] then xc[u, v] := 0;
                end;
        end;
    until (v = 1) or (begq = nil);
end;
```

Нарешті, можемо описати основну частину програми побудови максимального потоку в мережі:

```
begin
    for u ∈ V do
      for v ∈ V do F [u, v] := 0;
    repeat
      psa;                                          {Пошук допоміжної мережі}
      if lenght[t] <> ∞ then                        {Потік не є максимальним}
        begin
          maxpsa;            {Побудова максимального потоку в допоміжній мережі}
          for u ∈ XV do      {Перенесення потоку із допоміжної мережі в основну}
            for v ∈V do
              begin
                f [u, v] := f [u, v] + xf [u, v];
                if f [u, v] > c[u, v] then
```



```
                  begin
                    f [v, u] := f [v, u] – (f [u, v] – c[u, v]);
                    f [u, v] := c [u, v]);
                  end;
              end;
        end;
    until lenght[t] = ∞;
end.
```

Повну реалізацію на Паскалі ілюструє програма 3.4.

```
Program Ex3_4;
  uses crt;
  const max_n = 20;
  type p = ^core;                          {Елементи стека, черги}
       core = record
                num: byte;
                nextq: p;
                prevq: p;
              end;
  var  n: integer;
       W, XV, XN : set of byte;
       u, v, i, Pm, r, delta: byte;
       f, c, xc, xf: array [1..max_n, 1..max_n] of byte;
       lenght, Pr, P_in, P_out , Q: array [1..max_n] of byte;
       begq, endq, x, stk: p;

{Процедура додає значення до черги}
procedure addq(a: byte);
begin
    new(x);
    x^.num := a;
    if endq <> nil then
         endq^.nextq := x
      else
         begq := x;
    x^.nextq := nil;
    endq := x;
end;

{Функція вилучає значення з черги}
function outq: byte;
begin
    outq := begq^.num;
    x := begq;
    begq := begq^.nextq;
```

```
    if begq = nil then endq := nil;
    dispose(x);
end;

{Процедура додає значення до стека}
procedure push(a: byte);
begin
    new(x) ;
    x^.num := a;
    x^.nextq := stk;
    stk := x;
end;

{Функція вилучає елемент зі стека}
function pop: byte;
begin
    pop := stk^.num;
    x := stk;
    stk := stk^.nextq;
    dispose (x) ;
end;

{Функція будує додаткову ациклічну мережу}
procedure psa;
begin
    writeln;
    for u := 1 to n do
      begin
        lenght[u] := 255;
        for v := 1 to n do
          begin
            xc[u, v] := 0; xf [u, v] := 0;
          end;
      end;
    begq := nil;
    endq := nil;
    addq(1);
    XV := [];
    lenght[1] := 0;
    while begq <> nil do
     begin
      u := outq; XV := XV + [u];
      for v := 1 to n do
        begin
          if c[u, v] > 0 then
            if (lenght[u] < lenght[v]) and (lenght[v] <= lenght[n]) and (f [u, v] < c[u, v]) then
```



```
              begin
                if lenght[v] = 255 then addq(v);
                lenght[v] := lenght[u] + 1;
                xc[u, v] := c[u, v] – f [u, v];
              end;
            if c[v, u] > 0 then
              if (lenght[u] < lenght[v]) and (lenght[v] <= lenght[n]) and (f [v, u] > 0) then
                begin
                  if lenght[v] = 255 then addq(v);
                  lenght[v] := lenght[u] + 1;
                  xc[u, v] := xc[u, v] + f [v, u];
                end;
        end;
    end;
end;

{Побудова псевдомаксимального потоку в допоміжній ациклічній мережі}
procedure maxpsa;
begin
    stk := nil;
    for v := 1 to n do
      if v in XV then
          begin
            P_in[v] := 0;
            P_out[v] := 0;
            if v = 1 then P_in[v] := 255
                else
                    for u := 1 to n do P_in[v] := P_in[v] + xc[u, v];
            if v = n then P_out[v] := 255
                else
                    for u := 1 to n do P_out[v] := P_out[v] + xc[v, u];
            if P_in[v] < P_out[v] then Pr[v] := P_in[v]
                else Pr[v] := P_out[v];
            if Pr[v] = 0 then
              begin
                push(v);
                xc[v, u] := 0;
              end;
          end;
    XN := XV;
    while stk <> nil do
      begin
          v := pop;
          XN := XN – [v];
          for u := 1 to n do
            begin                                                    {*}
              if xc[u, v] > 0 then
```

```
            begin
              P_out[u] := P_out[u] – xc[u, v];
              xc[u, v] := 0;
              if P_out[u] = 0 then push(u);
            end;
          if xc[v, u] > 0 then
            begin
              P_in[u] := P_in[u] – xc[v, u];
              xc[v, u] := 0;
              if P_in[u] = 0 then push(u);
            end;
        end;                                                    {*}
    end;
  for u := 1 to n do
    for v := 1 to n do
      if v in XV then Q[v] := 0;
  if XN <> [] then                                  {Поток не є максимальним}
    begin
      Pm := 255;
      for v := 1 to n do
        if v in XN then
          if Pr[v] < Pm then
            begin
              r := v;
              Pm := Pr[r];
            end;
      endq := nil;
      begq := nil;
      addq(r);
      Q[r] := Pm;
      repeat
        v := outq;
        P_in[v] := P_in[v] – Q[v];
        P_out[v] := P_out[v] – Q[v];
        Pr[v] := Pr[v] – Q[v];
        if v = n then Q[v] := Pm
          else
            begin
              for u := 1 to n do
                if xc[v, u] > 0 then
                  while Q[v] > 0 do
                    begin
                      if Q[u] = 0 then addq(u);
                      if Q[v] < (xc[v, u] – xf [v, u]) then delta := Q[v]
                        else delta := (xc[v, u] – xf [v, u]);
                      xf [v, u] := xf [v, u] + delta;
```



```
                      Q[v] := Q[v] – delta;
                      Q[u] := Q[u] + delta;
                      if xf [v, u] = xc[v, u] then xc[v, u] := 0;
                    end;
            end;
        until (v = n) or (begq = nil);
      end;
    endq := nil;
    begq := nil;
    addq(r) ;
    Q[r] := Pm;
    repeat
      v := outq;
      if v <> r then
        begin
          P_in[v] := P_in[v] – Q[v];
          P_out[v] := P_out[v] – Q[v];
          Pr[v] := Pr[v] – Q[v];
        end;
      if v = 1 then Q[v] := 0
        else
          begin
            for u := 1 to n do
              if xc[u, v] > 0 then
                while Q[v] > 0 do
                  begin
                    if Q[u] = 0 then addq(u);
                    if Q[v] < (xc[u, v] – xf [u, v]) then delta := Q[v]
                      else
                        delta := (xc[u, v] – xf [u, v]);
                    xf [u, v] := xf [u, v] + delta;
                    Q[u] := Q[u] + delta;
                    Q[v] := Q[v] – delta;
                    if xf [u, v] = xc[u, v] then xc[u, v] := 0;
                  end;
          end;
    until (v = 1) or (begq = nil);
    writeln('Auxiliration is: ');
    for u := 1 to n do
      for v := 1 to n do
        if xf [u, v] <> 0 then write('xf [', u, ', ', v, '] = ', xf [u, v], ' ');
end;

{Процедура читає граф з файла}
procedure ReadGraph(fname: string);
  var  inf: text;
       i, j: integer;
```

```
        d: integer;
 begin
     assign(inf, fname);
     reset(inf);
     readln(inf, n);
     for i := 1 to n do
       begin
         for j := 1 to n do
           begin
             read(inf, d);
             c[i, j] := d;
             f[i, j] := 0;
           end;
         readln(inf);
       end;
     close(inf);
 end;

 begin
     ReadGraph('input.txt') ;
     writeln('Програма будує максимальний потік з вершини 1 до ', n) ;
     repeat                                                    {Головний цикл}
       psa;
       if lenght[n] <> 255 then
         begin
           maxpsa;                                  {Побудова максимального потоку}
           for u := 1 to n do
             for v := 1 to n do
               begin
                 f[u, v] := f[u, v] + xf[u, v];
                 if f[u, v] > c[u, v] then
                   begin
                     f[v, u] := f[v, u] - (f[u, v] - c[u, v]);
                     f[u, v] := f[u, v] - (f[u, v] - c[u, v]);
                   end;
               end;
         end;
     until lenght[n] = 255;
     writeln;
     writeln('Максимальний потік: ');
     for u := 1 to n do
       for v := 1 to n do
         if f[u, v] <> 0 then
           write('flow [', u, ', ', v, '] = ', f[u, v], ' ');
     writeln;
 end.
```

Програма 3.4. Знаходження максимального потоку в мережі

Задачі та вправи до розділу 3

1. Розробіть алгоритм складності $O(n)$ та напишіть програму перевірки, чи містить даний орієнтовний граф, заданий матрицею суміжності, вершину, яка є суміжною з іншими $n - 1$ вершинами і з якої не виходить жодна дуга.
2. За допомогою методу пошуку в глибину розробіть алгоритм побудови остового лісу для неорієнтованого графа.
3. Проаналізуйте, як зміняться описані алгоритми, в основу яких покладено метод, що відрізняється від стандартного пошуку в ширину тільки тим, що вдруге досягнуту вершину заносять у стек.
4. Дослідити метод пошуку в графі, при якому відвідані, але ще не використані вершини зберігають в черзі.
5. За допомогою методу пошуку в глибину розробіть алгоритм перевірки на ациклічність орієнтованого графа.
6. Побудуйте алгоритм та напишіть програму, що використовує пошук в глибину для знаходження у зв'язному графі всіх циклів. Дослідіть часову складність алгоритму.
7. Мостом графа G називається кожне ребро, вилучення якого зумовлює збільшення числа зв'язних компонент графа. Розробіть алгоритм, що використовує пошук в глибину для знаходження всіх мостів графа за час $O(m + n)$.
8. За допомогою методу пошуку в глибину розробіть ефективний алгоритм, що знаходить такий порядок вузлів ациклічного орієнтованого графа, для якого $v < w$, якщо із v у w веде шлях ненульової довжини.
9. Граф $G = \langle V, E\rangle$ називають дводольним, якщо існує розбиття $V = A \cup B$, $A \cap B = \varnothing$, таке що кожне ребро $e \in E$ має вигляд $e = \{a, b\}$, $a \in A$, $b \in B$. Розробіть два алгоритми, що перевіряють, чи буде такий граф дводольним за час $O(m + n)$: один, оснований на пошуку в глибину, інший – на пошуку в ширину.
10. Написати алгоритм, що перевірить за час $O(m + n)$ ациклічність даного орієнтованого графа.
11. Довести, що в графі без контурів з від'ємною довжиною рівняння
    $$d(s, s) = 0,$$
    $$d(s, v) = \min\{d(s, u) + a(u, v) : u \in V \setminus \{v\}\}$$
    однозначно визначають відстані $d(s, v)$, $v \in V$.
12. Напишіть алгоритм пошуку мінімальних шляхів від вершини-джерела до всіх інших вершин графа $G = (V, E)$ за час $O(m \log n)$, де $|V| = n$, $|E| = m$.
13. Модифікуйте алгоритм Флойда так, щоб окрім відстаней $D[i,j]$ він визначав матрицю $M[i,j]$, де $M[i,j]$ – найбільший номер вершини деякого найкоротшого шляху з v_i в v_j ($M[i,j] = 0$, якщо цей шлях не містить проміжних вершин).
14. Під пропускною здатністю шляху розумітимемо найменшу вагу дуги цього шляху. Написати алгоритм, що визначає найбільшу пропускну здатність шляхів між фіксованою парою вершин.
15. Довести, що якщо кожне збільшення потоку відбувається вздовж найкоротшого ланцюга зростання, то починаючи з довільного потоку, ми отримаємо максимальний потік з використанням не більш ніж $mn/2$ ланцюгів зростання.
16. Модифікуйте алгоритм Дініца для мережі з пропускною здатністю кожної дуги, що дорівнює нулю або одиниці.
17. Показати, що якщо в мережі з цілочисловою пропускною здатністю потенціал кожної вершини, відмінної від s і t, дорівнює нулю або одиниці, то алгоритм Дініца можна реалізувати за час $O(n^{1/2}m)$.

Розділ 4

КОМБІНАТОРНІ АЛГОРИТМИ РОЗМІЩЕННЯ І РОЗБИТТЯ

Для розв'язання різних задач методом перебору слід ефективно реалізовувати такі класичні комбінаторні задачі, як розміщення, генерування перестановок, побудова підмножин з певними ознаками деякої множини, генерування k-елементних підмножин і розбиття множин [66]. Всі вони мають давню історію дослідження і можуть мати нетрадиційне застосування. Наприклад, в середньовічній Великій Британії виник спеціальний тип дзвону в церквах, головна особливість якого полягала у вибиванні на n різних дзвонах всіх $n!$ перестановок без повторень.

Для більшості комбінаторних алгоритмів характерною є значна часова складність. Тому їх використання на найсучасніших комп'ютерах має бути обережним. Наприклад, всі перестановки 15-елементної множини обчислюватимуться на сучасному комп'ютері середньої потужності близько 20 год.

Розглянемо основні підходи до розв'язання задач цього типу.

4.1. Генерування підмножин множини

Для розв'язання багатьох задач слід вміти будувати всі підмножини $P(x)$ деякої множини $X = \{x_1, x_2, ..., x_n\}$. Постає питання, скільки таких множин? Кожній підмножині $Y \subseteq X$ поставимо у відповідність бінарну послідовність $b_1, b_2, ..., b_n$, де

$$b_i = \begin{cases} 0, & \text{якщо } x_i \notin Y, \\ 1, & \text{якщо } x_i \in Y. \end{cases}$$

Таких послідовностей буде 2^n.

Одним із найпростіших підходів до побудови $P(x)$ є такий [66]. Кожна бінарна послідовність $b_{n-1}, b_{n-2}, \ldots, b_0$ взаємно однозначно відповідає цілому числу, яке отримують із двійкового задання числа $b_{n-1}, b_{n-2}, \ldots, b_0$ в інтервалі $0 \le r \le 2^n - 1$, а саме числу $r = \sum_{i=0}^{n-1} b_i 2^i$. Отже, будуючи числа від 0 до 2^{n-1} шляхом додавання 1 до попереднього і переводячи їх у

двійкове задання, отримаємо всі підмножини n-елементної множини. Наприклад, для $n = 3$ такий алгоритм побудує такі бінарні послідовності:

```
0 0 0
1 0 0
0 1 0
1 1 0
0 0 1
1 0 1
0 1 1
1 1 1
```

Програма 4.1 реалізує цей алгоритм.

```
Program Ex4_1;
 const MAX_N = 100;
 var   b: array [1..MAX_N] of byte;                    {Масив бінарного зображення}
       n, j: integer;
       range, i: longint;

procedure IncB;                          {Процедура додає 1 до бінарного числа (масив b)}
 var i: integer;
begin
    for i := 1 to n do
      begin
        if (b[i] = 0) then
          begin
            b[i] := 1;
            exit;
          end;
        b[i] := 0;
      end;
end;

begin
    write('Введіть кількість елементів у множині: ');
    readln(n);
    range := 1;
    for i := 1 to n do
      range := range * 2;
    writeln('Генерування всіх ', range, ' підмножин: ');
    for i := 1 to n do
      b[i] := 0;
    for i := 1 to range do
      begin                                                {Виведення перестановки}
        for j := 1 to n do
        write(b[j]);
```

```
        writeln;                                    {b := b + 1}
        IncB;
      end;
end.
```

Програма 4.1. Генерування підмножин множини

4.2. Розбиття чисел

Розглянемо задачу: скількома способами можна записати задане натуральне число n у вигляді суми цілих чисел

$$n = b_1 + b_2 + \ldots + b_k, \qquad (4.1)$$

де k, b_1, b_2, ..., $b_k > 0$. Наприклад, 6 = 3 + 2 + 1. Слід також враховувати, що сума 3 + 2 + 1 = 3 + 1 + 2 = 2 + 3 + 1 = 2 + 1 + 3 = 1 + 2 + 3 = 1 + 3 + 2. Тому клас еквівалентних сум однозначно задають послідовністю a_1, a_2, ..., a_k, де $a_1 \geq \ldots \geq a_k$ і числа a_1, a_2, ..., a_k є числами b_1, b_2, ..., b_k, які розміщені за спадним порядком. Кожну послідовність a_1, a_2, ..., a_k називають розбиттям числа n на k доданків. Кількість розкладів числа n на k доданків позначають $P(n, k)$, а кількість всіх розкладів числа n позначають $P(n)$.

Зрозуміло, що $P(n) = \sum_{k=1}^{n} P(n,k)$, $n > 0$, вважаючи $P(0, 0) = P(0) = 1$.

Відомо багато цікавих результатів знаходження розбиття. Розглянемо один із них [66], в якому розбиття генеруватиметься за порядком, оберненим до лексикографічного порядку, тобто розбиття $n = c_1 + c_2 + \ldots + c_s$ буде побудовано (не обов'язково відразу) після розкладу

$$n = a_1 + \ldots + a_k \qquad (4.2)$$

тоді і тільки тоді, коли існує індекс $p \leq \min(k, s)$, такий що $c_p < a_p$ і $c_m = a_m$ для $1 \leq m < p$. Починаємо з розбиття, яке містить тільки один доданок, рівний n, а закінчуємо розбиттям, в якому $k = n$, $b_1 = b_2 = \ldots = b_k = 1$.

Наступний розклад за розбиттям (4.2) будуватиметься так. Відшукаємо розбиття, яке має найбільшу кількість початкових доданків, рівних початковим доданкам розкладу (4.2). Позначимо ці доданки відповідно через $a_1, \ldots, a_{t-1}$. Тоді решта доданків визначатиметься розкладом, що йде за розкладом $S = a_t + a_{t+1} + \ldots + a_k$. Ці умови визначають $t = \max\{i: a_i > 1\}$.

Тому розв'язання полягає у знаходженні розбиття, яке знаходиться безпосередньо за розкладом $S = a_t + \underbrace{1 + \ldots + 1}_{k-t \text{ разів}}$, $a_t > 1$ і набуває такого вигляду:

$$S = \underbrace{r + \ldots + r}_{\lfloor S/r \rfloor \text{ разів}} + (S \bmod r),$$

де $r = a_t - 1$.

Наприклад, розклади числа 8, виходячи з вищенаведених міркувань, такі:

перший розклад – 8 = 8
8 = 7 + 1
8 = 6 + 2
8 = 6 + 1 + 1
8 = 5 + 3
8 = 5 + 2 + 1
8 = 5 + 1 + 1 + 1
8 = 4 + 4
8 = 4 + 3 + 1
8 = 4 + 2 + 2
8 = 4 + 2 + 1 + 1
8 = 4 + 1 + 1 + 1 + 1
8 = 3 + 3 + 2
8 = 3 + 3 + 1 + 1
8 = 3 + 2 + 2 + 1
8 = 3 + 2 + 1 + 1 + 1
8 = 3 + 1 + 1 + 1 + 1 + 1
8 = 2 + 2 + 2 + 2
8 = 2 + 2 + 2 + 1 + 1
8 = 2 + 2 + 1 + 1 + 1 + 1
8 = 2 + 1 + 1 + 1 + 1 + 1 + 1
8 = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

Програма 4.2 демонструє один із варіантів запису цього алгоритму в Паскалі.

```
Program Ex4_2;
 uses Crt;
 const m = 1000;
 type int = 1..m;
      arr = array [int] of int;
 var  S: arr;              {S[1], ..., S[d] доданки розбиття, в якому S[1] > ... > S[d]}
      R: arr;              {R[i] > 0, величина якого визначає, скільки разів}
                           {доданок S[i] трапляється в розкладі}
      n, d, li, i, sum, q: integer;
      pl: char;
begin
   clrScr;
   writeln('Введіть значення числа n');
   read(n);
   S[1] := n; R[1] := 1; d := 1;                          {Перший розклад}
   writeln ('Перший розклад числа ', n, ' = ', S[1]);
   while S [1] > 1 do
```

```
begin
  sum := 0;                                  {Сума видалених доданків}
  if S[d] = 1 then
    begin                                    {Видалення доданків, рівних 1}
      sum := sum + R[d] * s[d];
      d := d - 1;
    end;
  sum := sum + S[d];
  R[d] := R[d] - 1;
  li := S[d] - 1;
  if R[d] > 0 then d := d + 1;
  S[d] := li;
  R[d] := sum div li;
  li := sum mod li;
  if li <> 0 then
    begin                                    {Додати останній доданок, рівний li}
      d := d + 1;
      S[d] := li;
      R[d] := 1;
    end;
  pi := ' + ';
  write('Черговий розклад ', n, ' = ');
  for i := 1 to d do
    for q := 1 to R[i] do
      begin
        if (i = d) and (q = R[i]) then pi := ' ';
        write(' ', S[i], ' ', pi);
      end;
  writeln;
end;
end.
```

Програма 4.2. Розбиття чисел

4.3. Генерування перестановок

4.3.1. *Основні поняття*

Однією з найвідоміших в комбінаториці є задача розміщення [2, 66]. Потрібно визначити число можливих способів розміщення деяких об'єктів у фіксованому наборі *«ящиків»* за певних обмежень. Точніше можна дати таке визначення цієї задачі. Нехай задано дві множини X та Y, ($|X| = n$, $|Y| = m$). Потрібно з'ясувати, скільки існує функцій $f: X \to Y$, які задовольняють визначеним обмеженням. Відомо, що якщо немає ніяких обмежень на розміщення, тоді число всіх функцій $f: X \to Y$ дорівнює m^n.

У разі обмеження, коли в кожен ящик можна розмістити тільки по одному об'єкту, розміщення відповідають взаємно однозначним функціям та їхнє число (позначають $[m]_n$) визначають формулою

$$[m]_n = m(m-1)(m-n+1), \tag{4.3}$$

де $[m]_0 = 1$.

Якщо $m = n$, тоді кожна взаємно однозначна функція $f: X \to Y$ є відображенням множини X у множину Y. За умови $f: X \to X$ таке відображення називають перестановкою. У цьому разі із (4.3) маємо

$$[n]_n = n(n-1)\cdots 1.$$

Такий добуток називають n-факторіал і позначають $n!$. Можна довести [66], що число перестановок n-елементної множини дорівнює $n!$.

Для позначення конкретної перестановки використовують таблиці з двома рядками. Перший рядок містить всі елементи X, а другий – значення функції f на цих елементах, які стоять за таким порядком: якщо $x \in X$, тоді $f(x)$ розміщуватиметься під елементом x. Наприклад, розглянемо таку перестановку f множини $\{a,b,c,d\}$, що $f(a) = b$; $f(b) = a$; $f(c) = d$; $f(d) = c$. Її можна трансформувати в таку таблицю:

$$f = \begin{pmatrix} a & b & c & d \\ b & a & d & c \end{pmatrix}.$$

Якщо порядок елементів першого рядка фіксований, тоді кожній перестановці однозначно відповідає послідовність елементів другого рядка. Тому іноді довільну ін'єктивну послідовність довжини n елементів множини X називають перестановкою n-елементної множини X.

У подібних дослідженнях природа елементів множини X неістотна. Для простоти беруть $X = \{1,2,\ldots,n\}$. Множину всіх перестановок множини X позначають через S_n. Довільну перестановку $f \in S_n$ можна ототожнити з послідовністю $\langle a_1, a_2, \ldots, a_n \rangle$, де $a_i = f(i)$. Під суперпозицією перестановок f і q розуміють перестановку fq, яку визначають так: $fq(i) = f(q(i))$.

Перестановку

$$e = \begin{pmatrix} 1 & 2 & \ldots & n \\ 1 & 2 & \ldots & n \end{pmatrix}$$

називають тотожною. Зрозуміло, що для довільної $f \in S_n$ виконуватиметься $fe = ef = f$ і знайдеться така перестановка f^{-1}, що $ff^{-1} = f^{-1}f = e$. Перестановку f^{-1} у цьому разі називають оберненою до перестановки f Для знаходження оберненої перестановки достатньо поміняти рядки таблиці визначення f Для нашого прикладу

$$f^{-1} = \begin{pmatrix} b & a & d & c \\ a & b & c & d \end{pmatrix}.$$

З такого визначення перестановки випливає, що для довільних перестановок $f,g,h \in S_n$ виконуватимуться умови:

$$(fg)h = f(gh), \tag{4.4}$$

$$fe = ef = f, \tag{4.5}$$

$$f^{-1}f = ff^{-1} = e, \tag{4.6}$$

звідки можна дійти висновку, що S_n утворює симетричну групу степеня n відносно операції суперпозиції.

Довільну підмножину $G \subseteq S_n$, для якої виконуються умови

$$f,q \in G \Rightarrow fq \in G,$$
$$f \in G \Rightarrow f^{-1} \in G,$$

називають групою перестановок степеня n.

Кожну перестановку $f \in S_n$ можна також задавати графічно за допомогою такого орієнтованого графа $G = (V,E)$, для якого $V = X = \{1,2,\ldots,n\}$, а $\langle x,y\rangle \in E$ тоді і тільки тоді, коли $f(x) = y$. Із кожної вершини x виходить рівно одне ребро $\langle x,f(x)\rangle$ і в кожну вершину x входить рівно одне ребро $\langle f^{-1}(x),x\rangle$.

Зрозуміло, що для довільної вершини x в графі G знайдеться такий скінченний цикл $x,x_1,x_2,\ldots,x_e$, для якого виконуватимуться умови $x_1 = f(x)$, $x_2 = f(x_1)$, ..., $f(x_e) = x$, $(e \geq 0)$. Тому можна дійти висновку, що граф G містить деяке число елементарних циклів з різними множинами вершин, які в сумі дорівнюють всій множині X.

Довільну перестановку, яка є циклом довжини 2, називають транспозицією. Транспозицію сусідніх елементів позначають $[i\_\_i+1]$. Пару $\langle a_i,a_j\rangle$, $i<j$ називають інверсією перестановки $\langle a_i,\ldots,a_n\rangle$, якщо $a_i > a_j$. Число інверсій перестановки $f \in S_n$ позначають $I(f)$. Відомо [66], що довільну перестановку $f \in S_n$ можна задати у вигляді суперпозиції $I(f)$ транспозицій сусідніх елементів. Маємо ідею алгоритму побудови перестановок.

4.3.2. *Алгоритми генерування всіх перестановок*

Зупинимось на основних ідеях генерування послідовності всіх $n!$ перестановок n-елементної множини і опишемо алгоритм Джонсона – Троттера [130, 148] реалізації однієї з них.

Вважатимемо, що елементи множини $X = \{1,2,\ldots,n\}$ задають відповідними елементами масиву $P[1]$, $P[2]$, ..., $P[n]$. Елементарною операцією, що застосовують до масиву P, є операція одноелементної транспозиції (обмін значеннями змінних $P[i]$, $P[j]$, де $1 \leq i,j \leq n$). Згідно [66] позначають її $P[i] :=: P[j]$ для скорочення запису послідовності команд $temp := P[i]$; $P[i] := P[j]$; $P[j] := temp$, де $temp$ є допоміжною змінною.

На множині всіх перестановок (всіх послідовностей довжини n із множини X) поряд з лексикографічним порядком можна визначити антилексикографічний порядок (позначатимемо $\Diamond$):

$$(x_1,x_2,\ldots,x_n) \Diamond (y_1,y_2,\ldots,y_n) \Leftrightarrow$$

існує таке $k \leq n$, *що* $x_k > y_k$ *і* $x_e = y_e$ *для кожного* $e > k$.

Тоді із визначення випливає, що послідовність перестановок множини $\{1,\ldots,n\}$ має такі властивості:

1. У першій перестановці елементи розміщуватимуться за зростанням, в останній – за спаданням; інакше кажучи, остання перестановка буде оберненою до першої.

2. Послідовність можна розділити на n блоків завдовжки $(n-1)!$, які відповідатимуть спадним значенням елементу останньої позиції. Перші $n-1$ позицій блока, який містить елемент p в останній позиції, визначають послідовність перестановок множини $\{1,\ldots,n\}\backslash\{p\}$ за антилексикографічним порядком.

У цій схемі побудови чергової перестановки число транспозицій буде змінною величиною. Кожна друга перестановка отримуватиметься однією транспозицією $P[1] := : P[2]$. Поряд з ними будуть і такі, які затрачуватимуть $\lfloor (n-1)/2 \rfloor + 1 = \lfloor (n+1)/2 \rfloor$ транспозицій. Тому ліпшим є алгоритм, в якому кожну наступну перестановку отримували із попередньої за допомогою тільки однієї транспозиції.

Використовують два основні підходи.

У першому – всі перестановки $P[1]$, ..., $P[n]$ будують послідовною генерацією всіх перестановок $P[1]$, ..., $P[n-1]$ і заміною елементу $P[n]$ на один із елементів $P[1]$, ..., $P[n-1]$, визначений в допоміжному масиві $B[m,i]$, $1 \leq i, m \leq n$. Для того щоб ця схема працювала вірно, масив має бути визначеним так, щоб гарантувати на кожній транспозиції $P[B[m,i]] : = : P[m]$ введення в $P[m]$ нового елементу. Традиційно значення масиву $B[m,i]$ обчислюють динамічно як деяку функцію від m та i.

Напишіть алгоритм (програму) реалізації такого підходу до побудови перестановок і визначте його часову оцінку.

У другому підході (алгоритм Джонсона – Троттера) кожну наступну перестановку отримують з попередньої за допомогою однієї транспозиції сусідніх елементів. Розглянемо приклад [66]. Припустимо, що вже побудована послідовність перестановок елементів 2, 3, ..., n, яка задовольняє цим властивостям, наприклад, 2, 3 і 3, 2 для $n = 3$. Тоді бажану послідовність перестановок елементів 1, 2, ..., n отримаємо, вставляючи 1 усіма можливими способами в кожну перестановку елементів 2, 3, ...:

1	2	3
2	1	3
2	3	1
3	2	1
3	1	2
1	3	2

Зазначимо, що елемент 1 переміщується між першою і останньою позиціями позмінно вперед і назад $(n-1)!$ раз.

На основі цього прикладу можна запропонувати рекурсивний алгоритм. Проте у цьому разі послідовність перестановок можна виводити тільки після закінчення побудови. Потрібен значний об'єм пам'яті. Тому під час практичної реалізації перевагу надають нерекурсивному варіанту алгоритму. У цьому разі для кожного i, $1 \le i < n$, вводять: булеву змінну $PR[i]$, яка характеризує перенесення елементу як вперед ($PR[i] = true$), так і назад ($PR[i] = false$); змінну $e[i]$, що показує, яку із можливих $n-i+1$ позицій елемент i займає відносно елементів $i+1,\ldots,n$ на своєму шляху вперед або назад. Позицію елементу i в таблиці P можна визначити на основі його позиції в блоці, що містить $i, i+1, \ldots, n$, а також на основі числа елементів із $1, 2, \ldots, i-1$, які знаходяться зліва від цього блока. Це число (змінна x) обчислюють як число елементів $j < i$, які, рухаючись назад, досягли б свого крайнього лівого положення ($e[j] = n-j+1$, $PR[j] = false$). Кожну нову перестановку утворюють транспозицією найменшого з елементів j, який ще не був у крайній позиції ($e[j] < n-j+1$) з його лівим або правим сусідом. Програма 4.3 описує цей алгоритм генерування всіх перестановок.

```
Program Ex4_3;
 const m = 1000;
 type arr = array [1..m] of integer;
      ar = array [1..m] of boolean;
 var  x, i, n, l, k, temp: integer;
      P, C: arr;
      PR: ar;
begin
   writeln('Введіть значення n');
   readln(n);
   for i := 1 to n do                          {Початкова ініціалізація}
     begin
       P[i] := i; C[i] := 1; PR[i] := true;
     end;
   C[n] := 0;
   writeln('Перша перестановка');
   for l := 1 to n do
     write(P[l], ' ');
   i := 1;
   while i < n do
     begin
       i := 1;
       x := 0;
       while C[i] = n - i + 1 do
         begin
           PR[i] := not(PR[i]); C[i] := 1;
           if PR[i] then x := x + 1;
           i := i + 1
         end;
       if i < n then
         begin                                 {Виконання транспозиції}
           if PR[i] then k := C[i] + x
             else k := n - i + 1 - C[i] + x;
           temp := P[k];
           P[k] := P[k + 1];
           P[k + 1] := temp;
           writeln('Чергова перестановка');
           for l := 1 to n do
             write(P[l], ' ');
           C[i] := C[i] + 1
         end
     end
end.
```

Програма 4.3. Реалізація на Паскалі алгоритму побудови перестановок

Задачі та вправи до розділу 4

1. У змаганні беруть участь 8 спортсменів. Скількома способами можна розподілити медалі (золоті, срібні та бронзові)?
2. Скільки чисел між 1000 і 10000 складається з непарних цифр, скільки з різних цифр?
3. Скільки існує можливих результатів, якими можуть закінчитися змагання, в яких стартує 10 чоловік, за трьома видами спорту, якщо кожна людина стартує в одному довільно вибраному виді спорту? (Під результатом змагання ми розумітимемо розподіл місць для всіх спортсменів, що стартують у кожному виді).
4. Скільки паліндромів довжини n можна утворити, використовуючи 26 літер алфавіту? (Паліндромом називають довільний вираз, який можна прочитати однаково як зліва направо, так і в оберненому напрямку, наприклад *did*).
5. Довести, що перестановки $f, g \subseteq S_n$ є перестановками одного і того самого типу тоді і тільки тоді, коли існує перестановка $h \subseteq S_n$, така що $g = hfh^{-1}$.

Розділ 5

МЕТОД «РОЗДІЛЯЙ І ПАНУЙ»

5.1. Загальна схема методу

Нехай нам потрібно розв'язати на ЕОМ деяку задачу на вхідних даних завдовжки n. Іноді в обчисленні задачі виникає проблема, пов'язана з розміром даних: дані можуть не поміститися в пам'яті існуючого обчислювального пристрою або їх природа така, що задачу ліпше обчислювати окремо на виділених певним чином піднаборах вхідних даних, а потім скомбінувати побудову загального розв'язання з отриманих часткових. Для розв'язання таких задач використовують метод [9, 62, 127], що дістав назву «розділяй і пануй» (РП). Часто для розв'язання підзадачі слід знову використовувати метод РП. Коли такий поділ має характер базової проблеми, тоді отримуємо рекурсивні алгоритми. Спробуємо формалізувати вищезазначене [127].

Нехай вхід задачі завдовжки n задано даними глобального масиву $A(1..n)$, а загальна проблема, яку потрібно вирішити на цих даних, описує процедура *dandc*. Параметри процедури $dandc(1,n)$ свідчать про розв'язання цієї проблеми на початковому вході завдовжки n, а $dandc(p,q)$ визначає розв'язання підзадачі на вхідних даних, розмір яких визначається величинами індексів масиву $A(p..q)$, $1 \le p \le q \le n$. Булева функція $small(p,q)$ визначає достатність (*true*) поділу даних для розв'язання задачі, тобто на вході завдовжки $q - p + 1$ можна отримати якийсь проміжний розв'язок за допомогою функції G. Якщо припустити, що значенням $devide(p,q)$ є таке m, що розділяє вхід $A(p,q)$ на дві підзадачі $A(p..m)$ і $A(m+1,q)$, а x та y – це розв'язання задачі на цих підвходах, тоді функція $combine(x,y)$ буде функцією побудови загального розв'язання на $A(p,q)$. Нехай в головній програмі описано:

```
...
const n = 10000;
type int = 1..n;
var A: array [1..n] of ... ;
...
```

Тоді наступна процедура описуватиме структуру узагальненого абстрактного алгоритму обчислення за методом РП:

```
procedure dandc (p, q: int);
  var m: int;
begin
  if small(p, q) then G(p, q)
    else
      begin
        M := devide(p, q);                                  {p <= m < q}
        combine(dandc(p, m), dandc(m + 1, q))
      end;
end.
```


Враховуючи рекурсивну природу алгоритмів, які ґрунтуються на методі РП, зрозуміло, що для визначення часових оцінок можна використовувати техніку розв'язання рекурентних рівнянь [9, 62], побудова яких описується такою схемою:

$$T(n) = \begin{cases} g(n), & \text{якщо } n \text{ достатнє для розв'язання,} \\ 2T(n/2) + f(n) & \text{в іншому разі,} \end{cases}$$

де $g(n)$ є часовою оцінкою розв'язання задачі на досить малому (достатньому) вході, а $f(n)$ є часом, потрібним для комбінації загального розв'язання з підрозв'язань.

Виділяють [62] три основні підходи до розв'язання рекурентних рівнянь: метод підстановки, метод ітерацій та використання основної теореми про рекурентні співвідношення.

Перший метод передбачає насамперед вгадування оцінки (розв'язок), а потім доведення її за методом математичної індукції з підстановкою оцінки у праву частину співвідношення. Важливим у цьому разі є вибір коефіцієнтів. Вони мають бути такими, щоб справджувалось індуктивне доведення. Зрозуміло, що методом математичної індукції знаходять тільки нижні або верхні оцінки. Наприклад [62], для знаходження верхньої оцінки функції

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

можна покласти, що $T(n) = O(n \log n)$, тобто, враховуючи визначення часової оцінки, можемо вважати, що $T(n) \le cn \log n$, для деякого $c > 0$. Зробимо гіпотезу індукції. Нехай оцінка правильна для $\lfloor n/2 \rfloor$, тобто справджується $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor$. Підставимо її в базове співвідношення

$$T(n) \le 2\left(c \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor\right) + n \le cn \log(n/2) + n =$$
$$= cn \log n - cn \log 2 + n = cn \log n - cn + n \le cn \log n.$$

Останній перехід є вірним при $c \ge 1$.

Перевірити базис індукції для $n = 1$ не вдається, $\log 1 = 0$. Але для доведення асимптотичної оцінки досить доведення для n, починаючи з деякого фіксованого. Можна підібрати деяке c, для якого $T(n) \le cn \log n$ буде правильна для $n = 2$ і $n = 3$.

Отже, оцінку доведено.

Виникає питання: як вгадувати оцінку? Чітких інструкцій не існує. Пропонують тільки рекомендації. Наприклад, можна використовувати аналогію, заміну змінних або послідовні наближення.

У другому підході «розгортають» рекурентну формулу у вигляді деякої суми, яку потім можна оцінити, використавши відомі математичні результати. Побудову часової оцінки на основі цього підходу демонструємо далі, на прикладі оцінювання процедури maxmin.

Використати ж основну теорему про рекурентні співвідношення можна у разі, коли вони мають такий вигляд:

$$T(n) = aT(n/b) + f(n),$$

де $a \geq 1$ і $b > 1$ – деякі константи, а f – додатна (хоча б для великих значень аргументу) функція. Теорема 5.1 дає загальну формулу, яку можна застосовувати в конкретних випадках, без доведень [62].

Теорема 5.1. Нехай $a \geq 1$ і $b > 1$ – константи, $f(n)$ – функція, $T(n)$ визначають для невід'ємних n формулою $T(n) = aT(n/b) + f(n)$, де під n/b розуміють або $\lfloor n/b \rfloor$, або $\lceil n/b \rceil$. Тоді

1) якщо $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ для деякого $\varepsilon > 0$, тоді $T(n) = \Theta\left(n^{\log_b a}\right)$;

2) якщо $f(n) = \Theta\left(n^{\log_b a}\right)$, тоді $T(n) = \Theta\left(n^{\log_b a} \log n\right)$;

3) якщо $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ для деякого $\varepsilon > 0$ і якщо $af(n/b) \leq cf(n)$ для деякої константи $c < 1$ і достатньо великих n, тоді $T(n) = \Theta\left(f(n)\right)$.

Наведемо приклад [62] застосування теореми. Нехай маємо співвідношення $T(n) = 9T(n/3) + n$, у цьому разі $n^{\log_b a - \varepsilon} = n^{\log_3 9} = \Theta\left(n^2\right)$. Оскільки $f(n) = O\left(n^{\log_3 9 - \varepsilon}\right)$ для $\varepsilon = 1$, тоді можна застосувати перше твердження теореми та дійти висновку, що $T(n) = \Theta\left(n^2\right)$.

Розглянемо перехід від загальної схеми використання методу РП до розв'язання конкретних задач на прикладі розв'язання класичних задач (бінарного пошуку, знаходження максимуму та мінімуму, впорядкування).

5.2. Бінарний пошук

Нехай маємо послідовність a_i, $i = 1..n$ елементів, які впорядковані за зростанням. Потрібно розв'язати задачу належності елементу x цій послідовності чисел, інакше кажучи, потрібно визначити таке j, що $a_j = x$, або присвоїти $j = 0$, якщо послідовність не має елементу, рівного x.

Для зручності програмування бажано ввести якийсь формальний опис, що характеризував би загальну задачу і давав би змогу просто описувати розподіл її на підзадачі.

Наприклад, розв'язання нашої задачі бінарного пошуку позначимо так:

$$I = (n, a_1, a_2, a_3, ..., a_n, x).$$

Для його знаходження використаємо метод РП. Тоді його початкове застосування можна описати у такий спосіб:

$$I = I_1 \cup I_2 \cup I_3,$$

де $I_1 = (k - 1, a_1, ..., a_{k-1}, x)$, $I_2 = (1, a_k, x)$, $I_3 = (n - k, a_{k+1}, ..., a_n, x)$, $\cup$ – позначає операцію об'єднання.

Якщо $x = a_k$, тоді $j = k$, і підзадачі I_1 та I_3 не потрібно розв'язувати. В іншому разі визначають ту підзадачу, де можливий розв'язок: якщо $x < a_k$, тоді потрібно шукати розв'язок у аналогічний спосіб в I_1, якщо $x > a_k$, тоді в I_3. У разі подальшого перегляду підзадач I_1 або I_3 і переходу правої межі I_1 через ліву межу I_2 процес пошуку закінчується невдало, тому $j := 0$.

Якщо k завжди вибирають у такий спосіб, що a_k є серединним елементом ($k = \lfloor (n + 1) / 2 \rfloor$), тоді алгоритм результуючого пошуку називають бінарним пошуком.

Приклад 5.1. Розглянемо таку гру. Грають дві особи. Одна вибирає із заданого числового відрізка певне число (загадане), а інша має відгадати, яке число (вгадане) загадала перша. За невдалої спроби перша особа каже, що вгадане число більше або менше за загадане. Можливо, за першою ж спробою можна одразу вгадати число, і гру буде закінчено, але це малоймовірно. Можна перебирати числа без будь-якого алгоритму, але зрозуміло, що в цьому разі процес угадування затягнеться до перебору всіх чисел зазначеного відрізка. Тому бажано використати алгоритм, який би обмежував цей перебір. Одним з таких алгоритмів є алгоритм бінарного пошуку.

Наприклад, заданий відрізок становить множину цілих чисел від 1 до 1000. Загадане число – 60. Відгадувач називає число 500, тому що 500 є серединою зазначеного відрізка (ділення націло). Перша особа каже, що загадане число менше. Відгадувач називає число 250, з тих самих міркувань. Загадане число менше, тому відгадувач вказує число 125. Процес вгадування відповідно до нашого алгоритму зумовить далі таку послідовність чисел вгадування: $62 = \lfloor (125) / 2 \rfloor$, $31 = \lfloor (62) / 2 \rfloor$. Оскільки 31 менше за загадане, ми беремо середину відрізку 31..62 – це $46 = 31 + \lfloor (62 - 31) / 2 \rfloor$, аналогічно $52 = 46 + \lfloor (62 - 46) / 2 \rfloor$ і далі: $57 = 52 + \lfloor (62 - 52) / 2 \rfloor$, $59 = 57 + \lfloor (62 - 57)/2 \rfloor$, $60 = 59 + \lfloor (62 - 59) / 2 \rfloor$, бо 60 і є загадане число. Отже, гру закінчено.

Один із варіантів реалізації алгоритму бінарного пошуку наведено у програмі 5.1.

```
Program Ex5_1;
 const n = 100;                {Визначає максимальний розмір вхідної послідовності ai}
    type  arr = array [1..n] of integer;
          int = 0..n;
 var   i, j, li: int;
          a: arr;                    {Вхідну послідовність ai задають масивом з іменем a}
          x: integer;    {Позначає те значення, яке потрібно знайти в послідовності ai}

procedure binsr(n: int; a: arr; x: integer; var j: int);  {j позначає розв'язок задачі I}
var high, low, mid: int;
          {high – права межа послідовності розв'язання підзадачі,
           low – ліва межа послідовності розв'язання підзадачі,
           mid – права межа послідовності розв'язання підзадачі}
label exit;
                        {Позначку використовують для швидкого виходу з циклу
                        пошуку у разі отримання позитивного результату пошуку}
begin
    low := 1; high := n;
    while low <= high do
      begin
        mid := (low + high) div 2;
        if x < a[mid] then high := mid – 1
           else if x > a[mid] then low := mid + 1
              else
                 begin
                    j := mid;
                    goto exit
                 end;
      end;
    j := 0;
    exit:
end;

begin
    readln(li);                          {Вводять реальну довжину послідовності ai}
    for i := 1 to li do
      read(a[i]);
    readln;
    read(x);
    binsr(li, a, x, j);
    writeln(j)
end.
```

Програма 5.1. Реалізація алгоритму бінарного пошуку

У праці [127] можна знайти доведення наступних тверджень щодо реалізованого алгоритму бінарного пошуку.

Теорема 5.2. Процедура *binsrch*(A, n, x, j) працює коректно.

Теорема 5.3. Якщо $n \in \lfloor 2^{k-1}, 2^k \rfloor$, тоді *binsrch*$(A, n, x, j)$ зробить не більше k порівнянь у разі, якщо $x \in A(n)$, і $k-1$ або k, якщо $x \notin A(n)$. Інакше кажучи, час вдалого пошуку – $O(\log n)$, а час невдалого – $\Theta(\log n)$.

Тому дамо такі основні характеристики часу роботи алгоритму бінарного пошуку:

Вдалий пошук	**Невдалий пошук**
$O(1)$, $O(\log n)$, $O(\log n)$	$\Theta(\log n)$
Ліпший, середній, гірший	Ліпший, середній, гірший

Чи можна очікувати, що інший алгоритм буде значно швидшим, ніж бінарний пошук у найгіршому разі? Доведіть, що відповідь на це питання – «Ні».

5.3. Знаходження максимуму та мінімуму

Для більш детального висвітлення застосування загальної схеми побудови часових оцінок алгоритмів розв'язання за використання методу РП розглянемо на прикладі рекурсивного розв'язання задачі знаходження максимального та мінімального елементу в послідовності a_i з n елементів.

Введемо формальний запис постановки задачі $I = \max(n, a_1, a_2, a_3, ..., a_n)$ і $I' = \min(n, a_1, a_2, a_3, ..., a_n)$. Згідно із загальною схемою, задачу слід розбити на підзадачі (наприклад, дві).

Нехай

$$r = \lfloor n/2 \rfloor,\ I = I_1 \cup I_2,\ I' = I'_1 \cup I'_2,$$

де

$$I_1 = \max(r, a_1, a_2, a_3, \ldots, a_r), \quad I_2 = \max(n-r, a_{r+1}, a_{r+2}, a_{r+3}, \ldots, a_n),$$

$$I'_1 = \min(r, a_1, a_2, a_3, \ldots, a_r), \quad I'_2 = \min(n-r, a_{r+1}, a_{r+2}, a_{r+3}, \ldots, a_n).$$

Тобто

$$\max(I) = \max(\max(I_1), \max(I_2)),$$

а

$$\min(I') = \min(\min(I'_1), \min(I'_2)).$$

Для подальшого розбиття можна використати подібні поділи. Звідси дійдемо рекурсивної реалізації. Оберненим кроком рекурсії (розбиття далі не потрібне) буде наявність у черговій підпослідовності розбиття одного або двох елементів. У першому разі *max = min = значення елементу*, у другому – слід порівняти ці елементи для визначення *max* і *min*.

Приклад 5.2. Нехай існує вхідна послідовність цілих чисел (5, 2, 10, 11, 25, 1, 36, 44, 55, 28, 39, 4, 19). Розбиваємо цю послідовність за зазначеним алгоритмом доти, доки не отримаємо мінімальних підпослідовностей чисел (5), (2, 10), (11), (25, 1), (36) , (44, 55), (28, 39), (4, 19). Прийшли до умови спрацювання оберненого кроку рекурсії. Записуємо, яке число у кожній підпослідовності відповідає *max* і яке *min*; у разі, якщо є одне число, яке лишилося, ми записуємо його у *max* і *min*. Потім починаємо обчислювати передні рекурсивні виклики, тобто відшукуємо максимальний і мінімальний елементи з *max* і *min* попереднього розбиття. Процес продовжують, поки не лишиться одна пара *max* і *min*. Це і є відповіддю. Детальний опис розв'язання початкової задачі ілюструє рис. 5.1.

Нехай вхідну послідовність, як і в попередньому параграфі, задають зовнішнім (глобальним) масивом *A* типу *arr*. Тоді процедура *maxmin* реалізовуватиме описаний алгоритм застосування методу РП для знаходження максимального і мінімального елементу у вхідній послідовності.

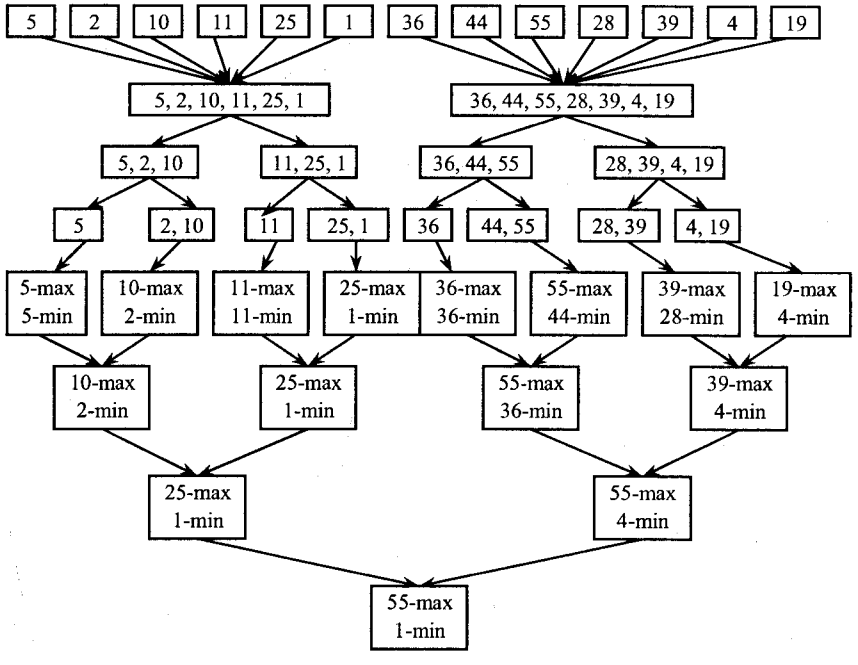


Рис. 5.1. Знаходження максимального і мінімального елементу згідно з методом РП


```
procedure maxmin ( i, j: int; a: arr; var fmin, fmax: integer);
      {i, j – характеризують межі (відповідно ліву і праву) розгляду чергової
       підпослідовності a_i , fmin, fmax позначають кінцевий результат}
 var  gmax, gmin, hmax, hmin: integer;
                    {gmin, gmax позначають проміжний максимум,
                     a hmin, hmax – проміжний мінімум}
      mid: int;     {mid – визначає середину чергової підпослідовності розгляду}

begin
  case (j – i) of                    {Обернений крок рекурсії}
       0:                            {У послідовності розгляду є лише один елемент}
          begin
             fmax := a[i]; fmin := a[j];
          end;
       1:                 {У послідовності розгляду залишилося два елементи}
          if a[i] < aj] then
             begin
                fmax := a[j]; fmin := a[i] ;
             end;
             else
                begin
                   fmax := a[i]; fmin := a[j];
                end;
     else
        begin
           mid := (i + j) div 2;
           maxmin(i, mid, a, gmin, gmax);
           maxmin(mid + 1, j, a, hmin, hmax);
           if gmin < hmin then fmin := gmin else fmin := hmin;
           if gmax > hmax then fmax := gmax else fmax := hmax;
        end;
end.
```

Побудуємо часову оцінку роботи процедури *maxmin*, використовуючи ітераційний підхід до побудови часових оцінок для методу РП. Після знаходження розв'язку підзадач для комбінування загального розв'язку нам слід зробити дві операції (порівняти два мінімуми і два максимуми), $f(n) = 2$. Обернений крок рекурсії (розмір задачі достатній для отримання часткового розв'язання) має два випадки (в послідовності залишився або один, або два елементи), тому $g(1) = 0$, а $g(2) = 1$. Поділ послідовності завжди робимо навпіл. Звідки

$$T(n) = \begin{cases} T(n/2) + T(n/2) + 2, & n > 2; \\ 1, & n = 2; \\ 0, & n = 1. \end{cases}$$

Припустимо, що $n = 2k$ для деякого додатного k, тоді

$$T(n) = 2T(n/2) + 2 = 2(2T(n/4) + 2) + 2 = (4T(n/4) + 4) + 2 = \ldots =$$

$$= 2^{k-1}T(2) + \sum_{i=1}^{k-1} 2^i = 2^{k-1} + 2^k - 2 = 3n/2 - 2.$$

5.4. Впорядкування

Розглянемо застосування методу РП для розв'язання задачі впорядкування послідовності із n цілих чисел.

5.4.1. *Обмінне впорядкування*

Використовуючи метод обмінного впорядкування за зростанням задачу $I = \text{sort}(n, a_1, a_2, \ldots, a_n)$ розділяють на $I_1 = \text{sort}(1, a_1)$ та $I_2 = \text{sort}(n-1, a_2, \ldots, a_n)$. Для цього потрібно знайти найменше значення серед $a_1, a_2, \ldots, a_n$ і поміняти місцями значення a_1 з цим найменшим значенням. Для розв'язання задачі розмірності $n-1$ можна знову використати переднє розбиття. Повторюючи процес n разів, отримаємо розв'язок початкової задачі.

Приклад 5.3. Нехай існує вхідна послідовність цілих чисел (15, 2, 11, 9). Потрібно впорядкувати цю послідовність за зростанням. Спочатку зафіксуємо крок нашого розподілу $i := 1$. i-й елемент заданої послідовності присвоюють змінній *min*, а індексу вибраного мінімального числа *index* присвоюють значення i. Далі порівнюємо змінну *min* з наступними числами послідовності. Як тільки якесь число менше за неї, змінна *min* набуває значення цього числа, а в змінну *index* заносять значення індексу цього числа в послідовності. І так до кінця послідовності. Після досягнення кінця послідовності міняємо місцями i-й та *index*-й елементи послідовності. Процес повторюють, доки i не буде дорівнювати n (n – кількість елементів в послідовності). Детальний опис процесу впорядкування ілюструє рис. 5.2.

Побудуємо часову оцінку роботи обмінного впорядкування. Для роботи алгоритму потрібно n раз відшукувати найменше значення та записувати його на потрібне місце. Для першого разу слід провести $n-1$ порівнянь, на другому кроці – виконати $n-2$ порівнянь і т. д. Таким чином, загальна кількість порівнянь визначатиметься формулою

$$(n-1) + (n-2) + (n-3) + \ldots + (n-(n-1)) + (n-n) =$$

$$= n*n - n*(n+1)/2 = n*(n-1)/2.$$

Тому часова оцінка алгоритму буде $O(n^2)$.

Недоліком розглянутого алгоритму є незначне зменшення розмірності задачі на кожному кроці її розбиття на підзадачі.

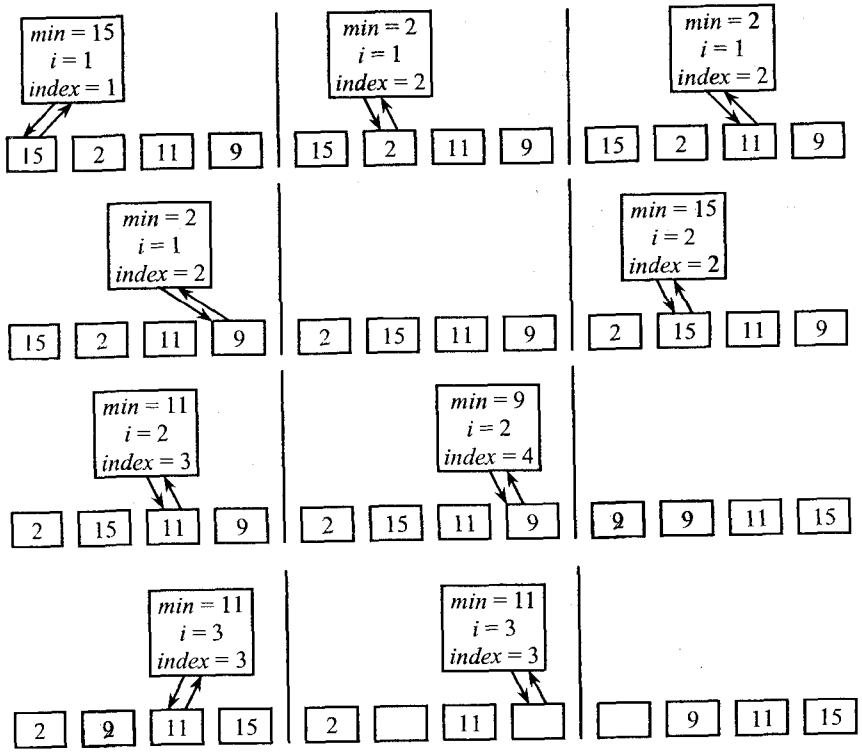


Рис. 5.2. Детальний опис процесу впорядкування

5.4.2. *Сортування злиттям*

Більш швидкий шлях розв'язання задачі впорядкування послідовності з n чисел – розбиття чергової задачі $I = \text{sort}(n, a_1, a_2, \ldots, a_n)$ на кожному кроці на дві рівні частини:

$$I_1 = \text{sort}(\lfloor n/2 \rfloor, a_1, a_2, \ldots, a_{\lfloor n/2 \rfloor}),\ I_2 = \text{sort}(\lceil n/2 \rceil, a_{\lceil n/2 \rceil}, \ldots, a_n).$$

Тоді задача комбінування загального розв'язання I полягатиме в злитті двох впорядкованих підпослідовностей. Для розв'язання підзадач I_1 та I_2 можна використати рекурсивний розв'язок базової задачі. Нехай $n = 2^k$ для деякого достатнього k. Тоді після $\log_2(n)$ поділів навпіл отримаємо підпослідовності значень завдовжки 1 (обернений крок рекурсії).

Приклад 3.4. Нехай існує вхідна послідовність цілих чисел (3, 20, 44, 1, 4). На першому кроці ми маємо розбити вхідну послідовність навпіл (за вищезазначеним алгоритмом), потім ці половинки ще навпіл і так доти, доки не залишиться по одному числу. Дійшли до оберненого кроку рекурсії. Впорядковуємо кожну отриману підпослідовність, а потім проводимо процедуру злиття.

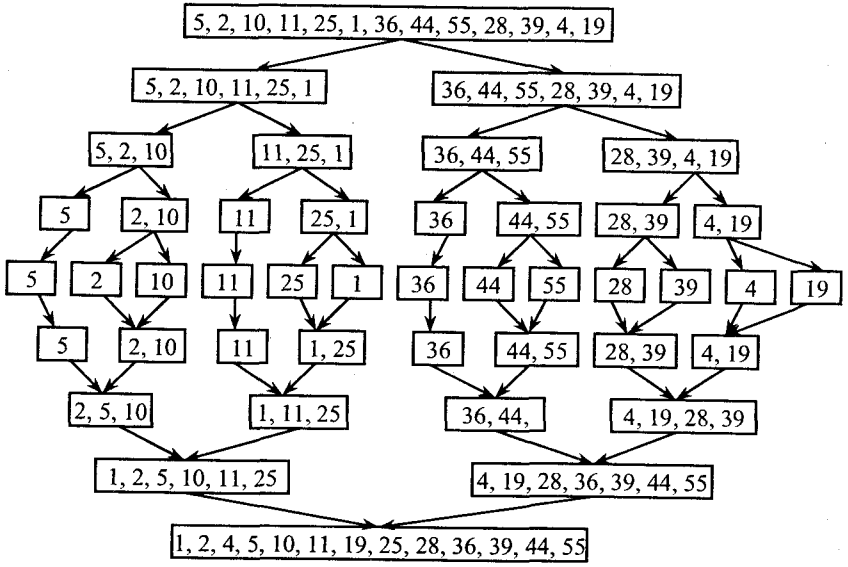


Рис. 5.3. Детальний опис процесу впорядкування

Детальний опис процесу впорядкування послідовності цілих чисел 5, 2, 10, 11, 25, 1, 36, 44, 55, 28, 39, 4, 19 ілюструє рис. 5.3.

Процедура *Mergsort* описує зазначений рекурсивний процес розв'язання поставленої задачі. Вона використовує процедуру *merge* для злиття докупи двох відсортованих підпослідовностей. Програма 5.2 реалізує запропонований алгоритм сортування злиттям (впорядкування фон Неймана).

```
Program Ex5_2;
  const n = 100;                    {Кількість елементів у послідовності впорядкування}
  type arr = array [1..n]of integer;
       int = 1..n;
  var  i, j, li: int;
       a: arr;

procedure merge(i, mid, j: int; var a: arr);
{merge – процедура злиття двох упорядкованих фрагментів масиву a; i – ліва межа
першого фрагмента масиву впорядкування, mid – права межа першого фрагмента
масиву впорядкування, mid + 1 – ліва межа другого фрагмента масиву впорядкування, j – права межа другого фрагмента масиву впорядкування; в масив a також
заносять і результат упорядкування}
  var  b: arr;                      {Додатковий масив злиття, проміжний}
       hh, ii, jj, kk: int;
begin
    hh := i; ii := i; jj := mid + 1;
```



```
    while (hh <= mid) and (jj <= j) do          {Поки ще не злитий повністю}
                                                {один із фрагментів}
      begin
{Зливати елементи фрагментів у проміжний масив, порівнюючи чергові порогові
елементи ще не злитих фрагментів, враховуючи їх упорядкування: доки елементи
першого масиву менші за перший ще не злитий елемент другого фрагмента, їх
переносять у проміжний масив; як тільки умова не виконається, починають
переносити елементи другого фрагмента; процес такого перемикання
продовжується, доки не вичерпають один із фрагментів}
        if a[hh] <= a[jj] then
          begin
            b[ii] := a[hh]; hh := hh + 1;
          end
          else
            begin
              b[ii] := a[jj]; jj := jj + 1;
            end;
        ii := ii + 1;
      end;
    if hh > mid then        {Визначають, який із фрагментів залишився ще не злитим}
      for kk := jj to j do
        begin       {Відповідний хвіст фрагмента дописують у проміжний масив}
          b[ii] := a[kk];
          ii := ii + 1;
        end
      else
        for kk := hh to mid do
          begin     {Відповідний хвіст фрагмента дописують у проміжний масив}
            b[ii] := a[kk];
            ii := ii + 1;
          end;
    for kk := i to j do
      a[kk] := b[kk];           {Проміжний масив переписують у результуючий масив}
end;

procedure mersort(i, j: int; var a: arr);
          {mergesort – процедура впорядкування методом злиття масиву a, i – ліва
          межа масиву впорядкування, j – права межа масиву впорядкування,
          в масив a також заносять і результат впорядкування}
    var mid: int;       {Індекс проміжного елементу серединного розбиття масиву
                         впорядкування}
begin
      if i < j then         {Умова рекурсивного виклику (ще є можливість розбиття)}
        begin
          mid := (i + j) div 2;                 {Знаходження середини поділу}
          mersort(i, mid, a);           {Рекурсивний виклик для лівого фрагмента}
```

```
        mersort(mid + 1, j, a);        {Рекурсивний виклик для лівого фрагмента}
        merge(i, mid, j, a)            {Злити в один упорядковані фрагменти}
      end;
end;

begin
   readln(li);
   for i := 1 to li do
     read(a[i]);
   mersort(1, li, a);
   for i := 1 to li do write(a[i], ' ')
end.
```

Програма 5.2. Впорядкування фон Неймана

Зрозуміло, що час, необхідний для процедури злиття, буде пропорційним до n, тобто $f(n) = cn$, де c – константа. Для впорядкування одного елементу також потрібен деякий константний час, нехай $g(n) = a$, де a – константа. Тоді часова оцінка роботи процедури *mersort* визначатиметься рекурентним співвідношенням

$$T(n) = \begin{cases} a, & n = 1; \\ 2T(n/2) + cn, \end{cases}$$

де a і c – константи.

Припустимо, що n є степенем двійки $n = 2^k$, тоді

$$T(n) = 2(2T(n/4) + cn/2) + cn = 4T(n/4) + 2cn = \ldots =$$
$$= 2^k T(1) + kcn = an + cn\log n.$$

Отже, якщо $2^k < n < 2^{k+1}$, тоді

$$T(n) \le T(2^{k+1}) \text{ і } T(n) = O(n\log n).$$

Задачі та вправи до розділу 5

1. Розробіть алгоритм множення двох n-розрядних двійкових чисел із застосуванням методу РП.
2. Модифікуйте розроблений алгоритм розв'язання задачі 1 для десяткових чисел.
3. Побудуйте часові оцінки розроблених алгоритмів множення чисел.
4. Нехай a, b і c – невід'ємні константи. Доведіть, що розв'язок рекурентних рівнянь

$$T(n) = \begin{cases} b, & n = 1; \\ aT(n/c) + bn, & n > 1, \end{cases}$$

де n є степінь числа c, має вигляд:

$$T(n) = \begin{cases} O(n), & \text{якщо } a < c; \\ O(n\log n), & \text{якщо } a = c; \\ O(n^{\log_c a}), & \text{якщо } a > c. \end{cases}$$

5. Нехай на вході маємо послідовність S із n елементів $a_1, a_2, \ldots, a_n$. На виході – отримати впорядковану послідовність S, використавши алгоритм процедури ШВИДКЕ_ВПОРЯДКУВАННЯ(S).

```
Procedure ШВИДКЕ_ВПОРЯДКУВАННЯ (S);
if S містить не більше одного елементу then return S
  else
     begin
       вибрати довільний елемент a із S;
       нехай S1, S2 і S3 послідовності елементів із S,
       відповідно менших, рівних і більших за a;
       return
         (ШВИДКЕ_ВПОРЯДКУВАННЯ (S1), S2,
              ШВИДКЕ_ВПОРЯДКУВАННЯ (S3))
     end.
```

Визначте середній час роботи алгоритму.

6. Розробіть алгоритм розв'язання задачі знаходження k-го найменшого елементу в n-елементній послідовності.

   k-м найменшим елементом послідовності $a_1, a_2, \ldots, a_n$ називають такий елемент b цієї послідовності, що $a_i < b$ не більше ніж для $k - 1$ значень i та $a_i \le b$ не менше ніж для k значень i.

   Для розв'язання використайте алгоритм: упорядкуйте цю послідовність за спаданням її елементів і візьміть k-й елемент. Розгляньте варіанти:

   а) алгоритм має використовувати $n\log(n)$ порівнянь;

   б) алгоритм, використовуючи стратегію методу РП, має знаходити k-й найменший елемент за $O(n)$ кроків.

7. У найгіршому випадку часова складність алгоритму *Mergesort* становить $O(n\log n)$. Який час він матиме в найліпшому випадку?
8. [Алгоритм Штрассена – множення матриць] Нехай A і B – дві матриці розмірності $(n \times n)$, де n – степінь числа 2. Відомо [9], що тоді кожну матрицю A і B розділяють на чотири $((n/2) \times (n/2))$-матриці і через них можна виразити добуток матриць A і B:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

де

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \quad C_{12} = A_{11}B_{12} + A_{12}B_{22},$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}, \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

Інакше кажучи, якщо розглядати A і B як (2×2)-матриці, елементами кожної з яких є $((n/2) \times (n/2))$-матриці, тоді їх добуток можна виразити через суми і добутки $((n/2) \times (n/2))$-матриць. Застосовуючи рекурсивно цей алгоритм, можна обчислити добуток двох матриць.

Напишіть програму обчислення добутку двох матриць за допомогою алгоритму Штрассена. Визначте часову оцінку алгоритму.

Розділ 6

ЗАДАЧІ ОПТИМІЗАЦІЇ

6.1. Загальна характеристика

Розглянемо ту частину теорії алгоритмів, яка пов'язана з проблемою знаходження екстремуму. Серед багатьох ідей знаходження екстремуму функцій і функціоналів можна виділити такі: лінійне програмування [8, 24, 75, 105], принцип максимуму Понтрягіна [82, 16, 17] і теорію локальних екстремумів [25, 87, 105]. Поряд із класичними ідеями оптимізації в останні десятиріччя почався розвиток ідей іншої природи. Це – ідеї послідовного аналізу варіантів, їх відсіву, послідовного звуження множини можливих розв'язків. Вони обумовили розвиток динамічного програмування.

Важливість розгляду оптимізаційних задач зазначав ще Гільберт. Розглядаючи свою знамениту програму, він плекав надію, що в XX ст. математики опанують методи розв'язання оптимізаційних задач. Це, дійсно, проблема, тому що обчислювальні характеристики більшості оптимізаційних задач належать до класу *NP*-повних задач. Отже, для їх розв'язання бажано використовувати і наближені евристичні алгоритми. Проте повною мірою його надії не виправдалися. Можна виділити тільки певні напрями оптимізації, в яких людство досягло істотного просування вперед у пошуку нових розв'язань. Наприклад, за останні десятиріччя значних успіхів було досягнуто в розділі теоретичної оптимізації, яка дістала назву конструювання і аналізу комбінаторних алгоритмів. Ефективні комбінаторні алгоритми широко застосовуються в багатьох галузях дискретної оптимізації.

6.2. Задачі оптимізації

Багато задач можна переформулювати у вигляді пошуку «найліпшої» конфігурації або множини параметрів для досягнення деякої цілі. Спробуємо прослідкувати ієрархічну структуру таких задач [88].

На верхньому щаблі ієрархії міститься загальна задача нелінійного програмування:

знайти таке x, яке буде мінімізувати $f(x)$ *за умов:*

$$\begin{aligned} g_i(x) &\geq 0 \text{ для } i = 1, \ldots, m, \\ h_j(x) &= 0 \text{ для } j = 1, \ldots, p, \end{aligned} \qquad (6.0)$$

де f, g_i, h_j – довільні функції параметра $x \in R^n$.

Методи розв'язання таких задач у більшості випадків мають ітераційний характер. Їх збіжність досліджують аналізом дійсної змінної. Впливає і тип функції.

Нехай задано функцію f, диференційовану на $[a, b]$. Її називають:

а) увігнутою на (a,b), якщо для довільних $x_1, x_2 \in (a,b)$, таких що $x_1 \neq x_2$: $f(x_2) \geq f(x_1) + f'(x_1)(x_2 - x_1)$;

б) опуклою на (a,b), якщо для довільних $x_1, x_2 \in (a,b)$, таких що $x_1 \neq x_2$: $f(x_2) \leq f(x_1) + f'(x_1)(x_2 - x_1)$, де $f'(x)$ – перша похідна.

Коли у (6.0) функція f опукла, функції g_i увігнуті і h_j – лінійні, тоді отримаємо задачу, яку називають задачею опуклого програмування. Для неї відомі достатні умови оптимальності – умови Куна–Таккера [48], і зі знаходження локальної оптимальності розв'язання випливає глобальна оптимальність.

Якщо у (6.0) f і всі g_i, h_j – лінійні, тоді отримаємо задачу лінійного програмування. Розв'язання задач такого типу вибирають із скінченної множини можливих розв'язань. Тому такі задачі називають ще комбінаторними. Зазначена скінченна множина кандидатів у розв'язки є вершинами опуклого багатогранника, який визначають лінійними обмеженнями.

Оптимальне розв'язання задачі лінійного програмування можна отримати за допомогою симплекс-методу Данціга [48]. Він потребує скінченної кількості кроків, тому що для поліпшення розв'язання перебирає вершини багатогранника, але часова складність має експоненційний характер. Проте нещодавно з'явився алгоритм еліпсоїдів [88], який гарантує знаходження оптимального розв'язку задачі за поліноміальний час. Рис. 6.1 ілюструє порівняльну діаграму вищезгаданих задач [88].

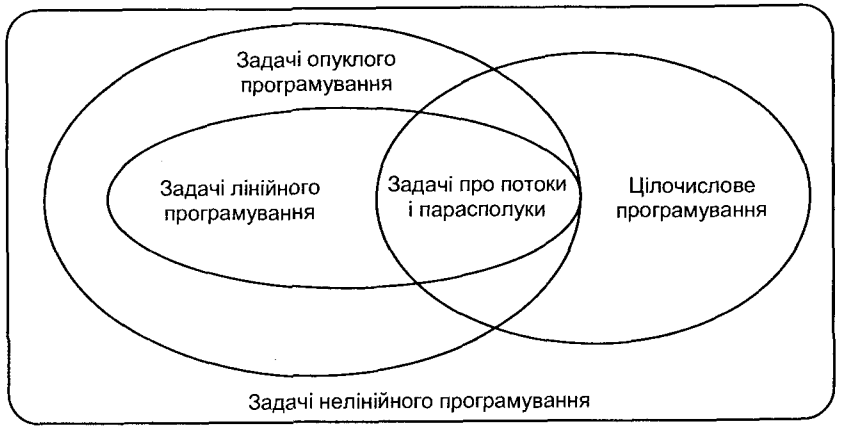


Рис. 6.1. Співвідношення оптимізаційних задач

Задачі про потоки і паросполуки можна розглядати як окремі випадки задач цілочисельного лінійного програмування. У них відшукують якусь «найліпшу» вартість за поліноміальний час за умови, що координати задають у цілих числах. Проте загальна задача цілочисельного програмування залишається *NP*-повною.

Як зазначалось, задачі оптимізації розподіляють на два класи: задачі з неперервними змінними і задачі з дискретними змінними (комбінаторні задачі). У неперервних задачах традиційно відшукують множину дійсних чисел. В комбінаторних задачах елементом пошуку є деякий об'єкт (ціле число, множина, перестановка, граф) із скінченної або, можливо, нескінченної множини. Методи розв'язання задач цих двох класів різної природи. Окреме важливе місце серед задач оптимізації посідає лінійне програмування. Задачі лінійного програмування є неперервними оптимізаційними задачами, які за своєю природою можна розглядати як комбінаторні. Тому є визначення задачі оптимізації [88], яке б задовольняло обидва класа.

Індивідуальна задача оптимізації – це пара (F, c), де F – довільна множина, область допустимих точок, а c – функція вартості, що виконує відображення $c : F \to R^1$. Потрібно знайти точку $f \in F$, для якої $c(f) \le c(y)$ для всіх $y \in F$. Таку точку f називають глобально-оптимальним розв'язком даної індивідуальної задачі або, коли не виникають непорозуміння, оптимальним розв'язком.

Індивідуальна задача – це задача із певного класу задач, для якої задані конкретні вхідні дані, на яких може бути отримано певний конкретний розв'язок. Тому під задачею оптимізації розумітимемо множину I індивідуальних задач оптимізації.

Ми вже розглядали задачу знаходження остових дерев графа. Спробуємо її дещо ускладнити. Розглянемо задачу знаходження мінімального остового дерева (*МОД*). Нагадаємо постановку задачі. Нехай $G\,(V, E)$ – зв'язний неорієнтований граф, для якого задана функція вартості, що відображає ребра в дійсні числа. Остовим деревом графа називають неорієнтоване дерево, що містить всі вершини графа. Вартість остового дерева визначають як суму вартостей його ребер. Наша мета – знаходження для G остового дерева мінімальної вартості. Зрозуміло, що її можна вирішити шляхом перебору: побудувати всі остові дерева, а потім вибрати з них дерево мінімальної вартості. Очевидно також, що часова оцінка такого алгоритму має експоненційний характер. Тому бажано було б її поліпшити. Спробуємо звести нашу задачу до розв'язання задачі лінійного програмування, використавши результати досліджень Е. Рейнгольта [88].

В індивідуальній задачі *МОД* задані ціле число $n > 0$ (кількість вершин у графі) і симетрична $(n \times n)$-матриця відстаней $[d_{ij}]$, де $d_{ij} \in Z^+$. Граф задають модифікованою матрицею суміжності. Модифікація полягає в

тому, що якщо дві вершини i та j графа з'єднані ребром, тоді d_{ij} визначатиметься вартістю цього ребра, в іншому разі d_{ij} дорівнює нулю. Задача полягає у знаходженні остового дерева на n вершинах, яке буде мати найменшу вартість. Переведемо останню в терміни індивідуальної задачі оптимізації:

$$F = \{\textit{Усі остові дерева}\ (V, E), \text{ де } V = \{1, 2, \ldots, n\}, c\colon (V, E) \to \sum_{[i,j]\in E} d_{ij}\}.$$

Проте цю суто комбінаторну задачу можна також розглядати і як задачу лінійного програмування (*ЛП*).

Нехай m, n – додатні цілі числа, $b \in Z^m$, $c \in Z^n$ і $A \in$ $(m \times n)$-матриця з елементами $a_{ij} \in Z$. Тоді індивідуальну задачу *ЛП* визначають так:

$$F = \{x\colon x \in R^n, Ax = b, x \ge 0\};\ c\colon x \to c'x.$$

Приклад 6.1. [88] Повернемось до задачі *МОД* з $n = 3$ точками (вершинами). Розглянемо відповідний граф, зображений на рис. 6.2.

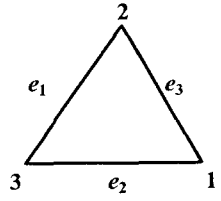


Рис. 6.2. Граф $G = (E, V)$

Для нього можна побудувати три остових дерева (рис. 6.3). Їх також можна вважати точками в тривимірному просторі, якщо покласти $x_j = 1$, коли ребро e_i міститься в дереві розгляду і $x_j = 0$ в іншому випадку ($j = 1, 2, 3$). Тоді ці три остових дерева можна ототожнити з вершинами v_1, v_2, v_3 допустимої множини F, показаної на рис. 6.4, яку визначають обмеженнями:

$$x_1 + x_2 + x_3 = 2,$$
$$x_1 \ge 0, x_2 \ge 0, x_3 \ge 0,$$
$$x_1 \le 1, x_2 \le 1, x_3 \le 1.$$

Тоді знаходження *МОД* з матрицею відстаней $d_{12} = c_3$, $d_{23} = c_1$ і $d_{31} = c_2$ полягає в розв'язанні задачі *ЛП* з допустимою множиною, зображеною на рис. 6.4.

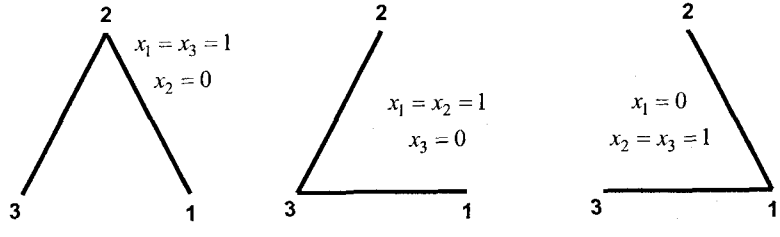


Рис. 6.3. Три остових дерева, що розглядають як точки в тривимірному просторі

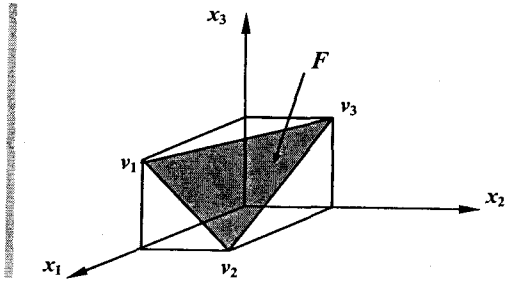


Рис. 6.4. Допустима множина F

Отже, чисто комбінаторну задачу можна розв'язати як деяку задачу *ЛП*. Розглянемо основні підходи до розв'язання задач такого типу.

Візьміть якусь суто комбінаторну задачу і розгляньте її як задачу ЛП.

6.3. Розв'язання задачі лінійного програмування

Задачі лінійного програмування (лінійного планування) з обмеженнями у вигляді нерівностей були першими детально вивченими задачами оптимізації.

6.3.1. *Основні визначення*

У загальному вигляді задачу лінійного програмування формулюють так: знайти величини $x^1, x^2, \ldots, x^n$, що максимізують лінійну функцію

$$f(x) = (c,x) = c^1x^1 + c^2x^2 + \ldots + c^nx^n$$

і задовольняють обмеження

$$a_{11}x^1 + \ldots + a_{1n}x^n \le b^i,$$
$$a_{21}x^1 + \ldots + a_{2n}x^n \le b^k,$$
$$\ldots$$
$$a_{m1}x^1 + \ldots + a_{mn}x^n \le b^e.$$

Серед обмежень задач *ЛП* також виділяють умови невід'ємності всіх або частини змінних:

$$x^i \ge 0,\ i = 1, 2, \ldots, n.$$

Цільову функцію $f(x) = (c, x)$ називають *критерієм* (функціоналом) задачі *ЛП*.

Джерелом розвитку математичної теорії розв'язання задач *ЛП* були економічні задачі [107, 108]. Розглянемо декілька з них.

Задача вибору оптимального плану. Припустимо, що для створення деякого виробу потрібно m видів ресурсів, і його можна створити, використавши n технологій виробництва. Позначимо через a_{ij} витрату ресурсу з номером i у разі одиничної інтенсивності використання технологій з номером j, а через c^j – кількість виробленої продукції. Вважатимемо, що

залежність витрат і випусків від інтенсивності має лінійний характер. Покладемо, що виробництву виділено b^i одиниць i-го ресурсу, і позначимо через x^j інтенсивність використання j-ї технології. Тоді під оптимізацією плану можна розуміти пошук максимуму об'єму випуску продукції, рівного $\sum_{j=1}^{n} c^j x^j$ за таких витрат ресурсів:

$$a_{11}x^1 + \ldots + a_{1n}x^n = b^1,$$
$$\ldots$$
$$a_{m1}x^1 + \ldots + a_{mn}x^n = b^m.$$

Інтенсивність використання технологій, по суті, невід'ємна, тому змінні x^j мають також задовольняти обмеженням

$$x^1 \ge 0, \ldots, x^n \ge 0.$$

Отримали постановку задачі *ЛП*.

Транспортна задача. Припустимо наявність m складів, де зберігаються певні вироби в кількостях $a_1, \ldots, a_m$, і n пунктів реалізації цих виробів, потреби яких визначають відповідно $b^1, \ldots, b^n$. Потрібно знайти найекономічніший спосіб перевезення продуктів зі складів до пунктів реалізації, якщо затрати на перевезення одиниці виробу з i-го складу в j-й пункт реалізації дорівнюють e_{ij}. Позначимо через x_{ij} кількість виробів, що перевозять з i-го складу в j-й пункт. Тоді задача мінімізації транспортних витрат формулюватиметься так:

знайти $\min \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij}$ з обмеженнями

$$\sum_{j=1}^{n} x_{ij} \le a_i,\ i = 1, \ldots, m,$$
$$\sum_{i=1}^{m} x_{ij} \ge b^i,\ j = 1, \ldots, n,$$
$$x_{ij} \ge 0;\ i = 1, \ldots, m;\ j = 1, \ldots, n.$$

Перша умова обмеження – зі складу не можна вивезти більше виробів, ніж зберігається; друга умова – потреба кожного пункту реалізації має задовольнятися; третя – від'ємні перевезення виробу заборонено.

Спробуйте сформулювати у вигляді задачі ЛП постановку задачі про дієту. У загальних рисах суть її полягає в такому. Господарці потрібно оптимізувати витрати на годівлю родини певним набором їжі за певний час (кожен продукт характеризується калорійністю, вартістю і масою), що забезпечує певну межу потрібної калорійності кожного члена сім'ї. Зверніть увагу на той факт, що в результаті розв'язання задачі можна отримати оптимум: слід годувати сім'ю тільки оцтом – він досить калорійний і найдешевший. Як уникнути отримання такого розв'язку?

Існує багато форм запису постановки задачі *ЛП*, але перехід від однієї форми до іншої досить простий. Наприклад, *перехід від задачі мінімізації до задачі максимізації* проводять за допомогою зміни знака критеріальної функції, а *за рахунок зміни знака вільного члена або коефіцієнтів* в обмеженні-нерівності можна змінити знак цієї нерівності на протилежний. Використовуючи ці та подібні їм перетворення, довільну задачу *ЛП* можна звести [48] до двох найбільш використовуваних форм: *канонічної* і з *однотипними обмеженнями*.

Канонічна форма задачі ЛП. Кажуть, що задачу *ЛП* записано в канонічній формі, якщо її формулюють так:

знайти

$$\max(c^1x^1 + \ldots + c^nx^n) \tag{6.1}$$

з обмеженнями

$$\begin{aligned} a_{11}x^1 + \ldots + a_{1n}x^n &= b^1, \\ \ldots & \\ a_{m1}x^1 + \ldots + a_{mn}x^n &= b^m, \end{aligned} \tag{6.2}$$

$$x^i \geq 0,\ i = 1, 2, \ldots, n. \tag{6.3}$$

У матричній формі цю задачу можна записати так:

$$\max(c, x), \tag{6.1'}$$

$$Ax = b, \tag{6.2'}$$

$$x \geq 0, \tag{6.3'}$$

де $A = ||a_{ij}||$ є *матрицею умов* розмірності $(m \times n)$, її стовпці $\{a_{1j}, \ldots, a_{mj}\}^T$, $j = 1, 2, \ldots, n$ називають *векторами умов*, $b = \{b^1, \ldots, b^m\}^T$ – *вектором правих частин*, $\{c^1, \ldots, c^n\}$ – *вектором коефіцієнтів лінійної форми*. Ранг матриці A має бути рівним числу її рядків. Якщо це не так і задача є *розв'язною*, тоді деякі з рівнянь (6.2) мають бути лінійними комбінаціями інших, і тому їх можна було б відкинути. Точку x, що задовольняє обмеженням (6.2), (6.3), називають *допустимим* розв'язком. Допустимий розв'язок, на якому форма (6.1) набуває максимального значення, називають *оптимальним* розв'язком або розв'язком задачі (6.1–6.3).

Задача лінійного програмування з однотипними умовами. У цьому разі задачу *ЛП* формулюють так:

знайти

$$\max(c^1x^1 + \ldots + c^nx^n) \tag{6.4}$$

з обмеженнями

$$\begin{aligned} a_{11}x^1 + \ldots + a_{1n}x^n &= b^1, \\ \ldots & \\ a_{m1}x^1 + \ldots + a_{mn}x^n &= b^m \end{aligned} \tag{6.5}$$

або (у матричному вигляді)

$$\max(c, x),\ Ax \leq b.$$

Цей запис називають також *спряженою канонічною формою*.

Важливе місце в теорії *ЛП* займають *опуклі множини* допустимих розв'язків.

Множину X називають *опуклою*, якщо для довільних двох її точок вона містить і відрізок, що з'єднує їх, тобто для довільних $x, y \in X$, $0 \leq \alpha \leq 1$, точка $y + \alpha(x - y)$ належить X. Якщо ввести число $\beta = 1 - \alpha$, тоді це визначення матиме таку форму: множина X опукла, якщо для довільних $x, y \in X$ і довільних α, β, таких що $\alpha \geq 0, \beta \geq 0, \alpha + \beta = 1$, точка $\alpha x + \beta y$ належить X.

Приведіть приклади опуклих множин.

Множина X допустимих розв'язків задачі *ЛП опукла*, і її можна задати у вигляді *опуклого* багатогранника. Особливої уваги потребують його *крайні* точки. Точку z називають крайньою точкою опуклої множини X, якщо не існує векторів $x, y \in X$, $x \neq y$, таких що $z = \alpha x + \beta y$, де α, β – деякі числа, що задовольняють умовам $\alpha + \beta = 1$, $\alpha > 0$, $\beta > 0$. Крайні точки опуклого багатогранника називають його вершинами. Зрозуміло, що хоча б одна точка-вершина буде входити до оптимального розв'язку. Враховуючи, що кількість вершин опуклого багатогранника скінченна, маємо можливість будувати скінченні алгоритми лінійної оптимізації.

Серед допустимих розв'язків задачі *ЛП* виділяють *базисні* допустимі розв'язки. Точку x називають *базисним допустимим* розв'язком задачі (6.1–6.3), якщо x задовольняє обмеженням (6.2–6.3) і вектори умов, які відповідають її додатним коефіцієнтам, є лінійно незалежними.

Нарешті, наступні дві теореми [75] є основою алгоритму знаходження розв'язку задачі *ЛП*.

Теорема 6.1. Для того щоб точка x була вершиною множини допустимих розв'язків задачі (6.1–6.3), необхідно і достатньо, щоб вектори умов, які відповідають її додатним координатам, були лінійно залежними.

Теорема 6.2. Серед оптимальних розв'язків довільної задачі (6.1–6.3) завжди знайдеться не менше однієї вершини допустимого багатогранника.

Із цих теорем випливає, що серед розв'язків задачі (6.1–6.3) знайдеться розв'язок, який буде базисним (буде вершиною допустимого багатогранника); точка цього розв'язку матиме таку кількість додатних координат k, що $k \leq m$, і відповідні цим координатам вектори умов є лінійно незалежними. У зв'язку з тим, що наша задача невироджена, кожна

вершина багатогранника матиме $k = m$ додатних координат. В іншому разі k може бути меншим за m. Проте ми завжди вважаємо, що ранг матриці умов дорівнює m, і з кожною вершиною можна неоднозначно зв'язати набір із m індексів, таких що стовпці матриці A з цими індексами лінійно незалежні, і координати вершини з номерами, які не увійшли до цього набору, дорівнюють нулю. Для цього, у разі потреби, список додатних координат вершини можна поповнити одним або кількома номерами її нульових координат. Звідси отримаємо ідею нового визначення допустимого базисного розв'язку [88].

Допустимий розв'язок задачі (6.1–6.3) називають базисним, якщо всі його координати вдається розбити на дві групи так, щоб до першої потрапило рівно m координат і відповідні їм вектори умов були лінійно незалежними, а друга складалася б тільки з нулів. Список координат першої групи називають допустимим базисом, а відповідні їм координати і вектори умов – базисними. За умови додатності всіх координат компонент допустимого базису говорять, що базис невироджений. Такими є всі допустимі базиси невиродженої задачі.

Зрозуміло [102], що у задачі (6.1–6.3) може бути найбільше C_n^m комбінацій допустимих базисних розв'язків. Серед них має бути і оптимальний. Для знаходження його потрібно:

а) відшукати розв'язок всіх можливих систем із m-рівнянь з m невідомими, кожну з яких отримують шляхом фіксування в (6.1.2) $n - m$ деяких координат вектора x;

б) позначити з них ті, що мають додатні компоненти;

в) вибрати із виділених розв'язків ті, які максимізують критерій (6.1).

За допомогою такого скінченного алгоритму можна розв'язати задачу *ЛП*. Проте величина C_n^m має астрономічно велике значення, навіть у разі досить малих значень n і m. Нагадаємо, що C_n^m позначає число комбінацій із n елементів по m. Комбінаціями із n елементів по m називають m-елементні підмножини деякої n-елементної множини: $C_n^m = n!/(m!\,(n-m)!)$. Тому виходом із цієї ситуації є організація впорядкованого перебору допустимих базисних розв'язків (симплекс-метод).

Обчисліть значення C_{10}^5.

6.3.2. *Симплекс-метод*

Симплекс-метод має багато варіантів реалізації: табличний, використання оберненої матриці тощо [40, 47, 48, 83]. Слід розглядати невироджений і вироджений варіанти розв'язання задачі (6.1–6.3). У першому варіанті всі алгоритми працюють скінченно, а в другому – можливе

зациклення у разі переходу до чергового допустимого розв'язку. Усі алгоритми ґрунтуються практично на одній ідеї розподілу координат на додатні та від'ємні.

Нехай маємо невироджену задачу (6.1–6.3) і припустимо, що $\tilde{x}$ – її допустимий базисний розв'язок. Він матиме m додатних (базисних) координат. З визначення базису випливає, що стовпці $a_1, a_2, \ldots, a_m$ лінійно незалежні. Тому рівняння задачі (6.1–6.3) будуть розв'язними відносно базисних змінних $x^1, x^2, \ldots, x^m$, які можна виразити лінійно через змінні, що залишились:

$$x^i = \mu^i - \sum_{k=m+1}^{n} z_{ik} x^k,\ i = 1, \ldots, m,$$

де μ^i, z_{ik} – деякі параметри. Проте ці ж рівняння мають виконуватися і для $x^i = \tilde{x}^i$, $i = 1, \ldots, m$, а небазисні координати $\tilde{x}^i$, $i = m+1, \ldots, n$ дорівнюють нулю. Тому отримаємо

$$x^i = \tilde{x}^i - \sum_{k=m+1}^{n} z_{ik} x^k,\ i = 1, \ldots, m. \tag{6.6}$$

Параметри z_{ik} можна обчислити тут, наприклад, за допомогою методу виключення Гауса. Вони є коефіцієнтами розкладу небазисних стовпців по базисних.

Враховуючи (6.6), значення критерію (6.1) в довільній точці, яка задовольняє обмеженням-рівнянням (6.2),

$$(c,x) = \sum_{i=1}^{m} c^i \tilde{x}^i - \sum_{i=1}^{m} c^i \sum_{k=m+1}^{n} z_{ik} x^k + \sum_{k=m+1}^{n} c^k x^k =$$

$$= \sum_{i=1}^{m} c^i \tilde{x}^i - \sum_{k=m+1}^{n} \left(\sum_{i=1}^{m} c^i z_{ik} - c^k \right) x^k. \tag{6.7}$$

Величини $\sigma^k = \sum_{i=1}^{m} c^i z_{ik} - c^k$, $k = m+1, \ldots, n$ називають оцінками заміщення, z_{ik} – коефіцієнтами заміщення. Ці дві компоненти разом із значеннями базисних координат $\tilde{x}$ і критерію утворюють симплекс-таблицю.

Тоді на кожному кроці алгоритму перебору допустимих базисних розв'язків невиродженої задачі (6.1–6.3) з використанням симплекс-таблиці виконуватимуться дві групи операцій:

1. *Визначення провідного стовпця і рядка.* Простежуються оцінки заміщення σ^k. Якщо всі вони невід'ємні, поточна вершина є оптимальною. В іншому разі, згідно з якоюсь оцінкою (традиційно, за мінімальністю σ^k), вибирають провідний стовпець k з оцінкою заміщення $\sigma^k < 0$. Далі простежуються значення коефіцієнтів заміщення z_{ik}, $i = 1, \ldots, m$. Якщо серед них немає додатних, задача не має розв'язку. В іншому разі,

перебором значень індексу i, для яких $z_{ik} > 0$, визначають однозначно номер s, такий що

$$\frac{\tilde{x}^{\tilde{e}_s}}{z_{sk}} = \min_{\{i: z_{ik}>0\}} \frac{\tilde{x}^{\tilde{e}_i}}{z_{ik}}, \tag{6.8}$$

де $\tilde{e}_s$, $\tilde{e}_i$ – номери базисних координат. Далі за допомогою операцій другої групи слід перейти до вершини, що відповідає новому допустимому базису, в якому є k-та змінна і немає змінної з номером $\tilde{e}_s$.

2. *Перерахунок симплекс-таблиці.* Визначають новий s-й рядок, рівний частці від ділення значень старого рядка на провідний елемент z_{sk}. Для кожного $i \neq s$ обчислюють значення нового i-го рядка як різницю між відповідними значеннями старого рядка і результатом множення z_{ik} на значення нового s-го рядка згідно з формулою

$$\overline{z}_{ij} = \begin{cases} z_{ij} - \dfrac{z_{sj} z_{ik}}{z_{sk}}, & i \neq s; \\ \dfrac{z_{ij}}{z_{ik}}, & i = s; \end{cases} \tag{6.9}$$

$$i = 0, \ldots, m, j = 0, \ldots, n,$$

де $\overline{z}_{0i} = \overline{x}^{\overline{e}_i}$, $i = 1, \ldots, m$ та $\overline{e}_1 = k$, $e_i = \overline{e}_i$ для $i \neq s$ і $z_{00} = (c, \overline{x})$, де $\overline{x}$ – нова вершина.

Збіжність описаного алгоритму в невиродженому випадку гарантується збільшенням критерію на кожному кроці зміни базису і сталістю C_m^n (кількості допустимих розв'язків). У виродженому випадку для усунення варіанта зациклювання застосовують додаткові правила вибору провідного рядка. Наприклад, згідно з лексикографічним правилом, як провідний рядок слід брати той, для якого вектор

$$\left\{ \frac{\tilde{x}^{\tilde{e}_i}}{z_{ik}}, \frac{z_{i1}}{z_{ik}}, \ldots, \frac{z_{in}}{z_{ik}} \right\}^T$$

лексикографічно мінімальний.

Розглянемо застосування симплекс-таблиці для розв'язання такої [53] задачі *ЛП*.

Приклад 6.2. Знайти максимум

$$F = 3x_1 + 2x_2,$$

при обмеженнях

$$x_1 - x_2 \leq 2;\ 2x_1 + x_2 \leq 6;\ x_1 \geq 0;\ x_2 \geq 0.$$

Приведемо цю модель до канонічного вигляду, ввівши вільні змінні x_3 і x_4, перетворюючи нерівності у рівності. Дійдемо задачі:

Знайти максимум

$$F = 3x_1 + 2x_2$$

при обмеженнях

$$x_1 - x_2 + x_3 + 0 \cdot x_4 = 2;$$
$$2x_1 + x_2 + 0 \cdot x_3 + x_4 = 6;$$
$$x_j \geq 0,\ 1 \leq j \leq 4.$$

Позначимо вектори умов задачі через $A_1 - A_4$, вектор обмежень – A_0.

Таблиця 6.1

C_j		0	3	2	0	0
	B_x	A_0	A_1	A_2	A_3	A_4
0	x_3	2	1	–1	1	0
0	x_4	6	2	1	0	1
	Δ	0	–3	–2	0	0

Як початкові вектори візьмемо вектори A_3 і A_4, що складають одиничну підматрицю розмірності m. Початковий допустимий базисний розв'язок має вигляд: $X = (0, 0, 2, 6)$. Заносимо у табл. 6.1 всі необхідні дані. Елементи рядка Δ розраховуємо за формулою (6.1.6). Значення функції для наведеного початкового базису дорівнює: $F = 3 \cdot 0 + 2 \cdot 0 = 0$.

Серед Δ_j є від'ємні, тому вибираємо новий базис.

Перша ітерація

Вводимо у базис вектор A_1, для якого значення Δ є мінімальним. Виводимо з базису вектор A_3. Таким чином елемент a_{11} буде напрямним. Заповнюємо табл. 6.2, використовуючи співвідношення (6.9).

Таблиця 6.2

C_j		0	3	2	0	0
	B_x	A_0	A_1	A_2	A_3	A_4
3	x_1	2	1	–1	1	0
0	x_4	2	0	3	–2	1
	Δ	6	0	–5	3	0

Знову обчислюємо значення відхилів Δ_j. Серед Δ_j є від'ємні, тому продовжуємо оптимізацію.

Результати другої і третьої ітерацій наведено у табл. 6.3 і 6.4.

Таблиця 6.3

C_j		0	3	2	0	0
	B_x	A_0	A_1	A_2	A_3	A_4
3	x_1	$2\,^2/_3$	1	0	$^1/_3$	$^1/_3$
2	x_2	$^2/_3$	0	1	$-^2/_3$	$^1/_3$
	Δ	$9\,^1/_3$	0	0	$-\,^1/_3$	$^5/_3$

Оскільки всі $\Delta_j \geq 0$, то план (табл. 6.4) буде оптимальним $X_{опт} = (0, 6, 8, 0)$, $F_{max} = 12$.

Таблиця 6.4

C_j		0	3	2	0	0
	B_x	A_0	A_1	A_2	A_3	A_4
0	x_2	8	3	0	1	1
2	x_3	6	2	1	0	1
	Δ	12	1	0	0	2

Реалізацію цього методу на Паскалі наведено в програмі 6.1.

```
Program Ex6_1;
uses crt;
const m = 4; n = 6;
      Debug = true;
type  arr1m0n = array [1..m, 0..n] of real;
      arr1n = array [1..n] of real;
      arr1m = array [1..m] of integer;
      arr0n = array [0..n] of real;

{Обчислення Delta}
procedure Delta_Calc(A: arr1m0n; c: arr1n; b: arr1m; var Delta: arr0n);
 var  i, j: integer;
      z: real;
begin
   Delta[0] := 0;
   for i := 1 to m do
      Delta[0] := Delta[0] + c[b[i]] * A[i, 0];
   for j := 1 to n do
      begin
         z := 0;
         for i := 1 to m do
            z := z + c[b[i]] * A[i, j];
         Delta[j] := z – c[j];
      end;
   if debug then
      begin
         write('Delta"s: ');
         for i := 0 to n do
            write(Delta[i]: 5: 1, ' ');
         writeln;
      end;
end;
```



```
{Оптимізація}
function Optimal(Delta: arr0n): boolean;
 var i: integer; Opt: boolean;
begin
   Opt := true;
   for i := 1 to n do
      if Delta[i] < 0 then Opt := false;
   if Opt then
      writeln('Optimal Solution has been reached --> ', Delta[0]: 8: 3);
   Optimal := Opt;
end;

{Провідний стовпець}
function Lead_Col(Delta: arr0n): integer;
 var i, Ind: integer;
begin
   Ind := 1;
   for i := 2 to n do
      if Delta[i] < Delta[Ind] then Ind := i;
   Lead_Col := Ind;
   if debug then
      writeln('Leading Column k = ', Ind);
end;

{Провідний рядок}
function Lead_Row(A: arr1m0n; k: integer): integer;
 var  i, Ind: integer;
      NoMin: boolean;
begin
   NoMin := true;
   i := 1;
   while i <= m do
      begin
         if ( (A[i, k] > 0) {and (A[i, 0] > 0)}) then
            if NoMin then
               begin
                  Ind := i;
                  NoMin := false
               end
            else
               if (A[i, 0] / A[i, k]) < (A[Ind, 0] / A[Ind, k])
                  then Ind := i;
         Inc(i);
      end;
```

```
    if NoMin then
      begin
        writeln('The function is not bounded on the range of it''s admissible values...');
        readkey;
        halt(0)
      end;
    Lead_Row := Ind;
    if Debug then writeln('Leading Row r = ', Ind);
end;

{Обчислення матриці A}
procedure A_Calc(r, k: integer; old_A: arr1m0n; var new_A: arr1m0n);
 var i, j: integer;
begin
    for i := 1 to m do
      for j := 0 to n do
        if i <> r then
          new_A[i, j] := old_A[i, j] - old_A[r, j] * old_A[i, k] / old_A[r, k]
        else
          new_A[i, j] := old_A[r, j] / old_A[r, k];
    if Debug then
      for i := 1 to m do
        for j := 0 to n do
          if j = n then
            writeln(new_A[i, j] : 6 : 1) else write(new_A[i, j] : 6 : 1);
end;

{Новий базис}
procedure New_Base(A: arr1m0n; var b: arr1m);
 var  i, j, Ind: integer;
      vector: string;
begin
    for j := 1 to n do
      begin
        vector := '';
        for i := 1 to m do
          if ( (A[i, j] = 1) and (vector = '')) then
            begin
              vector := '1';
              Ind := i
            end
          else
            if A[i, j] <> 0 then
              vector := 'not base';
        if vector = '1' then
          begin
            b[Ind] := j;
            Inc(Ind);
          end;
      end;
    if Debug then
      begin
        write('Basis: ');
        for i := 1 to m do
          write(b[i], ' ');
        writeln;
      end;
end;

 var  A: arr1m0n;
      c: arr1n;
      b: arr1m;
      Delta: arr0n;
      i, j, k, r, Step: integer;
begin
{Test #1, F_max = 12, F_min = 0, m = 2, n = 4
A[1, 0] := 2; A[1, 1] := 1; A[1, 2] := -1; A[1, 3] := 1; A[1, 4] := 0; A[2, 0] := 6; A[2, 1] := 2;
A[2, 2] := 1; A[2, 3] := 0; A[2, 4] := 1; c[1] := 3; c[2] := 2; c[3] := 0; c[4] := 0;}
{Test #2, F_max = 2.2, F_min = 1.00, m = 3, n = 5
A[1, 0] := 1; A[1, 1] := 1; A[1, 2] := 0; A[1, 3] := 0; A[1, 4] := 1; A[1, 5] := -2; A[2, 0] := 2;
A[2, 1] := 0; A[2, 2] := 1; A[2, 3] := 0; A[2, 4] := -2; A[2, 5] := 1; A[3, 0] := 3; A[3, 1] := 0;
A[3, 2] := 0; A[3, 3] := 1; A[3, 4] := 3; A[3, 5] := 1; c[1] := 0; c[2] := 0; c[3] := 0; c[4] := 1;
c[5] := -1;}
{Test #3, F_max = 10, m = 2, n = 3
A[1, 0] := 10; A[1, 1] := 1; A[1, 2] := 1; A[1, 3] := 0; A[2, 0] := 3; A[2, 1] := 1; A[2, 2] := 0;
A[2, 3] := 1; c[1] := 1; c[2] := 0; c[3] := 0;}
{Test #4, F_max = 20, m = 4, n = 6}
A[1, 0] := 10; A[1, 1] := 1; A[1, 2] := 0; A[1, 3] := 1; A[1, 4] := 0; A[1, 5] := 0; A[1, 6] := 0;
A[2, 0] := 10; A[2, 1] := 0; A[2, 2] := 1; A[2, 3] := 0; A[2, 4] := 1; A[2, 5] := 0; A[2, 6] := 0;
A[3, 0] := 0; A[3, 1] := -1; A[3, 2] := 0; A[3, 3] := 0; A[3, 4] := 0; A[3, 5] := 1; A[3, 6] := 0;
A[4, 0] := 0; A[4, 1] := 0; A[4, 2] := -1; A[4, 3] := 0; A[4, 4] := 0; A[4, 5] := 0; A[4, 6] := 1;
c[1] := 1; c[2] := 1; c[3] := 0; c[4] := 0; c[5] := 0; c[6] := 0;

    clrscr;
    if Debug then
      begin
        writeln('============');
        writeln('Step #0');
          for i := 1 to m do
            for j := 0 to n do
```

```
                if j = n then
                    writeln(A[i, j]: 6: 1)
                else
                    write(A[i, j]: 6: 1);
        end;
    New_Base(A, b);
    Delta_Calc(A, c, b, Delta);
    Step := 1;
    while not Optimal(Delta) do
        begin
            if Debug then
                begin
                    readkey; writeln; writeln; writeln('= = = = = = = = = = = = = ');
                    writeln('Step #', Step)
                end;
            k := Lead_Col(Delta);
            r := Lead_Row(A, k);
            A_Calc(r, k, A, A);
            New_Base(A, b);
            Delta_Calc(A, c, b, Delta);
            Inc(Step);
        end;
    repeat until keypressed;
end.
```

Програма 6.1. Табличний симплекс-метод

6.3.3. *Алгоритм з оберненою матрицею*

Під час застосування симплекс-таблиць для розв'язання задачі (6.1–6.3) потрібен значний об'єм пам'яті. Тому на практиці частіше використовують алгоритм розв'язання задачі *ЛП*, який дістав назву алгоритму з оберненою матрицею. Він потребує менше пам'яті. Розглянемо основні ідеї цього алгоритму.

Перепишемо задачу (6.1–6.3) в матричній формі:

$$\begin{aligned} &\max(c, x), \\ &Ax = b, \\ &x \geq 0. \end{aligned} \tag{6.10}$$

Для побудови симплекс-таблиць як основу використовують матриці систем рівнянь, які можна отримати із початкових обмежень – рівнянь задачі (6.10), якщо розв'язувати їх відносно базисних змінних, інакше кажучи, перемножуючи A зліва на матрицю A'^{-1}, де A' складають базисні стовпці.

Припустимо, що на черговій ітерації це перші m стовпців. Позначимо

$$x' = \{x_1, x_2, \ldots, x_m\}, x'' = \{x_{m+1}, \ldots, x_n\},$$

$$c' = \{c_1, c_2, \ldots, c_m\}, c'' = \{c_{m+1}, \ldots, c_n\},$$

A'' – матрицю, яку будують із останніх небазисних $n - m$ стовпців матриці A. Тоді обмеження-рівняння задачі (6.1.10) можна привести до вигляду:

$$A'x' + A''x'' = b.$$

Еквівалентна їм система рівнянь, розв'язна відносно вектора, має вигляд:

$$x' + A'^{-1}A''x'' = A'^{-1}b = \tilde{x}', \tag{6.11}$$

де $\tilde{x}'$ вектор, компонентами якого є координати вершини-розгляду. Значення критерію в довільній точці $x = \{x', x''\}^T$, що задовольняє (6.11), визначатиметься:

$$\begin{aligned} (c'x') + (c''x'') &= (c', \tilde{x}') - (c', A'^{-1}A''x'') + (c'', x'') = \\ &= (c', \tilde{x}') - ((A'^{-1}A'')^T c' - c'', x''). \end{aligned}$$

Порівнюючи цю рівність і (6.11) з (6.6) і (6.7), бачимо, що симплекс-таблицю можна переписати так:

$$\left[\begin{array}{c|cc} (c', \tilde{x}') & 0 & c'^T A'^{-1}A'' - c''^T \\ \tilde{x}' & E & A'^{-1}A'' \end{array}\right], \tag{6.12}$$

звідки отримуємо, що для переходу до нового допустимого базису за попередньою схемою достатньо знати матрицю A'^{-1} і вектор-рядок $\lambda = c'^T A'^{-1}$: послідовно перемножуючи λ на небазисні стовпці й віднімаючи від кожного із результатів відповідний елемент вектора c'', обчислимо оцінки заміщення і виділимо провідний стовпець; помноживши на нього матрицю A'^{-1}, знайдемо ті коефіцієнти заміщення, які використовують у формулі для визначення провідного рядка, і виділимо цей рядок; потім за попередніми формулами обчислимо значення координат нового базису. Отже, нам потрібно зберігати в пам'яті тільки матрицю A'^{-1} розміру $(m \times n)$, m-вимірний вектор λ, поточні значення базисних координат і критерій. Для підвищення швидкості потрібен ефективний алгоритм перерахунку матриці, оберненої до базисної.

Останній отримують так. Повернемося до системи рівнянь (6.11). Їх матриця

$$[E \mid A'^{-1}A''] = A'^{-1}A$$

с основною частиною симплекс-таблиці (6.12). Для переходу до нового базису її слід помножити на матрицю

$$M = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & -\frac{z_{ok}}{z_{sk}} & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 & -\frac{z_{ik}}{z_{sk}} & 0 & \dots & 0 \\ . & . & . & . & . & . & . & \dots & . \\ . & . & . & . & 1 & -\frac{z_{s-1k}}{z_{sk}} & 0 & \dots & 0 \\ . & . & . & . & 0 & \frac{1}{z_{sk}} & 0 & \dots & 0 \\ . & . & . & . & 0 & -\frac{z_{s+1k}}{z_{sk}} & 1 & \dots & 0 \\ . & . & . & . & . & . & . & \dots & . \\ 0 & 0 & . & . & 0 & -\frac{z_{mk}}{z_{sk}} & 0 & \dots & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots \\ 0 & M' & \\ 0 & & \end{bmatrix}.$$

У цьому разі $A'^{-1}A$ домножують зліва на M' і отримують матрицю, еквівалентну до системи рівнянь (6.11), розв'язних відносно нових базисних змінних:

$$M'A'^{-1} = \overline{A}'^{-1}A,$$

де $\overline{A}'^{-1}$ – матриця нового базису, звідки випливає, що $M'A'^{-1} = \overline{A}'^{-1}$, і тому матрицю, обернену до базисної, перераховують за тими формулами, що й симплекс-таблицю:

$$\overline{B}_{ij} = \begin{cases} \dfrac{B_{ij} - B_{sj}z_{ik}}{z_{sk}}, & i \neq s,\ i = 1,\dots, m,\ \ j = 1,\dots, m; \\ \dfrac{B_{sj}}{z_{sk}}, & i = s, \end{cases}$$

де B_{ij}, $\overline{B}_{ij}$, $i = 1, \dots, m, j = 1, \dots, m$, позначають (i, j)-ті елементи матриць A'^{-1}, $\overline{A}'^{-1}$. Поклавши $j = 0$, отримуємо формули перерахунку вектора λ

$$\overline{\lambda}^i = \overline{B}_{i0} = \begin{cases} \dfrac{\lambda^i - \lambda^s z_{ik}}{z_{sk}}, & i \neq s,\ i = 1,\dots, m; \\ \dfrac{\lambda^s}{z_{sk}}, & i = s. \end{cases}$$

Реалізацію цього методу на Паскалі наведено у програмі 6.2.

Застосуйте до розглянутого в попередньому параграфі табличного симплекс-методу тільки-но описану техніку використання оберненої матриці. Порівняйте отримані розв'язки.


```
Program Ex6_2;
uses crt;
const m = 3; n = 5; Debug = true;
type  arr1m0n = array [1..m, 0..n] of real;
      arr1n = array [1..n] of real;
      arr1m = array [1..m] of integer;
      arr1m0mzu1 = array [1..m, 0..m + 1] of real;
      arr0m = array [1..m] of real;

{Обчислення матриці e}
procedure e_Calc(r: integer; old_e: arr1m0mzu1; var new_e: arr1m0mzu1);
   var i, j: integer;
begin
   for i := 1 to m do
      for j := 1 to m do
         if i <> r then
            new_e[i, j] := old_e[i, j] – old_e[r, j] * old_e[i, m + 1] / old_e[r, m + 1]
         else
            new_e[i, j] := old_e[r, j] / old_e[r, m + 1];
   if Debug then
      for i := 1 to m do
         for j := 0 to m + 1 do
            if j = m + 1 then
               writeln(new_e[i, j]: 6: 1)
            else
               write(new_e[i, j]: 6: 1);
end;

{Оптимізація}
function Optimal(Delta: arr1n): boolean;
   var i: integer; Opt: boolean;
begin
   Opt := true;
   for i := 1 to n do
      if Delta[i] < 0 then Opt := false;
   Optimal := Opt;
end;

{Додати базис}
procedure Add_Basis(A: arr1m0n; r: integer; var e: arr1m0mzu1);
var   i, j: integer;
      Det: real;
      NoODR: boolean;
begin
   NoODR := true;
   for i := 1 to m do
      begin
         e[i, m + 1] := 0;
```

```
            for j := 1 to m do
                e[i, m + 1] := e[i, m + 1] + e[i, j] * A[j, r];
            e[i, r] := e[i, r];
            if e[i, r] >= 0 then NoODR := false;
        end;
    if NoODR then
        begin
            writeln('The function is not bounded on the range of it''s admissible values...');
            readkey;
            halt(0)
        end;
    if Debug then
        begin
            write('e[', m + 1, '] = ');
            for i := 1 to m do write(' ', e[i, m + 1]: 6: 1);
            writeln;
        end;
end;

{Обчислення V – evaluation of V}
procedure Calc_V(b: arr1m; c: arr1n; e: arr1m0mzu1; var V: arr0m);
var   Det: real;
      i, j: integer;
begin
    for i := 1 to m do
        begin
            V[i] := 0;
            for j := 1 to m do
                V[i] := V[i] + c[b[j]] * e[j, i];
            V[i] := V[i];
        end;
    if Debug then
        begin
            write('V = '); for i := 1 to m do write(' ', V[i]: 6: 1);
        end;
end;

{Delta_j-обчислення}
procedure Delta_j(V: arr0m; A: arr1m0n; c: arr1n; var Delta: arr1n);
var   i, j: integer;
begin
    for j := 1 to n do
        begin
            Delta[j] := 0;
            for i := 1 to m do
                Delta[j] := Delta[j] + V[i] * A[i, j];
            Delta[j] := Delta[j] - c[j];
        end;
```



```
    if Debug then
        begin
            writeln;
            write('Delta = ');
            for i := 1 to n do
                write(Delta[i]: 6: 1);
        end;
end;

{Провідний стовпець}
function Lead_Col(Delta: arr1n): integer;
var   i, Ind: integer;
begin
    Ind := 1;
    for i := 2 to n do
        if Delta[i] < Delta[Ind] then Ind := i;
    Lead_Col := Ind;
    if debug then writeln('Leading Column k = ', Ind);
end;

{Провідний рядок}
function Lead_Row(e: arr1m0mzu1): integer;
var i, Ind: integer; NoMin: boolean;
begin
    NoMin := true;
    i := 1;
    while i <= m do
        begin
            if e [i, m + 1] > 0 then
                if NoMin then
                    begin
                        Ind := i;
                        NoMin := false
                    end
                else
                    if (e[i, 0] / e[i, m + 1]) < (e[Ind, 0] / e[Ind, m + 1]) then Ind := i;
            Inc(i);
        end;
    if NoMin then
        begin
            writeln('The function is not bounded on the range of it''s admissible values...');
            readkey;
            halt(0)
        end;
    Lead_Row := Ind;
    if Debug then writeln('Leading Row r = ', Ind);
end;
```

```
var A: arr1m0n;
        c: arr1n;
        b: arr1m;
        Delta: arr1n;
        i, j, k, r, Step: integer;
        e: arr1m0mzu1;
        V: arr0m;
        Det: real;
begin

{Test #1, F_max = 12, m = 2, n = 4
A[1, 0] := 2; A[1, 1] := 1; A[1, 2] := –1; A[1, 3] := 1; A[1, 4] := 0; A[2, 0] := 6; A[2, 1] := 2;
A[2, 2] := 1; A[2, 3] := 0; A[2, 4] := 1; c[1] := 3; c[2] := 2; c[3] := 0; c[4] := 0; b[1] := 3;
b[2] := 4;}
{Test #2, F_max = 2.2, m = 3, n = 5}
A[1, 0] := 1; A[1, 1] := 1; A[1, 2] := 0; A[1, 3] := 0; A[1, 4] := 1; A[1, 5] := –2; A[2, 0] := 2;
A[2, 1] := 0; A[2, 2] := 1; A[2, 3] := 0; A[2, 4] := –2; A[2, 5] := 1; A[3, 0] := 3; A[3, 1] := 0;
A[3, 2] := 0; A[3, 3] := 1; A[3, 4] := 3; A[3, 5] := 1; c[1] := 0; c[2] := 0; c[3] := 0; c[4] := 1;
c[5] := –1; b[1] := 1; b[2] := 2; b[3] := 3;

    clrscr;
    if Debug then
      begin
        writeln('=============');
        writeln('Step #0'); writeln('A: ');
        for i := 1 to m do
          for j := 0 to n do
            if j = n then writeln(A[i, j]: 6: 1)
            else write(A[i, j]: 6: 1);
      end;
    for i := 1 to m do
      begin
        e[i, 0] := A[i, 0];
        for j := 1 to m do
          e[i, j] := A[j, b[i]];
      end;
    writeln('e: ');
    for i := 1 to m do
      for j := 0 to m do
        if j = m then writeln('', e[i, j]: 4: 1) else write(' ', e[i, j]: 4: 1);
    Calc_V(b, c, e, V);
    Delta_j(V, A, c, Delta);
    Step := 1;
    while not Optimal(Delta) do
      begin
        if Debug then
          begin
            readkey;
            writeln;
            writeln;
            writeln('=============');
            writeln('Step #', Step)
          end;
{1}     k := Lead_Col(Delta); Add_Basis(A, k, e);
{2}     r := Lead_Row(e); b[r] := k;
{3}     e_Calc(r, e, e);
{4}     for i := 1 to m do
          begin
            e[i, 0] := 0;
            for j := 1 to m do
              e[i, 0] := e[i, 0] + e[i, j] * A[j, 0];
              e[i, 0] := e[i, 0];
          end;
        writeln;
        write('e[i, 0]: ');
        for i := 1 to m do
          write(e[i, 0]: 6: 2);
        writeln;
{5}     Calc_V(b, c, e, V);
        if Debug then
          begin
            writeln;
            write('Current Max: ');
            Det := 0;
            for i := 1 to m do
              Det := Det + A[i, 0] * V[i];
            writeln(Det: 6: 1);
          end;
                                                    {Новий базис (A, b);}
{6}     Delta_j(V, A, c, Delta);
        Inc(Step);
      end;
    writeln;
    writeln;
    if Optimal(Delta) then
      begin
        write('Optimal Solution has been reached --> ');
        Det := 0;
        for i := 1 to m do
          Det := Det + A [i, 0] * v [i];
        writeln(Det: 6: 1);
      end;
    repeat until keypressed;
end.
```

Програма 6.2. Реалізація симплекс-методу з оберненою матрицею

Спробуємо обчислити часову оцінку і об'єм необхідної пам'яті ЕОМ для наведених вище алгоритмів, оперуючи такими поняттями, як мінімальна і максимальна можливі часові оцінки.

Як було зазначено, в обох алгоритмах для знаходження максимального значення функції в області допустимих значень проводять перебір допустимих базисних розв'язків за певними критеріями (додатні компоненти, що максимізують критерій (6.1)). Загалом може бути найбільше C_n^m допустимих базисних розв'язків. Число C_n^m – це кількість вершин на багатограннику області допустимих розв'язків. Тому кількість ітераційних кроків коливатиметься від 1 до C_n^m. У цьому разі об'єм використовуваної пам'яті не змінюється.

У разі лінійної форми з двома невідомими область допустимих розв'язків може бути порожньою, якщо система обмежень несумісна (рис. 6.5, *а*), однією точкою (рис. 6.5, *б*), опуклим багатогранником (рис. 6.5, *в*) або ж необмеженою опуклою багатогранною областю (рис. 6.5, *г*).

Аналогічно можна зобразити області допустимих рішень для тривимірних (геометрично) і *n*-вимірних координат (алгебрична модель). Випадки, зображені на рис. 3, *а*, *б*, зумовлюють віднаходження оптимального розв'язку вже на попередньому етапі заповнення симплекс-таблиці, тому зупинимося на випадку, що зображений на рис. 3, *в*.

Хоча кількість ітераційних кроків роботи алгоритмів наперед не відома, однак можна визначити часову оцінку окремого кроку (табл. 6.5 та 6.6).

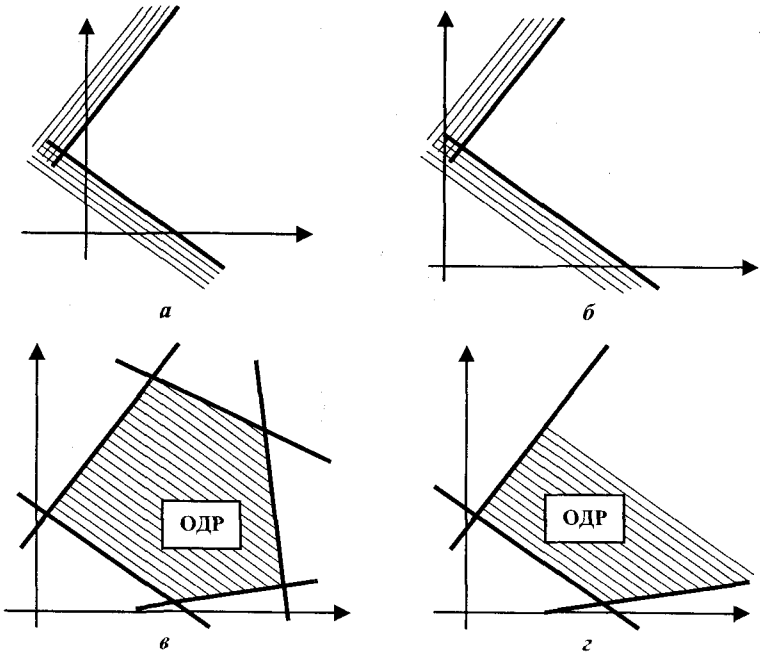


Рис. 6.5. Області допустимих розв'язків (ОДР)

Таблиця 6.5. Оцінка ефективності алгоритму для одного кроку табличного методу

Опис кроку	Спосіб обчислення (процедура)	Часова оцінка	Об'єм пам'яті
Обчислення і порівняння оцінок Delta[j]	Процедура **Delta_Calc** у програмі 6.1	$O(n \cdot m)$	$n + 1$ **of real**
Визначення напрямного елементу a_{rk}	Процедури **Lead_Col, Lead_Row** у програмі 6.1	$O((n-1)m)$	m **of integer**
Заповнення нової таблиці, що відповідає новому базисному розв'язанню	Процедура **A_Calc** у програмі 6.1	$O(m(n+1))$	$(n+1)m$ **of real**
Загальна часова оцінка		$\boldsymbol{O((n+1)m)}$	**real:** $\boldsymbol{(n+2)m}$ **integer:** $\boldsymbol{m}$

Таблиця 6.6. Визначення часової оцінки для одного кроку алгоритму з оберненою матрицею

Опис кроку	Спосіб обчислення (процедура)	Часова оцінка	Об'єм пам'яті
Обчислення і порівняння оцінок Delta[j]	Процедура **Delta_j** у додатку 2	$O(n+m)$	n **of real**
Визначення напрямного елементу a_{rk}	Процедури **Lead_Col, Add_Basis, Lead_Row** у додатку 2	$O(m \cdot m)$	m **of integer**
Заповнення нової таблиці, що відповідає новому базисному розв'язанню	Процедури **Calc_V, e_Calc Rev_Calc** у додатку 2	$O(m(m+1)) + O_{визн}$[1]	$n + m(m+1)$ **of real**
Загальна часова оцінка		$\boldsymbol{O(m(m+1)) + O_{визн}}$	**real:** $\boldsymbol{2n + m(m+1)}$ **integer:** $\boldsymbol{m}$

Тепер можна порівняти отримані оцінки для двох алгоритмів і визначити межі застосування кожного з них, які визначаються розмірами вхідних даних n та m. Результат занесено у табл. 6.7.

Таблиця 6.7. Часові оцінки симплекс-методу

Алгоритм	Часова оцінка	Об'єм пам'яті	Умови застосування
Табличний симплекс-метод	$O_s = O((n+1)m)$	**real:** $m(n+2)$ **integer:** m	при $n \gg m$, напр. при $n = 10m$ $O_r \cong {}^m/_2 \cdot O_s$
Алгоритм з оберненою матрицею	$O_r = O(m(m+1)) + O_{визн}$	**real:** $2n + m(m+1)$ **integer:** m	

[1] Часова оцінка обчислення визначника матриці степеня m.

6.4. Жадібний метод

6.4.1. *Основи методу*

Розглянемо ще раз постановку класичної задачі пошуку екстремумів скалярної функції $f(x)$ n-вимірного векторного аргументу x за деяких обмежень. Цю задачу можна описати так: $\min f(x)$, $x \in X$. Тут X – деяка підмножина n-вимірного евклідового простору E_n. Тоді X називають допустимою множиною задачі пошуку екстремуму, а точки, які належать X,– її допустимими точками. Якщо на область зміни аргументу функції мінімізації накласти більше обмежень, тоді пошук екстремуму функції стає легшим. Якщо ж обмеження досить жорсткі, наприклад, цю множину складають кілька точок і їх легко знайти, тоді загальна задача зводиться до простого перебору кількох чисел.

Розглянемо такий приклад. Нехай потрібно знайти найкоротший шлях між двома точками, які з'єднані вузьким коридором – допустимою областю руху. Тоді оптимізаційна задача стає досить тривіальною: просто потрібно прямувати тільки визначеним коридором – довільний шлях у ньому буде практично оптимальним. Тому в цьому розділі ми розглянемо алгоритми, що побудовані на схемі послідовного аналізу варіантів. Вони використовують процедури, які на підставі побічних оцінок відкидають всі ті допустимі розв'язки, серед яких не може бути оптимального. З часом відбувається поступове звуження множини конкурентних варіантів. Наприкінці залишається один або кілька з них, які можна безпосередньо порівняти. Істотним в цих алгоритмах є знання природи розв'язання задач.

Розглянемо один із оптимізаційних методів, який дістав назву «жадібний», тобто *алгоритм, що робить на кожному кроці локально оптимальний вибір із сподіванням на те, що кінцевий розв'язок також буде оптимальним*. Розглянемо ті задачі, для яких це справджується завжди, а також спробуємо розібратися, чому це так.

Наприклад, для класичної задачі пошуку екстремумів скалярної функції $f(x)$ n-вимірного векторного аргументу x за деяких обмежень можна запропонувати покроковий алгоритм, який за один крок розглядає один вхід n-вимірного вектора X і на кожному кроці робить перевірку входження цього часткового розв'язку до оптимального розв'язку.

Опишемо в процедурі **Greedy** загальну ідею таких алгоритмів [127]:

```
Procedure Greedy(n: int, X: arr; var solution: ...)
                                   {Масив X[n] позначає вектор X}
begin
   solution := 0;
   for i := 1 to n do
     begin
       x := select(X);
       if Feasible(solution, x) then
          solution := union(solution, x)
     end;
end.
```


Функція *SELECT* вибирає один вхід з масиву X. Функція *FEASIBLE* є булевою функцією, яка визначає, допустиме чи ні включення x до вектора розв'язку. Функція *UNION* включає x до загального розв'язку.

Уточнимо постановку задачі оптимізації в загальному вигляді: потрібно знайти $x = (x_1, \ldots, x_n)$, для якого $f(x_1, \ldots, x_n) \to \min$ і задовольняються обмеження

$$g_i(x_1, \ldots, x_n) \le 0; \quad i = \overline{1,m};$$

$$x_j \in D_j; \; j = \overline{1,n}.$$

Нагадаємо, що будь-який x, при якому $x_j \in D_j$, називають *варіантом* розв'язку, а будь-який x, при якому задовольняються обмеження, називають *допустимим* розв'язком. Допустимий розв'язок, на якому $f(x)$ досягає мінімуму, називають *оптимальним розв'язком*. Якщо обмежень немає, мають на увазі *безумовну оптимізацію*.

Одним з базових *методів* оптимізації є *метод послідовних локальних поліпшень*, який полягає в побудові послідовності $x^{(0)}, x^{(1)}, \ldots, x^{(n)}$, такої що $x^{(k)} \in D$ для всіх k; $x^{(k+1)} = h_k(x^{(k)})$, де h – деяка наперед задана функція, яка, узагалі беручи, може залежати від k, і $f(x^{(k+1)}) < f(x^{(k)})$.

Алгоритми на підставі методу послідовних локальних поліпшень можна назвати жадібними алгоритмами в широкому розумінні, оскільки вони на кожному кроці намагаються наблизитися до мети. Жадібним алгоритмом в більш вузькому розумінні називають алгоритм, який намагається на кожному кроці підійти до мети якнайближче.

Як один з класичних прикладів можна навести *градієнтний метод* [68], застосовуваний при $D_i = R^n$ і диференційованій цільовій функції. Градієнтом функції $f(x_1, \ldots, x_n)$ у точці x називають вектор

$$\nabla f(x^*) = \left(\frac{\partial f(x^*)}{\partial x_1}, \ldots, \frac{\partial f(x^*)}{\partial x_n} \right).$$

Слід нагадати, що градієнт задає напрямок найшвидшого зростання функції. Тоді в градієнтному методі $x^{(k+1)} = x^{(k)} - \lambda_k \nabla f(x_k)$, де λ_k – величина кроку в напрямку антиградієнта. Якщо λ_k вибирають з умови $\lambda_k = \min_{\lambda > 0} \left(f\left(x^{(k)}\right) - \lambda \nabla f\left(x^{(k)}\right) \right)$, алгоритм називають *методом найшвидшого спуску*.

Основним недоліком градієнтних методів (як і методів локальних послідовних поліпшень), які є певним різновидом жадібних алгоритмів, є те, що вони не завжди забезпечують знаходження глобального екстремуму. Згідно з нашою загальною стратегією розгляду жадібних алгоритмів,

слід виділити певні умови, за виконання яких наші методи забезпечуватимуть збіг локального і глобального екстремумів. У цьому разі такі достатні умови визначає [48] теорема 6.3.

Теорема 6.3. Якщо $f(x)$ – опукла функція, визначена на опуклій множині X, то локальний мінімум є водночас і глобальним мінімумом.

Все це зумовлює доцільність окремого розгляду задач опуклого програмування.

Для *цілочислового програмування* можна дати таке загальне визначення жадібного алгоритму. *Жадібним* називатимемо алгоритм, який на кожному кроці переходить від $x^{(k)}$ до $x^{(k+1)}$, причому $x^{(k+1)}$ вибирають з множини $P = \{x: x = h(x^{(k)})\}$, виходячи з умови оптимізації або самої цільової функції, або деякої евристичної функції $u(x^{(k)}, x^{(k+1)})$.

Як приклад можна навести задачу про рюкзак. Нехай є n предметів і рюкзак. Предмет i має вагову характеристику w_i, а рюкзак має об'єм M. Якщо візьмемо x_i, $0 \le x_i \le 1$, тоді розміщення предмета i в рюкзаку визначатиме зиск $p_i x_i$. Оптимальним наповненням рюкзака вважатимемо таке наповнення рюкзака предметами, яке максимізує загальний зиск. Оскільки вміст рюкзака обмежений M, вимагатимемо, щоб загальна вага предметів була не більшою за M. Формально задачу можна сформулювати так:

максимізувати $\sum_{i=1}^{n} p_i x_i$ *таких предметів* i, *для яких* $\sum_{i=1}^{n} w_i x_i \le M$ *та*

$0 \le x_i \le 1, p_i > 0, w_i > 0, i = \overline{1,n}$.

Коли $\sum_{i=1}^{n} w_i \le M$, для всіх x_i візьмемо $x_i = 1$, $i = \overline{1,n}$, яке й визначатиме оптимальний розв'язок. Тому розглянемо випадок, коли сума ваг всіх предметів більша за M.

Згідно з нашою загальною стратегією жадібних алгоритмів, щонайперше слід вибрати критерій жадібності. Звернувшись до здорового глузду, після нескладних роздумів або простих експериментів, приходимо до такого критерію: *на черговому кроці включатимемо до розв'язку той предмет, який має максимальний зиск за рахунок використаної частини об'єму рюкзака.* Це означає, що кандидати на занесення в рюкзак розглядатимуться стосовно відношення p_i/w_i. Зазначимо, що часова оцінка алгоритму цілком залежить від часової оцінки використаного алгоритму впорядкування, тому що для реалізації самої стратегії наповнення, після впорядкування предметів, потрібен час $O(n)$. Тоді, використавши для простоти опису обмінне впорядкування, отримаємо розв'язок нашої задачі. Програма 6.3 дає розв'язок задачі про рюкзак згідно з цією стратегією.


```
Program Ex6_3;
const d = 100;
type  ar = array [1..d] of real;
      int = 1..d;
var   i, n: int;
      p, w, x: ar;
      m: real;

procedure greknap(n: int; m: real; p, w: ar; var x: ar);
var   i: int;
      cu: real;
label exit;

procedure sort(n: int; p: ar; var w: ar);
var i, j, indmax: int;
      t: real;
      a: ar;
begin
   for i := 1 to n do
     a[i] := p[i] / w[i];
   for i := 1 to n do
     begin
        indmax := i;
        for j := i + 1 to n do
           if a[indmax] < a[j] then indmax := j;
        t := a[i]; a[i] := a[indmax]; a[indmax] := t;
        t := w[i]; w[i] := w[indmax]; w[indmax] := t;
     end;
end;

begin
   for i := 1 to n do x[i] := 0;
   sort(n, p, w);
   cu := m;
   for i := 1 to n do
     begin
        if w[i] <= cu then
           begin
              x[i] := 1;
              cu := cu – w[i];
           end
        else
           goto exit;
     end;
   exit: if i < = n then x[i] := cu / w[i]
end;
```

```
begin
    readln(n, m);
    for i := 1 to n do
      read(p[i], w[i]);
    greknap(n, m, p, w, x);
    for i := 1 to n do
      writeln(x[i])
end.
```

Програма 6.3. Розв'язок задачі про рюкзак

Ще одним прикладом ефективного застосування техніки жадібного методу в цілочисловому випадку може служити вдалий розв'язок задачі оптимального збереження програм у пам'яті. Одним із варіантів її постановки може бути такий.

Потрібно n програм розмістити послідовно в пам'яті завдовжки L. Нехай довжина кожної програми – l_i, $i = \overline{1,n}$. Зрозуміло, що всі програми можна зберегти тільки тоді, коли $\sum_{i=1}^{n} l_i \le L$. Припустимо, що після чергового звертання до якоїсь з програм читальний пристрій повертається на початкову позицію. Якщо для програм зберігається порядок $I = i_1, i_2, \ldots, i_n$, тоді час t_j, необхідний для доступу до програми i_j, буде пропорційним до $\sum_{k=1}^{j} l_{i_k}$. Якщо всі програми можна викликати рівномірно, то очікуваний час доступу (mean retrival time) *MRT* буде $\frac{1}{n}\sum_{j=1}^{n} t_j$. Тоді проблему оптимального збереження програм формулюють так. Потрібно знайти таке розміщення програм у пам'яті, щоб час доступу до них *MRT* був мінімальним. Мінімізація *MRT* є еквівалентною до мінімізації

$$D(I) = \sum_{j=1}^{n}\sum_{k=1}^{j} l_{i_k}.$$

Приклад 6.3. Нехай $n = 3$, а $l_1 = 6$, $l_2 = 8$, $l_3 = 4$. Тоді ці три програми ми можемо розмістити в пам'яті 3! = 6 способами. Порахуємо для кожного з варіантів розміщення $D(I)$:

Порядок *I*	**Значення *D*(*I*)**
1, 2, 3	6 + 6 + 8 + 6 + 8 + 4 = 38
1, 3, 2	6 + 6 + 4 + 6 + 4 + 8 = 34
2, 1, 3	8 + 8 + 6 + 8 + 6 + 4 = 40
2, 3, 1	8 + 8 + 4 + 8 + 4 + 6 = 38
3, 1, 2	4 + 4 + 6 + 4 + 6 + 8 = 32
3, 2, 1	4 + 4 + 8 + 4 + 8 + 6 = 34

Тому оптимальне розміщення буде 3, 1, 2.

Проаналізуємо цю задачу відповідно до загальної схеми жадібного алгоритму. Вибиратимемо на кожному кроці один розв'язок (одну роботу) і аналізуватимемо поведінку функції $D(I)$. Помітимо, що зростання $D(I)$ мінімізується, якщо чергова програма вибору має найменшу довжину серед програм, які залишилися.

Близькою до задачі оптимального збереження програм у пам'яті є задача оптимізації з'єднань файлів. Коли ми розглядали сортування злиттям, то відзначали, що два впорядковані файли, відповідно n та m записів, можна перетворити на один впорядкований файл за час $O(n + m)$. Якщо ми маємо більше початкових впорядкованих файлів, тоді кінцевий файл можна отримати шляхом поетапного попарного використання попереднього підходу.

Існує багато послідовностей попарного з'єднання файлів. Різні послідовності попарного з'єднання вимагатимуть різної кількості комп'ютерного часу. Перед дослідниками постає проблема оптимізації послідовності попарних з'єднань n файлів разом.

З нескладних роздумів можна сформулювати ідею обмежень жадібного алгоритму, які дадуть змогу отримати оптимальний розв'язок. Критерій відбору буде таким: на черговому кроці об'єднання з'єднуватимуться два найменших за розміром файли. Наприклад, якщо ми маємо набір з 7 файлів ($F_1 = 10$, $F_2 = 5$, $F_3 = 11$, $F_4 = 20$, $F_5 = 4$, $F_6 = 30$, $F_7 = 18$), тоді, згідно із запропонованим критерієм відбору, буде така послідовність об'єднань: $F_2 \,||\, F_5 = Z_1$ (9), $Z_1 \,||\, F_1 = Z_2$ (19), $F_7 \,||\, Z_2 = Z_3$ (37), $F_4 \,||\, F_6 = Z_4$ (50), $Z_3 \,||\, Z_4 = Z_5$ (87). Легко перевірити, що це оптимальне вирішення нашої проблеми.

На кожному кроці наша модель обчислень вимагає порівняння двох об'єктів. Тому графічно таку модель зручно описувати бінарним деревом. Листки дерева (квадрати) визначають набір початкових файлів впорядкування. Внутрішні вузли (кружки) визначають файли з'єднання. Їх сини – це файли, які беруть участь у черговому з'єднанні. Всередині вузлів – числа. Вони характеризують довжину відповідних файлів. Дерево, зображене на рис. 6.6, описує модель з'єднання файлів для останнього прикладу.

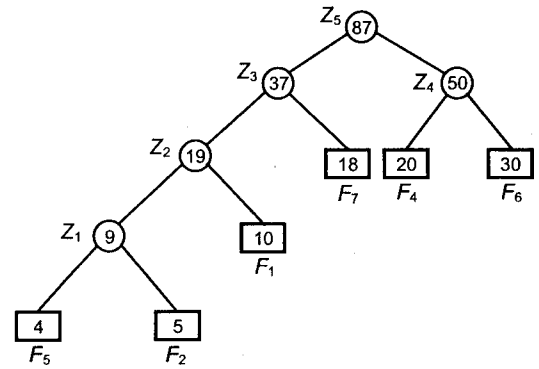


Рис. 6.6. Задання моделі поєднання бінарним деревом

Зовнішній вузол F_5 знаходиться на відстані 4 від кореня (вузол рівня i знаходитиметься на відстані $i - 1$ від кореня). Отже, записи файла F_5 пересуватимуть 4 рази для отримання остаточного з'єднання Z_5. Для файла F_i введемо відповідні позначення: d_i – відстань від кореня до вузла F_i та q_i – довжина файла F_i. Тоді загальне число пересувань записів для бінарного дерева поєднання визначатимуть формулою $\sum_{i=1}^{n} d_i q_i$, де n – кількість початкових файлів з'єднання. Цю суму називають *зваженою довжиною* зовнішнього шляху дерева.

Оптимальна модель поєднання файлів відповідатиме бінарному дереву поєднання, яке матиме мінімальну зважену довжину зовнішнього шляху. Процедура *tree* алгоритму використовує вищезазначене оптимізаційне правило для побудови дерева двошляхового злиття з n файлів. Спочатку на вхід алгоритму подають список L з n дерев. Кожен вузол дерева має три поля *ll*, *rl*, *w* (*eight*). У початковому списку L кожне дерево складається тільки з одного вузла. Ці вузли належать до множини зовнішніх вузлів (будуть листками). В полях *ll* та *rl* вони доти мають значення *nil*, доки поле *w* складається з довжини тільки одного з n файлів.

Процедура *heapify* будує з цих початкових дерев, використовуючи процедуру *adjust*, список дерев A у вигляді купи. Структура купи дає нам змогу на кожному кроці злиття легко вибирати 2 файли мінімальної довжини, які починають зливатись у вузлі t, тому $t\uparrow.w$ – довжина файла, визначатиметься як сума довжин файлів синів вузла t. Після злиття файли злиття видаляють з купи і до неї заносять значення довжини нового файла злиття $t\uparrow.w$. Процедура *typetree* друкує отримане дерево злиття з коренем, на який вказує покажчик Head в прямому порядку (корінь, ліве піддерево, праве піддерево).

```
Program Ex6_4;
const m = 100;
type  int = 1..m;
      ntree = ^node;
      node = record                                  {Вершина дерева}
               w: integer;
               ll, lr: ntree
             end;
      node1 = record
               w: integer;
               l: ntree
             end;
      arr = array [int] of node1;
var   i, j: int;
      a: arr;
      head: ntree;
```



```
procedure adjust(i, n, val: int; link: ntree; var aa: arr);
var   j: int;
      li: ntree;
      item: integer;
label exit;
begin
   aa[i].w := val;
   aa[i].l := link;
   j := 2 * i;
   item := val;
   li := link;
   while j <= n do
     begin
       if (j < n) and (aa[j].w > aa[j + 1].w) then j := j + 1;
         if item <= aa[j].w then
           goto exit
         else
           begin
             aa[(j div 2)].w := aa[j].w;
             aa[(j div 2)].l := aa[j].l;
             j := 2 * j
           end;
     end;
exit:
   aa[(j div 2)].w := item;
   aa[(j div 2)].l := li;
end;

{Побудова купи}
procedure heapify(n: int; var a: arr);
var i: int;
begin
   for i := n div 2 downto 1 do
   adjust (i, n, a [i].w, nil, a)
end;

{Побудова дерева злиття}
function tree(n: int; var lheap: arr): ntree;
var   i: int;
      t, p, q: ntree;
begin
   for i := n downto 2 do
     begin
       new(t);
       if lheap[1].l = nil then
         begin
           new(q);
```

```
                q^.ll := nil;
                q^.lr := nil;
                q^.w := lheap[1].w
              end
            else
              q := lheap[1].l;
            lheap[1].w := lheap[i].w;
            lheap[1].l := lheap[i].l;
            adjust(1, i – 1, lheap[1].w, lheap[1].l, lheap);
            if lheap[1].l = nil then
              begin
                new (p);
                p^.ll := nil;
                p^.lr := nil;
                p^.w := lheap[1].w
              end
            else
              p := lheap[1].l;
            t^.ll := q;
            t^.lr := p;
            t^.w := q^.w + p^.w;
            lheap[1].w := t^.w;
            lheap[1].l := t;
            adjust(1, i – 1, lheap[1].w, lheap[1].l, lheap);
        end;
      tree := t;
end;

{Виведення результату}
procedure typetree(p: ntree);
    begin
      write(p^.w, ' ');
      if p^.ll <> nil then typetree(p^.ll);
      if p^.lr <> nil then typetree(p^.lr)
    end;

begin
    readln(j);
    for i := 1 to j do
      read (a[i].w);
    heapify(j, a);
    head := tree(j, a);
    writeln;
    typetree(head);
end.
```

Програма 6.4. Побудова оптимального злиття набору файлів

Проаналізуємо роботу алгоритму побудови дерева злиття. Головний цикл **for** функції **tree** потребує $n - 1$ тактів часу. Враховуючи, що список кандидатів на з'єднання організований у вигляді купи, знайти і видалити елемент можна за $O(1)$, а для вставки елементу до купи потрібно $O(\log n)$. Тому загальний час роботи функції *tree* буде $O(n\log(n))$. Побудувати купу також можна за час $O(n\log(n))$. Отже, загальний час алгоритму побудови дерева буде $O(n\log(n))$.

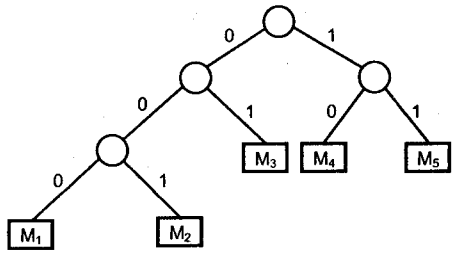


Рис. 6.7. Приклад дерева кодування

Мінімальну зважену довжину зовнішнього шляху дерева можна використовувати для отримання оптимальної множини кодів повідомлень M_1, M_2, ..., M_{n+1}. Кожен код буде двійковим рядком, який використовують під час передачі відповідного повідомлення. Для декодування повідомлення використовуватиметься дерево декодування. Дерево декодування є бінарним деревом, в якому зовнішні вузли містять інформаційні частини повідомлення.

Дерево будують аналогічно побудові дерева з'єднань файлів, тільки замість довжини файла з'єднань використовуватиметься частота використання повідомлення. Двійкові біти слова кодування відповідного повідомлення визначають потрібну гілку кожного рівня, досягнувши точно зовнішнього вузла. Розглянемо дерево, зображене на рис. 6.7.

Тоді 000 кодуватиме повідомлення M_1, 001 – M_2, 01 – M_3, 10 – M_4, 11 – M_5. Такі коди називають *кодами Хаффмана* [128]. Ціна декодування слова кодування прямо пропорційна кількості бітів у коді. Це число є еквівалентним до довжини шляху між коренем і відповідним зовнішнім вузлом. Якщо розглядати q_i як відносну частоту передачі повідомлення M_1, тоді очікуваний час розкодування визначатиметься $\sum_{1 \le i \le n+1} q_i d_i$, де d_i буде довжиною шляху між коренем і зовнішнім вузлом, що містить повідомлення M_i. Очікуваний час розкодування буде мінімальним для вибраного слова кодування, якщо дерево декодування матиме мінімальну зважену довжину зовнішнього шляху. Тому для мінімізації очікуваного часу декодування слід мінімізувати очікувану довжину повідомлення.

Формалізуйте запропоновані описові алгоритми оптимального кодування і розкодування.

6.4.2. *Алгоритм Дейкстри: пошук найкоротших шляхів для графа з одним джерелом*

Повернемось до задачі про найкоротші шляхи, яку ми вже вивчали в розділі 3.4. Розглянемо ациклічний граф (без контурів), вагами ребер якого можуть бути тільки невід'ємні числа. Для багатьох застосувань, наприклад в транспортній задачі, для графа достатньо знаходити найкоротші шляхи тільки із одного вузла, який називатимемо джерелом (витоком). Навіть якщо в нас є й інша задача – знаходження найкоротшого шляху між двома конкретними вершинами у графі, ми ще не знайшли алгоритм, який мав би в найгіршому разі часову оцінку, кращу за алгоритм розв'язання задачі з одним джерелом. Розглянемо алгоритм Дейкстри з погляду жадібного підходу [66].

Нагадаємо постановку задачі. Нехай маємо орієнтований граф $G = (V, E)$, джерело $v_0 \in V$ і функцію *cost*, яка відображає $E \times E$ в множину невід'ємних чисел. Нагадаємо, що $cost(u, v) = +\infty$, якщо ребро (u, v), $u \neq v$, не належить множині ребер графа, і $cost(v, v) = 0$.

Щоб сформулювати жадібний алгоритм для створення найкоротших шляхів, слід визначити поетапне (ітераційне) розв'язання проблеми із зазначенням оптимізаційного критерію. Зупинимось на такому підході. Шляхом приєднання однієї вершини на кожному кроці будуватимемо множину вузлів S, найкоротша відстань до яких від джерела вже відома. Оптимізаційний критерій визначимо так: черговим вузлом w під'єднання до S буде той з вузлів, що залишились ще незанесеними в S, відстань до якого від джерела буде найменшою.

Визначимо $dist(w)$ як довжину найкоротшого шляху, який починається у v_0 і проходить тільки через ті вершини, які є в S, та закінчується у $w \notin S$. Доведемо [66] (індукційним методом за розміром множини S), що мінімальний шлях від v_0 до w проходитиме через вершини, які вже є в S.

Нехай $S = \{v_0\}$. Найкоротший шлях із v_0 в v_0 завдовжки 0 (нуль) і шлях із v_0 у v включатиме тільки заздалегідь використану вершину v_0. Отже, якщо ми на початковому кроці визначимо $dist(v_0) = 0$ і *for* $v \in V - \{v_0\}$ *do* $dist(v) \leftarrow cost(v_0, v)$; тоді початкове значення масиву *dist* буде сформоване коректно.

Нехай черговим вузлом приєднання є вузол $w \in (V - S)$, для якого $dist(w)$ має найменше значення. Якщо значення $dist(w)$ не буде довжиною найкоротшого шляху із v_0 в w, тоді має існувати короткий шлях із v_0 в w, який містить вузол v, такий що $v \neq w$ і $v \notin S$. Тоді $dist(v) < dist(w)$, а найкоротший шлях у v повністю проходить через вершини із S за винятком вершини v. Дійшли протиріччя. Отже, такого шляху не може бути і $dist(w)$ – довжина найкоротшого шляху із v_0 у w.

Після занесення чергової вершини до S для всіх попередніх вершин ми повинні порівняти мінімальний шлях, який був, і шлях, який тепер може включати вершину w, та вибрати з них мінімальний:

(*) *for* $v \in V - S$ *do* $dist(v) \leftarrow \text{MIN}(dist(v), dist(w) + cost(w, v))$.

Вищезазначені спостереження ведуть до алгоритму, який вперше був запропонований Дейкстрою. У ньому вершини множини V графа $G = (V, E)$ мають імена 1, 2, ..., n. Множину S задає масив $S[1..n]$, де $S[i] = true$, якщо вершина i належить множині S, та $S[i] = false$ в іншому разі. Такий підхід дає змогу задавати початковий граф за допомогою матриці суміжності $cost[i, j]$, яку будують згідно з функцією вартості. Для реалізації $+\infty$ скористаємося значенням *Maxreal*.

Головною дією алгоритму є вибір на кожному кроці вершини, яка входить в ребро мінімальної вартості серед ще не розглянутих. Тому побудуємо купу, яка описуватиметься масивом $dist[n]$. Елементи цього масиву є структурами з полями: номер вершини та вартість шляху від джерела до вершини. Це дає змогу просто реалізувати операції вибору вершини, її видалення, поновлення робочої інформації (стрічка (*)), перевірку на належність множині вже розглянутих вершин.

Програма 6.5 реалізує алгоритм Дейкстри, в якому граф задає матриця.

```
Program Ex6_5;
const m = 100;
      maxreal = 1000.0;
type  int = 1..m;
      node = record                    {Структура елементу купи}
               i, j: int;              {Номер вершини}
               cost: real              {Вартість шляху до вершини і до джерела}
             end;
      arr = array [int] of node;       {Масив задання купи}
      ar = array [int] of boolean;     {Масив задання множини S}
      arrr = array [int, int] of real; {Задання вартості ребер графа}
var   u, v, l, kk, j, n, ii, jj, nonempty, q, p: int;
      s: ar;
      dist: arr;
      costt: arrr;
      mincost: real;                   {Вартість ребра, що знаходиться в голові купи}
      temp: node;

{Процедура пересипки; її параметри ii та n задають область масиву aa[ii..n], яка
задовольняє умові сортувального дерева; корінь нового дерева розміщується в aa[ii]}
procedure adjust(ii, n: int; var aa: arr);
var j: int;
      item: node;
label exit;
begin
   jj := 2 * ii;
   item := aa [ii];
   while jj <= n do
     begin
       if (jj < n) and (aa[jj].cost > aa[jj + 1].cost) then jj := jj + 1;
       if item.cost <= aa[jj].cost then
```

```
            goto exit
          else
            begin
              aa[(jj div 2)] := aa[jj];
              jj := 2*jj
            end;
        end;
    exit: aa[ (jj div 2)] := item;
end;

{Процедура, що робить все бінарне дерево купою; дерево задає масив aa[n]}
procedure heapify(n: int; var a: arr);
var i: int;
begin
    for i := n div 2 downto 1 do
      adjust (i, n, a)
end;

{Початок головної програми}
begin
    writeln('Введіть значення n кількості вершин у графі G ');
    readln(n);
    writeln('Введіть вартість ребер ');
    for l := 1 to n do                    {Введення матриці суміжності задання графа G}
      for j := 1 to n do
        begin
          writeln('Вартість ребра між вершинами ', l, ' та ', j);
          read(costt [l, j]);             {Якщо ребра (l, j) не існує, тоді вводимо}
        end;                              {значення maxreal}
    writeln('Введіть номер вершини v, яку ви обираєте джерелом');
    readln(v);
    for l := 1 to n do                    {Початкова ініціалізація множини S}
      begin                               {і вартості шляхів від усіх вершин до джерела}
        s[l] := false;
        dist[l].cost := costt[v, l];
        dist[l].i := l;
      end;
    s[v] := true;
    dist[v].cost := maxreal;
    heapify(n, dist);    {Побудова пріоритетної черги (купи), яку задає масив dist[n]}
    nonempty := n;       {Бар'єр купи в масиві dist[n]}
    for l := 1 to n – 1 do                {Поки купа не пуста ( S ≠ V ), робити}
      begin
        u := dist[1].i;                   {Узяти номер вузла u із голови купи}
        mincost := dist[1].cost;
        writeln;
```



```
{Можемо вже вивести довжину мінімального шляху від u до джерела dist[1].cost}
{та множину вершин, які входять у цей шлях: ті вершини w, для яких на цей}
{момент s[w] = true}
        writeln('Мінімальний шлях від джерела ', v, ' до вершини ', u, ' є ', dist[1].cost);
        writeln('Він включає вершини: ');
        for kk := 1 to n do
          if s[kk] then write(kk);
        s[u] := true;
        temp := dist[nonempty];                {Видалити елемент із голови купи}
        dist[nonempty] := dist[1];
        dist[1] := temp;
        nonempty := nonempty – 1;
        for kk := 1 to nonempty do
          if dist[kk].cost > mincost + costt[u, dist[kk].i] then
            dist[kk].cost := mincost + costt[u, dist[kk].i];
{Якщо черга Q – не пуста, тоді поновити для неї властивість купи}
        if nonempty > 1 then
          adjust(1, nonempty, dist);
      end;
end.
```

Програма 6.5. Алгоритм Дейкстри

Зрозуміло, що часова складність алгоритму буде $O(n^2)$.

З використанням для задання графа списків суміжності можна отримати алгоритм складності $O(m \log n)$; для задання множини можна також використати купу.

Нагадаємо, що бінарне дерево називають купою, якщо для довільної вершини дерева виконується умова: значення, яке містить вершина-батько, більше за значення вершин його синів. Висота такого дерева – $\log n$. Вершина, для якої значення $D[u]$ мінімальне, розміщуватиметься в корені, якщо під час побудови купи замінити знак ≤ на ≥ при порівнянні значень батька і його синів. Тому, видалення мінімального значення і відновлення властивості «купи» можна зробити за $O(\log n)$ кроків. Кожне ребро графа аналізують лише один раз, і з цією дією пов'язано $O(\log n)$ кроків на зсув відповідної вершини в дереві, що дає в сумі $O(m \log n)$ кроків. Якщо додати потребу в $O(n \log n)$ кроків, потрібних для побудови купи, і $n - 1$ кроків для усунення елементів із голови купи, тоді отримаємо, що складність алгоритму в цьому разі дорівнюватиме $O(m \log n)$.

Невідомо поки що нічого про існування для даної задачі алгоритму складності $O(m)$. Доведено [129], *що існує константа c, така що цю задачу для довільного $k > 0$ можна розв'язати за час $ck(m + n^{1+1/k})$. Спробуйте довести останнє твердження.*

6.4.3. *Остове дерево найменшої вартості*

Повернемося до розв'язання задачі знаходження остового дерева графа найменшої вартості. Ще раз нагадаємо постановку задачі.

Нехай $G = (V, E)$ – зв'язний неорієнтований граф. Для нього задана функція вартості, яка відображає ребра в дійсні числа. Остовим деревом для графа $G = (V, E)$ називають неорієнтоване дерево $T = (V, E')$, яке містить всі вершини графа G. Вартість остового дерева визначають як суму вартостей його ребер. Спробуємо знайти для G остове дерево мінімальної вартості.

Багато алгоритмів знаходження остових дерев ґрунтується на наступних двох лемах [9].

Лема 6.1. Нехай $G = (V, E)$ – зв'язний неорієнтований граф і $T = (V, E')$ – його остове дерево. Тоді
1) для довільних двох вузлів v_1 і v_2 із V шлях між ними в T – єдиний і
2) якщо до T додати довільне ребро із $E - E'$, виникне рівно один цикл.

Лема 6.2. Нехай $G = (V, E)$ – зв'язний неорієнтований граф і c – функція вартості, визначена на його ребрах. Нехай $\{(V_1, E'_1), (V_2, E'_2), \ldots, (V_k, E'_k)\}$ – довільний остовий ліс для G, $k > 1$ і $E' = \bigcup_{i=1}^{k} E'_i$. Припустимо, що $e = (v, w)$ – ребро найменшої вартості в $E - E'$, і є таким, що $v \in V_1$ і $w \notin V_1$. Тоді знайдеться остове дерево для G, яке містить $E' \cup \{e\}$, вартість якого не більша від вартості довільного остового дерева для G, яке містить E'.

Застосуємо для побудови остового дерева мінімальної вартості жадібний алгоритм. Він будуватиме остове дерево крок за кроком, додаючи до нього по одному ребру, згідно з якимось оптимізаційним критерієм. Один із найпростіших критеріїв такий. На черговому кроці вибираємо ребро, яке зумовлює мінімальне збільшення суми вартостей ребер, які вже введені в допустимий розв'язок. Існує два варіанти інтерпретації цього критерію.

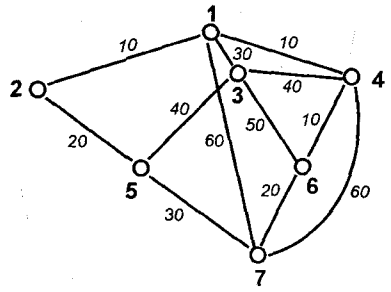


Рис. 6.8. Неорієнтований граф з функцією вартості ребер

Розглянемо перший варіант. Нехай A – множина ребер, вибраних на попередньому кроці, і A – дерево. Тоді чергове ребро під'єднання (u, v) є ребром з мінімальною вартістю, якого ще немає в A і приєднання якого до A залишає $A \cup \{(u,v)\}$ деревом. Відповідний алгоритм називають алгоритмом Пріма [144]. Для графа з рис. 6.8 роботу алгоритму Пріма описує рис. 6.9.

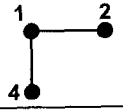
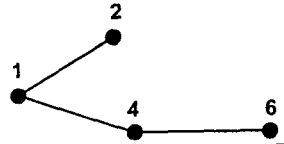
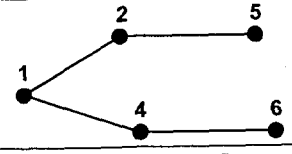
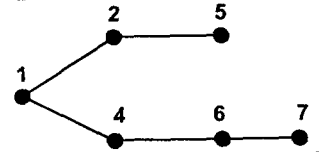
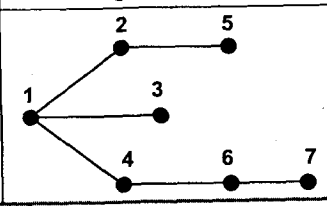
Кандидати	Ребра	Вартість	Остове дерево
(1, 2) (1, 4) (4, 6)	(1, 2)	10	
(2, 5) (1, 4)	(1, 4)	10	
(2, 5) (4, 6)	(4, 6)	10	
(2, 5) (6, 7)	(2, 5)	20	
(6, 7)	(6, 7)		
(1, 3) (5, 7)	(1, 3)	30	

Рис. 6.9. Остове дерево для графа з рис. 6.8

Спробуємо формалізувати алгоритм Пріма. Побудову дерева починаємо з розгляду ребер, які мають мінімальну вартість. Вибираємо одне з них, наприклад, відповідно до лексикографічного порядку кортежів відносно (i, j). Далі приєднуємо до остового дерева T таке ребро (l, k), в якого $l \in T$, а $k \notin T$ і вартість $cost(l, k)$ є мінімальною серед тих ребер, що задовольняють першій частині умови. До того ж приєднання цього ребра має залишити T деревом, конкретніше, дерево T має залишитись зв'язним, і не повинен з'явитися цикл. Процес побудови закінчиться, коли T матиме всі вершини із G.

Алгоритм Пріма реалізує програма 6.6.

```
Program Ex6_6;
 const m = 100;                          {Максимальна кількість вершин графа}
       maxreal = 100.0;                  {Максимальна вартість ребра}
 type int = 1..m;
```

```
    sett = set of int;                    {Множина розглянутих вершин графа}
    llisp = ^node;
    node = record                         {Структура вузла списків}
             i, j: int;                   {Вершини ребра}
             cost: real;                  {Вартість ребра}
             nextm: llisp;                {Покажчик на наступний вузол у головному списку}
             next: llisp;                 {Покажчик на наступний вузол списку суміжності}
                                          {вершини}
             pred: llisp                  {Покажчик на попередній елемент списку}
           end;
    arr = array [int, int] of real;       {Тип масиву задання дерева T}
  var i, j, k, n: int;
    cos, cot: real;                       {cot – значення мінімальної вартості остового дерева T}
    t: arr;
    head, head1: llisp;                   {Покажчики на голови списків}
    first, last: llisp;                   {Покажчики на бар'єрні елементи головного списку}
    fir, las: llisp;                      {Покажчики на бар'єрні елементи списку суміжності}
    p, q, qq: llisp;                      {Робочі покажчики}
    vused: sett;

{Функція перевірки виконання властивості дерева. Процедура Tree набуває значення}
{істина, коли приєднання чергового ребра-кандидата порушує цю властивість}
function tree(i, j: int): boolean;
  var k: int;
    fl: boolean;
begin
  fl := false;
  tree := fl;
  if (t[i, j] <> 0) or (t[j, i] <> 0) then tree := true
    else
      if ((i in vused) and (j in vused)) then tree := true
        else
          if (not(i in vused) and not(j in vused)) then tree := true
            else
              if (i in vused) then
                begin
                  k := 1;
                  while (k <= n) and (k in vused) and not(fl) do
                    begin
                      if (t[k, j] <> 0) or (t[j, k] <> 0) then
                        begin
                          fl := true;
                          tree := fl;
                        end;
                      k := k + 1
                    end;
                end
```



```
            else
              if (j in vused) then
                begin
                  k := 1;
                  while (k <= n) and (k in vused) and not(fl) do
                    begin
                      if (t[k, i] <> 0) or (t[i, k] <> 0) then
                        begin
                          fl := true;
                          tree := fl;
                        end;
                      k := k + 1
                    end;
                end;
end;

{Процедура вставки вузла в упорядкований за зростанням список вузлів суміжності,}
{де head – покажчик на голову списку, cost – вартість ребра (i, j)}
procedure inslisp(var head: llisp; cost: real; i, j: int);
  var p, q: llisp;
begin
    new(q);                               {Породження та наповнення вузла}
    q^.cost := cost;
    q^.i := i;
    q^.j := j;
    p := head;
    while(q^.cost > p^.cost) do
      p := p^.next;
    p^.pred^.next := q;
    q^.next := p;
    q^.pred := p^.pred;
    p^.pred := q;
end;

{Аналогічна за семантикою та параметрами процедура вставки елементу}
{в упорядкований за зростанням головний список}
procedure insmainlisp(var head, head1: llisp);
  var p, q: llisp;
begin
    new(q);
    q^.cost := head1^.next^.cost;
    q^.i := head1^.next^.i;
    q^.j := head1^.next^.j;
    p := head;
    while(q^.cost > p^.cost) do
      p := p^.nextm;
```

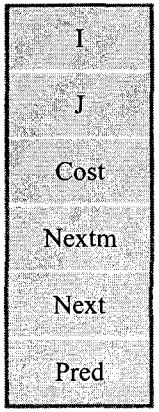
```
    p^.pred^.nextm := q;
    q^.nextm := p;
    q^.pred := p^.pred;
    p^.pred := q;
    q^.next := head1
end;

begin
    read(n);                          {n – реальна кількість вершин у графі}
    new(first);                       {Побудова бар'єрних вузлів головного списку}
    first^.cost := –1.0;
    head := first;
    new(last);
    first^.nextm := last;
    last^.cost := maxreal;
    last^.nextm := nil;
    last^.pred := first;
    for i := 1 to n do                {Введення графа, побудова початкового головного}
      begin                           {списку вузлів і списку суміжності вузлів}
        readln;
        new(fir);
        fir^.cost := –1.0;
        head1 := fir;
        new(las);
        fir^.next := las;
        las^.cost := maxreal;
        las^.next := nil;
        las^.pred := fir;
        read(cos, j);
        while cos <> maxreal do
          begin
            inslisp(head1, cos, i, j);
            read(cos, j)
          end;
        insmainlisp(head, head1);
      end;
    for i := 1 to n do                {Початкова ініціалізація результуючого дерева T}
      for j := 1 to n do
        t[i, j] := 0;
    p := head^.nextm;
    vused := [ p^.i];                 {Вибір перших двох вершин дерева T,}
    vused := vused + [p^.j];          {які входять у ребро мінімальної вартості}
    t[p^.i, p^.j] := p^.cost;
    cot := p^.cost;
{Початкове значення вартості мінімального остового дерева T}
```



```
    head^.nextm := p^.nextm;
{Видалення обробленого вузла із головного списку та пошук чергового ребра-
кандидата серед вершин, суміжних до занесеної вершини}
    head1 := p^.next;
    q := head1^.next;
    qq := q^.next;
    if q^.cost <> maxreal then
      begin
        head1^.next := qq;
        if qq^.cost <> maxreal then insmainlisp(head, head1);
{Занесення нового ребра до головного списку}
      end;
    for k := 1 to n – 2 do            {Головний цикл побудови остового дерева}
      begin
        p := head^.nextm;             {Покажчик вказує на чергове ребро-кандидат}
        while (tree(p^.i, p^.j)) do
{Знаходження першого ребра-кандидата, яке не порушує властивості дерева}
          p := p^.nextm;
        t[p^.i, p^.j] := p^.cost;
        if (p^.i in vused) then
          vused := vused + [p^.j]
        else
          vused := vused + [p^.i] ;
        cot := cot + p^.cost;
        p^.pred^.nextm := p^.nextm;
        head1 := p^.next;
        q := head1^.next;
        qq := q^.next;
        if q^.cost <> maxreal then
          begin
            head1^.next := qq;
            if qq^.cost <> maxreal then insmainlisp(head, head1);
          end;
      end;
    writeln;
    for i := 1 to n do                {Виведення результату}
      begin
        for j := 1 to n do
          write(t[i, j]: 2, ' ');
        writeln;
      end;
    writeln('Мінімальна вартість остового дерева = ', cot);
end.
```

Програма 6.6. Реалізація алгоритму Пріма.



Перша вершина ребра

Друга вершина ребра

Вартість ребра

Покажчик на наступний елемент у головному списку

Покажчик на наступний вузол або на відповідний список суміжності

Покажчик на попередній елемент списку

Рис. 6.10. Структура вузла

Для вибору мінімального ребра в програмі основним елементом є список з вузлом, структуру якого ілюструє рис. 6.10.

Списки суміжності вузла одразу будують за зростанням відносно вартості ребер. Головний список впорядкований за зростанням перших елементів списків суміжності. Для відображення вислідного остового дерева мінімальної вартості використовують матрицю суміжності, яка реалізована за допомогою масиву. Щоб спростити роботу перебігу по списках, в них використовують бар'єрні перший і останній елементи. Робота алгоритму полягає в такому. Вибирають перший значущий елемент із головного списку. Відповідне ребро заносять в остове дерево. Вузол видаляють з головного списку. Далі йде ітераційний процес. Поки в дереві не з'являться всі вершини графа, в головному списку вибирають чергового кандидата, який задовольняє умові мінімальної вартості ребра і не порушує властивості дерева не мати циклу, у разі його занесення до результуючого дерева. Відповідний вузол видаляють з головного списку. Список модифікують занесенням чергового вузла із відповідного списку суміжностей так, щоб не порушити попередню впорядкованість вузла.

Очевидно, що часову складність алгоритму визначає той фрагмент, у якому відбувається початкове заповнення (ініціалізація) матриці задання остового дерева.

Звернувшись до леми 6.2, можемо отримати іншу інтерпретацію оптимізаційного критерію, яка дістала назву алгоритму Крускала – знаходження остового дерева мінімальної вартості графа [132].

Введемо до розгляду ліс остових дерев, який описує набір VS неперетинних множин вузлів. Кожна множина W із VS задаватиме зв'язну множину вузлів остового дерева в остовому лісі. Спочатку $VS = \{W_i, i = 1..n\}$, де $W_i = v_i$, $i = 1..n$. Ребра вибирають за зростанням вартості. Нехай на черговому кроці вибрали ребро $(v, w) \in E$. Якщо ребро (v, w) належить тільки одному дереву в остовому лісі дерев VS, тоді це ребро видаляють з пріоритетної черги і з подальшого розгляду. В іншому разі, коли $v \in W_1$, $w \in W_2$, $W_1 \in VS$, $W_2 \in VS$, тоді зливаємо два дерева W_1 і W_2 в одне і додаємо ребро (v, w) до кінцевого остового дерева T. Процес закінчується, коли ліс VS складатиметься лише з одного дерева. Цей алгоритм називають алгоритмом Крускала. Структуру алгоритму описує процедура *Kruscal*.

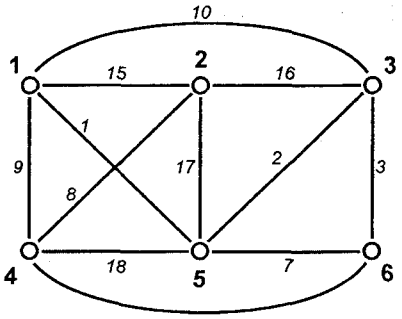


Рис. 6.11. Неорієнтований граф

Procedure Kruscal (V – множина вершин, E – множина ребер і C – вартість ребра)
begin
1. $T \leftarrow 0$;
2. $VS \leftarrow 0$;
3. Побудувати купу Q всіх ребер E згідно зі зростанням функції вартості $c(e)$;
4. Для $v_i \in V$ створити множину $W_i = \{v_i\}$ у VS, $i = 1, 2, ..., n$;
5. Поки ліс остових дерев VS не є одним деревом, робити
початок
6. Взяти ребро (v, u) із голови купи Q;
7. Видалити (v, u) із купи і поновити в Q властивість купи;
8. Якщо в лісі VS знайдуться такі два дерева W_i і W_j, що $v \in W_i$, а $u \in W_j$, тоді зробити таке:
початок
9. В лісі VS замінити множини W_i і W_j на $W_i \cup W_j$
10. $T \leftarrow (v, u)$
кінець
кінець
end.

Розглянемо роботу алгоритму Крускала на графі, зображеному на рис. 6.11.

Таблиця 6.8. Послідовність побудови остового дерева для неорієнтованого графа (див. рис. 6.11)

Голова купи Q	Остовий ліс дерев VS після дії пунктів 8–9	Дія пунктів 8–9	Результуюче дерево T
Пуста	Ще не дійшла справа	Ще не дійшла справа	0
(V_1,V_5)	$\{V_1,V_5\}$	Додати	$\{V_1,V_5\},\{V_2\},\{V_3\},\{V_4\},\{V_6\}$
(V_3,V_5)	$\{(V_1,V_5)(V_3,V_5)\}$	"	$\{V_1,V_3,V_5\}\{V_2\},\{V_4\}\{V_6\}$
(V_3,V_6)	$\{(V_1,V_5)(V_3,V_5)(V_3,V_6)\}$	"	$\{V_1,V_3,V_5,V_6\}\{V_2\}\{V_4\}$
(V_5,V_6)	$\{(V_1,V_5)(V_3,V_5)(V_3,V_6)\}$	Залишити	$\{V_1,V_3,V_5,V_6\}\{V_2\}\{V_4\}$
(V_2,V_4)	$\{(V_1,V_5)(V_3,V_5)(V_3,V_6),(V_2,V_4)\}$	Додати	$\{V_1,V_3,V_5,V_6\},\{V_2,V_4\}$
(V_1,V_4)	$\{(V_1,V_5)(V_3,V_5)(V_3,V_6)(V_2,V_4)(V_1,V_4)\}$	"	$\{V_1,V_2,V_3,V_4,V_5,V_6\}$

Теорема 6.4. Алгоритм Крускала знаходить остове дерево найменшої вартості для зв'язного графа G за час $O(e\log e)$, де $e = |E|$.

Доведення. Коректність алгоритму випливає безпосередньо із леми 6.4.2 і умов алгоритму, зосереджених в рядках 8–9.

Для кроків 1–2 алгоритм потребує константного часу. Купу (крок 3) можна побудувати за час $O(\log e)$. На виконання кроку 4 потрібно $O(n)$. Якщо припустити, що виконання кроків 5–10 проходить d разів, де $d \le e$, і врахувати ті факти, що вибрати елемент з голови купи можна за $O(1)$, поновити властивість купи можна за $O(\log e)$; кроки 8–9 (знаходження і об'єднання неперетинних множин) можна зробити за час $O(eG(e))$, де $G(n)$ визначають як найменше число k, для якого $F(k) \ge n(F(0) = 1, F(i) = 2, ...,$ для $i > 0)$; тоді загальний час роботи алгоритму визначатиметься $O(e\log e)$.

Реалізацію алгоритму Крускала на Паскалі наведено в програмі 6.7.

```
Program kruskal;
  const m = 100;
  type int = – m..m;
       node = record                          {Структура базової вхідної інформації}
                i, j: int;                    {Вершини ребра}
                cost: integer                 {Вартість ребра}
              end;
       arr = array [int] of node;             {Початкова інформація про граф}
       ar = array [int] of integer;           {Задання множин дерев}
       arrr = array [int, int] of int;        {Задання результуючого остового дерева}
  var  u, v, l, kk, j, n, ii, jj, nonempty, q, p: int;
       parent: ar;
       a: arr;
       t: arrr;
       mincost, costvu: integer;              {Мінімальна вартість остового дерева}
       temp: node;

{Процедура пересипки adjust; її параметри ii та n задають область масиву aa[ii..n],
яка задовольняє умові сортувального дерева; корінь нового дерева розміщується
в aa[ii]}
procedure adjust(ii, n: int; var aa: arr);
  var  j: int;
       item: node;
  label exit;
begin
    jj := 2 * ii;
    item := aa[ii];
    while jj <= n do
      begin
        if (jj < n) and (aa[jj].cost > aa[jj + 1].cost) then jj := jj + 1;
        if item.cost <= aa[jj].cost then goto exit
        else
          begin
            aa[(jj div 2)] := aa[jj];
            jj := 2 * jj
          end;
      end;
    exit: aa[(jj div 2)] := item;
end;

{Процедура heapify робить все бінарне дерево купою, дерево задається масивом aa[n]}
procedure heapify(n: int; var a: arr);
  var  i: int;
begin
    for i := n div 2 downto 1 do
      adjust(i, n, a)
end;

{Процедури union і find дають змогу виконати n операцій, об'єднати і знайти за оптимальний час. Для задання множин (елементами яких є ребра графа, які задані у вигляді кортежу вершин ребра (u, v)) використовують ліс дерев VS. Кожну множину W описує бінарне дерево. Кореню цього дерева приписують ім'я множини. Процедура union за допомогою масиву parent знаходить корені множин з іменами i, j та реалізує операцію об'єднання множин шляхом під'єднання кореня дерева з меншою кількістю вузлів як сина кореня дерева з більшою кількістю вузлів. parent[k] = l, якщо l > 0, тоді l – батько вузла k, а якщо l < 0, тоді abs(l) визначає кількість вершин в дереві k}
procedure union(i, j: int; var parent: ar);
  var  x: integer;
begin
    x := parent[i] + parent[j];
    if parent[i] > parent[j] then
      begin
        parent[i] := j;
        parent[j] := x
      end
    else
      begin
        parent[j] := i;
        parent[i] := x
      end
end;

{Процедура find, починаючи з вершини i (ребро-кандидат (u, v)), йде по шляху, який веде до кореня j (знаходить множину W). Після досягнення кореня видається ім'я множини j та приєднується кожен вузол пройденого шляху як син до кореня}
procedure find(i: int; var j: int; var parent: ar);
  var t, k, l: int;
```

```
begin
    l := i;
    while parent[l] > 0 do
      l := parent[l];
    k := i;
    while (k <> l) do
      begin
        t := parent[k];
        parent[k] := l;
        k := t;
      end;
    j := l;
end;

{Початок головної програми}
begin
    readln(j, n);          { j – кількість ребер графа G, n – число вершин графа G}
    for l := 1 to j do
      read(a[l].i, a[l].j, a[l].cost);      {a[l].i – початок ребра, a[l].j – кінець ребра,}
                                            {a[l].cost – вартість ребра}
    heapify(j, a);                          {Побудова пріоритетної черги Q у вигляді купи,}
                                            {яку задає масив a[j]}
    for l := 1 to n do
      parent[l] := –1;                      {Для всіх вершин графа створити ліс VS дерев W_i}
                                            {1 <= i <= n; кожне дерево містить тільки одну}
                                            {вершину (корінь)}
    ii := 0;
    mincost := 0;       {Початкове значення мінімальної вартості остового дерева}
    nonempty := j;
    while (ii < n – 1) and (nonempty > 0) do        {Поки купа не пуста – робити}
      begin
        u := a[1].i;                        {Узяти чергове ребро (u, v) з голови купи}
        v := a[1].j;
        costvu := a[1].cost;                {Визначити його вартість}
        temp := a[nonempty];                {Видалити ребро з купи}
        a[nonempty] := a[1];
        a[1] := temp;
        nonempty := nonempty – 1;
{Якщо черга не пуста, тоді підправити її так, щоб вона задовольняла
властивостям купи}
        if nonempty > 1 then adjust(1, nonempty, a);
        find(u, q, parent);         {Знайти множину (q), яка містить вершину u}
        find(v, p, parent);         {Знайти множину (p), яка містить вершину v}
        if q <> p then              {Якщо ці множини рівні, тоді}
          begin
            ii := ii + 1;       {занести ребро (u, v) до дерева T, додавши вартість}
            t[ii, 1] := u;      {цього ребра та об'єднавши відповідні множини}
            t[ii, 2] := v;
            mincost := mincost + costvu;
            union (q, p, parent)
          end
      end;
{Виведення результатів роботи програми}
    if ii <> n – 1 then                     {Граф незв'язний}
      writeln('Остового дерева не існує')
    else
      begin
        writeln('Остове дерево мінімальної вартості');
        writeln('містить ребра: ');
        for kk := 1 to ii do
          writeln('(', t[kk, 1], ', ', t[kk, 2], ')');
        writeln(', Вартість остового дерева: ', mincost)
      end;
end.
```

Програма 6.7. Алгоритм Крускала

Проведіть порівняльний аналіз переваг і недоліків застосування алгоритмів Пріма і Крускала для розв'язання задачі знаходження мінімального остового дерева.

6.4.4. *Матроїди*

Розглянуті приклади застосування жадібних стратегій підтвердили важливість питання: як узнати, чи дасть жадібний алгоритм оптимальний розв'язок? Загальних рецептів не існує, але можна виділити дві особливості, характерні для цього підходу: *принцип жадібного вибору і оптимальність для підзадач.*

Принцип жадібного вибору. Кажуть, що до оптимізаційної задачі можна застосувати принцип жадібного вибору, якщо послідовність локально оптимальних (жадібних) виборів дає глобально оптимальний розв'язок. Довести це не завжди просто, хоча схема доведення стандартна (згадайте таке доведення для алгоритму Дейкстри). Спочатку доводять, що жадібний алгоритм не закриває шлях до оптимального розв'язку за принципами: для деякого розв'язку знайдеться інший, узгоджений з жадібним вибором і не гірший за перший. Потім доводять, що підзадача, яка виникла після жадібного вибору на першому кроці, аналогічна початковій, і далі роздуми проводять за індукцією.

Оптимальність для підзадач. Ця властивість нагадує принцип оптимальності Белмана [11] і означає, що оптимальний розв'язок всієї задачі містить у собі і оптимальне розв'язання підзадач.

Остання властивість є характерною і для задач динамічного програмування, тому іноді жадібний алгоритм використовують і там, де він може не дати глобального оптимума. Продемонструємо останнє на прикладі вже розглянутої нами задачі про рюкзак, уточнивши два варіанти її постановки: дискретна задача про рюкзак вимагає, щоб ми працювали з цілими числами (предмети не можна дробити на частини), і неперервна задача про рюкзак дає змогу працювати з дійсними числами (предмети можна дробити на частини та складати ці частини в рюкзак).

Зрозуміло, що в обох випадках справджується принцип оптимальності для підзадач. Дійсно, вибравши деякий предмет з номером j із оптимального набору предметів, на наступному кроці отримаємо аналогічну задачу, тільки зі зміненими параметрами, тобто максимальний розмір рюкзака стане $M - w_i$, а набір предметів складатиметься із $n - 1$ предмета (всі, крім j-го). Жадібний алгоритм знаходить глобальний оптимум тільки в неперервному випадку. Для дискретного випадку розглянемо контрприклад.

Приклад 6.4. Нехай в рюкзак можна покласти предмети із загальною максимально можливою масою 80 кг. Серед предметів наповнення є три предмети з вагою 20, 30 і 50 кг і відповідною вартістю 80, 90 і 100 грн. Зиск їх наповнення в рюкзак відповідно становитиме 80 : 20 = 4, 90 : 30 = = 3, 100 : 50 = 2. Тому жадібний алгоритм покладе спочатку перший предмет (20 кг), потім другий (30 кг), а третій вже не поміститься. Загальна вартість занесених предметів становитиме 80 + 90 = 170 (грн). Проте очевидно, що оптимальним розв'язанням буде занесення в рюкзак другого і третього предметів. *Дійсно, вони вміщуються в рюкзаку*: 30 + + 50 = 80 (кг), а їх сумарна вартість: 90 + 100 = 190 (грн) більша за 170 грн.

Останній приклад ще раз наголошує на тому, що за використання жадібних алгоритмів найактуальнішим є питання: коли вигідно бути жадібним? Відповідь на це питання дає теорія матроїдів.

Розглянемо оптимізаційні задачі такого типу. Нехай задані скінченна множина E, набір її підмножин $I \subseteq P(E)$; функція w: $E \to R^+$ позначає множину дійсних невід'ємних чисел. Потрібно знайти підмножину $S \in I$ з найбільшою сумою $\sum_{e \in S} w(e)$.

Тоді питання про корисність використання жадібності можна перефразувати так: за яких умов відносно набору I жадібний алгоритм вірно розв'язав поставлену задачу для довільної функції w? На це питання існує відома відповідь: достатньо, щоб пара $\langle E, I \rangle$ була матроїдом [66, 127, 145].

Матроїди було введено Х. Уїтні у праці [151] для дослідження абстрактної теорії лінійної залежності. Використовуватимемо таке визначення матроїда [62].

Матроїдом називають довільну пару $M = (S, \Im)$, що задовольняє умовам:

1) S – скінченна непуста множина;

2) $\Im$ – непустий набір незалежних підмножин S. Множини набору $\Im$ називають незалежними множинами, наголошуючи на зв'язок теорії матроїдів з теорією лінійної залежності. До того ж має виконуватися така властивість: із $B \in \Im, A \subseteq B$ випливає $A \in \Im$. Набір $\Im$, що задовольняє цій умові, називають *спадковим*;

3) якщо $A \in \Im, B \in \Im, |A| < |B|$, тоді існуватиме такий елемент $x \in B \setminus A$, що $A \cup \{x\} \in \Im$. Цю властивість набору $\Im$ називають *властивістю заміни*.

Розглянемо приклади найпростіших матроїдів.

1. **Матричний матроїд.** S – множина всіх рядків матриці. Множину рядків вважають незалежною, якщо вони є незалежними в традиційному алгебричному визначенні.

2. **Графовий матроїд** $(S_G, \Im_G)$ будують із непустого неорієнтованого графа G так: S_G складається з множини ребер графа, а $\Im_G$ складається зі всіх ациклічних множин ребер (ліс остових дерев).

Доведіть, що визначена таким чином пара $(S_G, \Im_G)$ є матроїдом. Для цього необхідно і достатньо показати, що справджуються три властивості матроїда із визначення.

Якщо $M = (S, \Im)$, тоді елемент $x \notin A \in \Im$ називають **незалежним від** A, якщо множина $A \cup \{x\}$ незалежна. Наприклад, у графовому матроїді ребро e незалежне від лісу A тоді й тільки тоді, коли його внесення до A не створює циклу. Незалежну підмножину матроїда називають **максимальною**, якщо вона не міститься більше у жодній незалежній підмножині. Відома теорема [62].

Теорема 6.5. Всі максимально незалежні підмножини даного матроїда складаються з однакового числа елементів.

Матроїд $M = (S, \Im)$ називають **навантаженим**, якщо на множині S задано деяку вагову функцію w зі значеннями із множини додатних чисел. Функція w розповсюджується за адитивністю на всі підмножини множини A; вагу підмножини визначають як суму ваг його елементів: $w(A) = \sum_{x \in A} w(x)$. Наприклад, якщо M_G – графовий матроїд, а $w(e)$ – довжина ребра e, тоді $w(A)$ – сума довжин ребер підграфа A.

Нарешті, більшість оптимізаційних задач, що розв'язуються жадібними алгоритмами, зводиться до задачі знаходження в деякому навантаженому матроїді $M = (S, \Im)$ незалежної підмножини $A \subseteq S$ максимальної ваги. Цю підмножину називають **оптимальною** підмножиною навантаженого

матроїда. Оскільки ваги всіх елементів додатні, оптимальна підмножина автоматично буде і максимально незалежною підмножиною.

Згідно з працею [62], можна привести жадібний алгоритм, що знаходить оптимальну подмножину A в довільному навантаженому матроїді M. (Якщо $M = (S, \Im)$, тоді скористаємося позначенням $S = S[M]$ і $\Im = \Im[M]$, вагова функція позначається w.) Структуру цього алгоритму наведено у процедурі *Greedymatr*.

Procedure *Greedymatr*(M, w);
begin
1. $A \leftarrow 0$.
2. Впорядкувати $S[M]$ відповідно до незростання ваг.
3. **for** $x \in S[M]$ (перебираємо всі x в зазначеному порядку) **do**
4. **if** $A \cup \{x\} \in \Im[M]$
5. **then** $A \leftarrow A \cup \{x\}$;
6. return A;
end.

Оцінимо час роботи алгоритму. Зрозуміло, що впорядкування можна зробити за час $O(n \log n)$, де $n = |S|$. Перевірку незалежності множини (стрічка 4) проводять n разів; якщо кожна така перевірка займає час $f(n)$, тоді загальний час роботи алгоритму визначатиметься $O(n \log n + nf(n))$.

У працях [62, 116] також можна знайти доведення того, що алгоритм дійсно знаходить оптимальну підмножину.

Матричні матроїди. Розглянемо матрицю

$$A = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & & \dots & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}.$$

Кожна така матриця визначає матроїд $M(A) = (S, \Im)$, де S – множина її стовпців, а $B \in \Im$ тоді й тільки тоді, коли множина стовпців B лінійно незалежна. Цей матроїд називають матроїдом матриці A. Матроїд називають матричним, якщо він є (ізоморфним з) матроїдом деякої матриці.

У випадку матричного матроїда загальна схема жадібного відбору перетворюється в метод виключення Гауса для матриці A розмірності $m \times n$, стовпці якої впорядковані за спаданням ваг (ваги невід'ємні). Нехай на вхід алгоритму подається матриця A розмірності $m \times n$, стовпці якої впорядковані за спаданням ваг (ваги невід'ємні). Тоді процедура *Greedymatr* трансформується в алгоритм 6.2, який знаходить незалежну множину стовпців з найбільшою сумою ваг. Номери стовпців знаходитимуться в S.


```
begin
  S := ∅
  for j := 1 to n do
    begin
      i := 0;
      while (A[i, j] = ∅) and (i < m) do i := i + 1;
      if a[i, j] <> 0 then                          {j-й стовпець ненульовий}
        begin
          S := S ∪ {j};
          for k := j = 1 to n do
            for l := 1 to m do
              A[l, k] := A[l, k] – A[l, j] * A[i, k] / A[i, j]
        end                                         {A [l, k] = 0 для n ≥ m ≥ j + 1}
    end
end.
```

Алгоритм 6.2. Жадібний алгоритм для матричного матроїда

Доведіть, що числова оцінка цього алгоритму – $O(n^2 m)$.

Графові матроїди. Для неорієнтованого графа $G = (V, E)$ матроїд визначають так: $M(G) = (S, \Im)$, де $\Im = \{A \subseteq S$: *граф* (V, A) – *ациклічний*$\}$.

Матроїд $M(G)$ називають матроїдом графа G. Довільний матроїд називають графовим, якщо він є (ізоморфним з) матроїдом деякого графа.

Враховуючи, що граф можна задати матрицею інцидентності, кожен графовий матроїд можна трактувати як матричний матроїд. Останній відповідає матриці інцидентності графа G, елементи якої набувають значення із поля $Z^2 = \{0, 1\}$. Додавання в такому полі виконують за модулем 2, а множення – звичайне множення цілих чисел.

Відома теорема [66].

Теорема 6.6. Нехай $G = \langle V, E \rangle$ – звичайний граф, і A – його матриця інцидентності. Підмножина стовпців матриці A є лінійно залежною над полем Z^2 тоді й тільки тоді, коли відповідна їй підмножина ребер містить цикл.

Тому для пошуку бази матроїда $M(G)$ з найменшою вагою (мінімального дерева стягування, в загальному випадку – графа, всі компоненти зв'язності якого є деревами) можна використати жадібний алгоритм для матроїдів. Отже, за умови, що $u = |V|$ і $m = |E|$, таку базу матроїда можна знайти за $O(n^2 m)$.

Розглянуті нами (див. 6.4.3) алгоритми Пріма і Крускала є ліпшими для розв'язання цієї задачі. Останній посідає значне місце в теорії матроїдів.

Жадібний алгоритм для матроїдів був сформульований як узагальнення алгоритму Крускала з матроїда графа на довільний матроїд.

Задачі та вправи до розділу 6

1. Розглянемо класичну задачу лінійного програмування – задачу оптимального використання (розподілу) ресурсів виробництва. Для виробництва певних товарів використовують деяке число різного типу ресурсів. Відомо, скільки одиниць кожного ресурсу використовують для виробництва одиниці кожного товару. Потрібно так запланувати виробництво товарів, щоб за використання наявних ресурсів загальний прибуток від виробництва був найбільшим. Як приклад візьміть підприємство, що виробляє три типи продукції і використовує у цьому разі чотири типи ресурсів. Витрати ресурсів на кожен тип продукції, запаси ресурсів і прибутки від реалізації одиниці кожного типу продукції задайте самі;
   а) побудуйте математичну модель цієї задачі;
   б) розв'яжіть цю задачу;
   в) проведіть післяоптимізаційний аналіз.
2. Запрограмуйте описаний в розділі 6.4.1 метод найшвидшого спуску.
3. Напишіть програму k-го злиття файлів, що розглядалося в розділі 6.4.1.
4. Напишіть процедури побудови коду Хаффмана та його розкодування (розділ 6.4.1).
5. Доповніть деталі опису й аналізу варіанта алгоритму Дейкстри зі складністю $O(m \log n)$.
6. Намалюйте деякий навантажений граф. Застосуйте до нього алгоритм Дейкстри.
7. Напишіть програму реалізації алгоритму 6.2.

Розділ 7

ДИНАМІЧНЕ ПРОГРАМУВАННЯ

7.1. Загальна характеристика методу

Розглянемо ще раз метод розв'язання задач, побудований за схемою послідовного аналізу варіантів, тобто з використанням процедур, які на основі побічних оцінок відкидають усі допустимі розв'язання, серед яких не може бути оптимального. З часом відбувається поступове стиснення множини конкурентоспроможних варіантів. Наприкінці залишаються один або два варіанти, які й порівнюють.

Перетворити цей загальний підхід у систему формальних процедур дуже важко. Нині багато зроблено в цьому напрямі. Перші ідеї належать А. А. Маркову, їх розвивали американці Д. Вальд, Р. Айзекс, Р. Беллман [11]. Значний внесок у створення загального формалізму послідовного аналізу варіантів (схема формалізації Михалевича, метод «київський вінник») належить київській школі математиків на чолі з В. С. Михалевичем [73, 74].

Методи динамічної оптимізації значною мірою використовують глибоке знання сутностей задач розв'язання. Аналогічно методу РП задачу розбивають на підзадачі. Останні розв'язують, а з отриманих підрозв'язків будують загальний розв'язок. Підзадачі залежні між собою, тобто у них є загальні підпідзадачі. Алгоритми динамічного програмування розв'язуватимуть кожну із підзадач тільки один раз, запам'ятовуючи отримані розв'язки у вигляді спеціальних таблиць. Тому тут ми звернемо увагу не на загальну методологію підходу, а обмежимося розглядом декількох задач, які досить повно ілюструють загальні концепції підходу.

Динамічне програмування – це алгоритмічний метод, який використовують, коли шуканий розв'язок задачі можна зобразити як об'єднання деяких результатів оптимальних розв'язків певних підзадач, що трапляються в різних гілках побудови загального розв'язання і обраховуватимуться тільки один раз.

У попередніх розділах описано чимало таких задач. Наприклад, розв'язання задачі про рюкзак (див. 6.4.1) можна розглядати і так. Для знаходження значення x_i, $1 \le i \le n$, потрібно знайти розв'язки x_1, x_2, x_3 і т. д. Оптимальна послідовність цих розв'язків має максимізувати цільову функцію $\sum p_i x_i$ та задовольняти умовам $\sum w_i x_i \le M$ і $0 \le x_i \le 1$.

Для деяких задач, які можна розглядати в цій інтерпретації, оптимальну послідовність розв'язків можна знайти, *приймаючи на кожному кроці правильний розв'язок* (наприклад, задачі, що можна розв'язати жадібним методом). Для багатьох інших задач знайти такий розв'язок неможливо.

Зрозуміло, що одним із способів розв'язання задач, для яких неможливо на кожному кроці знаходити розв'язки, що призводять до оптимальної послідовності, є перевірка всіх можливих послідовностей розв'язків. Динамічне програмування значно зменшує кількість варіантів, що слід перевірити, за рахунок виключення послідовностей розв'язків, які не можуть бути оптимальними. У динамічному програмуванні оптимальної послідовності розв'язків досягають завдяки явному зверненню до *принципу оптимальності*. Він стверджує: ***оптимальна послідовність розв'язків має таку властивість, що, незважаючи на початковий стан та розв'язок, серед розв'язків, що залишились, завжди міститься оптимальна послідовність розв'язків щодо стану, який утворився після прийняття першого розв'язку.*** Отже, істотна різниця між жадібним методом і динамічним програмуванням полягає в тому, що жадібний метод завжди генерує *одну* послідовність розв'язків, а у разі застосування динамічного програмування може утворюватись *декілька* послідовностей розв'язків.

Уточнимо вищенаведене на прикладі одного із варіантів розв'язання вже розглянутої нами задачі знаходження найкоротшого шляху. Припустимо, що $i, i_1, i_2, ..., i_k, j$ – найкоротший шлях від i до j. Після аналізу вершини i було прийнято рішення відвідати вершину i_1. Після цього стан задачі визначатиметься вершиною i_1 та необхідністю знайти шлях від i_1 до j. Цілком зрозуміло, що послідовність $i_1, i_2, ..., i_k, j$ мусить містити найкоротший шлях від i_1 до j. Якщо це не так, тоді нехай $i_1, r_1, r_2, ..., r_q, j$ – найкоротший шлях від i_1 до j. Тоді шлях $i, i_1, r_1, ..., r_q, j$ буде коротшим за шлях $i, i_1, i_2, ..., i_k, j$. Отже, принцип оптимальності можна застосувати і до цієї задачі.

Е. Горовіц виділяє такі загальні риси методу [127]. Нехай S_0 – початковий стан задачі. Припустимо, що треба прийняти n розв'язків d_i, $1 \le i \le n$. Нехай $D_1 = \{r_1, r_2, ..., r_j\}$ – це множина можливих значень для розв'язку d_1. Нехай S_i – це стан задачі, який утворюється після вибору розв'язку r_i, $1 \le i \le j$. Нехай Γ_i буде оптимальною послідовністю розв'язків відповідно до стану задачі S_i. Тоді, згідно з принципом оптимальності, оптимальну послідовність розв'язків відповідно до S_0 визначають як найліпшу послідовність розв'язків $r_i\Gamma_i$, $1 \le i \le j$.

У застосуванні до задачі про найкоротший шлях ця стратегія набуде такого вигляду. Нехай A_i – множина вершин, суміжних з вершиною i. Для кожної вершини $k \in A_i$ нехай Γ_k буде найкоротшим шляхом від k до j. Тоді найкоротший шлях від i до j визначають як найкоротший шлях з множини $\{i, \Gamma_k | k \in A_i\}$.

Для більшості задач рекурсивна природа задання принципу оптимальності приводить до рекурентного типу співвідношень. Алгоритми динамічного програмування розв'язують ці рекурентні рівняння, щоб знайти оптимальний розв'язок заданої задачі. Існує окрема теорія розв'язання рекурентних рівнянь [62, 111, 133, 143].

Формулювати рекурентні співвідношення динамічного програмування можна за допомогою використання двох підходів перегляду варіантів

розв'язань: прямого та оберненого. Нехай $x_1, x_2, ..., x_n$ – змінні, що характеризують шукану послідовність розв'язків. У прямому перегляді побудова розв'язку x_i формулюється в термінах оптимальної послідовності розв'язків для $x_{i+1}, ..., x_n$. Для оберненого перегляду формулювання тверджень щодо розв'язку x_i здійснюють у термінах оптимальної послідовності розв'язків для $x_1, ..., x_{i-1}$. Відзначимо, якщо рекурентні співвідношення сформульовані з використанням прямого підходу, тоді рівняння *розв'язують за технікою «перегляду назад»*, тобто починаючи з останнього розв'язку; якщо співвідношення сформульовані з використанням оберненого підходу, їх *розв'язують за технікою «перегляду вперед»*. Завдяки використанню принципу оптимальності послідовності розв'язків, що містять підпослідовності, які не є оптимальними, здебільшого *не розглядаються*. Тому алгоритми динамічного програмування, як правило, мають поліноміальну оцінку.

Інша важлива риса динамічного програмування полягає у тому, що оптимальні розв'язки підзадач можна зберігати в такому вигляді, який запобігає перерахуванню їх значень у разі подальших використань цих підзадач. Здебільшого для цього використовують табличне зберігання. Використання табличних значень робить природним перетворення рекурсивних рівнянь в ітеративні програми. Більшість алгоритмів динамічного програмування, описаних у цій частині, подаватиметься саме в такий спосіб.

За оберненого підходу побудова загального розв'язку із розв'язків підзадач меншої розмірності, обчислення йде від менших підзадач до більших, і відповіді запам'ятовуються в таблиці. Перевага використання таблиці полягає в тому, що, як тільки будь-яку підзадачу розв'язано, її відповідь заносять у таблицю і ніколи вже не обчислюють знову. Розглянемо цю техніку на класичному прикладі побудови послідовності *чисел Фібоначчі*.

Для розв'язання багатьох задач використовують спеціальну послідовність чисел, яку ввів Фібоначчі. N-не число Фібоначчі рекурсивно визначають так:

$$F(N) = \begin{cases} 0, \text{ якщо } N = 0; \\ 1, \text{ якщо } N = 1; \\ F(N-1) + F(N-2), \text{ якщо } N > 1. \end{cases}$$

Потрібно написати програму, яка обчислює N-не число Фібоначчі.

Цю задачу можна розв'язати двома способами: за допомогою рекурсії або динамічного програмування.

Розглянемо перший варіант. Його реалізовує рекурсивна функція *fib*.

```
function fib(n: integer): integer;
begin
  if n = 0 then fib := 0
    else if n = 1 then fib := 1
      else fib := fib(n – 1) + fib(n – 2)
end;
```

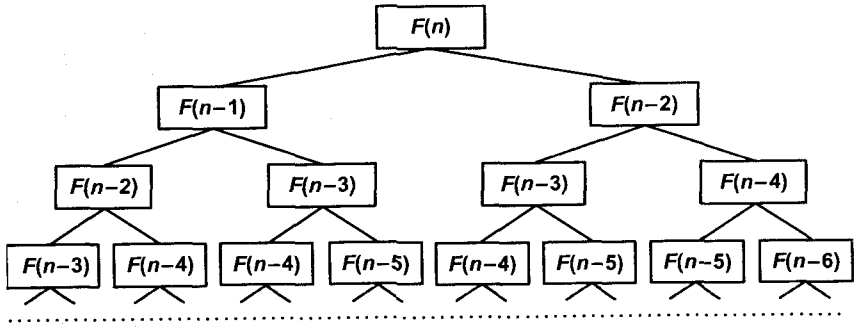


Рис. 7.1. Структура викликів процедури для обчислення *N*-го числа Фібоначчі

Таке використання рекурсії не завжди буває оптимальним. Недолік полягає в тому, що кожного разу для обчислення більшого числа Фібоначчі ми змушені менші числа обчислювати декілька разів (рис. 7.1).

Розглянемо інший підхід. Почнемо обчислення з породження значень менших чисел Фібоначчі. Зберігатимемо два останні числа в таблиці, яка містить дві змінні. Наступне число Фібоначчі утворюють шляхом додавання цих двох змінних. Потім таблицю оновлюють і процес продовжується. У цьому разі алгоритм, що реалізується функцією *fibdin*, обчислюватиметься швидше.

```
function fibdin(n: integer): integer;
  var  i, next, old, new: integer;
begin
    if n = 0 then fibdin := 0
      else
        if n = 1 then fibdin := 1
          else
            begin                          {Занесення початкових значень у таблицю}
              old := 0;
              next := 1;
              for i := 2 to n do
                begin
                  new := old + next;       {Обчислення чергового числа}
                  old := next;             {Модифікація таблиці}
                  next := new;
                end;
              fibdin := new;
            end
end.
```

Отже, динамічне програмування – це такий спосіб побудови алгоритмів, який застосовують, коли розв'язання загальної задачі можна розбити на розв'язання її підзадач і отримувати розв'язок наступної задачі як суперпозицію попередніх розв'язків з використанням принципу оптимальності.

Природним є також питання: коли ж використання основних принципів динамічного програмування буде ефективним? У праці [62] виділено дві ознаки: *оптимальність для підзадач* і *перетинні підзадачі*. Нагадаємо, що задача *задовольняє властивості оптимальності* для підзадач, якщо оптимальний розв'язок задачі містить і оптимальні розв'язки її підзадач. Другу ознаку можна визначити так. Нехай розв'язок загальної задачі можна знайти шляхом комбінування розв'язків деяких підзадач. Число неперетинних (різних) підзадач у цій сукупності підзадач має бути невеликим. Завдячуючи цьому за рекурсивного розв'язання ми багато разів виходитимемо на розв'язок перетинних задач.

Узагалі, здебільшого метод динамічного програмування ліпше застосовувати до розв'язання задач оптимізації. Використовуючи термінологію останніх, загальний алгоритм динамічного програмування можна зобразити так:

1. Опис побудови (структури) оптимальних розв'язань.
2. Виписування рекурентних співвідношень, які зв'язують оптимальне значення параметра для підзадач.
3. За умови використання техніки «перегляду назад» обчислення оптимального значення параметра для підзадач.
4. За отриманою інформацією побудова оптимального розв'язання.

7.2. Знаходження найкоротших шляхів: акценти динамічного програмування

Повернемось до задачі знаходження найкоротших шляхів між усіма парами вершин графа G, що не має циклів від'ємної довжини, акцентуючи нашу увагу на використанні ідей динамічного програмування (метод Форда – Белмана). Нехай $A(i, j)$ – матриця ваг ребер графа. Нагадаємо, що нам потрібно визначити матрицю D, таку що $d(i, j)$ – це довжина найкоротшого шляху від i до j.

Доведіть, що коли ми дозволимо графу G мати цикли від'ємної довжини, тоді найкоротший шлях між двома вершинами буде мати довжину $-\infty$.

Розглянемо найкоротший шлях від i до j в G, $i \neq j$. Цей шлях починається з вершини i, проходить через кілька проміжних вершин (їх може й не бути) та закінчується у вершині j. Ми можемо припустити, що цей шлях не має циклів, тому що існуючий цикл можна знищити без збільшення довжини шляху (немає циклів з від'ємною ціною). Якщо k – проміжна вершина на цьому найкоротшому шляху, тоді підшляхи від i до k та від k до j мають бути найкоротшими шляхами від i до k та від k до j

відповідно. Інакше шлях від i до j буде не мінімальної довжини. Отже, принцип оптимальності справджується. Це дає можливість використовувати загальні ідеї динамічного програмування.

Якщо k – проміжна вершина з найбільшим індексом, тоді шлях від i до k буде найкоротшим шляхом від i до k в G, що не проходить через вершини з індексом, більшим за $k-1$. Шлях від k до j буде також найкоротшим шляхом від k до j в G, якщо він проходить через вершини з індексом, не більшим за $k-1$.

Тому доходимо думки, що побудова найкоротшого шляху від i до j спершу потребує визначення найбільшого індексу проміжної вершини k. Після цього нам слід знайти два найкоротші шляхи. Один від i до k, другий – від k до j. Жоден з них не може проходити через вершину з індексом, більшим за $k-1$. Використовуючи $d^k(i,j)$ для позначення довжини найкоротшого шляху від i до j, що не проходить через вершини з індексом, більшим за k, отримаємо:

$$d(i,j) = \min\left\{\min_{1\le k\le n}\left\{d^{k-1}(i,k)+d^{k-1}(k,j)\right\}, a(i,j)\right\}. \qquad (7.1)$$

Очевидно, $d^0(i,j) = a(i,j)$, $1 \le i \le n$, $1 \le j \le n$. Можемо отримати рекурентне співвідношення для $d^k(i,j)$, використовуючи міркування, схоже з використаним раніше. Найкоротший шлях від i до j проходить через вершини, менші за k, або проходить через k, або не проходить. Якщо проходить, $d^k(i,j) = d^{k-1}(i,k) + d^{k-1}(k,j)$. Якщо ні, тоді жодна проміжна вершина не має індексу, більшого за $k-1$. Тому $d^k(i,j) = d^{k-1}(i,j)$. Поєднуючи, отримаємо:

$$d^k(i,j) = \min\{d^{k-1}(i,j),\ d^{k-1}(i,k) + d^{k-1}(k,j)\},\ k \ge 1. \qquad (7.2)$$

Нагадаємо, що (7.2) не справджується для графів з циклами від'ємної довжини.

Рекурентне співвідношення (7.2) можна розв'язати для d^n, якщо знайти спочатку d^1, потім d^2, d^3 і т. д., тому що в G немає індексу більшого за n, $D(i,j) = d^n(i,j)$. Враховуючи, що визначення роблять ітераційно та, крім того, $d^k(i,k) = d^{k-1}(i,k)$ і $d^k(k,j) = d^{k-1}(k,j)$, тобто коли d^k формується, k-й стовпчик і рядок не змінюються, можна дійти висновку, що верхній індекс у d непотрібний.

Уточніть алгоритм знаходження всіх шляхів та напишіть відповідну програму. Визначте часову оцінку запропонованого алгоритму знаходження мінімальних шляхів.

Деякі уточнення наведеного алгоритму запропонували Уоршал [150] і Флойд [118]. Розглянемо їх.

Процедура *finddistfloyd* реалізує відповідну ідею знаходження відстані між усіма парами вершин в ациклічному графі без контурів від'ємної довжини.


```
Procedure finddistfloyd(n: arr11; A: arr22; var D:arr22);   {A – матриця ваг дуг графа}
  var i, j, m: integer;
begin
    for i := 1 to n do
      for j := 1 to n do
        D[i, j] := A[i, j];
    for i := 1 to n do
      D[i, j] := 0;
    for m := 1 to n do
      for i := 1 to n do
        for j := 1 to n do
          if D[i, m] + D[m, j] < D[i, j] then
            D[i, j] := D[i, m] + D[m, j]
end.
```

Зрозуміло, що часова складність процедури буде $O(n^3)$.

У програмі 7.1 наведено повну реалізацію алгоритму знаходження відстані між усіма парами вершин в ациклічному графі без контурів від'ємної довжини.

```
Program Ex7_1;
  uses crt;
{У розділі const задаємо розмірність матриці (n – кількість вершин у графі)}
  const n = 10;
  type arr = array [1..n, 1..n] of real;
  label again, rep;
{Таблицю ваг ребер графа задають матрицею A. Вона міститься у файлі «graf». За
домовленістю, якщо вага ребра дорівнює 100, то це фактично означає, що прямого
сполучення між цими вершинами немає. Цілі значення вибрано для скорочення запису
   0   10  100   30  100  100   22  100  100    6
  10    0  100  100   11   26  100  100   18  100
 100  100    0    5    8  100  100   20  100   27
  30  100    5    0  100   13  100  100   25  100
 100   11    8  100    0  100   11  100  100   30
 100   26  100   13  100    0  100  100  100  100
  22  100  100  100   11  100    0  100  100   12
 100  100   20  100  100  100  100    0    3  100
 100   18  100   25  100  100  100    3    0  100
   6  100   27  100   30  100   12  100  100    0}
  var A, D: arr; ans: char;
procedure floyd(a: arr; d: arr);
{У розділі var задаються індекси матриці i та j, а також індекс проміжної
вершини m. Файли початкової та результуючої матриць задаються як текстові}
  var i, j, m: integer;
      file1: text;
      file2: text;
```

```
begin
    assign(file1, 'graf');
    reset(file1);
    assign(file2, 'result');
    rewrite(file2);
{У цьому фрагменті відбувається зчитування матриці ваг з файла «graf»}
    i := 0;
    repeat
      i := i + 1;
      while not eoln(file1) do
        for j := 1 to n do
          read(file1, a[i, j]);
      readln(file1);
    until eof(file1);
{Цей уривок програми є ключовим. У ньому відбувається пошук найкоротших шляхів
між вершинами графа. Це виконують за допомогою проміжної вершини m. Якщо
шлях через m є коротшим за попередній, то відбувається переприсвоєння. Таким
чином, у вузлах матриці опиняться найкоротші відстані між вершинами}
    for i := 1 to n do
      for j := 1 to n do
        d[i, j] := a[i, j];
    for m := 1 to n do
      for i := 1 to n do
        for j := 1 to n do
          if d[i, m] + d[m, j] < d[i, j] then
            d[i, j] := d[i, m] + d[m, j];
    clrscr;
{У цьому фрагменті відбувається запис результатів у файл «result» у вигляді
матриці}
    i := 0;
    repeat
      i := i + 1;
      for j := 1 to n do
        write(file2, d[i, j]:2:0, ' ');
      writeln (file2);
    until i = n;
```

{У результаті файл «result» матиме такий вигляд:

```
 0  10  29  30  21  36  18  31  28   6
10   0  19  24  11  26  22  21  18  16
29  19   0   5   8  18  19  20  23  27
30  24   5   0  13  13  24  25  25  32
21  11   8  13   0  26  11  28  29  23
36  26  18  13  26   0  37  38  38  42
18  22  19  24  11  37   0  39  40  12
31  21  20  25  28  38  39   0   3  37
```



```
28  18  23  25  29  38  40   3   0  34
 6  16  27  32  23  42  12  37  34   0
```

Цей фрагмент є підсумковим у поданій програмі. Користувач вводить дві вершини, і програма видає мінімальний шлях між ними}

```
    writeln('Введіть першу вершину'); readln(i);
    writeln('Введіть другу вершину'); readln(j);
    writeln('Найкоротша відстань між цими вершинами – ', d[i, j]:2:0);
    close(file1);
    close(file2);
end;

begin                                          {Початок основної програми}
    textcolor(red);
    textbackground(green);
    clrscr;
    writeln('Обчислення найменших відстаней між усіма вершинами графа');
    readln;
again:
    floyd(a, d);
{Цей фрагмент програми дає змогу повторити пошук найкоротшого шляху для
іншого набору вершин}
rep:
    writeln('Повторити для інших вершин? (1 – так / 2 – ні)');
    readln(ans);
    if ans = '1' then goto again;
    if ans = '2' then exit else goto rep;
    readln;
    clrscr;
end.
```

Програма 7.1. Знаходження мінімальних відстаней між вершинами графа

Важливо відзначити, що поки невідомий алгоритм, який знаходив би відстань між двома фіксованими вершинами за час, кращий за $O(n^3)$.

7.3. Транзитивне замикання бінарного відношення

Під бінарним відношенням на множині V розуміємо довільну підмножину $E \subseteq V \times V$. Відношення буде транзитивним, якщо дотримується умова: при $\langle x, y\rangle \in E$, $\langle y, z\rangle \in E$, буде і $\langle x, z\rangle \in E$ для довільних x, y, $z \in V$. Зрозуміло, що довільне бінарне відношення можна зобразити орієнтованим графом $G = \langle V, E\rangle$. Для нього можна виразити транзитивне відношення $E^* = \{\langle x, y\rangle$: в $\langle V, E\rangle$ існує шлях ненульової довжини із x в $y\}$. До того ж E^* є найменшим транзитивним відношенням, що містить E. Відношення E^* називають *транзитивним замиканням* відношення E.

Якщо відношення E задати графом (V, E) з вагами дуг, рівними одиниці, тоді транзитивне замикання E^* можна отримати за допомогою процедури *finddistfloyd* за час $O(n^3)$. Після завершення роботи алгоритму матимемо: $\langle v_i, v_j \rangle \in E$ тоді й тільки тоді, коли $D[i,j] < +\infty$.

Згідно з Уоршаллом [150], граф зручно задавати так:

$$A[i,j] = \begin{cases} 0, \text{ якщо } \langle v_i, v_j \rangle \notin E; \\ 1, \text{ якщо } \langle v_i, v_j \rangle \in E. \end{cases}$$

Тоді стрічку результату визначення добутку в процедурі *finddistfloyd* можна замінити на такий запис:

$$D[i,j] := D[i,j] \text{ and } (D[o,m] \text{ or } D[m,j]),$$

де and і or – традиційні булевські операції диз'юнкції та кон'юнкції відповідно. Тоді після завершення алгоритму

$$D[i,j] = \begin{cases} 1, \text{ якщо } \langle v_i, v_j \rangle \in E^*; \\ 0, \text{ якщо } \langle v_i, v_j \rangle \notin E^*. \end{cases}$$

Коли відношення E симетричне, зручно будувати його транзитивне замикання за допомогою методів пошуку в глибину або в ширину.

Яка оптимальна часова оцінка за побудови транзитивного замикання методом пошуку в глибину (ширину) для симетричного відношення E?

7.4. Задача обчислення добутку матриць

З погляду яскравішого висвітлення основних характерних рис динамічного програмування розглянемо задачу [9] мінімізації кількості елементарних операцій за обчислення добутку n матриць: $M_1 \times M_2 \times ... \times M_n$, де M_i – матриця розміру $[r_{i-1}, r_i]$.

Зрозуміло, що порядок, за яким ці матриці перемножують, істотно впливає на загальну кількість виконаних операцій. Нагадаємо, що добуток матриці розмірності $p \times d$ на матрицю розмірності $d \times r$ потребує pdr операцій. Нехай є потреба перемножити 4 матриці:

$$M = M_{1[10\times20]} \times M_{2[20\times50]} \times M_{3[50\times1]} \times M_{4[1\times100]}.$$

Якщо обчислювати M за порядком $M_1 \times (M_2 \times (M_3 \times M_4))$, то потрібно буде використати 125 000 операцій, якщо ж за порядком $(M_1 \times (M_2 \times M_3)) \times M_4$ – 2200 операцій. За звичайним порядком $((M_1 \times M_2) \times M_3) \times M_4$ потрібно 11500 операцій.

Процес перебору всіх порядків, в яких можна обчислити добуток n матриць має експоненціальну складність. Проте зрозуміло, що задача задовольняє умовам застосування методу динамічного програмування: загальну задачу можна розбити на підзадачі (кількість операцій, затрачених на множення двох матриць) і справджується принцип оптимальності (оптимальна послідовність розв'язків має таку властивість, що, незалежно від початкового стану і початкового рішення, решта рішень мають бути оптимальною послідовністю розв'язків щодо стану, який отримують в результаті першого розв'язку).

Нехай m_{ij} – найменша складність обчислення $M_i \times M_{i+1} \times ... \times M_j$, $1 \le i \le j \le n$, тоді очевидно, що

$$m_{ij} = \begin{cases} 0, \text{ якщо } j = i; \\ \min\limits_{i \le k < j} \left(m_{ik} + m_{(k+1)j} + r_{i-1} r_k r_j \right), \text{ якщо } j > i, \end{cases}$$

де m_{ik} – найменша складність обчислення $M' = M_i \times M_{i+1} \times ... \times M_k$; $m_{(k+1)j}$ – найменша складність обчислення $M'' = M_{k+1} \times ... \times M_j$; r_{i-1}, r_k, r_j – складність обчислення $M' \times M''$.

Фрагмент алгоритму знаходження оптимального порядку обчислення матриць, що реалізує розв'язання відповідного рекурентного співвідношення і знаходитиме таку послідовність множення матриць, яка забезпечуватиме мінімальну кількість операцій для реалізації множення всіх матриць, наведено в алгоритмі 7.1.

```
begin
   for i := 1 to n do m[i, j] := 0
   for p := 1 to n – 1 do
      for i := 1 to n – 1 do
         begin
            j := i + p;
            m[i, j] := min(m[i, k] + m[k + 1, j] + r[i – 1] * r[k] * r[j])
                       i≤k<j
         end
   write(m[1, n])
end
```

Алгоритм 7.1. Фрагмент алгоритму знаходження порядку обчислення матриць

Зрозуміло, що у разі кожного знаходження оптимального k для деякої підзадачі бажано його десь запам'ятовувати, щоб потім можна було відновити сам порядок множення, а не лише кінцеву мінімальну кількість операцій. Очевидно також, що алгоритм 7.1 знаходить оптимальний порядок множення матриць за час $O(n^3)$.

Повну реалізацію алгоритму 7.1 наведено у програмі 7.2.

```
Program Ex7_4;
{Вхідні дані вводять з файла, що має такий формат: у першому рядку стоїть число,
що відповідає n (кількість матриць); у другому – вказують послідовність цілих
чисел, яка характеризує перелік розмірностей матриць (наприклад, для трьох
матриць розмірностей 10×20, 20×30, 30×40 слід написати послідовність цілих
10 20 30 40)}
  const max_n = 20;
  var  n: integer;
       d: array [0..max_n] of longint;
       m: array [1..max_n, 1..max_n] of longint;
       i, p, j, k, t, w: longint;

procedure ReadInputData(fname: string);
  var inf: text;
      i, j: integer;
begin
    assign(inf, fname);
    reset(inf);
    readln(inf, n);
    for i := 0 to n do
      read(inf, d[i]);
    close(inf);
end;

begin
    ReadInputData('input.txt');
    for i := 1 to n do
      m[i, j] := 0;
      for p := 1 to n – 1 do
        for i := 1 to n – 1 do
          begin
            j := i + p;
{Min}
            t := maxint;
            for k := i to j – 1 do
              begin
                w := m[i, k] + m[k + 1, j] + d[i – 1] * d[k] * d[j];
                if (w < t) then t := w;
              end;
            m[i, j] := t;
          end;
    writeln(m[1, n]);
end.
```

Програма 7.2. Знаходження оптимальної послідовності множення матриць

Задачі та вправи до розділу 7

1. [Задача комівояжера]. Комівояжер має відвідати n міст і повернутися назад (додому), побувавши в кожному лише один раз так, щоб сумарна довжина шляху була найменшою. Відстань між містами i та j позначатимемо C_{ij}.
   Вказівка. Нехай $T(i; J_1, \ldots, J_k)$ – вартість оптимального переходу з міста i в місто 1 (місто 1 – дім), причому маршрут проходить через міста $J_1, \ldots, J_k$ за довільним порядком і не проходить через інші міста. Зрозуміло, що

$$T(i; J_1) = C_{ij_1} + C_{j_1 1};$$

$$T(i; J_1 J_2) = \min\{C_{ij_1} + C_{j_1 j_2} + C_{j_2 1}, C_{ij_2} + C_{j_2 j_1} + C_{j_1 1}\} = \min\{C_{ij_1} + T(J_1; J_2), C_{ij_2} + T(J_2; J_1)\}$$

$$\ldots$$

   Використовуючи метод математичної індукції можна вивести

$$T(i; J_1 \ldots J_k) = \min_{1 \le m \le k} \{C_{ijm} + T(J_m; J_1 \ldots J_{m-1} J_{m+1} \ldots J_k)\}.$$

   Написати програму реалізації алгоритму і визначити часову оцінку алгоритму.
2. Нехай є контекстно-вільна граматика графа G, яка задає мову $L(G)$ в алфавіті $\sum$. Перевірити чи $w \in L(G)$, де $w \in \sum^*$. Написати алгоритм розв'язання цієї задачі за час $O(|w|^3)$ з використанням ідей динамічного програмування.
3. [Задача про тріангуляції]. Задано n-кутник. Тріангуляція – розбиття його діагоналями на трикутники, що не перетинаються. Оцінка тріангуляції – сума довжин діагоналей. Зробити тріангуляцію з мінімальною оцінкою.
4. [Проблема корекції]. Задано два слова x та y та операції видалення і вставки символу. Як за мінімальну кількість операцій можна з x зробити y?
5. Припустимо, що n програм зберігаються на двох стрічках. Нехай l_i буде довжиною стрічки, яка потрібна для зберігання i-ї програми. Покладемо, що $\sum l_i \le L$, де L – довжина кожної стрічки. Програма може зберігатись на будь-якій з двох стрічок. Якщо S_1 – набір програм на стрічці 1, тоді найгірший час доступу до програми пропорційний $\max\left\{\sum_{i \in S_1} l_i, \sum_{i \notin S_1} l_i\right\}$. Оптимальний розподіл програм на стрічках мінімізує найгірший час доступу. Сформулюйте, згідно з методом динамічного програмування, підхід до визначення найгіршого випадку часу доступу за оптимальним розташуванням. Яка часова складність вашого алгоритму?
6. N робіт слід виконати на двох машинах A і B. Якщо роботу i виконують на машині A, тоді потрібно a_i одиниць часу для обробки, якщо на машині B – потрібно b_i одиниць часу. Цілком можливо, що $a_i \ge b_i$ для деяких i, коли $a_j < b_j$ для деяких $j, j \ne i$.
   А. Сформулюйте, згідно з принципами динамічного програмування, підхід до визначення мінімального часу, потрібного для виконання всіх робіт.
   Б. Вкажіть, як ви розв'язуватимете отримані рекурентні співвідношення і визначатимете оптимальний розподіл робіт на машинах.
7. N робіт слід виконати на одній машині. З роботою i зв'язаний кортеж (p_i, t_i, d_i): t_i – час обробки, потрібний для виконання i. Якщо роботу i виконано до

терміну d_i, тоді одержують прибуток p_i, якщо ні, то не одержують нічого. Нехай J – це підмножина робіт, всі з яких можна завершити за їх терміни виконання тоді і тільки тоді, коли роботи в J можна виконати за спадним порядком їх термінів виконання без порушення жодного з термінів. Припустимо, що $d_i \le d_{i+1}$, $1 \le i < n$. Нехай $f_i(x)$ – максимальний прибуток, що можна одержати з підмножини J робіт, коли $n = i$, $f_n(d_n)$ – значення оптимальної вибірки робіт J; $f_0(x) = 0$. Покажіть, що для $x \le t_i$, $f_i(x) = \max\{f_{i-1}(x), f_{i-1}(x - t_i) + p_i\}$.

Розділ 8

БЕКТРЕКІНГ

8. 1. Загальна характеристика

Серед фундаментальних принципів побудови багатьох алгоритмів розв'язання задач значне місце посідає бектрекінг, або пошук розв'язання з можливістю повернення назад на визначену кількість кроків за досягнення такого стану розв'язання, коли не можна отримати подальших просувань у вибраному раніше напрямі досягнення скінченного результату розв'язання задачі. Термін «бектрекінг» дістав початкову теоретичну розробку в працях Р. Уолкера (R. J. Walker) у 1960-х роках [149]. С. Голомб і Л. Баумен [125] продемонстрували широке практичне застосування методу. Найбільш вдале поєднання опису різнобічних характеристик бектрекінгу можна знайти у працях [9, 66, 112, 145].

Для більшості застосувань методу бектрекінгу зручною формою задання бажаного розв'язку є можливість його вираження у вигляді n-вимірного кортежу $(x_1, ..., x_n)$, де x_i вибрано з деякого скінченного набору A_i (для багатьох задач множина A_i містить деякі надлишкові елементи, що не з'являються в i-й компоненті жодного розв'язку). Починаємо будувати розв'язання з порожнього кортежу є завдовжки 0. Маючи частковий розв'язок $(x_1, ..., x_i)$, намагаються знайти таке допустиме значення x_{i+1}, що приводить або до розширення розв'язання, або до скінченного розв'язку проблеми у вигляді $(x_1, ..., x_i, x_{i+1})$. Якщо x_{i+1} забезпечує задоволення скінченного розв'язку проблеми (у разі знаходження тільки одного можливого розв'язку), тоді закінчуємо процес знаходження розв'язку проблеми, в іншому разі додаємо x_{i+1} до попереднього часткового розв'язку і продовжуємо процес, але вже для $(x_1, ..., x_i, x_{i+1})$. Якщо ж такого x_{i+1} не буде знайдено та i залишається більшим за 1, то слід повернутись на попередній крок до $(x_1, ..., x_{i-1})$ і повторити процес знаходження нового, ще не використаного допустимого значення x_i. Якщо перебрано всі варіанти знаходження x_1 і не досягнуто бажаного розв'язку, тоді процес пошуку розв'язку закінчується повідомленням «задача не має розв'язку».

Для перевірки входження нової компоненти в розв'язок припускають також існування деякої булевої функції обмеження, яка довільному частковому розв'язку $(x_1, ..., x_i)$ ставить у відповідність значення $P(x_1, ..., x_i)$ так:

$$P(x_1,\ldots,x_i) = \begin{cases} \text{хибність}, & x_i \text{ недопустиме для пошуку часткового} \\ & \text{розв'язку з } (x_1,\ldots,x_{i-1}); \\ \text{істина}, & x_i \text{ допустиме для пошуку часткового} \\ & \text{розв'язку з } (x_1,\ldots,x_{i-1}). \end{cases}$$

Останнє не означає, що $(x_1, ..., x_{i-1})$ обов'язково розшириться до повного розв'язку.

Бажані розв'язки можна умовно поділити на дві категорії: в задачі організовується пошук одного вектора розв'язку (один можливий розв'язок); інколи шукають усі можливі вектори розв'язку (усі можливі розв'язки задачі).

Наприклад, у разі розв'язку задачі впорядкування цілих чисел $B(1:n)$, розв'язок можна описати n-мірним кортежем, де x_i буде індексом у B найменшого i-го елементу. Функцію критерію P задаватиме нерівність $B(x_i) \le B(x_{i+1})$, де $1 \le i < n$. Набір A_i є скінченним і містить цілі числа від 1 до n.

Згідно з працею [66], процес бектрекінгу можна описати схемою, наведеною в алгоритмі 8.1.

```
begin
  i := 1;
  while i > 0 do
  if існує ще не використаний елемент y ∈ A_i, такий, що P(X[1], ..., X[i-1], y)
    then
      begin
        X[i] := y;
        if (X[1], ..., X[i]) є розв'язком then
          write(X[1], ..., X[i]);
        i := i + 1;
      end
    else                {Повернення на попередній частковий розв'язок;}
      i := i - 1        {всі елементи множини A_i знов стають невикористаними}
end.
```

Алгоритм 8.1. Загальна схема бектрекінгу

Цей алгоритм знаходить всі розв'язки з припущенням, що множини A_i – скінченні і що існує n, таке що у разі розв'язання задачі організовується пошук одного вектора, що задовольняє (або мінімізує, або максимізує) функцію критерію $P(x_1, ..., x_n)$.

Зрозуміло [66], що можна запропонувати рекурсивну схему організації бектрекінгу у вигляді процедури $AP(i)$, що наведено в алгоритмі 8.2. Елемент повернення в ній явно не присутній і реалізується за рахунок механізму рекурсії.

```
Procedure AP(i);
{Генерування всіх розв'язків, що є розширенням послідовності X[1], ..., X[i-1];
масив X – глобальний}
begin
  for y ∈ A_i такого, що P(X[1], ..., X[i-1], y) = істина do
    begin
      X[i] := y;
      if X[1], ..., X[i] є розв'язком then
        write(X[1], ..., X[i]);
      AP(i + 1);
    end
end.
```

Алгоритм 8.2. Варіант рекурсивної схеми реалізації бектрекінгу

В іншій загальноприйнятій термінології, наприклад [127], процес побудови бажаного розв'язку задачі, згідно з бектрекінгом, можна розглядати як ітераційний процес побудови вектора $(x_1, ..., x_n)$, що містить перевірку множин *експліцитних* (явні) та *імпліцитних* (неявні) обмежень на вибір компонент розвитку розв'язку.

Експліцитні обмеження – правила, що обмежують набір вибору кожного x_i, тобто правила побудови множин A_i. Кажуть, що експліцитні обмеження визначають можливий простір розв'язків задачі розгляду, імпліцитні – які з наборів у просторі розв'язків задачі фактично можуть просунути далі процес побудови бажаного розв'язку (задовольняють функцію критерію P), задаючи потрібні співвідношення між x_i, $1 \le i \le n$.

Розглянемо застосування запропонованої методології та термінології на прикладі задачі про n ферзів. Вона належить класу комбінаторних задач і полягає в тому, щоб розмістити n ферзів на шахівниці розміром $(n \times n)$ так, щоб жодні два не атакували один одного, тобто не знаходилися в одному рядку, стовпці або в одній діагоналі. Пронумеруємо рядки і стовпці шахівниці числами від 1 до n. Ферзі також можна пронумерувати числами від 1 до n. Без втрати загальності можемо припустити, що ферзь i слід помістити в рядок i. Тоді всі розв'язки задачі можна виразити у вигляді n-мірних кортежів $(x_1, ..., x_n)$, де x_i – стовпець розміщення i-го ферзя. *Експліцитні обмеження* набуватимуть вигляду: $A_i = \{1, 2, ..., n\}$, $1 \le i \le n$. Отже, простір розв'язку складається з n^n кортежів. *Імпліцитні обмеження* задачі полягають у тому, що жодні два x_i не можуть бути однаковими. Перше з цих двох обмежень свідчить, що всі розв'язки – перестановки з n-мірних кортежів $(1, 2, 3, ..., n)$. Це дає змогу зменшити розмір простору розв'язків від n^n кортежів до $n!$ кортежів. Пізніше ми встановимо, як сформулювати інші обмеження.

Розглянемо ще один приклад [127] застосування бектрекінгу для розв'язання задачі про суму підмножин. Нехай задано $n + 1$ додатних цілих чисел: w_i, $1 \le i \le n$ та M; потрібно знайти всі підмножини w_i-х, сума яких дорівнює M. Наприклад, якщо $n = 4$, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ і $M = 31$, тоді бажаними підмножинами будуть (11, 13, 7) і (24, 7). Замість запису у векторі розв'язку чисел w_i, які в сумі дають M, ми могли задати вектор розв'язків шляхом внесення до нього індексів цих w_i. Тоді наші два розв'язки опишуть вектори (1, 2, 4) і (3, 4). Експліцитні обмеження набувають вигляду: $x_i \in \{j \mid j$ – *ціле число* і $1 \le j \le n\}$. Імпліцитні обмеження вимагають, щоб жодні два доданки не були однаковими ($x_i < x_{i+1}$, $1 < i \le n$) та їхня сума дорівнювала M, оскільки ми хочемо уникнути гене-

рування кратних випадків тієї ж самої підмножини (для прикладу, (1, 2, 4) і (1, 4, 2) зображують ті ж самі підмножини).

Для більшості задач бектрекінгові варіанти побудови розв'язку допускають можливість існування кількох рівноправних підходів до задання розв'язку у вигляді деякого вектора. Наприклад, можна запропонувати і такий варіант задання розв'язку задачі знаходження суми підмножин: кожну підмножину розв'язків можна відображати таким вектором $(x_1, x_2, ..., x_n)$, що $x_i \in \{0, 1\}$, $1 \le i \le n$, де $x_i = 0$, якщо w_i не вибране, і $x_i = 1$ – в іншому разі. Тоді розв'язки згаданого вище випадку набуватимуть вигляду – (1, 1, 0, 1) і (0, 0, 1, 1), і для побудови всіх розв'язків використовуватимуться вектори фіксованого розміру.

Перевірте, що для обох вищезгаданих формулювань простір розв'язків складається з 2^n різних кортежів.

Зрозуміло, що для бектрекінгових алгоритмів пошук розв'язків полегшиться, якщо їх задати у вигляді дерева [125]. Пошук розв'язку тоді зручно описувати за допомогою техніки пошуку в дереві. Кожна вершина дерева відповідає деякій послідовності $(x_1, \ldots, x_n)$, за винятком кореня. Вершина кореня відповідає пустій послідовності ε. Вершини, позначені $(x_1, \ldots, x_k, y)$, є синами вершини $(x_1, \ldots, x_k)$.

Розглянемо повне дерево T, яке складається з усіх можливих послідовностей типу $(x_1, \ldots, x_k)$ де $0 \le k \le n$, $x_i \in A_i$ для $1 \le i \le k$. Тимчасово припустимо, що кожен елемент $y \in A_k$ є допустимим для $(x_1, \ldots, x_{k-1})$, якщо $k \le n$, і жоден елемент не буде допустимим для $(x_1, \ldots, x_{k-1})$, $k > n$, тобто

$$P(x_1,\ldots,x_k) = \begin{cases} true, \text{ якщо } k \le n; \\ false, \text{ якщо } k > n, \end{cases}$$

$x_i \in S_i$ для $1 \le i \le k$. Тоді зрозуміло, що виклик процедури $P(1)$ викличе пошук в глибину в дереві T. У разі складнішої функції P процес пошуку пропускає розгляд вершини піддерев з коренем у кожній «недопустимій» вершині, що трапилася. За рахунок цього й позбавляються повного перебору елементів. Для багатьох задач часова складність бектрекінгового алгоритму в «найгірших» випадках носитиме експоненційний характер.

Техніку оцінювання ефективності бектрекінгових алгоритмів вперше було запропоновано Кнутом [131].

Ефективність реалізації бектрекінгу залежить від чотирьох чинників: часу визначення, яка з множин A_i використовується; часу генерування наступного x_i; часу визначення, чи задовольняє x_i експліцитним обмеженням; часу визначення, чи задовольняє x_i імпліцитним обмеженням.

Функції обмеження перебору варіантів побудови компонент вектора розв'язку вважають вживаними, якщо вони в багатьох випадках зможуть зменшити число чергових вершин генерування. Якщо для задачі розмір простору станів дерева є занадто великим, щоб дозволити генерування

всіх вершин, тоді функції обмеження слід будувати так, щоб принаймні один розв'язок був знайдений в прийнятний час.

Використання перестановок спеціального типу – загальний принцип побудови ефективного пошуку. Для багатьох задач множини A_i можна використовувати в будь-якому порядку. Теоретично можна показати, що за пересічного вибору використання найменшої множини є ефективнішим.

Зрозуміло, що бектрекінг приводить до експоненціальних алгоритмів. Це досить просто довести. Всі розв'язки мають довжину не більшу n, тому досліджується десь близько $\prod_{i=i-1}^{n} |A_i|$ вузлів дерева, де $|A_i|$ – кількість елементів в A_i. В найліпшому варіанті $|A_i|$ – це константа (позначатимемо C), отже маємо дерева з кількістю вузлів, що наближається до C^n. Тому бажано здійснювати пошук, який би враховував оцінки числа вузлів в дереві.

Враховуючи природу бектрекінга, отримати аналітичний вираз побудови такої оцінки в загальному випадку неможливо. Тому використовують ймовірнісний підхід до побудови оцінки розмірів дерева – метод Монте-Карло.

Ідея полягає в проведенні кількох досліджень (спроб). Кожне таке дослідження є проведенням бектрекінгу з випадково вибраними значеннями x_i.

Нехай ми маємо частковий розв'язок $(x_1, x_2, ..., x_{k-1})$, і число виборів x_k дорівнює $a_k = |S_k|$, де S_k – підмножина кандидатів із A_i в x_i. Якщо $a_k \ne 0$, то x_k вибирається випадково із S_k і для кожного елементу ймовірність бути вибраним дорівнює $1/a_k$. Якщо $a_k = 0$, то дослідження закінчується. Тому, якщо $a_1 = |S_1|$, то $x_1 \in S_1$ вибирається випадково з ймовірністю $1/a_1$, якщо $a_2 = |S_2|$, то за умови, що x_1 було вибрано із S_1, $x_2 \in S_2$ вибирається випадково з ймовірністю $1/a_2$ і т. д. Математичне очікування

$$a_1 + a_1a_2 + a_1a_2a_3 + a_1a_2a_3a_4 + \ldots$$

буде рівним числу вузлів дерева без кореня, тобто визначатиме число випадків, що досліджуватимуться бектрекінговим алгоритмом.

Такі обчислення за методом Монте-Карло можна використовувати для оцінки ефективності бектрекінгового алгоритму шляхом порівняння його з еталоном, отриманим для тієї ж задачі, але з меншою розмірністю.

Продемонструємо використання описаної узагальненої техніки бектрекінгових алгоритмів на конкретних задачах.

8.2. Задача про 8 ферзів

Класична шахівниця має розмір 8 × 8. Тому розглянемо задачу про розміщення на ній 8 ферзів. Розв'язок можна зобразити кортежем $(x_1, ..., x_n)$, де x_i є стовпцем i-го рядка шахівниці, в якому буде розміщений

i-й ферзь. Всі x_i будуть відмінними, бо жодні два ферзі не можна помістити в один стовпчик. Залишається лише подумати, як перевіряти, чи два ферзі знаходяться на одній діагоналі.

Для вирішення задачі пронумеруємо вертикальні та горизонтальні рядки шахової дошки числами від 1 до n. Для однозначності нумерації діагоналі розділимо на два типи так, щоб діагональ визначали тип і номер. Останній може бути вирахований згідно з такими формулами:

$$Diagonal1 = N + Column - Row \text{ (type 1)},$$

$$Diagonal2 = Row + Column - 1 \text{ (type 2)},$$

де N – кількість ферзів (розмір шахівниці), *Column* – номер стовпця, *Row* – номер рядка.

Якщо ви дивитесь на шахівницю, на ряд 1 по горизонталі та колонку 1 по вертикалі з лівого боку, тоді type 1 розділяє діагоналлю клітинку як символ похилої риски вліво (\), а type 2 – вправо (/). Рис. 8.1 ілюструє нумерацію діагоналей type 2.

Позицію ферзя можна описати за допомогою зазначення номера рядка та колонки. Отже, уявімо, що квадрати шахівниці є елементами двовимірного масиву $C(1: n, l: n)$. Припустимо, що два ферзі розміщені відповідно (i, j) і (k, l). Тоді вони лежать на одній діагоналі, тільки якщо $i - j = k - l$ або $i + j = k + l$. Перше рівняння є рівносильним $j - l = i - k$, а друге – $j - l = k - i$. Отже, два ферзі лежать на одній діагоналі тоді й тільки тоді, коли $|j - l| = |i - k|$.

Функція PLACE(A) повертає булеве значення «істина», якщо k-й ферзь можна розмістити в поточному значенні $x(k)$. Вона перевіряє також чи $x(k)$ відмінне від усіх попередніх значень $x(1)$, ..., $x(k-1)$ і чи немає іншого ферзя на тій самій діагоналі. Час обчислення становить $O(k-1)$.

	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8
2	2	3	4	5	6	7	8	9
3	3	4	5	6	7	8	9	10
4	4	5	6	7	8	9	10	11
5	5	6	7	8	9	10	11	12
6	6	7	8	9	10	11	12	13
7	7	8	9	10	11	12	13	14
8	8	9	10	11	12	13	14	15

Рис. 8.1. Нумерація діагоналей шахівниці другого типу


```
function Place(k: int): boolean;
  var i: int;
  label exit;
begin
    i := 1;
    Place := true;
    for i := 1 to k – 1 do
      if (x[i] = x[k]) or (abs(x[i] – x[k]) = abs(i – k)) then
        begin
          Place := false;
          goto exit;
        end;
end.
```

Використовуючи процедуру PLACE і алгоритм опису загальної стратегії бектрекінгу приходимо до розв'язку задачі про n ферзів:

```
procedure NQUEENS(n: int);
  var k: int;
begin
    x[1] := 0;                          {k – поточний рядок, x[k] – поточний стовпець}
    k := 1;                             {Для всіх рядків зробити}
    while k > 0 do
      begin
        x[k] := x[k] + 1;               {Перейти на наступний стовпець}
        while x[k] <= n and not(Place(k)) do  {Чи може ферзь бути розміщеним?}
          x[k] := x[k] + 1 ;
        if x[k] <= n then               {Позицію знайдено}
          if k = n then                 {Чи всі ферзі розставлені?}
            PRINT;
          else
            begin
              k := k + 1;
              x[k] := 0;                {Перехід на наступний рядок}
            end;
        else k := k – 1;                {Повернення назад}
      end;
end;
```

Повну паскалівську реалізацію алгоритму наведено в програмі 8.1.

```
program Ex_8_1;
 const max_n = 100;                     {Максимальний розмір дошки}
 type int = 0. . 1000;
      arr = array [int] of int ;
```

```
var  x: arr;
     n: int;
     i: integer;

procedure PRINT; forward;

function Place(k: int): boolean;
  var  i: int;
  label exit;
begin
    i := 1;
    Place := true;
    for i := 1 to k – 1 do
      if (x[i] = x[k]) or (abs(x[i] – x[k]) = abs(i – k)) then
         begin
            Place := false;
            goto exit;
         end;
exit:
end;

procedure NQUEENS(n: int);
  var  k: int;
begin
    x[1] := 0;                    {k – поточний рядок, x[k] – поточний стовпець}
    k := 1;                       {Для всіх рядків зробити}
    while k > 0 do
      begin
         x[k] := x[k] + 1;                    {Перейти на наступний стовпець}
         while (x[k] <= n) and not(Place(k)) do {Чи може ферзь бути розміщеним}
            x[k] := x[k] + 1 ;
         if x[k] <= n then                    {Позицію знайдено}
            begin
               if k = n  then PRINT           {Чи розміщення повне?}
               else
                  begin
                     k := k + 1;
                     x[k] := 0;               {Перехід на наступний рядок}
                  end;
            end
         else k := k – 1;                     {Бектрекінг}
      end;
end;

procedure PRINT;
  var  I: int ;
```



```
begin
    for I := 1 to n do
       write('|_', x[i]);
    write ('|');
    writeln;
end;

begin
    read(n);
    for i := 1 to n do
       x[i] := 0;
    NQUEENS(n);
end.
```

Програма 8.1. Всі розв'язки задачі про *n* ферзів

8.3. Сума підмножин

У підрозділі 4.2 розглядали розв'язання задачі знаходження суми підмножин. Розглянемо бектрекінговий варіант розв'язання [127]. Нагадаємо постановку задачі. Припустимо, дано n різних невід'ємних чисел і додатне число M. Потрібно знайти всі послідовності цих чисел, сума яких дорівнює M. Під час обговорення загальної стратегії бектрекінгових алгоритмів ми зазначали, що переформулювати цю задачу можна з використанням фіксованих або змінних кортежів. Зупинимось на першому варіанті. Нагадаємо, що в цьому разі елемент $x(i)$ з вектора розв'язків дорівнює 0 або 1 залежно від того, введений $w(i)$ доданок у суму чи ні.

Як видно, для цієї задачі підійде така функція обмеження:

$$P_k(x(1), ..., x(k)) = \textbf{\textit{true}}, \text{ якщо } \sum_{i=1}^{k} w(i)x(i) + \sum_{i=k+1}^{n} w(i) \geq M.$$

Зрозуміло, що перебір варіантів можна поліпшити, якщо розмістити доданки за зростанням. У цьому разі $(x(1), \ldots, x(k))$ не може вести до вершини відповіді, якщо

$$\sum_{i=1}^{k} w(i)x(i) + w(k+1) > M.$$

Отже, функцію обмеження можна підсилити так:

$$P_k(x(1), ..., x(k)) = \textbf{\textit{true}}, \text{ якщо}$$

$$\sum_{i=1}^{k} w(i)x(i) + \sum_{i=k+1}^{n} w(i) \geq M \text{ і } \sum_{i=1}^{k} w(i)x(i) + w(k+1) > M$$

за умови, що доданки розміщені за зростанням.

Оскільки визначили вже все потрібне для використання бектрекінгового алгоритму, виберемо лише схему реалізації. З останнього зауваження щодо визначення функції обмеження природно випливає надання переваги рекурсивному варіанту. Можна також уникнути обчислення

$$\sum_{i=1}^{k} w(i)x(i), \quad \sum_{i=k+1}^{n} w(i)$$

щоразу за рахунок збереження цих значень у вигляді змінних s і r відповідно. Алгоритм припускає $w(1) \le M$ і $\sum w(i) \ge M$. Цікаво зазначити, що можна явно не використовувати умову $k > n$, щоб завершити рекурсію. Ця умова не потрібна, тому що на вході в алгоритм $s \ne M$ і $s + r \ge M$. Отже, $r \ne 0$ і, таким чином, k не може бути більшим за n.

Повну паскалівську реалізацію алгоритму наведено в програмі 8.2.

```
Program Ex_8_2;
  const max_len = 100;
  var  M, n, i: integer;
       W: array [1.. max_len] of integer;
       sum: real;
       X: array [1.. max_len] of boolean;

procedure ReadInputData(fname: string);
  var  inf: text;
       i: integer;
begin
     assign(inf, fname);
     reset(inf);
     readln(inf, n);                    {Перше значення, що міститься у файлі,–}
                                        {кількість можливих доданків}
     for i := 1 to n do
        read(inf, W[i]);                {Перелік доданків}
     readln(inf);
     readln(inf, M);                    {Значення суми}
     close (inf);
     sum := 0;
     for i := 1 to n do
        sum := sum + w[i];
end;

{Головна процедура}
procedure SUMOFSUB(s, r: real; k: integer);
  var  j: integer;
begin
     X[k] := true;
     if (s + W[k] = M) then
        begin                           {Виведення результату}
           for j := 1 to k do
              if (X [j]) then write (W [j]: 2: 2, ' ');
           writeln;
        end
     else
        if(s + W[k] + W[k + 1] <= M) then SUMOFSUB(s + W[k], r – W[k], k + 1);
     if (s + r – W[k] >= M) and (s + W[k + 1] <= M) then
        begin
           X[k] := false;
           SUMOFSUB(s, r – W[k], k + 1);
        end;
end;

begin
     ReadInputData('input. txt');
     i := 1;
     SUMOFSUB(0, sum, i);
end.
```

Програма 8.2. Розв'язок задачі про суму підмножин

Побудуйте дерево простору станів, що згенерувалося б процедурою SUMOFSUB під час роботи над випадком $n = 5$, $M = 40$ і $W(1:5) = (8, 11, 14, 15, 17)$.

8.4. Гамільтонові цикли

Розглянемо задачу знаходження гамільтонового циклу в графі [66, 127]. Спочатку згадаємо задачу знаходження ейлерового циклу, розглянуту в підрозділі 3.3.4. Різниця між цими задачами полягає в тому, що у разі гамільтонового циклу нас цікавитиме шлях, що проходить тільки один раз через кожну вершину (а не ребро, як у задачі про ейлерів цикл) даного графа. Такий шлях називають *гамільтоновим*. Аналогічно визначається і гамільтонів цикл. Така незначна модифікація задачі істотно змінює характер проблеми. Зрозуміло, що не для кожного графа існуватиме гамільтонів цикл. Для задачі про гамільтонів цикл невідомі необхідна й достатня умова існування гамільтонових циклів, а також поліномний алгоритм, який би перевіряв існування гамільтонового шляху в довільному графі.

Нехай маємо зв'язний граф $G = \langle V, E \rangle$. Тоді можна згенерувати $n!$ різних послідовностей вершин для перевірки на відповідність умові гамільтонового шляху. Такі дії потребують не менше $n!n$ різних кроків. Функція

такого типу зростає швидше за експоненційну функцію. Тому знову використаємо ідею бектрекінгу для скорочення кількості кроків в алгоритмі повного перебору варіантів.

Уточнимо введене визначення. Гамільтонів цикл – шлях, що проходить через n вершин графа G, за умови, що кожну вершину відвідують один раз і остання вершина відвідування є вершиною початку руху. Інакше кажучи, якщо гамільтонів цикл починається в деякій вершині $v_1 \in G$ і вершини графа G відвідані за порядком $v_1, v_2, ..., v_{n+1}$, тоді ребра (v_i, v_{i+1}) належать E, $1 \le i \le n$ і v_i відмінні, за винятком v_1 та v_{n+1}, які рівні між собою.

Розглянемо алгоритм бектрекінгу, що знаходить всі гамільтонові цикли в орієнтованому або неорієнтованому графах. Кортеж розв'язку $(x_1, ..., x_n)$ побудуємо так, що x_i визначатиме, чи була відвідана i-та вершина запропонованого циклу. Для вибору множини можливих вершин – кандидатів до x_k, за умови, що $x_1, ..., x_{k-1}$ вже вибрані, дотримуватимемось правила:

if k = 1 **then**
    X(1) *може бути будь-якою з* n *вершин*
  else
    if 1 < k < n **then**
      X(k) *може бути будь-якою вершиною* v, *відмінною від* X(1), X(2), ..., X(k – 1), *і* v *зв'язана ребром з* X(k – 1)
    else
      X(n) – *вершина* v, *що має бути зв'язана з* X(n – 1) *та* X(1).

Для уникнення виведення одного циклу n разів накладають умову $X(1) = 1$.

Повну реалізацію в Паскалі відповідного алгоритму наведено в програмі 8.3.

```
Program Ex_8_3;
{Вхідні дані задають у файлі наступного формату: в першому рядку визначають
n – кількість вершин графа; наступні n рядків описують матрицю суміжності
задання графа}
  const max_n = 20;
  var  n: integer;
       g: array [1. . max_n, 1. . max_n] of integer;
       cycle: array [1. . max_n] of integer;

function in_array(nod: integer; pos: integer): boolean;
 var i: integer;
begin
    for i := 1 to pos do
      if (cycle[i] = nod) then
        begin
          in_array := true;
          exit;
        end;
    in_array := false;
end;

procedure ReadInputData(fname: string);
  var  inf: text;
       i, j: integer;
begin
    assign(inf, fname);
    reset(inf);
    readln(inf, n);
    for i := 1 to n do
      begin
        for j := 1 to n do
          read(inf, g[i, j]);
          readln(inf);
      end;
    close(inf);
end;

procedure G_Cycle(depth: integer);
  var  i: integer;
begin                                                    {Знайдено?}
    if (depth = n) then
      begin                                              {Друк}
        write('Cycle: ');
        for i := 1 to n do
          write(cycle[i], ' ');
        writeln;
        exit;
      end;
    for i := 1 to n do                                   {Спроба}
      if not(in_array(i, depth)) and (g[cycle[depth], i] > 0) then
        begin
          cycle[depth + 1] := i;
          G_Cycle(depth + 1);
        end;
end;

begin
    ReadInputData('input. txt');
    cycle[1] := 1;
    G_Cycle(1);
end.
```

Програма 8.3. Розв'язок задачі про гамільтонів цикл

За допомогою процедури Hamiltonian *можна розв'язати й інші задачі, наприклад задачу про комівояжера, задачу знаходження шляху мінімальної вартості. Запропонуйте варіанти такого застосування.*

Задачі та вправи до розділу 8

1. Зробити наступні модифікації в алгоритмах розділу:
   а) де шукали всі розв'язки, знайти один, і навпаки;
   б) де застосовували рекурсивні алгоритми, використати ітераційний підхід, і навпаки.
2. Запропонуйте оптимізацію процедури *Nqueens*.
3. Дано шахівницю ($n \times n$), кінь поміщений на довільний квадрат з координатами (x, y). Визначити $n^2 - 1$ переміщень коня за умови, що кожен квадрат дошки відвідують один раз, якщо така послідовність ходів існує. Напишіть алгоритм до задачі.
4. Нехай дано n чоловіків і n жінок, та два $n \times n$ масиви P і Q, таких що $P(i, j)$ вказує на надання переваги i-го чоловіка j-й жінці, а $Q(i, j)$ – надання переваги i-ї жінки j-му чоловіку. Придумайте алгоритм, що знаходить такі парос полуки чоловіків та жінок, які оптимізують переваги.
5. Нехай $A(1: n, 1: n)$ буде $n \times n$ матрицею. Детермінантом матриці A є число

$$\det(A) = \sum_s \operatorname{sgn}(s) a_{1,s(1)} a_{2,s(2)} \cdots a_{n,s(n)},$$

   де беруть суму всіх перестановок $s(1), \ldots, s(n)$ набору $\{1, 2, \ldots, n\}$, і $\operatorname{sgn}(s)$ є або +1 або –1, відповідно до того, парною чи непарною є s перестановка. Перманент A визначають як

$$\operatorname{per}(A) = \sum_s a_{1,s(1)} a_{2,s(2)} \cdots a_{n,s(n)}.$$

   Детермінант можна обчислити як побічний продукт гаусового вилучення, що вимагає $O(n^3)$ операцій, але невідомо жодного алгоритму, який обчислював би детермінант за поліноміальний час. Напишіть бектрекінговий алгоритм, що обчислює перманент матриці, генеруючи елементи s. Проаналізуйте часову оцінку вашого алгоритму.
6. Нехай MAZE(1: n, 1: n) – двовимірний масив задання лабіринту, що набуває значення 0 або 1. Він визначає такий шлях, де нуль означає відкриту позицію. Потрібно розробити алгоритм, що починається з MAZE(1, 1) і намагається знайти шлях до MAZE(n, n). Знову ж таки використання бектрекінгу є необхідним.
7. Проблему призначення формулюють так: є n людей, яких призначають на n робіт. Вартість призначення i-людини до j-роботи – COST(i, j). Потрібно розробити бектрекінговий алгоритм, що призначає кожній людині роботу за мінімальної загальної вартості призначення.
8. Цю задачу називають задачею поштової марки. Є країна, що видає n різноманітних марок, але дає змогу використовувати не більш ніж m марок на один конверт. Для заданих значень m та n напишіть алгоритм, що обчислює найбільший послідовний діапазон цінності поштових марок від одного і вище, та всі можливі набори, що підходять цьому діапазону. Наприклад, для $n = 4$ та $m = 5$ марки зі значеннями (1, 4, 12, 21) допускають значення від 1 до 71. Чи є будь-які інші найменування чотирьох марок, що мають той самий діапазон?
9. Припустимо, є n завдань для виконання, але тільки k процесорів, що можуть працювати паралельно. Час, необхідний для виконання роботи – i – t_i. Написати алгоритм, що визначає, які завдання на яких процесорах мають бути виконані й за яким порядком, щоб остаточний час виконання завдань був мінімальним.
10. Побудуйте алгоритм з поверненням для розв'язку задачі знаходження розміщення 8 взаємно не атакуючих тур на шахівниці.
11. Побудуйте алгоритм з поверненням для розв'язку задачі знаходження мінімального числа кольорів для розфарбування вершин графа, так щоб жодне ребро не з'єднувало двох вершин одного кольору.
12. Побудуйте алгоритм з поверненням для розв'язку задачі знаходження кліки найбільшої потужності в неорієнтованому графі. (Клікою називають довільну підмножину вершин, в якій кожна пара різних вершин з'єднана ребром графа.)

Розділ 9

МЕТОД ГІЛОК ТА ГРАНИЦЬ

9.1. Загальна характеристика методу

Цей метод дає змогу серед елементів деякої множини можливих розв'язків задачі знайти найліпший розв'язок за умови, що цю множину можна розділити на такі підмножини, які не перетинаються, і для будь-якої з них можна визначити деяку оцінку найліпшого (найоптимальнішого) можливого розв'язку. Тобто має бути можливість визначити «оптимальність» розв'язку, ліпше якої не можна досягти, якщо шукати розв'язок тільки у межах вибраної підмножини (насправді такої оптимальності на цій підмножині може й не бути).

Окрім цього, має існувати можливість розділяти кожну з вибраних неперетинних підмножин знаходження на нові неперетинні підмножини у подібний спосіб і так продовжувати далі розділення на підмножини з аналогічним оцінюванням «оптимальності», поки буде змога поліпшення попередньої оцінки [48, 62].

Для ілюстрації особливостей застосування методу гілок та границь розглянемо розв'язок задачі лінійного програмування в інтерпретації пошуку розв'язку відомої нам задачі про рюкзак. У цьому разі початкову множину можливих розв'язків складає деякий набір предметів, що можна розмістити в рюкзаку, а оцінкою найліпшого розв'язку є загальна цінність предметів, покладених у рюкзак.

Прикладом застосування методу для розв'язку задач на графах може слугувати також уже розглянута нами задача про комівояжера. У цьому разі множину можливих розв'язків складатимуть усі допустимі цикли, а оцінками виступатимуть їх довжини. Найліпшим розв'язком вважатимемо, звичайно ж, той, що має найкоротшу довжину.

Уточнимо запропоновану методологію пошуку розв'язку.

Нехай маємо задачу, оптимальність можливих розв'язків x якої визначає деяка функція $f(x)$, а усі можливі розв'язки складають множину G^0. Тоді пошук розв'язку проходитиме таким шляхом.

Першим кроком обов'язково є розбиття початкової множини G^0 допустимих розв'язків x на підмножини G_i^1 так, щоб потім для кожної підмножини визначити оцінку найліпшого можливого результату $\varepsilon\left(G_k^1\right)$ або знайти точно найліпший для цієї підмножини розв'язок x, $x \in G_k^1$, $0 < k \le i$. Якщо для якоїсь k-ї підмножини вдалося знайти точний розв'язок, визначаємо міру його оптимальності функцією $f(x)$. Якщо таких розв'язків виявиться кілька, вибираємо (тимчасово) той з них, що має найліпшу $f(x^{\text{опт}})$. Усі множини, що мають оцінку $\varepsilon\left(G_i^1\right)$, гіршу за тимчасову $f(x^{\text{опт}})$, відкидають з подальшого розгляду.

Наступні кроки повторюють доти, доки є невідкинуті підмножини з незнайденими на них точними найліпшими розв'язками. Кожен з кроків полягає у наступному.

Вибирають підмножину G_k^m з найліпшою оцінкою $\varepsilon\left(G_k^m\right)$ і розбивають на підмножини G_1^{m+1}. Над цими підмножинами виконують усі ті дії, що виконували над множинами G_i^1 на першому кроці, тобто знаходять нові оцінки $\varepsilon\left(G_1^{m+1}\right)$ або точні розв'язки $f\left(x_1^{m+1}\right)$. Якщо якийсь із цих розв'язків x_k^{m+1} виявиться ліпшим за $x^{\text{опт}}$, то цей розв'язок стає «тимчасово» оптимальним, і слід відкинути усі підмножини, чия оцінка гірша за нове значення $f(x^{\text{опт}})$.

Розв'язком буде останнє знайдене значення $x^{\text{опт}}$, після досягнення умови припинення пошуку (тобто знайдуть точні найліпші розв'язки на всіх невідкинутих підмножинах).

Нарешті, за даними [48, 83], метод гілок і границь є універсальним методом розв'язання комбінаторних задач дискретного програмування. Складність його застосування визначає спосіб поділу множин на підмножини і обчислення оціночних функцій, які залежать від специфіки конкретної задачі.

9.2. Розв'язок задач цілочислового лінійного програмування

Нехай маємо цілочислову задачу лінійного програмування: мінімізувати $z = f(x) = \sum c_j x_j$ за обмежень

$$\sum_{j=1}^{n} a_{ij} x_j \ge b_i, \quad i = 1 \ldots m;$$

$$0 \le x_j \le d_i, \quad j = 1 \ldots n;$$

$$x_j \in Z.$$

Почнемо шукати розв'язок з розгляду неперервної задачі лінійного програмування, тобто тимчасово не братимемо до уваги умову цілочисельності x_j. Якщо виявиться, що знайдений розв'язок x_0 задовольнить і умову цілочисельності, початкову задачу розв'язано, якщо ж умова цілочисельності не виконується – тоді оцінкою ε_0 є значення $f(x_0)$.

Нехай деяка змінна x_{i0} ще не набула у розв'язку x_0 цілочислового значення. Тоді треба x_{i0} або зменшити хоча б до першого меншого цілого, або збільшити хоча б до найближчого більшого цілого.

Якщо межі зміни x_{i0} не задано, то їх можна обчислити, розв'язавши дві допоміжні задачі лінійного програмування. Ці задачі полягають в оптимізації x_{i0} за обмежень, що їх накладено на змінні x_j у початковій задачі.

Далі розв'язують початкову задачу, знов відкинувши умови цілочисельності, але наклавши умову $x_{i0} \le [x_{i0}]$ або $x_{i0} \ge [x_{i0}] + 1$, де $[x_{i0}]$ – ціла частина x_{i0}. Знайдені розв'язки знову аналізують на задоволення умови цілочисельності, сформульованої у початковій задачі. Кожну з отриманих підзадач розв'язують аналогічно до досягнення цілочислових розв'язків в усіх невідкинутих підмножинах.

Іноді, для поліпшення ілюстративності пошуку розв'язку за методом гілок і границь, розглядають графову інтерпретацію методу.

Уявімо собі дерево, в якому вершина 0 відповідає нецілочисловому (у загальному випадку) розв'язку x_0, а два нащадки цієї вершини відповідають розв'язкам таких задач: мінімізувати z з урахуванням наведених у початковій задачі обмежень, за винятком обмеження на цілочисельність, але за умови, що $x_{i0} \le k_{i0}$ (або $x_{i0} \ge k_{i0} + 1$), де k_{i0} – певне ціле число. Кожній з таких вершин приписують оцінку $\varepsilon = \varepsilon(i_0, k)$, яка задовольняє умові мінімальності z за встановлених для цієї вершини обмежень. Якщо оптимальні розв'язки для кожної з підзадач відповідають умовам цілочисельності, то за відповідь обирають найменший з них. Якщо ж ні, то слід продовжити розкриття вершин (тобто розбиття відповідних до них множин наборів x), починаючи з тієї, що має найменшу оцінку. Якщо оцінка якоїсь з вершин виявиться більшою за найменший зі знайдених раніше розв'язків, то цю вершину далі не розглядаємо.

Які зміни потрібно ввести в цей метод для розв'язання задачі максимізації z?

Розглянемо запропоновану методологію використання методу гілок і границь на прикладі [48].

Нехай необхідно мінімізувати функцію $f(x_1, x_2) = -(x_1 + x_2)$ за таких обмежень:

$$2x_1 + 11x_2 \le 38;$$
$$x_1 + x_2 \le 7;$$
$$4x_1 - 5x_2 \le 5;$$
$$x_1, x_2 \ge 0;$$
$$x_1, x_2 \in Z.$$

Оптимальне рішення цієї задачі лінійного програмування без умови цілочисельності $-\left(\frac{40}{9}; \frac{23}{9}\right)$, оцінка $\xi = -7$.

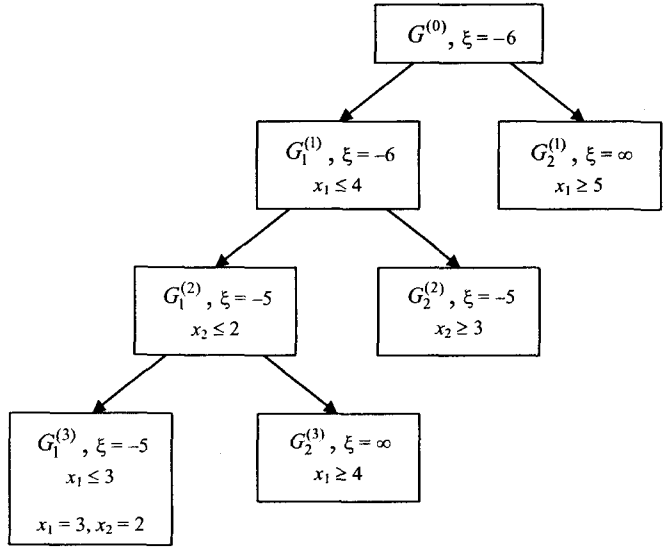


Рис. 9.1. Дерево розв'язків і додаткові обмеження

Отримане рішення, між іншим, не задовольняє умові цілочисельності, тож необхідно розбити вихідну множину G_0 на

$$G_1^{(1)} = \left\{x : x \in G^{(0)} \cap x_1 \le 4\right\},$$
$$G_2^{(1)} = \left\{x : x \in G^{(0)} \cap x_2 \ge 5\right\}.$$

Розв'язавши ці дві задачі лінійного програмування, отримаємо наступні оцінки:

$$\xi\left(G_1^{(1)}\right) = -6 \ (x_1 = 4, x_2 = 30/11);$$
$$\xi\left(G_2^{(1)}\right) = \infty \text{ (нерозв'язна задача).}$$

Очевидно, що надалі додаткові обмеження слід накладати на множину $G_1^{(1)}$ ($x_2 \le 2$ та $x_2 \ge 3$). Продовжуючи цей процес накладання умов, прийдемо до розв'язку задачі (3, 2) з оцінкою $\xi = -5$. Дерево розв'язків та додаткові обмеження наведено на рис. 9.1.

Повний опис методу гілок і границь розв'язання задачі цілочислового лінійного програмування наведено в алгоритмі 9.1.

0. Нульова ітерація

0.1. *Визначимо множини G_0 за умов*

$$\sum_{j=1}^{n} a_{ij} x_j \ge b_i, \ i = 1 \ldots m;$$
$$0 \le x_j \le d_j, j = 1 \ldots n;$$

0.2. *Знаходимо оптимальний розв'язок за умов, наведених вище.*

0.3. *Обчислюємо оцінку* $\varepsilon(G_0) = f(x_0)$. *Якщо* x_0 *задовольняє умові цілочисельності, то він є шуканим. Якщо ні – переходимо до першої ітерації.*

1. Перша ітерація

1.1. *Вибираємо з* x_0 *нецілочислову компоненту* $x_i = x_{i0}$, $1 \le i \le n$. *Множину* G_0 *розділяємо на дві підмножини, що не перетинаються, у наступний спосіб:*

$$G_1^{(1)} = \{x : x \in G_0, x_i \le [x_{i0}]\};$$

$$G_2^{(1)} = \{x : x \in G_0, x_i \ge [x_{i0}] + 1\}.$$

1.2. *Розв'язуємо початкову задачу лінійного програмування на множинах* $G_1^{(1)}$ *і* $G_2^{(1)}$ *і знаходимо оптимальні розв'язки* $x_1^{(1)}$ *і* $x_2^{(1)}$. *Для подальшої перевірки на ознаку оптимальності обчислюємо оцінки* $\varepsilon(G_1^{(1)}) = f(x_1^{(1)})$, $\varepsilon(G_2^{(1)}) = f(x_2^{(1)})$.

Якщо $x_i^{(1)}$ *– цілочисловий і* $\varepsilon(G_i^{(1)}) = \min(\varepsilon(G_1^{(1)}), \varepsilon(G_2^{(1)}))$, *то* $x_i^{(1)}$ *– шуканий розв'язок. Інакше переходимо до наступної ітерації.*

(Нехай вже пройдено k ітерацій і побудовано підмножини $G_i^{(k)}$, $i = 1 \ldots r_k$ *з оцінками* $\varepsilon(G_i^{(k)})$ *і ще не знайдено оптимальний розв'язок.)*

$k+1$. ($k+1$)-ша ітерація

$k+1.1$. *Тоді для подальшого розділення вибираємо таку підмножину* $G_v^{(k)}$, *що*

$$\varepsilon\left(G_v^k\right) = \min_i\left(\varepsilon\left(G_v^k\right)\right).$$

Множину $G_v^{(k)}$ *розділяють на підмножини* $G_{v_1}^{(k)}$ *і* $G_{v_2}^{(k)}$ *так, що* $G_v^{(k)} = G_{v_1}^{(k)} \cup G_{v_2}^{(k)}$ *і* $G_{v_1}^{(k)} \cap G_{v_2}^{(k)} = \varnothing$. *Для цього вибираємо деяку цілочислову компоненту з набору* $x_v^{(k)}$, *наприклад* $x_{s,v}^{(k)} = x_{s_0}$, $s = 1 \ldots n$.

Тоді підмножини визначають з умов:

$$G_{v_1}^{(k)} = \left\{x : x \in G_v^{(k)}, x_s \ge \left[x_{s_0}\right] + 1\right\};$$

$$G_{v_2}^{(k)} = \left\{x : x \in G_v^{(k)}, x_s \ge \left[x_{s_0}\right] + 1\right\}.$$

$k+1.2$. *Розв'язуємо початкову задачу лінійного програмування на підмножинах* $G_{v_1}^{(k)}$ *і* $G_{v_2}^{(k)}$ *і знаходимо оптимальні розв'язки* $x_{v_1}^{(k)}$ *і* $x_{v_2}^{(k)}$ *та оцінки* $\varepsilon\left(G_{v_1}^{(k)}\right) = f\left(x_{v_1}^{(k)}\right)$ *і* $\varepsilon\left(G_{v_2}^{(k)}\right) = f\left(x_{v_2}^{(k)}\right)$ *для них відповідно.*

$k+1.3$. *Якщо* $x_i^{(k)}$ *задовольняє умові цілочисельності, то вершина* $G_0^{(k)}$ *– кінцева і розглядати її далі не треба. Якщо ж для цієї вершини виконується ще й ознака оптимальності, то її вважають шуканою.*

$k+1.4$. *Відкидаємо всі нерозділенні вершини з оцінкою, гіршою за найоптимальнішу з усіх знайдених цілочислових розв'язків.*

$k+1.5$. *Якщо ще є підмножини з нецілочисловими розв'язками початкової задачі і оцінками, ліпшими за значення цільової функції для знайдених цілочислових розв'язків, повторюємо ітерацію.*

Алгоритм 9.1. Метод гілок і границь розв'язку задачі ЦЛП

Загальну реалізацію алгоритму 9.1 в Паскалі наведено в програмі 9.1.

```
Program Ex9_1;
{
G_data                              {Структура для множини розв'язків}
                                    {(залежить від конкретної задачі)}
function GetBestEstimate(): G_data; {Шукає із сукупності альтернативних множин}
                                    {розв'язків множину з найліпшою оцінкою}
procedure Delete(argument: G_data); {Видаляє деяку множину розв'язків argument}
                                    {із сукупності альтернативних множин}
procedure Push(argument: G_data);   {Додає множину argument до сукупності}
                                    {альтернативних множин}
procedure Divide(original: G_data; var G1, G2, ... , Gn: G_data);
                        {Процедура розділяє множину original на деяку кількість}
                        {підмножин G1, ..., Gn, що не перетинаються}
function Admissible(argument: G_data): boolean;
                        {Перевіряє допустимість розв'язку на множині argument}
}

function BranchAndLimitMethod(Original_G: G_data): G_data;
  var i: integer;
      select, G1, ..., Gn: G_data;
      flag: boolean;
begin
    Push(Original_G);
    repeat
      select := GetBestEstimate();
      flag := Admissible(select);
      if not(flag) then
        begin
          Divide(select, G1, ..., Gn);
          Pop(select);
          Push(G1);
          ...
          Push(Gn);
        end;
    until(flag);
    BranchAndLimitMethod := select;
end;

uses crt;
const max_m = 3 + 4;          {m – кількість обмежень, n – кількість змінних}
      max_n = 2 + max_m*2;
      Debug = false;
      max_sol = 50;
```

```
type  arr1m0n = array [1..max_m, 0..max_n] of real;
      arr1n = array [1..max_n] of real;
      arr1m = array [1..max_m] of integer;
      arr0n = array [0..max_n] of real;
      Solution = record
                   A: arr1m0n;
                   estimate: real;
                   res: arr1n;
                 end;
 var  m, n, m_real, n_real: integer;
      A: arr1m0n;                                  {Обмеження}
      c, res: arr1n;                               {Рівняння, результат}
      b: arr1m;
      Delta: arr0n;
      i, j, k, r, Step: integer;
      Sols: array [1..max_sol] of Solution;
      top: integer;

{Обчислення Delta}
procedure Delta_Calc(A: arr1m0n; c: arr1n; b: arr1m; var Delta: arr0n);
 var  i, j: integer;
      z: real;
begin
    Delta[0] := 0;
    for i := 1 to m do
      Delta[0] := Delta[0] + c[b[i]] * A[i, 0];
    for j := 1 to n do
      begin
        z := 0;
        for i := 1 to m do z := z + c[b[i]] * A[i, j];
        Delta[j] := z - c[j];
      end;
    if debug then
      begin
        write('Delta"s: ');
        for i := 0 to n do write(Delta[i]: 5: 1, ' ');
        writeln;
      end;
end;

{Оптимізація}
function Optimal(Delta: arr0n): boolean;
 var  i: integer;
      Opt: boolean;
begin
    Opt := true;
```



```
    for i := 1 to n do
      if Delta[i] < 0 then Opt := false;
    if Opt then
      begin
        writeln('Optimal Solution has been reached -- > ', Delta[0]: 8: 3);
        for i := 1 to m do res[i] := 0;
        for i := 1 to m do
          if (b[i] <= m) then res[b[i]] := A[i, 0];
      end;
    Optimal := Opt;
end;

{Провідний стовпець}
function Lead_Col(Delta: arr0n): integer;
 var  i, Ind: integer;
begin
    Ind := 1;
    for i := 2 to n do
      if Delta[i] < Delta[Ind] then Ind := i;
    Lead_Col := Ind;
    if debug then writeln('Leading Column k = ', Ind);
end;

{Провідний рядок}
function Lead_Row(A: arr1m0n; k: integer): integer;
 var  i, Ind: integer;
      NoMin: boolean;
begin
    NoMin := true;
    i := 1;
    while i <= m do
      begin
        if ((A[i, k] > 0) {and (A[i, 0] > 0)}) then
          if NoMin then
            begin
              Ind := i;
              NoMin := false;
            end
          else
            if (A[i, 0] / A[i, k]) < (A[Ind, 0] / A[Ind, k]) then Ind := i;
        Inc (i);
      end;
    if NoMin then
      begin
        writeln('The function is not bounded on the range of it"s admissible values. . . ');
        readkey;
```

```
            halt(0)
        end;
    Lead_Row := Ind;
    if Debug then writeln('Leading Row r = ', Ind);
end;

{Обчислення матриці A}
procedure A_Calc(r, k: integer; old_A: arr1m0n; var new_A: arr1m0n);
  var  i, j: integer;
begin
    for i := 1 to m do
      for j := 0 to n do
        if i <> r then new_A[i, j] := old_A[i, j] – old_A[r, j] * old_A[i, k] / old_A[r, k]
          else new_A[i, j] := old_A[r, j] / old_A[r, k];
    if Debug then
      for i := 1 to m do
        for j := 0 to n do
          if j = n then writeln(new_A[i, j]: 6: 1)
            else write(new_A[i, j]: 6: 1);
end;

{Новий базис}
procedure New_Base(A: arr1m0n; var b: arr1m);
  var  i, j, Ind: integer;
       vector: string;
begin
    for j := 1 to n do
      begin
        vector := '';
        for i := 1 to m do
          if ((A[i, j] = 1) and (vector = '')) then
            begin
              vector := '1';
              Ind := i
            end
          else
            if A[i, j] <> 0 then vector := 'not base';
            if vector = '1' then
              begin
                b[Ind] := j;
                Inc(Ind);
              end;
      end;
    if Debug then
      begin
        write('Basis: ');
        for i := 1 to m do write(b[i], ' ');
        writeln;
      end;
end;

{Функція вирішує задачу лінійного програмування}
function Solve_Linear(var A: arr1m0n; var c: arr1n): real;
begin
    if Debug then
      begin
        writeln('=============');
        writeln('Step #0');
        for i := 1 to m do
          for j := 0 to n do
            if j = n then
              writeln(A[i, j]: 6: 1)
            else write(A[i, j]: 6: 1);
      end;
    New_Base(A, b);
    Delta_Calc(A, c, b, Delta);
    Step := 1;
    while not(Optimal(Delta)) do
      begin
        if Debug then
          begin
            readkey; writeln; writeln; writeln('=============');
            writeln('Step #', Step)
          end;
        k := Lead_Col(Delta);
        r := Lead_Row(A, k);
        A_Calc(r, k, A, A);
        New_Base(A, b);
        Delta_Calc(A, c, b, Delta);
        Inc(Step);
      end;
    {Repeat until keypressed;}
    Solve_Linear := Delta[0];
end;

procedure Print_A(var A: arr1m0n);
  var  i, j: integer;
begin
    writeln('Print_A');
    for i := 1 to m do
      begin
        for j := 0 to n do write(A[i, j]: 5: 2, ' ');
```

```
            writeln;
        end;
end;

{Функція здійснює підготовку варіанта вирішення}
procedure Prepare_Solution(index: integer);
  var  i: integer;
begin
    A := Sols[index].A;
    Sols[index].estimate := Solve_Linear(A, c);
    Sols[index].res := res;
{
    write('Solution #', index: 2, ': ');
    for i := 1 to n_real do
      write(Sols[index].res[i]: 10: 3, ' ');
    writeln;
    readln;
}
end;

{Функція встановлює додаткове обмеження на змінну var_num <= value}
procedure SET_COND(index: integer; var_num: integer; value: real);
  var  nn: integer;
begin
    if (value < 0) then nn := var_num * 2
      else nn := var_num * 2 – 1;
    Sols[index].A[m_real + nn, 0] := value;
    Sols[index].A[m_real + nn, var_num] := 1;
    {Print_A(Sols [index].A);}
end;

{Функція знаходить рішення з найліпшою оцінкою}
function GetBestEstimate: integer;
  var  i, mm: integer;
begin
    mm := 1;
    for i := 1 to top do
      if (Sols[i].estimate > Sols[mm].estimate) then mm := i;
    GetBestEstimate := mm;
end;

{Функція перевіряє допустимість розв'язку (цілочисельність)}
function Admissible(index: integer): boolean;
  var  i: integer;
begin
    for i := 1 to m do
      if (int(Sols[index].res[i]) <> Sols[index].res[i]) then
```



```
        begin
          Admissible := false;
          exit;
        end;
    Admissible := true;
end;

{Функція розділяє розв'язок на дві множини}
procedure Divide(index: integer);
  var  i, sel: integer;
       r: real;
begin
    sel := 0;
    for i := 1 to m_real do
      if (int(Sols[index].res[i]) <> Sols[index].res[i]) then
        begin
          sel := i;
          r := Sols[index].res[i];
          break;
        end;
    inc(top);
    Sols[top] := Sols[index];
    SET_COND(index, sel, int(r));
    SET_COND(top, sel, –int(r));
    Prepare_Solution(index);
    Prepare_Solution(top);
end;

{Метод гілок та границь}
procedure BranchAndLimitMethod;
  var  sel, i: integer;
       flag: boolean;
begin
    repeat
      sel := GetBestEstimate;
      flag := Admissible(sel);
      if not(flag) then Divide(sel);
    until flag;
    writeln('Result: ');                                    {Друк результату}
    writeln('Function value: ', Sols[sel].estimate: 10: 3);
    write('Variables: ');
    for i := 1 to n_real do
      write (Sols[sel].res[i]: 7: 0);
    writeln;
end;
```

```
{Початок}
begin
    m_real := 3;                                    {3 обмеження}
    n_real := 2;                                    {2 змінна}
    m := m_real + n_real * 2;
    n := n_real + m;
    for i := 1 to max_m do
      for j := 0 to max_n do A[i, j] := 0;
    A [1, 0] := 7; A [1, 1] := 1; A [1, 2] := 1;    {x + y <= 7}
    A [2, 0] := 5; A [2, 1] := 4; A [2, 2] := -5;   {4x - 5y <= 5}
    A [3, 0] := 38; A [3, 1] := 2; A [3, 2] := 11;  {2x + 11y <= 38}
    for i := 1 to m do A[i, n - m + i] := 1;
{Рівняння: max(x + y) те саме, що min(-x - y)}
    c[1] := 1; c[2] := 1; c[3] := 0; c[4] := 0; c[5] := 0;
    clrscr;
    Sols[1].A := A;
    top := 1;
    Prepare_Solution(1);
    BranchAndLimitMethod;
end.
```

Програма 9.1. Метод гілок і границь розв'язання задачі ЦЛП

9.3. Застосування методу для розв'язання задачі комівояжера

Нагадаємо постановку задачі комівояжера. Комівояжеру потрібно розповсюдити свої товари в містах деякого штату і повернутися додому. Слід мінімізувати довжину шляху відвідування міст за умови, що жодне місто не можна відвідувати більше одного разу. *Тобто у зваженому графі потрібно знайти найкоротший простий цикл, який містив би всі вершини графа.*

Звичайно, цей цикл складатиметься з деякої підмножини ребер, що входить до множини G_0 усіх ребер графа. Вдало розділяючи множину G_0 на підмножини, можна виокремити цикли (не обов'язково всі), серед яких буде найкоротший. Слід ввести засіб визначати оцінку найкоротшого циклу в поданій підмножині ребер. Якщо ця оцінка виявиться більшою за довжину якогось вже знайденого циклу, можна бути впевненим, що у цій підмножині ребер графа неможливо виокремити найкоротший повний цикл.

Проілюструємо застосування методу гілок і границь для задачі комівояжера на графі, зображеному на рис. 9.2.

Мінімально можливу довжину циклу в ньому можна визначити (і це не єдиний метод), якщо скласти довжини пар найкоротших ребер, що інцидентні кожному ребру, та поділити на два. Наприклад, два найкоротші

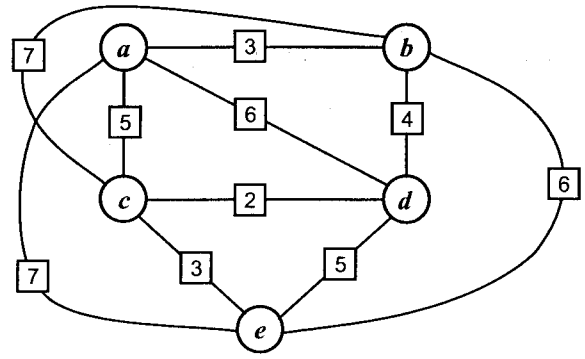


Рис. 9.2. Графічна інтерпретація деякої території відвідин комівояжера

ребра, що інцидентні вершині a, мають довжину 3 і 5, 3 + 5 = 8. Для вершини b ця сума становитиме 7, для c – 5, для d – 6, для e – 8; сума – 34. Отже, найкоротший можливий шлях буде не меншим за 17.

Розділяти множину ребер на підмножини будемо за ознакою наявності чи відсутності певного ребра, тоді для кожного розділення множини $G_v^{(k)}$ (v-та підмножина після k-го розбиття) на підмножини буде $G_{v_1}^{(k+1)}$, $G_{v_2}^{(k+1)}$, і виконуватимуться умови:

$$G_v^{(k)} = G_{v_1}^{(k+1)} \cup G_{v_2}^{(k+1)},$$
$$G_{v_1}^{(k+1)} \cap G_{v_2}^{(k+1)} = \varnothing.$$

Наприклад, початкову множину ребер можна розділити на множини G_1^1 та G_2^1, якщо вважати, що множина G_1^1 обов'язково матиме ребро ab, множина G_2^1 не матиме ребра ab у жодному разі. До того ж, окрім запланованих вилучень чи включень ребра до множини, вилучатимемо ребро у тому разі, коли воно виявиться третім інцидентним ребром однієї вершини, а обидва інші інцидентні цій вершині ребра вже було включено до цієї підмножини як обов'язкові. Ми змушені також вилучати ребро з графа, якщо воно утворює неповний цикл з ребрами, які вже було включено до підмножини як обов'язкові. Ми також не можемо вилучити з підмножини якесь ребро, коли в результаті цього вилучення якась з вершин графа залишиться лише з одним інцидентним ребром.

Отже, якщо ми зафіксували, що до якоїсь підмножини мають обов'язково входити ребра cd і de, то ми повинні вилучити з цієї підмножини ребра ad і bd як «зайві» для вершини d і ребро ce як таке, що утворює неповний цикл. І за таких обставин ми не зможемо у подальшому розділенні цієї підмножини обійтися одночасно без ребер ae і be, оскільки тоді вершина e матиме лише одне інцидентне ребро.

Нарешті, уявімо собі, що кожній множині вершин, що не утворює цикл, відповідає вузол на деякому дереві (рис. 9.3). Листкам цього

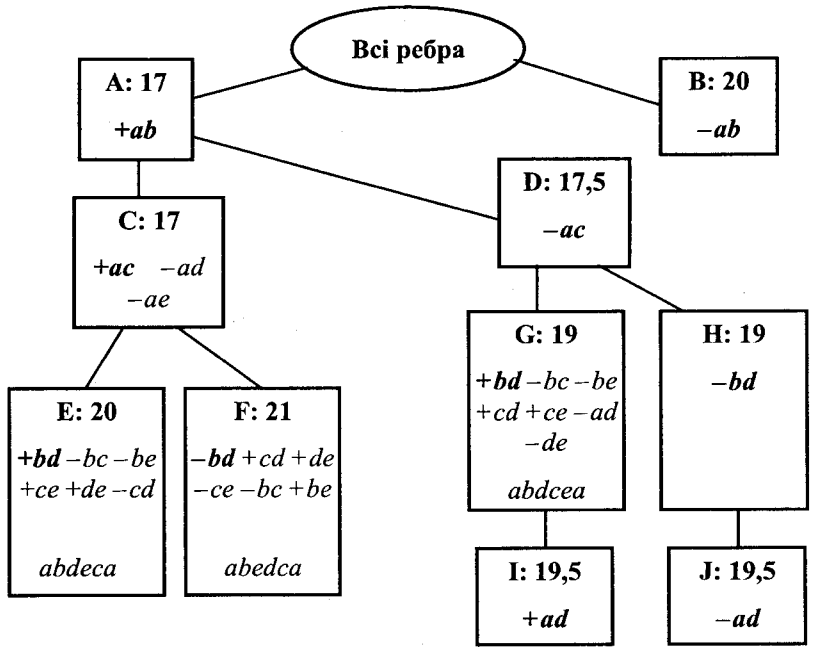


Рис. 9.3. Процес побудови маршруту комівояжера

дерева відповідатимуть знайдені цикли. Кожен вузол матиме двох нащадків (вузли чи листя).

Вузлу *A* відповідає множина вершин, що обов'язково містить ребро *ab*, і цей вузол, відповідно, має таку ж оцінку, як і корінь дерева. Вузлу *B* з оцінкою 20 відповідає множина всіх ребер, за винятком *ab*. Далі розділяємо множину, що відповідає вузлу *A* (оскільки в нього ліпша оцінка) за ознакою належності відповідній множині вузла *ac*. Вузол дерева *C* відповідає множині, що містить і *ab*, і *ac*, отже, до цієї множини не можуть входити ребра *ad* і *ae*. Для такої множини оцінка становитиме 17. Для множини, що відповідає вузлу *D*, – 17.5, тому далі розкриватимемо вузол *C*. Його нащадки відповідають множинам, що мають ребро *bd* (вузол *E*) чи не мають (*F*). За наведеними раніше правилами примусового додавання і вилучення ребер для обох вершин можна визначити повні цикли, у такий спосіб, вони насправді не вершини, а листи з оцінками 20 і 21 відповідно. На схемі позначено ці цикли у нижньому рядку листа. Нас цікавить той з них, що має меншу оцінку.

Нараз розкриваємо вузол *D*. Для його нащадка *G* наявність ребра *bd* також дає змогу визначити цикл довжини 19. Отже, можна напевне сказати, що серед циклів без ребра *ab* (якщо такі взагалі існують) найкоротшого циклу немає, оскільки вершина *B* має оцінку 20 і вона відтинається від дерева розгляду. Нащадок *H* вузла *D* є вузлом і має теж оцінку 19. Хоча його дослідження не може дати розв'язок, ліпший за отриманий, але ми можемо тут знайти ще один найкоротший повний цикл.

Для нащадків вузла *H* одночасну наявність вузлів *ad* і *ae* виключають. Оцінка їх обох становить 19,5, що менше за оцінку листа *G*, тому вузли *I* та *J* можна далі не розглядати.

Зауважимо, що порядок вибору ребер графа, за якими розділяють множини, може бути довільним.

```
Program Ex9_2;

{Розв'язання задачі комівояжера. Вхідні дані: файл input.txt. Формат вхідних
даних: n – кількість вершин графа, заданого матрицею суміжності A[n, n], де a_ij –
відстань між містом i та j. Вихідні дані: перелік ребер графа, що утворюють
оптимальний маршрут}

const MAX_MATRIX = 15;                    {Максимальна кількість вершин графа}
      debug = FALSE;
  type DataPtr = ^Data;
      Data = record                       {Множина розв'язків}
               matr: array [1.. MAX_MATRIX, 1.. MAX_MATRIX] of integer;
               n, e: integer;                          {n – розмірність, e – оцінка}
               a, b: array [1.. MAX_MATRIX] of integer;          {Оцінки ребер}
             end;
  var matr: Data;

{Процедура читання графа з файла}
procedure ReadInputData(fname: string; var res: Data);
  var inf: text;
      i, j: integer;
begin
    assign(inf, fname);
    reset(inf);
    readln(inf, res.n);
    for i := 1 to res.n do
      begin
        for j := 1 to res.n do
          read(inf, res.matr[i, j]);
        readln(inf);
      end;
    res.e := 0;
    close(inf);
end;

{Процедура виводить перелік ребер маршруту}
procedure WriteOutputData(var matr: Data);
  var i, j: integer;
begin
    writeln('Result path: ');
```

```
    for i := 1 to matr.n do
      for j := 1 to matr.n do
        if (matr.matr[i, j] = –10) or (matr.matr[i, j] = 0) then
          writeln(i, '', j);
end;

{Процедура «зведення» матриці та перерахування оцінки}
procedure Transform(sourc: Data; var matr: Data);
  var  i, j, k: integer;
       hi, Hj: array [1. . MAX_MATRIX] of integer;
begin
    matr := sourc;
    for i := 1 to matr.n do          {Підрахунок h[i] = min(1 < = j < = n) (matr[i][j])}
      begin
        hi[i] := –1;
        for j := 1 to matr.n do
          if (matr.matr[i, j] > = 0) then
            if (hi[i] > = 0) then
              begin
                if (hi[i] > matr.matr[i, j]) then hi[i] := matr.matr[i, j];
              end
              else hi[i] := matr.matr[i, j];
      end;
{Перетворення 1}
    for i := 1 to matr.n do
      for j := 1 to matr.n do
        if (matr.matr[i, j] >= 0) then
          matr.matr[i, j] := matr.matr[i, j] – hi[i];
    for i := 1 to matr.n do          {Підрахунок H[j] = min(1 <= i <= n) ( matr[i][j])}
      begin
        hj[i] := –1;
        for j := 1 to matr.n do
          if (matr.matr[j, i] >= 0) then
            if (hj[i] >= 0) then
              begin
                if (hj[i] > matr.matr[j, i]) then hj[i] := matr.matr[j, i];
              end
              else
                hj[i] := matr.matr[j, i];
      end;
{Перетворення 2}
    for i := 1 to matr.n do
      for j := 1 to matr.n do
        if (matr.matr[j, i] >= 0) then
          matr.matr[j, i] := matr.matr[j, i] – hj[i];
```



```
{Оцінювання}
    for i := 1 to matr.n do
      begin
        if (hi[i] >= 0) then matr.e := matr.e + hi[i];
        if (Hj[i] >= 0) then matr.e := matr.e + Hj[i];
      end;
{Підрахунок ai, bj}
    for i := 1 to matr.n do
      begin
        k := 0;
        matr.a[i] := –1;
        for j := 1 to matr.n do
          if (matr.matr[i, j] > 0) then
            begin
              if (matr.a[i] < 0) or (matr.a[i] > matr.matr[i, j]) then
                matr.a[i] := matr.matr[i, j];
            end
            else
              if (matr.matr[i, j] = 0) then
                begin
                  inc(k);
                  if (k > 1) then
                    matr.a[i] := 0;
                end;
      end;
    for i := 1 to matr.n do
      begin
        k := 0;
        matr.b[i] := –1;
        for j := 1 to matr.n do
          if (matr.matr[j, i] > 0) then
            begin
              if (matr.b[i] < 0) or (matr.b[i] > matr.matr[j, i]) then
                matr.b[i] := matr.matr[j, i];
            end
            else
              if (matr.matr[j, i] = 0) then
                begin
                  inc(k);
                  if (k > 1) then matr.b[i] := 0;
                end;
      end;
    if (DEBUG) then
      begin
        for i := 1 to matr.n do
```

```
            begin
              for j := 1 to matr.n do
                write(matr.matr[i, j]: 2, ' ');
              writeln;
            end;
          write('h[i] = ');
          for i := 1 to matr.n do
            write(hi[i]: 2, ' ');
          writeln;
          write ('H[j] = ');
          for i := 1 to matr.n do
            write (hj[i]: 2, ' ');
          writeln;
          write('a = ');
          for i := 1 to matr.n do
            write(matr.a[i]: 2, ' ');
          writeln;
          write('b = ');
          for i := 1 to matr.n do
            write(matr.b[i]: 2, ' ');
          writeln;
          writeln('E = ', matr.e);
        end;
end;

{Процедура поділу множини рішень matr на дві підмножини res1, res2}
function Divide(var matr: Data; var res1, res2: Data): boolean;
  var  i, j, ni, nj: integer;
       bb: boolean;
begin
    bb := false;
    for i := 1 to matr.n do
      begin
        if matr.a[i] >= 0 then bb := true;
        if matr.b[i] >= 0 then bb := true;
      end;
    if not(bb) then
      begin
        Divide := false;
        exit;
      end;
{Вибір ребра, ґрунтується на оцінках a, b}
    ni := -1;
    nj := -1;
    for i := 1 to matr.n do
```



```
      for j := 1 to matr.n do
        if (matr.matr[i, j] = 0) then
          if (ni = -1) or (matr.a[ni] + matr.b[nj] < matr.a[i] + matr.b[j]) then
            begin
              ni := i;
              nj := j;
            end;
    if DEBUG then writeln('Divide REBRO: ', ni, ' ', nj);
{Формування 1-го варіанта}
    res1 := matr;
    for i := 1 to matr.n do
      begin
        res1.matr[ni, i] := -1;
        res1.matr[i, nj] := -1;
      end;
    res1.matr[ni, nj] := -10;
    res1.matr[nj, ni] := -1;
    Transform(res1, res1);
{Формування 2-го варіанта}
    res2 := matr;
    res2.matr[ni, nj] := MaxInt;                                    {MaxInt;}
    Transform(res2, res2);
    if DEBUG then writeln('E1 = ', res1.e, ' E2 = ', res2.e);
    Divide := true;
end;

{Процедура перевіряє, чи містить множина розв'язок}
function CheckCycle(var matr: Data): boolean;
  var  i, j, k: integer;
begin
    k := 0;
    for i := 1 to matr.n do
      for j := 1 to matr.n do
        if (matr.matr[i, j] = -10) or (matr.matr[i, j] = 0) then inc(k);
    if (k = matr.n) then CheckCycle := true
      else CheckCycle := false;
end;

{Головна процедура (реалізує метод гілок і границь)}
procedure Process(var matr: Data);
  var  d1, d2, d3: data;
       sol: array [1. . 100] of DataPtr;
       top, i, k: integer;
       flag: boolean;
begin
    Transform(matr, d1);
```

```
    new(sol[1]);
    sol[1]^ := d1;
    top := 1;
    repeat
      k := 1;
      for i := 1 to top do
        if (sol[i]^.e < sol[k]^.e) then k := i;
      flag := Divide(sol[k]^, d2, d3);
      sol[k]^ := d2;
      inc(top);
      new(sol[top]);
      sol[top]^ := d3;
      if not(flag) then
        if not(CheckCycle(sol[k]^)) then
          begin
            sol[k]^.e := maxint;
            flag := true;
          end;
    until not(flag);
    WriteOutputData(sol[k]^);
    for i := 1 to top do dispose(sol[i]);
end;

begin
    ReadInputData('input.txt', matr);
    Process(matr);
end.
```

Програма 9.2. Розв'язання задачі комівояжера

Задачі та вправи до розділу 9

1. Розв'язати методом гілок і границь задачу: знайти $\max(x_1 + x_2)$ за умови

$$\begin{cases} 2x_1 + 5x_2 \le 24; \\ x_1 + x_2 \le 6; \\ 3x_1 - 4x_2 \le 4, \end{cases}$$

$x_1, x_2 \ge 0$, x_1, x_2 – цілі числа.

2. Запропонуйте методику використання методу гілок і границь для розв'язання задачі календарного планування роботи трьох станків. Нехай три станки r_1, r_2, r_3 послідовно обробляють n деталей d_i ($i = 1, 2, ..., n$). Технологічні маршрути обробки кожної деталі є однаковими, а час обробки деталі d_i на кожному із станків r_1, r_2, r_3 становитиме відповідно a_i, b_i, c_i. Визначити такий порядок запуску деталей на обробку, який забезпечить мінімально можливий час обробки всіх деталей.

3. [Задача про оптимальні призначення]. Нехай є n видів робіт та n кандидатів на їх виконання. Вважається, що кожен з кандидатів $i = 1, ..., n$ може виконувати будь-яку роботу $j = 1, ..., n$, при цьому c_{ij} – витрати, пов'язані з призначенням i-го кандидата на j-й вид роботи. Потрібно розподілити кандидатів на виконання робіт так, щоб кожен з кандидатів одержав єдине призначення, кожна з робіт одержала єдиного виконавця та сумарні витрати, пов'язані з призначенням, були мінімальними.

Математичну модель задачі можна задати у вигляді задачі лінійного програмування з цілочисловими змінними, тобто потрібно знайти матрицю $X = \|x_{ij}\|$, $i, j = 1, \ldots, n$, що мінімізує цільову функцію $L(X) = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$ з обмеженнями $\sum_{j=1}^{n} x_{ij} = 1$, $i = 1, \ldots, n$, $\sum_{i=1}^{n} x_{ij} = 1$, $j = 1, \ldots, n$, $x_{ij} = 1$ (якщо i-й виконавець призначений на j-ту роботу), $x_{ij} = 0$, а якщо ні, то $i, j = 1, \ldots, n$.

Знайдіть розв'язок цієї задачі, використавши метод гілок і границь, за умов: $n = 3$, матриця витрат C має значення $c[1, 1] := 1$, $c[1, 2] := 0$, $c[1, 3] := 2$, $c[2, 1] := 2$, $c[2, 2] := 1$, $c[2, 3] := 1$, $c[3, 1] := 1$, $c[3, 2] := 3$, $c[3, 3] := 0$.

4. [Задача про ранець]. Нехай потрібно помістити у ранець n предметів. Вважається, що відома вагова характеристика корисності c_j одного предмета j-го типу. Допускають варіант, що в рюкзак можна покласти кілька однакових предметів одного з типів. Ранець має m обмежень (об'єм, лінійні розміри, вага тощо). Введемо позначення a_{ij}, i-та характеристика ($i = 1, \ldots, m$) предмета j-го типу ($j = 1, \ldots, n$), b_j – максимальне значення i-ї характеристики ранця. Потрібно визначити, які предмети і в якій кількості слід завантажити в ранець, щоб їх сумарна корисність була максимальною.

Якщо через x_i позначити кількість предметів j-го типу, що планують помістити в ранець, то математичне формулювання задачі (тобто слід знайти вектор x, що максимізує цільову функцію) набуває вигляду:

$$L(x) = \sum_{j=1}^{n} c_j x_j$$

з обмеженнями $\sum_{j=1}^{n} a_{ij}x_j \le b_j$, $i = 1, \ldots, m$, $x_j \ge 0$, x_j – ціле, $j = 1, \ldots, n$.

Розв'яжіть задачу про ранець, використавши метод гілок і границь, для випадку: $n = 4$, а матриця витрат C має значення $a[1, 1] := 2$, $a[1, 2] := 1$, $a[1, 3] := 1$, $a[1, 4] := 3$, $a[2, 1] := 1$, $a[2, 2] := 1$, $a[2, 3] := 2$, $a[2, 4] := 3$, $a[3, 1] := 1$, $a[3, 2] := 1$, $a[3, 3] := 1$, $a[3, 4] := 2$, $a[4, 1] := 2$, $a[4, 2] := 1$, $a[4, 3] := 2$, $a[4, 4] := 1$, $x_j = 1, j = 1, \ldots, 4$, $c_1 = 2$, $c_2 = 3$, $c_3 = 1$, $c_4 = 5$, $b_1 = 3$, $b_2 = 4$, $b_3 = 2$, $b_4 = 6$.

Розділ 10

ІГРОВІ ЗАДАЧІ

Кожна освічена людина в повсякденному житті часто натрапляє на поняття «гра», «ігрова задача», але дати формальне визначення цих понять жоден з них не може, висловлюють лише велику кількість визначень описового типу. Неоднозначність випливає з філософського визначення поняття «гра». Класично розрізняють математичну *теорію ігор* та ігрові програми [17]. Перша – здебільшого належить теорії математичного програмування і числових методів розв'язання відповідних задач пошуку екстремумів [83]. Традиційно вона розглядає визначення предмету математичного програмування, його місце в науці про дослідження операцій, а також математичні моделі задач техніко-економічного змісту і слугує інструментарієм знаходження алгоритму розв'язання певної задачі оптимізації. Базою теорії вважають результат Льюїса і Райфа: існує алгоритм визначення оптимального ходу в довільній стадії гри *n* осіб, в якій кожен учасник має повну інформацію і число можливих ходів скінченне. Теорія ігор є важливою складовою частиною дослідження операцій. Її перше систематизоване викладення зроблено Нейманом і Моргенштерном у 1944 р. в [76], хоча перші результати отримано у 1920-х роках.

Ігрова програма займається побудовою ефективних наглядних програм, реалізованих на комп'ютері [1, 100]. Моделюють ігри, в які люди колись грали тільки між собою. Важливо звернути увагу на різні цілі ігрових програм і математичної *теорії ігор*.

10.1. Загальна характеристика

Ігрові задачі є типовим класом задач, які традиційно належать до інтелектуальних. Оскільки вибір чергового ходу в іграх є не що інше, як прийняття рішення, методи програмування ігрових задач найтіснішим чином пов'язані з методами планування цілеспрямованих дій і прийняття рішень. Характерною особливістю ігрових задач є *наявність суперника*, що активно перешкоджає здійсненню цілей, які ставить перед собою кожен гравець.

Теорія ігор – це математична дисципліна, що вивчає питання поведінки учасників конфліктних ситуацій та має на меті виробити оптимальну для кожного з учасників стратегію такої поведінки. Конфліктною при-

цьому називають ситуацію, коли гравці мають різні цілі (різні функції виграшу) та можуть вибирати різні засоби досягнення своїх цілей (стратегії).

Для побудови ігрових програм найбільший інтерес становлять методи знаходження планів гри і оптимальних стратегій для таких ігор, як шахи, шашки, хрестики-нулики тощо. З погляду теорії ці ігри є ідентичними між собою. Їх відносять до класу *позиційних ігор двох осіб*. Кожен гравець може по черзі зробити будь-який хід з тих, які дозволені правилами гри. Ці ігри є *детермінованими* у тому розумінні, що перебіг гри та вибір ходу не залежать від випадкових чинників. Крім того, це *ігри з повною інформацією*, тобто кожному гравцеві доступна вся інформація про будь-яку позицію, яка утворюється в процесі гри. Такі ігри відносять до класу *антагоністичних ігор*, або *ігор з нульовою сумою*. Це означає, що сума виграшів обох гравців дорівнює нулю, тобто виграш одного гравця дорівнює програшу іншого. З цього випливає, що замість двох функцій виграшу можна розглядати одну.

Є відомі позиційні ігри, які належать до інших класів. Так, нарди є грою з повною інформацією, але не є детермінованою у тому розумінні, що гравець не може зробити довільний хід; його вибір обмежений випадковими чинниками. Преферанс не є грою з повною інформацією і не є грою, детермінованою в тому розумінні, що початкова позиція залежить від випадку. Але після того як початкову позицію зафіксовано, гравець може зробити будь-який хід, дозволений правилами.

Загальновизнано, що реалізовані у сучасних шахових та інших ігрових програмах варіанти повного перебору є прикладами «нелюдського» підходу до гри: людина ніколи не робить перебору, до якого залучається величезна кількість безглуздих ходів. Шахіст швидко відчуває основні особливості позиції, помічає ходи, на які слід звернути увагу щонайперше. У мозку шахіста виникає *дерево планів* [109] на основі того, що під час аналізу позиції шахіст має змогу поставити перед собою *певні цілі*. План має визначену кінцеву мету, тому насамперед він аналізує ходи, які можуть сприяти або завадити реалізації цього плану. Такий підхід реалізований у програмі ROBIN [109].

Відзначимо також дослідження М. М. Ботвінника та його програму «Піонер» [12]. У найзагальніших рисах підхід ґрунтується на розгляді шахової гри на основі розв'язання задачі *багаторівневого керування з неточною метою*. Шахову гру розглядають як багаторівневу систему. Кожна шахова фігура має певну мету, інші фігури можуть допомагати або заважати їй у досягненні певних цілей. Локальні цілі слід узгоджувати з глобальними (виграш, досягнення матеріальної переваги), за цим слідкує система керування вищого рівня.

Людська гра спирається на ряд *евристик*, застосування яких може різко скоротити перебір і часто дає змогу зробити розумний хід майже без аналізу варіантів. До таких шахових евристик належать, наприклад, вимоги боротьби за центр, швидкого розвитку фігур, своєчасної рокіровки

тощо. Зрозуміло, що стратегія, яка спирається на такі правила, далеко не завжди є оптимальною. Вона може бути наближеною до оптимальної або просто раціональною.

Крім розглянутої гри з суперником, який завжди намагається робити ходи, що є для нас найгіршими, можна розглядати такі позиційні ігри: *гра з природою, яка робить випадкові ходи; гра з партнером, який робить найкращі для нас ходи* (таку гру часто називають *кооперативною*). Можна також говорити про *рефлексію*, тобто про врахування намірів і можливостей суперника. Так, гравець, роблячи ризикований хід, може сподіватись на те, що суперник не знайде найкращу відповідь. Гра у такому разі набуває рис імовірнісного, азартного характеру.

Для вирішення різних ігрових задач слід застосовувати алгоритми та процедури з елементами певної «інтелектуальності». Очевидно, говорити, що система стає більш інтелектуальною тільки від того, що для розв'язання нової задачі запрограмований новий алгоритм, не доводиться. Інтелектуальною слід вважати систему, яка вміє орієнтуватися і планувати свою поведінку в новій ситуації. Для цього вона повинна мати у своєму розпорядженні певні *загальноінтелектуальні метапроцедури* з елементами самонавчання.

Спочатку розвивалася *стимульно-реактивна* теорія, яка розглядає навчання як поступове вироблення правильної реакції на зовнішні подразники. Цю теорію згодом було поповнено теорією лабіринтів [24]. Згідно з нею, задачу планування цілеспрямованих дій розв'язують на основі метапроцедури пошуку в лабіринті можливостей. Лабіринт можливостей може бути не єдиним. Було висунуто модельну гіпотезу, згідно з якою центральною метапроцедурою інтелектуальної діяльності є знаходження нового лабіринту, в якому можна організувати ефективний пошук рішення. Іншою метапроцедурою є структуризація опису ситуації, тобто виділення базових елементів та характерних особливостей, на основі яких можна приймати рішення. Виділяють також такі метапроцедури, як зведення задач до підзадач, формування закономірностей на основі спостережень тощо.

Характерною особливістю «інтелектуального» ухвалення рішень під час вибору ходу в певній ігровій ситуації є практична неможливість проаналізувати всі елементи ситуації та всі варіанти рішень за прийнятний час. Тому необхідною частиною підсистеми вибору варіанта ходу і повинні стати метапроцедури евристичного скорочення пошуку та припинення такого пошуку, якщо його продовження видається недоцільним. Зрозуміло, що рішення не обов'язково буде оптимальним. Можна сформулювати такі критерії припинення аналізу ситуації та ухвалення деякого рішення:

Принцип толерантності. Нехай r – оцінка ситуації, в яку потрапить система за умови, що рішення ухвалюють на основі повного аналізу поточної ситуації, s – оцінка ситуації, в яку потрапить система за умови, якщо рішення ухвалюють негайно. Рішення можна виносити, якщо r і s

зв'язані відношенням толерантності, тобто якщо ситуації r і s оцінюють як досить схожі.

Принцип мінімізації затрат. Пошук припиняють і рішення ухвалюють негайно, якщо оцінка затрат, пов'язаних з продовженням пошуку, перевищує оцінку втрат від неоптимальності рішення.

Класифікацію ігор можна проводити за кількістю гравців і стратегій, за характером взаємодії гравців, характером виграшу, за кількістю ходів, станом інформації тощо [83].

Залежно від кількості гравців розрізняють ігри *двох* та *n* гравців. Ігри трьох і більше гравців менше досліджені через принципові складності й технічні можливості отримання розв'язку.

За кількістю стратегій ігри поділяють на скінченні та нескінченні. Якщо у грі всі гравці мають скінченне число можливих стратегій, то її називають *скінченною*. Якщо ж хоча б один із гравців має необмежену кількість можливих стратегій, то гру називають *нескінченною*.

За характером взаємодії ігри поділяються на такі:

- *безкоаліційні* (гравці не мають права вступати в угоди, утворювати коаліції);
- *коаліційні* (кооперативні), з дозволом вступати в коаліції.

У кооперативних іграх коаліції визначають заздалегідь.

За характером виграшу ігри класифікують так: ігри з *нульовою сумою* (загальний капітал усіх гравців не змінюється, а перерозподіляється між гравцями; сума виграшів усіх гравців дорівнює нулю) та ігри з *ненульовою сумою*.

За виглядом функцій виграшу ігри поділяють на матричні, біматричні, безупинні, опуклі, сепарабельні, типу дуелей та ін.

Матрична гра – це скінченна гра двох гравців із нульовою сумою, у якій задано виграш першого гравця у вигляді матриці (рядок матриці відповідає номеру застосованої стратегії першого гравця, стовпчик – номеру застосованої стратегії другого гравця; на перетині рядка і стовпця матриці – виграш першого гравця, що відповідає застосованим стратегіям). Для матричних ігор доведено, що будь-яка з них має рішення і його можна легко знайти шляхом зведення гри до задачі лінійного програмування.

Біматрична гра – це скінченна гра двох гравців із ненульовою сумою, у якій виграші кожного гравця можна задати матрицями окремо для відповідного гравця (у кожній матриці рядок відповідає стратегії першого гравця, стовпчик – стратегії другого гравця, на перетині рядка і стовпця в першій матриці знаходиться виграш першого гравця, у другій матриці – виграш другого гравця). Для біматричних ігор також розроблено теорію оптимальної поведінки гравців, однак пошук розв'язків для таких ігор складніший, ніж для звичайних матричних.

Неперервною вважають гру, в якій функція виграшів кожного гравця є неперервною, незалежно від стратегій. Доведено, що ігри цього класу мають розв'язки, однак практично не розроблено прийнятних методів їх знаходження.

Якщо функція виграшів є опуклою, то таку гру називають *опуклою*. Для таких ігор розроблено прийнятні методи рішення, суть яких у відшуканні чистої оптимальної стратегії (визначеного числа) для одного гравця та імовірностей застосування чистих оптимальних стратегій для іншого гравця. Таке завдання розв'язують порівняно легко.

Вивченням математичних особливостей «теорії ігор» займається такий розділ математики, який історично дістав назву дослідження операцій. Для докладнішого ознайомлення з цією наукою допитливого читача можна відіслати до [14, 17, 47]. Ми ж, на прикладі матричних ігор, спробуємо тільки охарактеризувати цей напрям математичних досліджень.

10.2. Основні математичні моделі «теорії ігор»

Матричну гру двох гравців із нульовою сумою можна розглядати як наступну абстрактну гру двох гравців.

Перший гравець має m стратегій $i = 1, 2, ..., m$, другий – n стратегій $j = 1, 2, ..., n$. Кожній парі стратегій (i, j) поставлено у відповідність число a_{ij}, яке виражає виграш першого гравця за рахунок другого гравця, якщо кожен з гравців візьме *свою* стратегію.

Кожен із гравців робить один хід: перший гравець вибирає свою i-ту стратегію ($i = \overline{1,m}$), а другий – свою j-ту стратегію ($j = \overline{1,n}$), після чого перший одержує виграш a_{ij} за рахунок другого (якщо $a_{ij} < 0$, то це означає, що перший гравець винен другому суму $|a_{ij}|$). На цьому гру закінчують.

Кожну стратегію гравців $i = \overline{1,m}$, $j = \overline{1,n}$ часто називають чистою стратегією.

Якщо розглянути матрицю

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ & & & \dots & & \\ a_{i1} & a_{i2} & \dots & a_{ij} & \dots & a_{in} \\ & & & \dots & & \\ a_{m1} & a_{m2} & \dots & a_{mj} & \dots & a_{mn} \end{pmatrix},$$

то проведення кожної партії матричної гри за матрицею A зводиться до вибору першим гравцем i-го рядка, а другим – j-го стовпця й одержання першим гравцем (за рахунок другого) виграшу a_{ij}.

Головним у дослідженні ігор є поняття оптимальних стратегій гравців. У це поняття інтуїтивно вкладають такий зміст: стратегія гравця є оптимальною, якщо застосування цієї стратегії забезпечує йому найбільший гарантований виграш за будь-яких стратегій іншого гравця. Враховуючи це, перший гравець досліджує матрицю виграшів A у такий спосіб: для кожного значення i ($i = \overline{1,m}$) визначають мінімальне значення виграшу, залежно від стратегій другого гравця:

$$\min_j a_{ij} \quad (i = \overline{1,m}),$$

тобто мінімальний виграш для першого гравця за умови, що він візьме свою i-ту чисту стратегію, далі із цих мінімальних виграшів відшукують таку стратегію $i = i_{\text{про}}$, за якої цей мінімальний виграш буде максимальним, тобто знаходять

$$\max_i \min_j a_{ij} = a_{i_0 j_0} = \underline{\alpha}. \qquad (10.1)$$

Число $\underline{\alpha}$, визначене за формулою (10.1), називають *нижньою чистою ціною гри*; воно показує, який мінімальний виграш може гарантувати собі перший гравець, застосовуючи свої чисті стратегії за будь-яких дій другого гравця.

Другий гравець за своєї оптимальної поведінки прагне завдяки своїм стратегіям максимально зменшити виграш першого гравця. Тому для другого гравця відшукують $\min_i a_{ij}$, тобто визначають найбільший можливий виграш першого, за умови, що другий гравець застосує свою j-ту чисту стратегію, а далі відшукає таку свою $j = j_1$ стратегію, за якої перший гравець одержить мінімальний виграш, тобто знаходить

$$\max_j \min_i a_{ij} = a_{i_1 j_1} = \overline{\alpha}. \qquad (10.2)$$

Число $\overline{\alpha}$, обчислене за формулою (10.2), називають *чистою верхньою ціною гри*; воно показує, який максимальний виграш завдяки своїм стратегіям може собі гарантувати перший гравець.

Інакше кажучи, застосовуючи свої чисті стратегії, перший гравець може забезпечити собі виграш не менше $\underline{\alpha}$, а другий гравець завдяки застосуванню своїх чистих стратегій може не допустити виграшу першого гравця більшого за $\overline{\alpha}$.

Якщо в грі з матрицею A $\underline{\alpha} = \overline{\alpha}$, то кажуть, що така гра має *сідлову точку* в чистих стратегіях та *чисту ціну* гри $v = \underline{\alpha} = \overline{\alpha}$.

Сідлова точка – це пара чистих стратегій $(i_{\text{про}}, j_0)$ першого і другого гравців відповідно, за яких досягають рівності $\underline{\alpha} = \overline{\alpha}$. Це поняття має такий зміст: якщо один з гравців дотримувався стратегії, що відповідає сідловій точці, то інший гравець не зможе зробити краще, ніж дотримуватись стратегії, що відповідає сідловій точці.

Математично це можна записати й так:

$$a_{ij_0} \le a_{i_0 j_0} \le a_{i_0 j}, \qquad (10.3)$$

де i, j – будь-які чисті стратегії першого і другого гравців відповідно; $(i_{\text{про}}, j_0)$ – стратегії, що мають сідлову точку.

У такий спосіб, виходячи з (10.3), сідловий елемент $a_{i_0 j_0}$ є мінімальним у i_0-му рядку і максимальним у j_0-му стовпчику матриці A. Відшукання сідлової точки матриці A виконуємо так: у матриці A послідовно в кожному рядку знаходимо мінімальний елемент і перевіряємо, чи є цей елемент максимальним у своєму стовпчику. Якщо так, то він і є сідловим

елементом, а пара стратегій, що йому відповідає, утворює сідлову точку. Пару чистих стратегій ($i_{\text{про}}$, $j_{\text{о}}$) першого і другого гравців, що утворює сідлову точку і сідловий елемент $a_{i_{\text{о}}j_{\text{о}}}$, називають *розв'язком гри*. При цьому $i_{\text{про}}$ і $j_{\text{про}}$ називають *оптимальними чистими стратегіями* для першого і другого гравців відповідно.

Приклад 10.1.

$$\min_j a_{ij}$$

$$A=\begin{pmatrix}1 & -3 & -2\\ 0 & 5 & 4\\ 2 & 3 & 2\end{pmatrix}\begin{matrix}-3\\ 0\\ 2\end{matrix}\quad \boxed{\max_i \min_j a_{ij}} = 2.$$

$$\max_i a_{ij} = \underbrace{2 \quad 5 \quad 4}$$

$$\min_j \max_i a_{ij} = 2$$

Сідловою точкою є пара ($i_{\text{про}} = 3; j_{\text{про}} = 1$), при якій $v = \underline{\alpha} = \overline{\alpha} = 2$.

Зазначимо, що хоча виграш у ситуації (3, 3) також дорівнює $2 = \underline{\alpha} = \overline{\alpha}$, ця точка не є сідловою, тому що цей виграш не є максимальним серед виграшів третього стовпчика.

Приклад 10.2.

$$\min_j a_{ij}$$

$$A=\begin{pmatrix}10 & 30\\ 40 & 20\end{pmatrix}\begin{matrix}\to 10\\ \to 20\end{matrix}\quad \max_i \min_j a_{ij} = 20.$$

$$\max_i a_{ij} \quad \downarrow \quad \downarrow$$

$$\underbrace{40 \quad 30}$$

$$\min_j \max_i a_{ij} = 30$$

З аналізу матриці виграшів бачимо, що $\underline{\alpha} < \overline{\alpha}$, тобто дана матриця не має сідлової точки. Якщо перший гравець вибирає свою чисту стратегію $i = 2$, то другий гравець, вибравши свою мінімаксну $j = 2$, програє тільки 20. У цьому випадку першому гравцю вигідно вибрати стратегію $j = 1$, тобто відхилитися від своєї чистої максимінної стратегії і виграти 30. Тоді другому гравцю буде вигідно вибрати стратегію $j = 1$, тобто відхилитися від своєї чистої мінімаксної стратегії і програти 10. У свою чергу перший гравець має вибрати свою 2-гу стратегію, щоб виграти 40, а другий відповісти вибором 2-ї стратегії і т. д.

Дослідження в матричній грі починають зі знаходження її сідлової точки в чистих стратегіях. Якщо матрична гра має сідлову точку в чистих стратегіях, то знаходженням цієї сідлової точки закінчується дослідження

гри. Якщо ж у грі немає сідлової точки, тоді слід знайти нижню й верхню чисті ціни цієї гри, які вказують, що перший гравець не може сподіватись на виграш, більший за верхню ціну гри, і може бути впевненим в одержанні виграшу, не меншого за нижню ціну гри. Поліпшення рішень матричних ігор варто шукати у використанні таємності застосування чистих стратегій і можливості багатократного повторення ігор у вигляді партії. Цього результату досягають шляхом застосування чистих стратегій випадково, із визначеною ймовірністю.

Змішаною стратегією гравця називають повний набір імовірностей застосування його чистих стратегій.

Таким чином, якщо перший гравець має m чистих стратегій 1, 2, ..., m, то його змішана стратегія x – це набір чисел $x = (x_1, ..., x_m)$, які задовольняють відношенню

$$x_i \ge 0 \ \left(i = \overline{1,m}\right), \ \sum_{i=1}^{m} x_i = 1.$$

Аналогічно для другого гравця, який має n чистих стратегій, змішана стратегія y – це набір чисел:

$$y = \left(y_1, \ldots, y_n\right), \ y_j \ge 0, \left(j = \overline{1,n}\right), \ \sum_{j=1}^{n} y_j = 1.$$

Через те що застосування гравцем однієї чистої стратегії виключає застосування іншої, чисті стратегії є *несумісними* подіями, до того ж вони є єдиними можливими подіями.

Чиста стратегія є окремим випадком змішаної стратегії. Дійсно, якщо в змішаній стратегії якусь i-ту чисту стратегію застосовують з імовірністю 1, то всі інші чисті стратегії не розглядають. Ця i-та чиста стратегія є окремим випадком змішаної стратегії. Для збереження таємності кожен гравець має вибирати свої стратегії, незалежно від вибору іншого гравця.

Середній виграш першого гравця у матричній грі з матрицею A виражають у вигляді математичного очікування його виграшів:

$$E\left(A, x, y\right) = \sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij} x_i y_j = xAy.$$

Перший гравець має на меті за рахунок зміни своїх змішаних стратегій x максимально збільшити свій середній виграш $E(A, x, y)$, а другий – за рахунок своїх змішаних стратегій зробити $E(A, x, y)$ мінімальним, тобто для розв'язання гри необхідно знайти такі x і y, за яких досягають верхньої ціни гри:

$$\overline{\alpha} = \min_y \max_x E\left(A, x, y\right).$$

Аналогічною має бути ситуація і для другого гравця, тобто нижня ціна гри є такою:

$$\underline{\alpha} = \max_x \min_y E\left(A, x, y\right).$$

Подібно до ігор, які мають сідлові точки в чистих стратегіях, дамо визначення: *оптимальними змішаними стратегіями* першого і другого гравців називають такі набори x°, y° відповідно, які задовольняють рівності

$$\min_y \max_x E(A,x,y) = \max_x \min_y E(A,x,y) = E\left(A,x^\circ,y^\circ\right).$$

Величину $E(A, x^\circ, y^\circ)$ називають *ціною гри* і позначають через v.

Існує й інше визначення оптимальних змішаних стратегій: x°, y° – оптимальні змішані стратегії першого і другого гравців відповідно, якщо вони утворюють сідлову точку

$$E\left(A,x,y^\circ\right) \le E\left(A,x^\circ,y^\circ\right) \le E\left(A,x^\circ,y\right).$$

Оптимальні змішані стратегії та ціну гри називають *розв'язком матричної гри*.

Відома основна теорема матричних ігор [83]:

Теорема 10.1 (про мінімакс). Для матричної гри з будь-якою матрицею A величини $\underline{\alpha} = \max_x \min_y E(A,x,y)$ і $\overline{\alpha} = \min_y \max_x (A,x,y)$ існують і є рівними.

Для знаходження розв'язку в матричних іграх використовують різні математичні підходи. Одним з них є використання добре розробленої методики розв'язання задач лінійного програмування.

Припустимо, що ціна гри додатна (v > 0). Якщо це не так, то завжди можна підібрати таке число, додавши яке до всіх елементів матриці виграшів, отримаємо матрицю з додатними елементами, і отже, із додатним значенням ціни гри. При цьому оптимальні змішані стратегії обох гравців не змінюються.

Отже, нехай дано матричну гру з матрицею A порядку $m \times n$. Оптимальні змішані стратегії $x = (x_1, ..., x_m)$, $y = (y_1, ..., y_n)$ першого і другого гравців відповідно і ціна гри v мають задовольняти співвідношенням:

$$\begin{cases} \sum_{i=1}^{m} a_{ij} x_i \ge \mathrm{v} \quad \left(j=\overline{1,n}\right); \\ \sum_{i=1}^{m} x_i = 1; \\ x_i \ge 0 \quad \left(i=\overline{1,m}\right), \end{cases} \tag{10.4}$$

$$\begin{cases} \sum_{j=1}^{n} a_{ij} y_j \le \mathrm{v} \quad \left(i=\overline{1,m}\right); \\ \sum_{j=1}^{n} y_j = 1; \\ y_j \ge 0 \quad \left(j=\overline{1,n}\right). \end{cases} \tag{10.5}$$

Розділимо всі рівняння та нерівності в (10.4) і (10.5) на v (це можна зробити, тому що за припущенням v > 0) і введемо позначення:

$$\frac{x_i}{\mathrm{v}} = p_i \ \ (i=\overline{1,m}), \ \frac{y_i}{\mathrm{v}} = q_i \ \ \left(j=\overline{1,n}\right).$$

Тоді (10.4) і (10.5) можна переписати у вигляді:

$$\sum_{i=1}^{m} a_{ij} p_i \ge 1, \ \sum_{i=1}^{m} p_i = \frac{1}{\mathrm{v}}, \ p_i \ge 0 \ \ \left(i=\overline{1,m}\right),$$

$$\sum_{j=1}^{n} a_{ij} q_j \le 1, \ \sum_{j=1}^{n} q_j = \frac{1}{\mathrm{v}}, \ q_j \ge 0 \ \ \left(j=\overline{1,n}\right).$$

Оскільки перший гравець прагне знайти такі значення x_i, а отже, і p_i, щоб ціна гри v була максимальною, то розв'язок першого завдання зводиться до знаходження таких невід'ємних значень $p_i\left(i=\overline{1,m}\right)$, за яких

$$\sum_{i=1}^{m} p_i \to \min, \ \sum_{i=1}^{m} a_{ij} p_i \ge 1. \tag{10.6}$$

Оскільки другий гравець прагне знайти такі значення y_j і, отже, q_j, щоб ціна гри v була найменшою, то розв'язок другого завдання зводиться до знаходження таких невід'ємних значень $q_j\left(j=\overline{1,n}\right)$, за яких

$$\sum_{j=1}^{n} q_j \to \max, \ \sum_{j=1}^{n} a_{ij} q_j \le 1. \tag{10.7}$$

Формули (10.6) і (10.7) виражають двоїсті одна одній задачі лінійного програмування (ЛП).

Вирішивши їх, отримаємо значення $p_i\left(i=\overline{1,m}\right)$, $q_j\left(j=\overline{1,n}\right)$ і v. Тоді змішані стратегії, тобто x_i і y_j обчислюють за формулами

$$x_i = \mathrm{v} p_i \quad \left(i=\overline{1,m}\right);$$

$$y_j = \mathrm{v} q_j \quad \left(j=\overline{1,n}\right).$$

Природним узагальненням матричних ігор є нескінченні антагоністичні ігри, у яких хоча б один із гравців має нескінченну кількість можливих стратегій. Формалізуючи реальну ситуацію з нескінченною кількістю варіантів вибору, можна кожну стратегію зіставити з визначеним числом з одиничного інтервалу, тому що завжди можна простим перетворенням будь-який інтервал перевести в одиничний і навпаки.

Велике значення в теорії нескінченних антагоністичних ігор має вигляд функції виграшів $M(x,y)$. На відміну від матричних ігор, не для кожної функції $M(x,y)$ існує розв'язок. Вважатимемо, що вибір визначе-

ного числа гравцем означає застосування його чистої стратегії, яка відповідає цьому числу. За аналогією з матричними іграми *чистою нижньою ціною* гри називають величину

$$V_1 = \max_x \inf_y M(x,y), \text{ або } V_1 = \max_x \min_y M(x,y),$$

а *чистою верхньою ціною* гри називають величину

$$V_2 = \min_y \sup_x M(x,y), \text{ або } V_2 = \min_y \max_x M(x,y).$$

Природно вважати, що, якщо для деякої нескінченної гри величини V_1 і V_2 існують і рівні між собою ($V_1 = V_2 = V$), то така гра має розв'язок в чистих стратегіях, тобто оптимальною стратегією першого гравця є вибір числа $x_0 \in X$ і другого гравця – числа $y_0 \in Y$, за яких $M(x_0,y_0) = V$. У цьому разі V називають ціною гри, а (x_0,y_0) – сідловою точкою в чистих стратегіях.

Теорема 10.2 (існування). Будь-яка антагоністична нескінченна гра двох гравців G з неперервною функцією виграшів $M(x,y)$ на одиничному квадраті має розв'язок (гравці мають оптимальні змішані стратегії).

Ігри з опуклими неперервними функціями виграшів називають *опуклими*.

Нагадаємо, що *опуклою* функцією f дійсної змінної x на інтервалі (a,b) називають таку функцію, для якої виконується нерівність

$$f(\alpha_1 x_1 + \alpha_2 x_2) \le \alpha_1 f(x_1) + \alpha_2 f(x_2),$$

де x_1 і x_2 – будь-які дві точки з інтервалу (a,b); $\alpha_1, \alpha_2 \ge 0$, причому $\alpha_1 + \alpha_2 = 1$.

Якщо для $\alpha_1 \ne 0$, $\alpha_2 \ne 0$, завжди є строга нерівність

$$f(\alpha_1 x_1 + \alpha_2 x_2) < \alpha_1 f(x_1) + \alpha_2 f(x_2),$$

то функцію f називають *строго опуклою* на (a,b). Геометрично опукла функція зображає дугу, графік якої розташований нижче її хорди стягування (рис. 10.1).

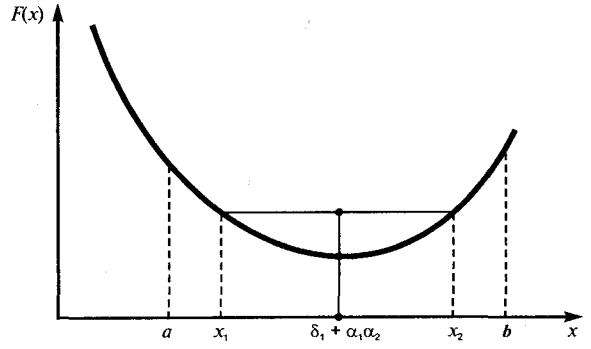


Рис. 10.1. Геометричний зміст строго опуклої функції

Неперервна і строго опукла функція f на замкненому інтервалі набуває мінімального значення тільки в одній точці інтервалу.

Теорема 10.3. Нехай $M(x,y)$ – неперервна функція виграшів першого гравця на одиничному квадраті і *строго опукла* відносно y для будь-якого x. Тоді існує єдина оптимальна чиста стратегія $y = y_0 \in [0;1]$ для другого гравця, ціну гри визначають формулою

$$V = \min_y \max_x M(x,y); \tag{10.8}$$

значення y_0 визначають як розв'язок рівняння

$$\max_x M(x,y_0) = V. \tag{10.9}$$

Таким чином, якщо $M(x,y)$ неперервна і опукла по y, то ціну гри визначають за формулою (10.8), і другий гравець має оптимальну чисту стратегію, визначену рівнянням (10.9).

Аналогічно і для першого гравця. Ціну гри визначають формулою

$$V = \max_x \min_y M(x,y), \tag{10.10}$$

а чисту оптимальну стратегію x_0 першого гравця визначають з рівняння

$$\min_y M(x_0,y) = V. \tag{10.11}$$

Антагоністичні ігри, що ми тільки-но розглянули, описують конфлікти часткового характеру. Для переважної більшості реальних конфліктів антагоністичні ігри або зовсім не можна вважати прийнятними, адекватними моделями, або, у ліпшому випадку, можна розглядати як перші грубі наближення.

По-перше, антагоністичні ігри не можуть описати конфлікти з числом рядків більшим ніж два, хоч такі багатосторонні конфлікти не тільки трапляються у дійсності, але є принципово складнішими, ніж конфлікти з двома учасниками, і взагалі не зводяться до останніх.

По-друге, навіть у конфліктах із двома учасниками інтереси сторін зовсім не мають бути протилежними; у багатьох конфліктах такого роду стає так, що одна із ситуацій виявляється ліпшою за іншу для обох учасників.

По-третє, навіть якщо деякі дві ситуації гравці порівнюють за перевагою у протилежний спосіб, різниця в оцінках цієї переваги залишає місце для компромісів та кооперацій.

Нарешті, по-четверте, гострота конфлікту за змістом не обов'язково відповідає його формальній антагоністичності.

Для усунення цих вад можна використовувати більш складні моделі ігор [83]. Як приклад розглянемо ігри двох учасників з довільною сумою та кооперативні ігри.

Ігри двох учасників з довільною сумою. У скінченній безкоаліційній грі двох гравців кожен з них робить один хід – обирає одну стратегію із скінченної кількості стратегій, і після цього він одержує свій виграш відповідно до визначених для кожного з них матриць виграшів. Інакше кажучи, таку гру цілком визначають двома матрицями виграшів для двох гравців. Тому такі ігри називають біматричними. Нехай у першого гравця є m стратегій, $i=\overline{1,m}$, у другого гравця є n стратегій, $j=\overline{1,n}$. Виграші першого і другого гравців відповідно задають матрицями

$$A=\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{m1} & \dots & a_{mn} \end{pmatrix},\ B=\begin{pmatrix} b_{11} & \dots & b_{1n} \\ \dots & \dots & \dots \\ b_{m1} & \dots & b_{mn} \end{pmatrix}.$$

Як і раніше, повний набір імовірностей $x = (x_1, \dots, x_m)$ застосування першим гравцем своїх чистих стратегій вважатимемо його змішаною стратегією, і $y = (y_1, \dots, y_n)$ – змішаною стратегією другого гравця; тоді середні виграші першого і другого гравців відповідно рівні

$$\begin{cases} E(A,x,y)=\sum\limits_{i=1}^{m}\sum\limits_{j=1}^{n}a_{ij}x_iy_j=xAy^T; \\ E(B,x,y)=\sum\limits_{i=1}^{m}\sum\limits_{j=1}^{n}b_{ij}x_iy_j=xBy^T. \end{cases}$$

Ситуацію рівноваги для біматричної гри складають пари (x, y) таких змішаних стратегій першого і другого гравців, які задовольняють нерівностям:

$$\begin{cases} \sum\limits_{j=1}^{n}a_{ij}y_j \le \sum\limits_{i=1}^{m}\sum\limits_{j=1}^{n}a_{ij}x_iy_j \quad \left(i=\overline{1,m}\right); \\ \sum\limits_{i=1}^{m}b_{ij}x_i \le \sum\limits_{i=1}^{m}\sum\limits_{j=1}^{n}b_{ij}x_iy_j \quad \left(j=\overline{1,n}\right) \end{cases}$$

або

$$\begin{cases} Ay^T \le xAy^T \quad \left(=E_1(A,x,y)\right); \\ xB \le xBy^T \quad \left(=E_2(B,x,y)\right). \end{cases}$$

Для визначення ситуацій рівноваги слід розв'язати ці системи нерівностей щодо невідомих $x = (x_1, \dots, x_m)$ і $y = (y_1, \dots, y_n)$ за умов

$$\sum_{i=1}^{m}x_i=1,\ \sum_{j=1}^{n}y_j=1,\ x_i\ge 0\ \left(i=\overline{1,m}\right),\ y_j\ge 0\ \left(j=\overline{1,n}\right).$$

Існування такої ситуації рівноваги визначає така теорема [48].

Теорема 10.4 (Неша). Кожна біматрична гра має принаймні одну ситуацію рівноваги.

Кооперативні ігри утворюють у тих випадках, коли у грі n гравцям дозволено утворювати визначені коаліції. Нехай N – множина всіх гравців, $N = \{1, 2, \dots, n\}$, а K – будь-яка її підмножина. Нехай гравці із K домовляються між собою про спільні дії і, у такий спосіб, утворюють одну коаліцію. Очевидно, що число таких коаліцій з r гравцями дорівнює C_n^r, а число всіх можливих коаліцій дорівнює

$$\sum_{r=1}^{n}C_n^r=2^n-1.$$

З цього випливає, що кількість усіх можливостей швидко зростає із збільшенням числа учасників даної гри. Для дослідження таких ігор слід врахувати всі можливі коаліції, і тому складність дослідження швидко зростає у разі збільшення n. Утворивши коаліцію, множина гравців K діє як один гравець проти інших, і виграш цієї коаліції залежить від вибору стратегії кожним із n гравців.

Функцію v, що ставить у відповідність кожній коаліції K найбільший напевно отримуваний виграш v(K), називають *характеристичною функцією гри*. Так, наприклад, для безкоаліційної гри n гравців v(K) існує, коли гравці з множини K оптимально діють як один гравець проти інших $N\backslash K$ гравців, які утворюють іншу коаліцію (другий гравець).

Характеристична функція v є *простою*, якщо вона набуває тільки двох значень: 0 і 1. Якщо характеристична функція v проста, то коаліції K, для яких v(K) = 1, називають *виграшними*, а коаліції K, для яких v(K) = 0,– *програшними*.

Якщо в простій характеристичній функції v *виграшними* є ті і тільки ті коаліції, що містять фіксовану не порожню коаліцію R, то характеристичну функцію v, що позначена в цьому разі через v_R, називають *найпростішою*.

Змістовно прості характеристичні функції виникають, наприклад, в умовах голосування, коли коаліція є виграшною, якщо вона збирає більше половини голосів (проста більшість) або не менше двох третин голосів (кваліфікована більшість). Складнішим є приклад оцінки результатів виборів у Раду безпеки ООН, де виграшними коаліціями є всі об'єднання, що складаються з усіх п'ятьох постійних членів Ради плюс ще хоча б один непостійний член, і тільки вони. Найпростіша характеристична функція з'являється, коли в голосуючому колективі є деяке «ядро», що голосує з дотриманням правила «вето», а голоси інших учасників виявляються несуттєвими.

Розподіл виграшів (поділ) гравців має задовольняти таким умовам: якщо позначити через x_i виграш i-го гравця, то, *по-перше*, має виконуватись умова *індивідуальної раціональності*

$$x_i \ge v(i),\ \text{для}\ i \in N, \qquad (10.12)$$

тобто будь-який гравець має одержати виграш у коаліції не менший, ніж він одержав, якби брав участь у ній (а якщо ні, то він не буде брати участі у коаліції); *по-друге*, має виконуватись умова *колективної раціональності*

$$\sum_{i \in N} x_i = \mathrm{v}(N), \qquad (10.13)$$

тобто сума виграшів гравців повинна відповідати можливостям (якщо сума виграшів усіх гравців менша, ніж v(N), то гравцям нема задля чого вступати в коаліцію; якщо ж вимагати, щоб сума виграшів була більша, ніж v(N), то це означає, що гравці мають поділити між собою суму більшу, ніж вони мають).

У такий спосіб, вектор $x = (x_1, ..., x_n)$, що задовольняє умовам індивідуальної і колективної раціональності, називають *поділом* в умовах характеристичної функції v.

Систему $\{N, \mathrm{v}\}$, яку складають множина гравців, характеристична функція над цією множиною і множина поділів, що задовольняють співвідношенням (10.12) і (10.13) в умовах характеристичної функції, називають *класичною кооперативною грою*.

У безкоаліційних іграх завершення є результатом дій тих гравців, які у цій ситуації одержують свої виграші. Результатом кооперативної гри є поділ, що виникає не як наслідок дії гравців, а як результат їх угод. Тому в кооперативних іграх порівнюють не ситуації, як це трапляється в безкоаліційних іграх, а поділи, і це порівняння має набагато складніший характер.

Кооперативну гру називають *нульовою*, якщо всі значення її характеристичної функції дорівнюють нулю. Істотне значення нульової гри полягає в тому, що в ній гравці не мають ніякої зацікавленості.

Довільна несуттєва гра стратегічно еквівалентна нульовій. Методику розв'язання таких ігор можна знайти в [48].

10.3. Методи розробки алгоритмів ігрових програм

Розглянемо другий напрям нашого дослідження – методи розробки алгоритмів ігрових програм (комп'ютерних ігор). У цьому разі учасник гри вибирає свої дії, виходячи із проведеного ним дослідження дерева майбутніх можливостей, спираючись на оцінку можливих у майбутньому ситуацій і на припущення відносно того, що робитимуть інші учасники гри.

Можна навести таку типову структуру ігрової програми [90]:

A. Людина задає довільну ігрову позицію. Перейти до кроку *B* чи *E* залежить від того, належить наступний крок машині чи людині.
B. Людина робить свій хід.
C. Породити нову позицію.
D. Якщо програма закінчена, надрукувати результат і зупинитись. В іншому разі перейти до кроку *E*.
E. Породити деякі або ж усі приймачі вихідної позиції машини.
F. Оцінити кожен приймач. Для деяких програм кожна оцінка потребує прискіпливого дослідження наступних можливостей.

G. Перейти в позицію приймача з найбільшою оцінкою.
H. Надрукувати хід, який робить машина.
I. Якщо гру закінчено, надрукувати результат і зупинитись. В іншому разі перейти до кроку *B*.

Зупинимось на деталізації розробки основних кроків цього алгоритму.

10.3.1. *Задання даних*

В останній час розроблено багато моделей задання даних для ігрових задач. Серед них можна виділити: семантичні мережі, фреймові моделі, логічні моделі та продукційні моделі [31]. Однією з найбільш використовуваних в теорії програмування «комп'ютерних ігор» є продукційна модель [90], в ній дані описують за допомогою правил, які мають структуру «якщо – тоді», тобто описують ситуацію «явище – реакція». Основними перевагами такого задання є простота доповнення, модифікацій і анулювання.

Продукційну систему складають три основні компоненти. Перша компонента – це набір правил, які використовують як базу знань, тому її ще називають базою правил. Наступною компонентою є робоча пам'ять, в якій зберігаються умови, що відносяться до конкретних задач предметної області, і результати виведень, отриманих на їх основі. Третя компонента є механізмом логічного виведення. Він використовує правила відповідно до робочої пам'яті. Загальну структуру системи зображено на рис. 10.2.

Розглянемо на прикладі як працює продукційна система. Припустимо, що дані, які записані у робочій пам'яті, мають вигляд зразків: «намір – хід», «хід – правило» і т. д. Правила, що накопичуються в базі правил, відображають зміст робочої пам'яті та вхідних даних. В їх умовній частині знаходяться або ж одиничні зразки, або декілька умов, об'єднаних сполучником «і», а заключна частина – зразки, які додатково реєструють в робочій пам'яті. Розглянемо два приклади подібних правил із гри в шахи.

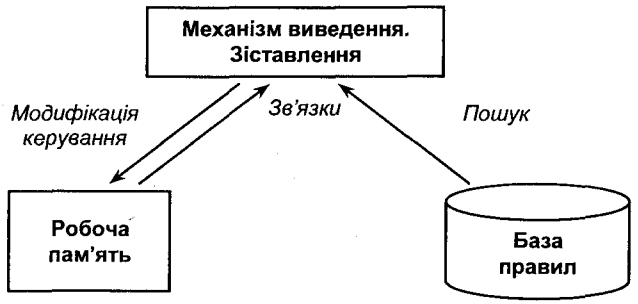


Рис. 10.2. Конфігурація продукційної системи

Правило 1.

ЯКЩО «кінь шахує короля» і «кінь напав на королеву», і «можна бити коня»,

ТОДІ «потрібно бити коня».

Правило 2.

ЯКЩО «кінь під боєм»,

ТОДІ «можна бити коня».

Нехай спочатку в робочу область заносять зразки «кінь шахує короля» і «кінь напав на королеву», а в базі правил знаходяться два наші правила та на вхід подають ситуацію «можна бити коня». Розглядаємо можливість застосування правил. Спочатку механізм виведення зіставляє зразки із умовної частини правила зі зразками, які зберігаються в робочій пам'яті. Якщо всі зразки містяться в робочій пам'яті, тоді умовну частину вважають істинною, а в іншому разі – хибною. Для нашого прикладу спочатку повністю виконуватиметься тільки умовна частина другого правила. Тому його використають, і в робочу пам'ять занесуть зразок «можна бити коня». Тільки після цього можна застосовувати перше правило і в робочу пам'ять занести зразок «потрібно бити коня».

У наведеному прикладі для отримання виведення здійснювали вибір заздалегідь записаного вмісту робочої пам'яті, застосували правила і доповнювали дані в пам'яті. Такі виведення називають прямими. Спосіб, в якому на основі фактів, що потребують підтвердження, щоб виступати в ролі виведення, досліджують можливість застосування правила, яке дає можливість підтвердження, називають оберненим виведенням.

У разі застосування оберненого виведення умова зупинки системи така: або досягають першочергової цілі, або закінчились правила, які застосовують для досягнення цілі в ході виведення.

Очевидно, що вибір чергового правила впливає на ефективність виведення. Якщо на кожному етапі логічного виведення існує кілька правил застосування, тоді ця множина правил носить назву *конфліктного набору*, а вибір одного з них називають *вирішенням конфлікту*.

Для багатьох практичних застосувань продукційних систем недостатньо запису в робочу пам'ять тільки одного зразка, і виникає потреба керувати даними, які уточнюють зміст. У таких випадках використовують спосіб задання конкретних даних за допомогою триплету: *об'єкт – атрибут – значення*. Тому кожну окрему субстанцію із предметної області розглядають як один об'єкт. Можна вважати, що дані, які зберігаються в робочій пам'яті, показують значення, що набувають атрибути цього об'єкта. Наприклад, дані (*кінь 1, колір чорний*) описують той факт, що існує деяка шахова фігура кінь, що має чорний колір.

Однією із переваг використання структури об'єкт – атрибут – значення є уточнення контексту, в якому застосовують правило. Так, правило, єдине для всіх об'єктів «кінь», має бути застосованим незалежно від того, йдеться про білого або чорного коня. Коли ж існує кілька об'єктів,

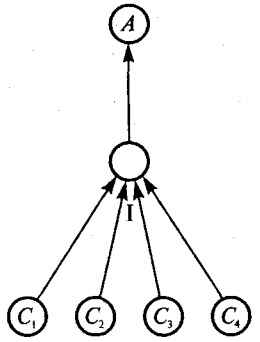


Рис. 10.3. Задання правила графом

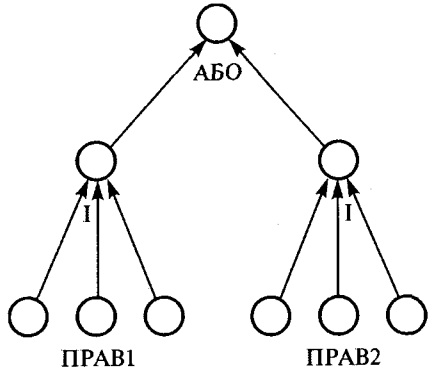


Рис. 10.4. Виведення, що роблять за допомогою кількох правил

для яких можна використати деяке правило, тоді його застосовують для кожного об'єкта не більше одного разу. Цю проблему називають проблемою контексту застосування правила, а об'єкт застосування правила – контекстом застосування правила.

На практиці умовна і заключна частина правил ускладнюються. Проте основу складає класичне правило, яке описує відношення виведення, встановлене між вмістом робочої пам'яті, звернення до якого приходить із умовної частини, і змістом, описаним в заключній частині.

Візуально таке відношення можна задати графом, який має структуру дерева (рис. 10.3).

Якщо існує кілька правил, з яких виводять один і той самий висновок, тоді, виконавши операцію АБО над усіма виведеннями, які можна отримати за допомогою цих правил, отримуємо граф І/АБО для всієї системи (рис. 10.4).

Згідно з таким заданням, обернене виведення можна трактувати як проблему пошуку шляху в даному графі. В іграх (шахи, шашки) проблема пошуку шляху показує, як може розвиватись ситуація. Тому ефективність виведення повністю залежить від ефективності пошуку шляхів на графі. Ширина графа – число етапів виведення і число правил, на основі яких роблять те саме виведення.

У реальних системах продукцій іноді буває необхідно працювати з *нечіткою інформацією*. У таких випадках виділяють дві категорії нечіткості:

- нечіткість безпосередніх даних;
- нечіткість виведення.

Коли виведення роблять за допомогою кількох правил, серед яких є і нечіткі, виникає проблема визначення ступеня нечіткості всього виведення.

Розглянемо випадок [90], коли одне виведення C роблять за допомогою множини правил $R_1 - R_n$ (рис. 10.5).

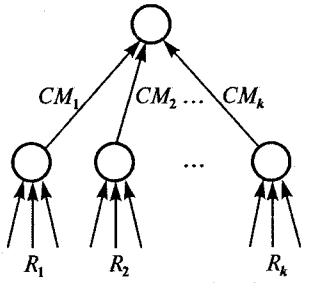


Рис. 10.5. Синтез вірогідності під час роботи з нечіткими даними

Припустимо, що нечіткість виведення C характеризується коефіцієнтом впевненості, заданим числовим значенням. Кожен коефіцієнт впевненості виведення C для кожного правила позначимо відповідно $CM_1 - CM_n$. Як метод визначення коефіцієнта впевненості CM_t розглянемо функцію синтезу:

$$CM_t = КОМБ(CM_1, ..., CM_n).$$

Спосіб пошуку залежить від вигляду *КОМБ*. Взагалі вважають, якщо хоча б одна з CM_t заперечує нечіткість, тоді і CM_t заперечує нечіткість, тому коли таке правило існує, потрібно спробувати використати його раніше за інші. Фактично, якщо коефіцієнт впевненості заперечує нечіткість, пошук можна закінчувати, в іншому разі аналізують *КОМБ*.

Як зазначалося в продукційній моделі, існує пряме й обернене виведення.

У прямому виведенні повторюється цикл пошуку даних у робочій пам'яті, генерація конфліктного набору, розв'язання конфлікту і застосування правил. Використовують основні методи:

1) генерація низки обмежень на породження конфліктного набору;
2) визначення більш строгого алгоритму розв'язання конфлікту.

Згідно з першим методом, є два підходи:

- правила розділяють на окремі категорії, і у визначених ситуаціях аналізують виконання правил певної категорії;
- пошук умовної частини правил визначеної категорії не проводять.

В останньому підході реалізується використання метаправил – правил, в умовній частині яких знаходяться умови, що відносять до змісту правил і змісту пам'яті, або ж правил, в умовній частині яких вказані атрибути, що визначають непотрібність пошуку.

Згідно з першим підходом, правила заздалегідь групують по атрибутах, для кожної групи вказують умову (форма якої аналогічна умовній частині правила) і, аналізуючи стан робочої пам'яті, проводять пошук застосування правил тільки з цієї групи.

Якщо у разі нашого виведення всі висновки, які можна визначити, знаходять, тоді розв'язання конфлікту проводять просто:

- правила застосовують за порядком їх визначення;
- у разі, коли задано умову зупинки, її можна прискорити, застосувавши виведення за пріоритетом глибини, що відповідає спробі пріоритетного застосування правила зі зв'язком на останню інформацію, якою було доповнено робочу область. За даного способу шлях логічного виведення – мінімальний. Можна використати й інші пріоритети, наприклад, правила з більшим числом підумов в умовній частині.

Все наведене стосується і оберненого виведення. Крім того, додають ще одну задачу, пов'язану з послідовністю оцінки умов в умовній частині правил.

10.3.2. *Дерева гри*

Із попереднього параграфа видно, що важливою частиною більшості ігрових програм є процедура аналізу «дерева логічних можливостей».

Існує два типи дерев [90]: дерева гри і дерева цілей. Гілки в дереві гри задають можливі ходи, ходи у відповідь і т. д. Дерево цілі показує, що деякої початкової цілі можна досягти, коли буде досягнуто певних підцілей; у свою чергу аналогічно перевіряють досягнення підцілей. Оскільки дерева мають тенденцію сильно розростатись, виникає потреба ефективного виділення істотних частин дерева. Дерево гри може бути експліцитним та імпліцитним. Експліцитне дерево гри задають у явному вигляді, імпліцитне ж дерево задають виділенням початкової позиції і правил формування дерева.

Нас більше цікавитиме імпліцитне дерево. Правила формування такого дерева визначають *критерій закінчення* і вказують, як формувати приймач для кожної позиції. У позиції, що задовольняє критерію закінчення, приймач відсутній. Процедуру, яка перетворює імпліцитне дерево в експліцитне, називають *породжувальною процедурою*. Найбільш використовуваними є дві породжувальні процедури: «*спочатку в ширину*» та «*спочатку в глибину*». Нагадаємо, що перша процедура породжує спочатку всі вершини першого рівня, потім другого, потім третього і т. д. Друга процедура породжує дерево, починаючи з лівого боку. Спочатку породжується перший (лівий) приймач початкової позиції, потім його перший (лівий) приймач і т. д.

Отже, ми приходимо до того, що ключовим для позиційних ігор є поняття «***дерева гри***». Корінь цього дерева збігається з початковою позицією. Кожен вузол цього дерева характеризується номером гравця, що має робити хід. Дуги відповідають ходам, тобто, якщо в позиції 1 можливий хід, який переводить позицію 1 в позицію 2, то з позиції 1 до позиції 2 йде орієнтована дуга, яка відповідає цьому ходу. Право вибору ходу в позиції 2 належить, звичайно, уже іншому гравцеві. Кожен вузол дерева корисно характеризувати також його ***рівнем***, тобто відстанню від кореня. Якщо на k-му рівні право вибору ходу належить одному з гравців, то на $(k + 1)$-му рівні воно переходить до його суперника.

Нехай грають два гравці – A і B. Для визначеності вважатимемо, що право вибору ходу в початковій позиції належить гравцеві A. Функція виграшу збігається з функцією виграшу цього гравця, тобто гравець A аналізує дерево гри зі свого погляду та прагне максимізувати виграш. Вершини, в яких право ходу належить гравцеві A, прийнято називати ***α-вершинами***; вершини ж, у яких право ходу належить його суперникові, називають ***β-вершинами.***

Теорія стверджує, що антагоністичні детерміновані позиційні ігри з повною інформацією (такі, як шахи), дерево гри яких має скінченну кількість вершин, мають оптимальну стратегію, відступати від якої невигідно жодному із суперників. Інакше кажучи, ***у разі правильної гри обох сторін гра завжди має закінчуватися однаковим результатом.***

Для будь-якої гри, що завершується за скінченну кількість ходів, є теоретично можливим побудувати повне дерево гри, що охоплює всі можливі позиції. Почнемо аналіз дерева з завершальних позицій (листків дерева). Розглянемо довільні завершальні позиції, які мають одного батька. Нехай для визначеності це α-вершини. Кожна завершальна позиція має оцінку, що збігається з функцією виграшу та є результатом гри. Ясно, що гравець *B*, якому належить право вибору ходу у батьківській вершині, вибере хід, що мінімізує цю оцінку (мінімізує його програш). Ця мінімальна оцінка передається даній вершині знизу.

Нехай всі наступники довільної β-вершини уже проаналізовані та кожному передано знизу певну оцінку. Загальне правило можна сформулювати у такий спосіб: *з β-вершини має бути зроблений хід, що веде до позиції-наступника з найменшою оцінкою.*

За допомогою аналогічних міркувань можна встановити правило, згідно з яким повинен вибирати ходи гравець *A*: *з α-вершини має бути зроблений хід, що веде до позиції-наступника з найбільшою оцінкою.*

Такий аналіз можна зробити для будь-якої позиції, включаючи початкову. Ми *конструктивно* довели, що для будь-якої позиції результат гри є повністю визначеним, тобто сформулювали правила, які визначають процедуру, що дає змогу досягти цього результату і знайти стратегію гри, оптимальну для кожного гравця. Ця процедура носить назву ***мінімаксної процедури***.

Проведіть аналогію з процедурою досягнення результату в матричних іграх.

Оцінки позицій, передані знизу, називатимемо ***мінімаксними оцінками***. Зрозуміло, що за правильної гри обох сторін вони збігаються із залишковими виграшами гравців.

Мінімаксна процедура є процедурою повного перебору, оскільки у разі її застосування слід породити, починаючи з вершини, все дерево гри, а потім отримати оцінки, рухаючись за оберненим порядком. Для практичної реалізації мінімаксної процедури застосовують, як правило, бектрекінгові алгоритми, хоча теоретично ніщо не заважає застосувати і перебір в ширину. Застосування мінімаксної процедури [31] ілюструє рис. 10.6. Жирним виділено варіант, оптимальний для обох гравців. Оцінки всіх вершин, крім завершальної, є мінімаксними.

Для таких ігор, як хрестики-нулики, побудова дерева гри та реалізація мінімаксної процедури не викликає особливих проблем. Для складніших ігор розв'язок можна знайти тільки теоретично (повний перебір).

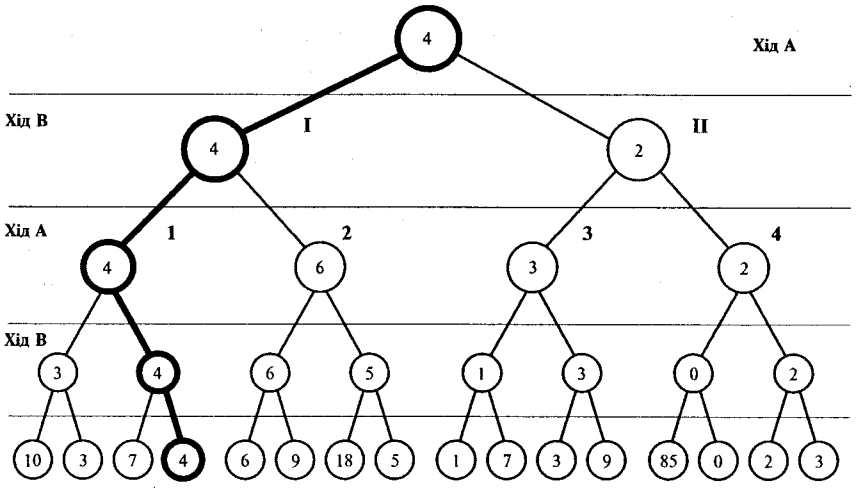


Рис. 10.6. Мінімаксна процедура вибору ходу

Таким чином, за практичної реалізації ключовою слід вважати проблему *ефективного скорочення перебору.*

Звичайним підходом є обмеження *глибини перебору*, тобто пошук реалізації мінімаксної процедури для дерева гри з обмеженою кількістю ходів. За невеликої глибини (5–10 напівходів) дерево перебору скорочується до розмірів, що є прийнятними для перебору на сучасних комп'ютерах.

10.3.3. *Оцінка позиції*

Опишемо процедуру, необхідну для виконання кроку *F* побудови оцінки позиції загального алгоритму створення ігрових програм.

Процедура оцінки позиції *n* складається з двох кроків, що виконують послідовно у разі, коли не зроблено спеціальних зауважень.

A. *Якщо критерій закінчення для оцінки позиції справджується, використати статичну оціночну функцію. В іншому разі перейти до кроку B.*

B. *Оцінити деякі або ж всі приймачі позиції.*

Щоб оцінити кожен приймач, використовують вищезазначену двокрокову процедуру. Процедура рекурсивна. Оцінка позиції формується по оцінках приймачів, для формування оцінок яких використовують мінімаксні процедури.

Процедура *m^n*, запропонована Слейглом і Діксоном [90], є ще одним варіантом процедури формування робочої оцінки. Щоб отримати робочу оцінку макс-позиції, процедура *m^n* знаходить оцінку, яка залежить від *m* найліпших приймачів макс-позиції. Ідея полягає в тому, що якщо інші приймачі є такими ж або майже такими ж добрими, як і вибраний найліпший приймач, тоді до мінімаксної оцінки додають відносно велику величину, а якщо всі інші приймачі значно гірші, ніж найліпший приймач,

тоді до мінімаксної оцінки роблять маленьку добавку або нічого не додають. Інакше кажучи, за заданої оцінки найліпшого приймача процедура вважає, що ліпше мати кілька добрих приймачів на противагу одному. Щоб отримати робочу оцінку мін-позиції, процедура знаходить оцінку, яка залежить від n найліпших приймачів мін-позиції.

Основна проблема, пов'язана з обмеженням глибини перебору, полягає у тому, що для позицій, які знаходяться на максимальній глибині та на яких завершується аналіз варіантів з поточної вершини, оцінку позиції ще не визначено. У таких випадках замість остаточної оцінки використовують ***статичні оціночні функції***.

Статична функція оцінювання є функцією, що дає змогу здійснювати оцінку ігрової позиції без породження якихось приймачів. Найбільш використовуваною є функція присвоєння відносно високих оцінок позиціям, що є добрими з погляду обчислювальної машини. Наприклад, функція оцінювання в шашках може мати вигляд: $6k + 4m + u$, де k – перевага в дамках, m – в шашках, u – беззаперечна перевага в рухливості (сумарна кількість ходів, які не ведуть до прямого взяття фігур).

Статичною оціночною функцією називають числову функцію, яка призначена для оцінки виграшу гравця у поданій позиції та залежить виключно від цієї позиції.

Це визначення є простим та інтуїтивно зрозумілим. Проте воно не зовсім чітке, оскільки і мінімаксну оцінку позиції врешті-решт однозначно визначає ця позиція. Справа в тому, що статичні оцінки, на відміну від мінімаксних, не враховують *динаміки* позиції, тобто можливостей її зміни; саме тому їх і називають статичними.

Більш чітке визначення можна дати [31], якщо ввести до розгляду деяку ***формальну модель гри Олецького.***

Формальною моделлю гри називатимемо п'ятірку $M = \langle S, S_0, P, Q, F \rangle$, де S – множина всіх можливих позицій; S_0 – початкова позиція; P – множина правил зумовлення можливості переходу від однієї позиції до іншої, інакше кажучи,– множина припустимих ходів; Q – множина завершальних позицій; F – функція, яку визначено на завершальних позиціях, визначає виграші гравців (для антагоністичної гри достатньо задати функцію виграшів для одного з гравців).

Тепер можна дати визначення мінімаксної та статичної оціночних функцій.

Мінімаксною оціночною функцією називатимемо функцію f: $M \to R$, яку визначено для будь-якої позиції та яку обчислюють на основі описаної раніше мінімаксної процедури; значення функції є мінімаксними оцінками.

Статичною оціночною функцією називатимемо функцію f: $S \to R$, яку визначено для будь-якої позиції та яку обчислюють на основі власних чинників цієї позиції; значення функції називають статичними оцінками.

Ясно, що за реалізації мінімаксної процедури оцінка позиції, що передана знизу, зовсім не зобов'язана збігатися з її власною статичною

оцінкою (і, як правило, не збігається). Зрозуміло, що якби передані оцінки повністю збігалися зі статичними, така статична оціночна функція однозначно визначала б виграші гравців, і жодні мінімаксні процедури не були б потрібні.

Очевидно, що статичні оцінки позицій, які знаходяться на горизонті, мають бути певним чином пов'язані з мінімаксними оцінками цих позицій, отриманими для повного дерева (невідповідності істотно впливають на якість гри).

Часто використовують *лінійні* статичні оціночні функції, які обчислюють за формулою

$$S(P) = \sum_{i=1}^{n} a_i x_i,$$

де x_i, $i = 1, \ldots, n$ – числові значення факторів, що впливають на оцінку позиції, a_i, $i = 1, \ldots, n$ – вагові коефіцієнти визначення міри важливості того чи іншого фактора.

Так, під час побудови шахових програм загальноприйнятим є аналіз матеріального співвідношення; пішак оцінюють одним очком, легку фігуру – трьома, тура має коефіцієнт 5 або 5,5; ферзь – 9 або 10 (ферзь слабший за дві тури). Втрата короля рівнозначна програшу, і тому його оцінюють великим числом (наприклад, 200). Певним чином оцінюють також позиційні фактори (активність фігур, пішакову структуру тощо).

Важливо розуміти, що скорочене дерево гри залежить від поточної позиції, для позиції-наступника воно зміниться.

Побудуйте статичні оціночні функції для відомих вам ігор.

Мінімаксна процедура, застосована до скороченого дерева, дає змогу знайти оптимальну стратегію лише для цього скороченого дерева, а не для всієї гри. Програма аналізує лише варіанти, що не виходять за «*горизонт*» (тобто, які не перевищують максимальної глибини). Наприклад, у шахах цілком імовірна ситуація, коли з «*оптимального*» вузла програма зразу отримує мат. Якби глибина перебору була більшою, вона б вибрала зовсім інший варіант. Тому глибина перебору є одним з вирішальних чинників, що визначає силу ігрових програм та якість їх гри.

Слід зазначити, що варіант може обірватися і до завершення максимальної глибини через остаточний виграш (мат) або теоретичну нічию, це можна легко обрахувати.

10.3.4. *Процедури пошуку*

Процедури пошуку утворюють певною комбінацією поєднання статичної оціночної і породжувальної процедур. Можна виділити головні типи таких процедур:

1) *мінімаксна процедура пошуку в глибину*;

2) α-β-*процедура*.

Мінімаксна процедура «спочатку в глибину» є процедурою пошуку, яка спільно використовує оціночну функцію, що породжує процедуру «спочатку в глибину» і мінімаксну процедуру формування робочої оцінки. Вважають, що дерево гри задається імпліцитно. Далі можна застосувати всі наші роздуми щодо процедури пошуку на графі, які описано в розділі 3.

Іноді α-β-процедуру ще називають «процедурою з обмеженим рухом в оберненому напрямі». Вона подібна до мінімаксної процедури «спочатку в глибину», але містить модифіковану процедуру породження «спочатку в глибину». Ці дві процедури еквівалентні в тому значенні, що за однакових початкових позицій, критеріїв закінчення і оціночних функцій обидві вони виберуть один хід. Перед тим як вибрати свій хід, α-β-процедура майже завжди породжує значно меншу частину дерева порівняно з опоненткою.

Проілюструємо суть α-β-відтинання на прикладі. Нехай гравець *A* робить аналіз дерева гри із рис. 10.6, починаючи з кореня. Він має зробити вибір між ходами, позначеними I та II. З позиції, що утворюється після ходу 1, ведуть гілки, позначені цифрами 1 та 2, а з II – гілки 3 та 4.

Нехай уже проаналізовано гілку 1. Вона дала оцінку 4. Тому з цього випливає, що гарантований виграш складає 4, тому варіанти, які не дадуть більшого виграшу, розглядати не варто. Так, якщо далі проаналізовано гілку 4, вона дає максимальну оцінку 2 для позиції II. Звідси випливає, що хід до позиції II не дасть виграшу, більшого ніж 2, всі інші гілки, що виходять з цієї позиції (у нашому випадку гілку 3) аналізувати не варто.

Інакше кажучи, якщо деякий хід досягає максимально можливого виграшу, інші його альтернативи не розглядають. Якщо для деякого ходу знайдене спростування, інші відповіді суперника також не розглядають.

Дамо більш формальний опис α-β-відтинання. З кожним вузлом пов'язують ***попередню оцінку***, що змінюється в ході аналізу його наступників. В α-вершині вона дорівнює $-\infty$ перед початком аналізу. Якщо остаточна оцінка вузла-наступника перевищує поточну попередню оцінку, цій попередній оцінці присвоюють нове значення.

Відтинання у α-вершині (припинення аналізу цієї вершини та її наступників) відбувається, коли попередня оцінка цієї вершини стає більшою, ніж попередня оцінка попередника (батька) цієї вершини.

Аналогічне правило відтинання у β-вершині сформулюйте самостійно.

Детальніший опис та можливі алгоритмізації мінімаксної процедури і α-β-відтинання можна знайти в багатьох джерелах, наприклад [93, 67, 78].

Ефективність скорочення перебору у разі α-β-відтинання істотно залежить від порядку перегляду варіантів. Згідно з [93], кількість позицій,

що доводиться аналізувати за максимальних α-β-відтинань, приблизно дорівнює подвоєному квадратному кореню від кількості позицій перегляду у разі повного перебору. Там само наведено типове дерево гри, яке аналізують за максимальних α-β-відтинань.

На перший погляд, подвоєний квадратний корінь – це не таке велике скорочення. Проте якщо врахувати, що в шахах уже під час аналізу на 3–5 ходів доводиться переглядати 10^{10}–10^{14} позицій, виявляється, що α-β-відтинання може скоротити час на вибір ходу в сотні тисяч або мільйони разів.

Існує ряд практичних рекомендацій з приводу упорядкування перебору позицій-наступників у разі використання α-β-відтинання. Наприклад, передусім аналізувати наступників з найліпшою статичною оцінкою.

У літературі можна знайти багато досліджень про α-β-відтинання. Нехай ми під час аналізу позиційної гри не враховуємо конкретної специфіки такої гри, а обмежимося лише інформацією, що гра є скінченною, детермінованою та з повною інформацією. Тоді інших правил відтинання, еквівалентних повному перебору, не існує.

Є, звичайно, низка рекомендацій, як підвищити якість таких ігор. Більшість відомих програм має у своєму складі «*службу ліпших ходів*», яка спирається на рекомендації «*у таких-то позиціях грай так-то*» та використовує розвинуті бібліотеки стандартних позицій. Добре зарекомендував себе метод *форсованих варіантів*. Метод полягає у тому, що після досягнення максимальної глибини аналіз продовжують, але розглядають не всі ходи, а лише форсовані, наприклад, в шахах – це хід, що веде до взяття фігур. Метод підвищує якість комбінаційної гри та дає змогу різко скоротити ймовірність суттєвих помилок.

Механізм (глибокого) α-β-пошуку може містити низку процедур *неглибокого пошуку*. Наприклад, глибокий пошук може проглядати 10 рівнів, а неглибокий – усього 3 рівня від кожної позиції, яка належить будь-якій із 6 верхніх рівнів процедурі глибокого пошуку. Процедура неглибокого пошуку може бути або ж мінімаксною процедурою «пошуку в глибину», або ж α-β-процедурою. Механізм неглибокого пошуку можна використати для «*n*-найліпшого спрямованого скорочення», інакше кажучи, для впорядкування ходів, згідно з імовірністю, що пояснюється далі.

Вірогідне впорядкування ходів. У процесі відшукання α-β-процедурою «добрих» ходів, можлива поява багатьох α-β-відтинань. Досягнута у такий спосіб економія пошукових затрат може бути досить значною, особливо коли відтинання з'являються недалеко від вершини дерева. Тому α-β-процедура може дати собі змогу затратити деякі зусилля на впорядкування ходів за їх ймовірністю. Для позиції, яка знаходиться близько до вершини дерева, виграш від впорядкування ходів більший, ніж для позицій, що знаходяться ближче до нижньої частини дерева. Наприклад, під час пошуку, в якому розглядають 10 рівнів, ходи можна

впорядковувати для позицій, які лежать на 6-му або вищих рівнях, і не впорядковуються в іншому разі.

Впорядкування можна проводити за допомогою *генератора ходів* або *процедурою неглибокого пошуку* чи, нарешті, *динамічним впорядкуванням*. Прикладом впорядкування за допомогою генератора ходів є правило, згідно з яким ходи, що ведуть до шаху або ж взяття фігури в шахах, породжують і аналізують в першу чергу.

Якщо приймачі деякої позиції раз і назавжди впорядковують процедурою неглибокого пошуку, α-β-процедуру пошуку називають процедурою жорсткого впорядкування (fixed-ordering). Жорстке впорядкування означає, що порядок приймачів незмінний. Наприклад, для деякої позиції, в якій процедура пошуку може проглядати три рівні для оцінки приймачів позиції, приймачі впорядковують відповідно до їхніх робочих оцінок – від найліпшої до найгіршої.

Процедура динамічного впорядкування проводить впорядкування приймачів один раз, але з часом, в процесі отримання нових знань про приймачі, вона може ухвалити рішення про їх перевпорядкування.

Припустимо, що процедура оцінила деякий приймач макс-позиції як найліпший. Потім вона може встановити, що ця попередня оцінка повністю помилкова і що приймач, ймовірно, буде мати зовсім низьку робочу оцінку. (Приклад жертви фігури і мат.) Інтуїція підказує, що якщо на оцінку приймачів потрібні невеликі затрати, може статись, що необхідно присвоїти заданому приймачу як нову цю низьку оцінку, а потім перевпорядкувати приймачі та зробити інший вибір першого приймача, який потрібно відшукати.

Цей процес продовжується доти, доки деякий приймач не отримає глибоку відносно точну робочу оцінку. Через самий характер процедури ця оцінка буде високою, і тому дасть високе значення α. Тому в цьому випадку ймовірна поява багатьох α-відтинань.

Критерії закінчення вказують процедурі пошуку, коли потрібно зупинити пошук. Критерії закінчення часто використовують в парі з іншими критеріями. Ми вже розглянули деякі з них: «кінець гри», мінімальна (максимальна) глибина і направлене скорочення. Інший критерій закінчення завершує пошук в «мертвій» позиції. *Мертвою* називають таку позицію, статична оцінка якої мало відрізнятиметься від робочої оцінки, отриманої в результаті додаткового пошуку. Наприклад, шахову позицію, в якій неможливо брати фігуру або робити шах, можна розглядати як мертву.

За іншого критерію закінчення – мінімальна глибина – завжди аналізують приймачі позиції на деякій спеціально встановленій мінімальній глибині, за винятком тих випадків, коли гра в цій позиції закінчується. Це гарантує повне дослідження найважливіших верхніх рівнів дерева.

На практиці часто використовують поєднання цих правил. Припустимо, що α-β-процедура пошуку вирішує питання про закінчення пошуку в деякій позиції *P*. Пошук завершують. З іншого погляду, якщо позиція *P* знаходиться на деякій мінімальній глибині, скажімо 4, не вище, аналізують приймачі позиції *P*. Якщо позиція *P* лежить на максимальній глибині, скажімо 10, пошук завершують. Якщо позиція *P* знаходиться на рівнях 5, 6, 7, 8, 9, пошук завершують тільки тоді, коли її видаляє процедура спрямованого скорочення або ж вона є мертвою позицією.

10.3.5. *Деякі підходи до самонавчання ігрових програм*

Зрозуміло, що ігрові програми створюють не тільки для шахів. Так, дуже цікавою є програма для гри в шашки, створена Семюелем. Головною метою роботи Семюеля була не стільки гра, скільки вивчення можливостей «самовдосконалення» і «самонавчання» ігрових програм [43].

Ідея одного з підходів Семюеля полягала у такому. В найпростішому випадку статична оціночна функція являє собою лінійний вираз

$$s = k_1a_1 + k_2a_2 + \ldots + k_na_n,$$

де a_i – числові оцінки факторів, що впливають на оцінку позиції, k_i – деякі вагові коефіцієнти. Якість гри визначають цими коефіцієнтами, і тому їх слід оптимізувати. Їх можна підібрати шляхом пробних ігор або безпосереднього аналізу дерева гри.

Якби статична оціночна функція збігалася з оцінкою, переданою знизу, така оцінка була б ідеальною, а її отримання було б еквівалентне повному аналізу дерева. Нехай s – статична оцінка даної позиції; q – оцінка, передана знизу; $e = s - q$ – різниця між цими оцінками. Ідея Семюеля полягала в адаптивній мінімізації цієї різниці. Він визначав *коефіцієнти кореляції* між a_i та e. Якщо кореляція додатна, відповідний коефіцієнт k_i слід зменшити, якщо від'ємна – збільшити. Подібні підходи часто називають *декореляцією помилок*.

Останнім часом шахові програми інтенсивно використовують для аналізу шахових закінчень [57]. При цьому використовують *ретроспективний аналіз*, який іде за оберненим порядком від матових позицій або від позицій з відомою оцінкою.

За допомогою шахових програм було повністю проаналізовано низку ендшпілів, наприклад, «*ферзь з пішаком проти ферзя*», «*тура і кінь проти тури*» та ін. Часто такий комп'ютерний аналіз змінював попередню теоретичну оцінку. Наприклад, закінчення «тура і кінь проти тури» в теорії вважалося нічийним, але комп'ютер виявив великий процент виграшних позицій. У шаховому кодексі є правило, згідно з яким партія закінчується внічию, якщо протягом 50 ходів не було жодного взяття і жоден пішак не зробив ходу. Для деяких типів позицій роблять виняток. На основі детального комп'ютерного аналізу встановлено, що для зміни матеріального співвідношення у багатьох закінченнях 50 ходів не вистачає, і тому ці закінчення теж слід ввести до кодексу як винятки. Шахові програми використовують також для перевірки шахових задач, в яких є вимога поставити мат за визначену наперед кількість ходів. Унаслідок цього в багатьох таких задачах було виявлено серйозні помилки.

Чемпіонати світу серед шахових програм регулярно проводять. Перший з них відбувся у 1974 р., і його переможцем стала програма *Каїса* (колишній СРСР). Досконалість сучасних шахових програм продовжує зростати. Такі програми, як *Hiarch 6.0, Rebel 8.0, MChess Pro 6.0* у разі виконання на процесорах *Pentium* показують досить високу якість гри. Можна згадати спеціалізований шаховий комп'ютер *Deep Blue* з серії *RS/6000*, який влітку 1997 р. в шаховому матчі з шести партій виграв у Г. Каспарова з рахунком 3,5 на 2,5.

10.3.6. *Приклад ігрової програми*

Програма Crest (програма 10.1) – програмна реалізація усім добре відомої гри «хрестики-нулики». Гра проходить у діалоговому режимі «користувач – комп'ютер». У цій програмі реалізовано вищезазначені принципи побудови алгоритмів та використовуються певні евристики, наприклад такі, як «правило боротьби за центр» та ін.

Гра проходить у графічному режимі, де користувач може бачити свої та комп'ютерні ходи. Їх видно на дошці в центрі екрана. Гра супроводжується коментарями. Програма Crest дає змогу гравцеві вибрати фігури, якими він буде грати (хрестики або нулики), а також пріоритет першого ходу (перший хід належить користувачу або перший хід належить комп'ютеру).

Програма Crest містить декілька процедур та функцій, які побудовано згідно з загальними принципами теорії ігор. Це такі процедури та функції як: procedure PutCriss – рисування хрестика; procedure PutZero – рисування нулика; procedure PutNext – процедура вибору хрестика або нулика залежно від вибору користувача; procedure Showc – показує де на даному кроці знаходиться курсор; procedure PutN – знаходить координати фігури на дошці за індексом *N*; procedure PutNextMas – заносить належність ходу в масив за координатами *x*, *y* (1 – комп'ютер, 4 – гравець, 0 – відсутність ходу); procedure PutMasN – заносить належність ходу в масив за індексом *N* (1 – комп'ютер, 4 – гравець, 0 – відсутність ходу); procedure UnputMas – очищує елемент масиву за індексом *N*; procedure Out_End – виводить відповідний до результату гри коментар; **function** The_End – визначення кінця гри і встановлення результату (виграв комп'ютер або нічия); **function** Otsinka – мінімаксна функція, яка шляхом пошуку в глибину в дереві усіх можливих станів гри встановлює робочу оцінку ходу (–1 – програш, 1– виграш, 0 – нічия); **function** MinIndex, MaxIndex – визначає мін- та макс-позиції ходу.

Після запуску програми на екран виводять повідомлення:

Game X O.

Натисніть будь-яку клавішу для подальшого діалогу. Далі буде виведено повідомлення:

What do you prefer?
1. Criss
2. Zero

Ви можете вибрати фігури, якими гратимете. Натисніть клавішу «1» для хрестиків, «2» – для куликів. Після цього програма запропонує вам вибрати, кому належатиме перший хід:

Who'll be the first?
1. You
2. Computer

Натисніть «1», щоб ходити першим, або «2», щоб віддати перший хід комп'ютеру. Далі на екран буде виведено ігральну дошку. Клавішами керування курсором виберіть належну позицію та натисніть «Enter» або «Space».

Для виходу із програми ви можете в будь-який момент натиснути клавішу Esc.

```
Program Ex10_1;
 uses graph, crt;
 const q = 9;
       zer: boolean = true;
       you: boolean = true;
 type n4 = 0..4;
      nn4 = array [1..3, 1..3] of n4;
      nn = 1..q;
      mm = –1..1;
      ss = set of nn;
 var  k, hod: integer;
      mas: nn4;
      s: ss;
      who: n4;                          {Змінна визначає належність ходу}
      index: nn;                        {Покажчик на елемент в масиві}
      temp, max, min: integer;
      x, y, i: integer;
      flag: byte;
      l, m, grdriver, z, w, grmode, j, errcode: integer;
      p1, p2, p3: pointer;              {Покажчики на зображення фігур}
      size: word;
      x1, y1: word;
      gd, gm, ec: integer;

procedure putzero(a1, a2: integer);                    {Нарисувати нулик}
begin
   putimage(300 + a1 * 20 – 20, 185 + a2 * 20, p1^, xorput);
end;

procedure putcriss(a1, a2: integer);                   {Нарисувати хрестик}
begin
   putimage(300 + a1 * 20-20, 185 + a2 * 20, p3^, xorput);
end;
```

```
procedure putnext(x1, y1: integer);                    {Вибір фігури}
begin
    if zer then
       begin
          putzero(x1, y1); zer := false;
       end
     else
       begin
          putcriss(x1, y1); zer := true;
       end;
end;

procedure putn(n: integer);                    {Знаходження координат за індексом n}
 var y, x: integer;
begin
    case n of
       3, 6, 9: y := 3;
       2, 5, 8: y := 2;
       else y := 1;
    end;
    case n of
       1, 2, 3: x := 1;
       4, 5, 6: x := 2;
       else x := 3;
    end;
    putnext(x, y);
end;

procedure showc(a1, a2: integer);              {Активує комірку, знаходження курсора}
begin
    putimage(300 + a1 * 20 – 20, 185 + a2 * 20, p2^, xorput);
    repeat until keypressed;
    putimage(300 + a1 * 20 – 20, 185 + a2 * 20, p2^, xorput);
end;

procedure putnextmas(x1, y1: integer; hod: integer);    {Занесення належності}
begin                                                   {ходу в масив}
    if odd(hod) then mas[x1, y1] := 4
       else mas[x1, y1] := 1
end;

procedure putmasn(n, hod: integer);            {Занесення належності ходу в масив}
 var x, y: integer;
begin
    case n of
       3, 6, 9: y := 3;
       2, 5, 8: y := 2;
       else y := 1;
    end;
```



```
    case n of
       1, 2, 3: x := 1;
       4, 5, 6: x := 2;
       else x := 3;
    end;
    if odd(hod) then mas[x, y] := 4 else mas[x, y] := 1
end;

procedure unputmasn(n: integer);               {Вивільнення комірки масиву}
 var y, x: integer;
begin
    case n of
       3, 6, 9: y := 3;
       2, 5, 8: y := 2;
       else y := 1;
    end;
    case n of
       1, 2, 3: x := 1;
       4, 5, 6: x := 2;
       else x := 3;
    end;
    mas[x, y] := 0
end;

function mas_i(x, y: integer): nn;
begin
    mas_i := 3 * (x – 1) + y
end;

function the_end(mas: nn4; var who: n4): boolean;          {Визначення кінця гри}
 var code, where: integer;
     i, j: integer;
     flag: boolean;
begin
    the_end := false;                                      {Не кінець гри}
    code := 0;
    who := 0;
    flag := true;
    for i := 1 to 3 do                          {Перевірка на порожні місця на дошці}
       for j := 1 to 3 do
          if mas[i, j] = 0 then
             begin
                flag := false;                 {На дошці є порожні місця}
                break;                         {Припинити виконання функції}
             end;
    if flag then code := 4;                    {На дошці нема порожніх місць}
    for i := 1 to 3 do
```

```
    begin                                   {Перевірка на виграшну комбінацію}
      if (mas[i, 1] = mas[i, 2]) and (mas[i, 2] = mas[i, 3]) and (mas[i, 3] <> 0) then
        begin
          code := 1;
          where := i;
          break
        end;
      if (mas[1, i] = mas[2, i]) and (mas[2, i] = mas[3, i]) and (mas[3, i] <> 0) then
        begin
          code := 2;
          where := i;
          break
        end;
    end;
  if (mas[1, 1] = mas[2, 2]) and (mas[3, 3] = mas[2, 2]) and (mas[2, 2] <> 0) then
    begin
      code := 3;
      where := 1;
    end;
  if (mas[1, 3] = mas[2, 2]) and (mas[3, 1] = mas[2, 2]) and (mas[2, 2] <> 0) then
    begin
      code := 3;
      where := 3;
    end;
  case code of
    1: who := mas[where, 1];
    2: who := mas[1, where];
    3: who := mas[where, 1];
    4: who := 0;
  end;
  if code <> 0 then the_end := true;
end;

function otsinka(i: nn; hod: integer; s: ss): integer;      {Визначає робочу оцінку хода}
  var k, temp, max, min: integer;
begin
  who := 0;                                                 {Переможця немає (нічия)}
  putmasn (i, hod);                                         {Занести в масив хід}
  s := s + [i];                                  {Додати цей хід до набору зроблених ходів}
  if the_end(mas, who) then
    begin
      case who of                                           {Визначення переможця}
        4: begin otsinka := 1 ; temp := 1; end;
        1: begin otsinka := -1; temp := -1; end;
        0: begin otsinka := 0; temp := 0; end;
      end;
      unputmasn(i);
      exit;
```



```
    end
  else
    begin
      max := -2;
      min := 2;
      if odd(hod) then
        begin                                   {Якщо хід комп'ютера}
          for k := 1 to q do
            if not(k in s) then                 {Якщо ходу не було}
              begin                             {Знаходження максимальної оцінки}
                temp := otsinka(k, hod + 1, s);
{Мінімаксна функція рекурсивно реалізує пошук в глибину на дереві усіх можливих
розв'язків}
                if temp = -1 then
                  begin
                    otsinka := -1;
                    unputmasn(i);
                    exit;
                  end
                else
                  if min > temp then min := temp;
              end
        end
      else
        for k := 1 to q do
          if not(k in s) then                   {Якщо ходу не було}
            begin                               {Знаходження максимальної оцінки}
              temp := otsinka(k, hod + 1, s);
              if temp = 1 then
                begin
                  otsinka := 1;
                  unputmasn(i);
                  exit;
                end
              else
                if max < temp then max := temp;
            end;
      unputmasn(i);
      if odd(hod) then otsinka := min else otsinka := max;
    end;                {Якщо хід комп'ютера, то оцінка дорівнює min, інакше – max}
end;

function minindex: integer;                                 {Визначення мін-позиції}
  var min, temp, i, index: integer;
begin
  min := 2;
  for i := 1 to q do
    if not(i in S) then
```

```
        begin
          temp := otsinka(i, hod, S);
          if temp = –1 then
            begin
              index := i;
              min := –1;
              break;
            end;
          if temp < min then                    {Якщо даний хід програшний}
            begin
              min := temp;
              index := i;
            end;
        end;
    minindex := index;
end;

function maxindex: integer;                     {Визначення макс-позиції}
  var  max, temp, i, index: integer;
begin
    max := –2;
    for i := 1 to q do
      if not(i in S) then                       {Перебір всіх незроблених ходів}
        begin
          temp := otsinka(i, hod, S);
          if temp = 1 then
            begin
              index := i;
              max := 1;
              break;
            end;
           if temp > max then                   {Якщо хід виграшний}
            begin
              max := temp;
              index := i;
            end;
        end;
    maxindex := index;
end;

procedure out_end(you: boolean; who: n4);       {Виведення коментаря відповідно}
begin                                           {до кінця гри}
    randomize;
      setcolor(trunc(random(13) + 1));
        if you then
          if who = 0 then outtextxy(175, 60, 'Nobody...')
          else
            begin                               {Вивести фразу зі звуковим гудком}
              outtextxy(105, 60, 'You failed, ha-ha.');
              sound(300); delay(200); nosound
            end
        else
          case who of
            4: begin                            {Вивести фразу зі звуковим гудком}
                 outtextxy(105, 60, 'Yes! I win !');
                 sound(300);
                 delay (200);
                 nosound
               end;
            0: outtextxy(175, 60, 'Nobody...');  {Вивести фразу}
          end;
end;

BEGIN                                           {Початок головної програми}
    gd := 9;
    gm := vgahi;
    initgraph(gd, gm, 'd: \tp.70\bgi\');        {Ініціалізація графічного режиму}
    cleardevice;                                {Очищення монітора}
    grdriver := detect;
    initgraph(grdriver, grmode, '');
    errcode := graphresult;
    if errcode <> 0 then halt;
    setcolor(yellow);                           {Реалізація робочого інтерфейсу}
    settextstyle(0, 0, 3);
    outtextxy(10, 50, 'Game X O ');
    readkey;
    cleardevice;
    settextstyle(0, 0, 2);
    setcolor(green);
    outtextxy(10, 20, 'What do You prefer? ');
    outtextxy(10, 40, '1. Criss');              {Вибір фігури (хрестик, нулик)}
    outtextxy(10, 60, '2. Zero');
    case(readkey) of
      '1': zer := false;
      '2': zer := true;
    end;
    cleardevice;
    outtextxy(10, 20, 'Who"ll be the first? ');  {Вибір першого ходу}
    outtextxy(10, 40, '1. You');
    outtextxy(10, 60, '2. Computer');
    case (readkey) of
      '1': you := true;
      '2': you := false;
    end;
    cleardevice;
```

```
{Формування зображення нулика і зв'язування його з покажчиком p1}
    setcolor(white);
    for m := 5 to 7 do circle(300, 200, m);
    setcolor(lightgreen);
    size := imagesize(290, 190, 310, 210);
    getmem(p1, size);
    getimage(290, 190, 310, 210, p1^);
    cleardevice;
{Формування зображення хрестика і зв'язування його з покажчиком p3}
    line(300, 190, 320, 210);
    line(300, 210, 320, 190);
    line(301, 190, 320, 209);
    line(301, 210, 320, 191);
    size := imagesize(301, 191, 319, 209);
    getmem(p3, size);
    getimage(301, 191, 319, 209, p3^);
    cleardevice;
{Формування зображення активної рамки і зв'язування її з покажчиком p2}
    setcolor(14);
    rectangle(296, 196, 314, 214);
    size := imagesize(296, 196, 314, 214);
    getmem(p2, size);
    getimage(295, 195, 314, 214, p2^);
    cleardevice;
    setcolor(green);
    setbkcolor(0);
    settextstyle(0, 0, 1);
    cleardevice;
    setcolor(14);
    randomize;
    setcolor(random(15) + 1); settextstyle(0, 0, 3); outtextxy(150, 10, 'Crisses-Zeroes');
    setcolor(15);
    for i := 0 to 4 do for j := 0 to 4 do                    {Рисування ігрової дошки}
      mas[i, j] := 0;
    for i := 2 to 3 do
      line(300 + – 20 + i * 20, 185 + 20, 300 + – 20 + i * 20, 185 + 80);
    for i := 2 to 3 do
      line(300 + 0, 185 + i * 20, 300 + 60, 185 + i * 20);
    hod := 1;
    if not(you) then
      begin
        randomize;                          {Якщо перший хід робить комп'ютер,}
        x := trunc(random(3)) + 1;          {то він робить його випадково}
        y := trunc(random(3)) + 1;
        zer := not(zer);
        putnext(x, y); mas[x, y] := 4;
        inc(hod);
```



```
        s := s + [mas_i(x, y)];
      end;
    x := 1; y := 1;
    repeat                                          {Переміщення активної рамки}
      showc(x, y);
      k := byte(readkey);
      if k = 0 then
        case byte(readkey) of
          72{Вверх}: if y > 1 then dec(y) else y := 3;
          80{Вниз}: if y < 3 then inc(y) else y := 1;
          77{Вправо}: if x < 3 then inc(x) else x := 1;
          75{Вліво}: if x > 1 then dec(x) else x := 3;
        end
      else
        case k of
          32, 13: if mas[x, y] = 0 then
                    begin
                      putnextmas(x, y, hod);
                      putnext(x, y);
                      s := s + [mas_i(x, y)];
                      inc(hod);                                  {hod – not odd}
                      if the_end(mas, who) then out_end(you, who)
                        else
                          begin
                            setcolor(trunc(random(13) + 1));
                            outtextxy(175, 75, 'wait, please...');
                            if you then index := minindex else index := maxindex;
                            setcolor(0);
                            outtextxy(175, 75, 'wait, please...');
                            setcolor(15);
                            sound(220);
                            delay(200);
                            nosound;
                            putmasn(index, hod);
                            putn(index);
                            if the_end(mas, who)      {Якщо кінець, то вивести}
                              then out_end(you, who); {відповідний коментар}
                            s := s + [index];  {Занесення нового ходу в масив}
                            inc(hod);                {Наступний хід}
                          end;
                    end;
        end;                                                     {case}
    until (k = 27);                                  {Для виходу натиснути Esc}
    closegraph;
end.
```

Програма 10.1. Програма гри в хрестики-нулики Crest

Напишіть програму реалізації гри Го-Моку. Це нескладна позиційна гра, яка відома зі стародавніх часів і, напевно, вперше виникла у Японії. Гра схожа на хрестики-нулики, але набагато цікавіша. За класичними правилами, грають на дошці розміром 19×19 клітин (в програмі реалізуйте альтернативні варіанти: скорочений – 9×9 та розширений – 29×29 клітин). Гравці ставлять свої фішки на вільні клітини дошки по черзі. Виграє той, кому першому пощастить побудувати власну лінійку з п'яти фішок – горизонтальну, вертикальну або діагональну.

Звичайно, Го-Моку простіша за шашки або шахи, але її алгоритмізація вимагає використання більшості положень загальної теорії ігор. Як і програми гри в шашки або шахи, алгоритм ігри в Го-Моку містить оцінку поточної позиції і прогноз на кілька ходів наперед. У такій реалізації гри використайте алгоритм прогнозу лише на два ходи за допомогою спрощеного алгоритму α-β-відтинання. Досить поверховий аналіз (який можна легко посилити) обумовлений дуже сильним коефіцієнтом розростання дерева варіантів і можливістю ввести дуже ефективну лінійну оціночну функцію.

Використайте у програмі оціночну функцію, що обчислює вартість ходу в дану клітину (або, що аналогічно, перехід у наступну позицію після ходу) за допомогою емпірично-визначеної ваги кожної з чотирьох лінійок (з 4+1+4 клітинок), які перетинаються в даній клітині. Оціночна функція кожної з лінійок може бути такою:

$$\varphi = g^2 - 8g - 2h + o + n + t + 16,$$

де g – кількість вільних позицій, h – характеризує наявність фішок суперника, o – характеризує заблокованість лінійки з її кінців, n – сусідство із зайнятою клітиною, t – неперервність усієї лінійки.

Загальну оціночну функцію, яка враховує вартість усіх чотирьох лінійок, що перетинаються, можна подати так:

$$F = 64a + 16b + 4c + d,$$

де a, b, c, d – відсортовані за спаданням значення функції φ для кожної лінійки.

Оскільки якість гри програми в Го-Моку дуже сильно залежить від якості оціночної функції, оптимізація її коефіцієнтів значно поліпшить саму гру. Для цього може бути використаний підхід Семюеля – процедура ***декореляції помилок.***

Задачі та вправи до розділу 10

1. Поясніть зв'язок між ігровими задачами і плануванням цілеспрямованих дій.
2. Назвіть характерну особливість ігрових задач, яка вирізняє їх серед інших задач прийняття рішень.
3. Дайте поняття дерева гри; побудуйте дерево довільної конкретної гри.
4. Напишіть програму, яка грає в Го-Моку.
5. Напишіть програму для гри в калах на основі зрізаної мінімаксної процедури з α-β-відтинанням.
6. Проілюструйте на прикладі довільного дерева гри мінімаксну процедуру на повному дереві, мінімаксну процедуру на зрізаному дереві та α-β-відтинання. Покажіть на цьому ж прикладі вплив глибини перебору на якість гри.
7. Наведіть відому вам теоретичну оцінку максимального скорочення перебору при α-β-відтинанні. Покажіть на прикладі, як порядок перебору ходів-альтернатив впливає на ефективність α-β-відтинання.
8. Покажіть на дереві гри позиції, які аналізують за максимальних α-β-відтинань.
9. Покажіть, як можна побудувати І/АБО-граф для довільної ігрової задачі.
10. Дайте загальну характеристику сучасних шахових програм.
11. Чим відрізняється підхід, реалізований у сучасних шахових програмах, від підходу людини-шахіста до гри?

Розділ 11

АЛГЕБРИЧНІ ПЕРЕТВОРЕННЯ ТА СПРОЩЕННЯ

11.1. Загальна характеристика

Розглянемо методи швидкої і точної реалізації операцій з числами та поліномами. Для цього потрібно мати формальний апарат, що давав би змогу зручно оперувати математичними виразами [134, 141]. Наприклад, у разі роботи з поліномами бажано розглядати дві моделі: *щільну* – для задання поліномів, що мають більшість коефіцієнтів, відмінних від нуля; *розріджену* – для задання поліномів, що мають більшість коефіцієнтів, рівних нулю. Остання – особливо корисна для задання поліномів від багатьох змінних.

Для розв'язання багатьох задач, пов'язаних з обробкою поліномів, також корисним є використання різних спеціалізованих перетворень, які зменшують загальну похибку обчислень під час роботи з дійсними числами. Прикладом може слугувати перетворення Фур'є (швидке перетворення Фур'є). Його використовують для побудови багатьох ефективних алгоритмів. Отже, бажано мати швидку реалізацію перетворення Фур'є. Прикладом його використання може слугувати обчислення *добутку двох поліномів. Один з алгоритмів обчислення добутку складається* з такої послідовності кроків. До векторів коефіцієнтів поліномів спочатку слід застосувати деяке лінійне перетворення (дискретне перетворення Фур'є). Потім над образами коефіцієнтів потрібно виконати операцію, простішу за згортку, і до результату застосувати обернене перетворення Фур'є.

Алгоритм швидкого перетворення Фур'є ґрунтується на техніці обчислення поліномів за допомогою ділення, з урахуванням того, що поліноми обчислюють для аргументів, які дорівнюють кореням із одиниці. Під згорткою розумітимемо обчислення поліномів у коренях з одиниці, множення цих значень та інтерполяцію поліномів. За допомогою швидкого перетворення Фур'є можна розділити ефективний алгоритм реалізації згортки. Його можна застосувати для формального (символьного) множення поліномів і цілих чисел (алгоритм Шьонхаге–Штрасена).

Отже, для роботи з многочленами необхідно, по-перше, вирішити спосіб задання (опису) і, по-друге, написати алгоритми (програми), кожен(на) з яких ефективно реалізує певну операцію.

Система, що дає змогу оперувати з математичними виразами (зазвичай, під математичними виразами розуміють цілі числа довільної точності, поліноми та раціональні функції), називають *математичною системою з операціями над символами* [127].

Перший метод нашого опису називають алгебричним перетворенням. Припустимо, що на вході ми маємо I з множини S_1 і функцію $f(I)$, яка описує певну операцію, що потрібно здійснити над I. Як правило, результат $f(I)$ також належить S_1, але спосіб визначення $f(I)$ шляхом уточнення операцій на елементах з множини S_1 є неефективним. Метод *алгебричного перетворення* передбачає задання входу у вигляді елемента з S_2. Множина S_2 складається з тих же елементів що і S_1, але має іншу форму їх задання. Перетворення входу в іншу форму проводять для надання можливості простішого обчислення функції f на елементах з множини S_2, ніж на елементах з S_1. Коли отримують відповідь для S_2, застосовують обернене перетворення, щоб встановити результат на множині S_1.

Наприклад, нехай S_1 – множина цілих чисел, наведених у вигляді арабських цифр (десяткова система числення), і S_2 – множина цілих чисел у двійковій системі числення. Якщо задано два цілих числа з S_1 і потрібно провести деякі арифметичні операції над ними, то за більшості обчислень над цими числами за допомогою комп'ютера, останній перетворює числа в елементи з S_2, виконує операції з ними та знову, для зручності сприйняття результату людиною, подає результат в арабській формі запису (у десятковій системі числення). Алгоритм таких перетворень знайомий всім студентам, що вивчають комп'ютерні науки. Для переходу від множини S_1 в множину S_2 необхідно застосувати операцію ділення на 2, а для оберненого переходу – операцію множення.

Світ алгебричних алгоритмів настільки різноманітний, що ми лише спробуємо розкрити декілька найцікавіших тем. Звернемо увагу на проблеми цілих і модульну арифметику, яку можна зобразити як схему перетворень, що є корисною для прискорення арифметичних операцій з цілими; розглянемо найбільш відомі алгоритми. Арифметичні операції над цілими числами і поліномами доцільно вивчати разом, тому що багато алгоритмів, які працюють з цілими числами, фактично еквівалентні алгоритмам над поліномами від однієї змінної. Це стосується як простих (множення, ділення тощо), так і складніших операцій. Наприклад, знаходження лишку цілого числа за модулем, що задається іншим цілим числом, еквівалентне обчисленню полінома у точці. Зображення цілого числа його лишками еквівалентне зображенню полінома його значеннями у кількох точках. Поновлення цілого числа за його лишками («китайська теорема про залишки») еквівалентне інтерполяції полінома.

Дотримуючись методології [9, 127], закцентуємо також увагу на тому, що для деяких операцій над цілими числами і поліномами, такими як ділення і піднесення до квадрата, потрібен час того ж порядку, що й для множення [113]. Час виконання інших операцій, таких як вищезгадані операції з лишками чи обчислення найбільших спільних дільників, може

перевищувати час множення не більше ніж в $\log n$ разів, де n – довжина двійкового задання цілого числа або степінь полінома.

Під час висвітлення відповідного матеріалу чергуватимемо результати для цілих чисел з відповідними результатами для поліномів, проводячи формалізацію алгоритму тільки для якогось одного варіанта. Серед алгоритмів розгляду віддаватимемо перевагу алгоритмам, що є асимптотично найефективнішими серед відомих, а для задання поліномів використовуватимемо дві основні моделі: щільну й розріджену.

Найбільш очевидна аналогія між невід'ємними цілими числами й поліномами від однієї змінної полягає в можливості зображення тих й інших скінченними степеневими рядами. У разі цілих чисел коефіцієнти a_i можна вибирати з множини $\{0,1\}$ при $x^i = 2^i$, у разі поліномів – a_i можна вибирати з деякої множини коефіцієнтів, вважаючи x змінною. Далі вважатимемо множину коефіцієнтів полем дійсних чисел, хоча результати вірні для будь-якого поля коефіцієнтів, якщо вимірювати обчислювальну складність числом операцій в цьому полі [63]. Тому в нашій моделі розмір коефіцієнтів не беруть до уваги. Існує природна міра «розміру» – це фактично довжина степеневого ряду, яким зображене ціле число або поліном, а у разі двійкового цілого числа за розмір беруть число бітів, потрібних для його зображення; у разі полінома, розмір – це число його коефіцієнтів.

Отже, якщо i – невід'ємне ціле число, то РОЗМІР(i) = $\lfloor \log i \rfloor + 1$; якщо $p(x)$ – поліном, то РОЗМІР(p) = СТ(p) + 1, де СТ(p) – степінь полінома p, тобто найбільший степінь змінної x з ненульовим коефіцієнтом.

Аналогію між цілими числами й поліномами продемонструємо на прикладі реалізації операції наближеного ділення. Цю операцію визначають так. Якщо a і b – два цілих числа і $b \neq 0$, то знайдеться єдина пара цілих чисел q і r, для яких $a = bq + r$ і $r < b$, де q і r – відповідно частка і залишок від ділення a на b. Аналогічно, якщо a і b – поліноми, причому b відрізняється від константи, то можна знайти такі поліноми q і r, що $a = bq + r$ і СТ(r) < СТ(b).

Згідно з [9], результати арифметичних операцій над поліномами і цілими числами є дуже схожими, якщо користуватися двома різними мірами складності (арифметичною і бітовою). Ці дві міри аналогічні у тому сенсі, що бітові операції – це операції над коефіцієнтами степеневих рядів, якими зображені цілі числа.

11.2. Базові обчислення

11.2.1. *Обчислення многочлена*

Розглянемо операції оцінки та інтерполяції на многочленах і метод, що дістав назву *методу алгебричних спрощень*. Алгебричне спрощення передбачає перетворення формули обчислення в таку форму, щоб кількість операцій для її обчислення була мінімальною.

Многочлен від однієї змінної, як правило, записують так:

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0, \qquad (11.1)$$

де x – змінна, a_i – ціле або дійсне число (або більш загальний елемент комутативного кільця чи поля [63]).

Якщо $a_n \neq 0$, то n називають степенем полінома A.

Для задання многочлена його коефіцієнтами існує принаймні два шляхи.

Перший – (n, a_n, a_{n-1}, ..., a_1, a_0). Таке задання називають *щільним*, оскільки воно точно зберігає всі коефіцієнти, незважаючи на їх значення (нульові вони чи ні). У разі, коли багато коефіцієнтів дорівнюють нулю, таке задання було б марнотратним. Тому існує ще один спосіб зображення (*розріджений*), в якому зберігаються лише ненульові коефіцієнти: (n, a_n, $n-1$, a_{n-1}, ..., 1, a_1, 0, a_0). Припустимо, що доступ до i-го коефіцієнта відбувається шляхом написання a_i.

Розглянемо задачу. Нехай задано многочлен типу (11.1) і потрібно його визначити (оцінити) в точці v, тобто треба обчислити $A(v)$. Прямий метод додає a_0 до $a_1 v$, $a_2 v^2$ до цієї суми і т. д. Аналіз цього алгоритму досить простий, фактично виконують $2n$ добутків, n додавань і $2n + n$ присвоєнь.

Ньютон, а потім Горнер запропонували методику поліпшення часової оцінки, яка з часом дістала назву правила (схеми) Горнера. Вона передбачає запис многочлена у такій формі:

$$A(x) = (\ldots ((a_n x + a_{n-1}) x + a_{n-2}) x + \ldots + a_1) x + a_0. \qquad (11.2)$$

Це є першим прикладом алгебричного спрощення. Алгоритм обчислення, що ґрунтується на такому заданні (правило Горнера), описує функція *horner*.

```
Function HORNER(a: array of real; v: real; n: integer): real;
  var s: real;
      i: integer;
begin
   s:=a[n+1];
   for i := n downto 1 do
     s := s * v + a[i];
   HORNER := s;
end;
```

Правило Горнера потребує лише n добутків, n додавань і $n + 1$ операцій присвоєння. Отже, цей алгоритм буде вдвічі ліпший за прямий метод.

Розглянемо випадок розрідженого зображення многочлена типу

$$A(x) = a_m x^{e_m} + a_{m-1} x^{e_{m-1}} + \ldots + a_1 x^{e_1}, \qquad (11.3)$$

де $a_i \neq 0$ і $e_m > e_{m-1} > \ldots > e_1 \geq 0$.

Якщо вважати, що v^e обчислюють шляхом знаходження повторних добутків v, тоді ця операція потребує $(e-1)$ операцій множення, а пря-

мий алгоритм обчислення потребує $e_m + e_{m-1} + ... + e_1$ добутків, m додавань і $(m + 1)$ присвоєнь. Це дуже неефективно. Можна поліпшити цей алгоритм, ґрунтуючись на обчисленні v^{e_1}, $v^{e_2-e_1}v^{e_1}$, $v^{e_3-e_2}v^{e_2}$, ... Такс обчислення многочлена реалізовуватиме функція *nstraiteval*.

```
Function NSTRAITEVAL(a: array of real; e: array of integer; v: real; m: integer): real;
  var  s, r, t: real;
       i: integer;

begin
    s := 0;
    e[0] := 0;
    t := 1;
    for i := 1 to m do
      begin
        if (e[i] – e[i – 1]) = 0 then r := 1
           else r := exp(v * ln(e[i] – e[i – 1]));
        t := t * r;
        s := s + a[i] * t;
      end;
    NSTRAITEVAL := s;
end;
```

Зрозуміло, що функція *nstraiteval* потребує лише $e_m + m$ операцій множення і $3m + 3$ присвоєнь, m додавань і m операцій віднімання.

Для використання схеми Горнера розглянемо обчислення $A(x)$ у вигляді:

$$A(x) = (... ((\ a_m x^{\,e_m - e_{m-1}} + a_{m-1})\, x^{\,e_{m-1} - e_{m-2}} + ... + a_2)\, x^{\,e_2 - e_1} + a_1)\, x^{\,e_1}. \qquad (11.4)$$

Тоді функція *shorner* обчислює за цією формулою (правило Горнера для розрідженного зображення).

```
Function SHORNER(a: array of real; e: array of integer; m: integer; v: real): real;
  var  s: real;
       i: integer;
begin
    s := 0; e[0] := 0;
    for i := m downto 1 do
      s := (s + a[i]) * exp(v, ln(e[i] – e[i – 1]));
    SHORNER := s;
end;
```

У цьому разі, кількість операцій множення визначатиметься

$$(e_m – e_{m-1} – 1) + ... + (e_1 – e_0 – 1) + m = e_m.$$

Крім того, маємо лише m операцій додавання, m віднімань та $m + 2$ присвоєнь.

Повну реалізацію у Паскалі схеми Горнера наведено у програмі 11.1.


```
Program Ex_11_1;
  const
    MaxCoefs = 100;                          {Максимальний степінь полінома}
  type
    TPolynom =record
                n: integer;
                coefs: array [0..MaxCoefs] of real;
              end;

{Функція обчислює значення полінома}
function evaluate(x: real ; p: TPolynom): real;
  var
    result: real;
    i: integer;
begin
    result := 0;
    if p.n >= 0 then
      begin
        result := p.coefs[0];
        for i := 1 to p.n do
          result := result * x + p.coefs[i];
      end;
    evaluate := result;
end;

{Процедура реалізує введення полінома з консолі}
procedure readpoly(var p: TPolynom);
  var
    i: integer;
begin
    write('Введіть степінь полінома: ');
    readln(p.n);
    for i := p.n downto 0 do
      begin
        write('A', p.n – i, ': ');
        readln(p.coefs[i]);
      end;
end;

  var
    p: TPolynom;
    x: real;
begin
    writeln('Введіть поліном: ');
    readpoly(p);
```

```
    write('Введіть значення X для обчислення: ');
    readln(x);
    writeln('Результат дорівнює: ', evaluate(x, p):0:5);
    write('Натисніть ENTER для закінчення...');
    readln;
end.
```

Програма 11.1. Реалізація алгоритму Горнера

Для виконання програми підрахунку значення полінома $5x^3 + 4x^2 + 3x + 2$ в точці $x = 5$ слід ввести такі значення:

Введіть поліном:
Введіть степінь полінома: 3
*A*0: 2
*A*1: 3
*A*2: 4
*A*3: 5
Введіть значення X для обчислення: 5
Результат дорівнює: 742.00000
Натисніть ENTER *для закінчення...*

11.2.2. *Інтерполяція*

Розглянемо задачу. Задано n точок (x_i, y_i). Потрібно знайти коефіцієнти многочлена $A_m(x)$ степеня $m \le n - 1$, графік якого проходить через задані точки. Математично цю задачу розв'язав Лагранж:

$$A(x) = \sum_{i=1}^{n} \prod_{k=1, k\neq i}^{n} \left(\frac{x - x_k}{x_i - x_k} \right) y_i. \qquad (11.5)$$

Для перевірки, чи задовольняє $A(x)$ n точкам, ми обчислюємо

$$A(x_i) = \prod_{k=1, k\neq i}^{n} \left(\frac{x_i - x_k}{x_i - x_k} \right) y_i = y_i, \qquad (11.6)$$

коли всі інші елементи дорівнюють нулю. Чисельник кожного елементу – це добуток $n - 1$ множників, тому степінь $A(x) \le n - 1$.

Згідно з [127], алгоритм Лагранжа матиме наступну часову складність:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} (j-1) = \sum_{i=1}^{n} \left(n \frac{n+1}{2} - n \right) = n^2 \frac{n+1}{2} - n^2 = O(n^3).$$

Тому знайдено алгоритм інтерполяції з ліпшими показниками за часом.

Нехай знайшли многочлен $A(x)$, який задовольняє умові $A(x_i) = y_i$ для $1 \le i \le n$, і бажають додати ще одну точку (x_{n+1}, y_{n+1}). Як тепер обчислити многочлен, щоб він задовольняв заданій умові, якщо $A(x)$ її вже задовольняє? Якщо існує ефективне вирішення цієї проблеми, то ми могли б застосувати його n разів, щоб отримати n точок, які б задовольняли наведеному многочлену.

Нехай $G_{i-1}(x)$ задовольняють $i - 1$ точкам (x_k, y_k), $1 \le k \le i$, тобто $G_{i-1}(x_k) = y_k a D_{i-1}(x) = (x - x_1) \dots (x - x_{i-1})$. Тоді ми можемо обчислити $G_i(x)$ за формулою

$$G_i(x) = (y_i - G_{i-1}(x_i)) \frac{D_{i-1}(x)}{D_{i-1}(x_i)} + G_{i-1}(x). \qquad (11.7)$$

Можна помітити, що

$$G_i(x_k) = (y_i - G_{i-1}(x_i)) \frac{D_{i-1}(x_k)}{D_{i-1}(x_i)} + G_{i-1}(x_k),$$

але $D_{i-1}(x_k) = 0$ для $1 \le k \le i - 1$, тоді $G_i(x_k) = G_{i-1}(x_k) = y_k$, до того ж

$$G_i(x_i) = (y_i - G_{i-1}(x_i)) \frac{D_{i-1}(x_i)}{D_{i-1}(x_i)} + G_{i-1}(x_i) = y_i - G_{i-1}(x_i) + G_{i-1}(x_i) = y_i. \qquad (11.8)$$

Отже, на основі цієї формули можна побудувати процедуру *interp*, яка обчислюватиме точки, що задовольняють многочлену (ньютонівська інтерполяція).

```
Procedure INTERP(x, y: array of real; var g: array of real);
  var
    num, denom: real;
    d: array [0. . n – 1] of real;
    i, j: integer;
begin
    for i := 0 to n – 1 do
      begin
        g[i] := 0;
        d[i] := 0;
      end;
    g[0] := y[0];
    d[0] := – x[0];
    d[1] := 1;
    for i := 1 to n – 1 do
      begin
        denom := HORNER(d, x[i], i);                    {Обчислює d(x) в x(i)}
        num := HORNER(g, x[i], i – 1);                  {Обчислює g(x) в x(i)}
        for j := n – 1 downto 0 do          {g = ADD(MULT((y(i) – num)/denom g), d)}
          g[j] := g[j] + d[j] * (y[i] – num)/denom;
        for j := n – 1 downto 1 do                      {d = MULT(d, x – x(i))}
          d[j] := d[j – 1] – d[j] * x[i];
        d[0]:= –d[0] * x[i];
      end;
end;
```

На i-му кроці D має степінь $i-1$ та G має степінь $i-2$, тому виклик процедури HORNER потребує

$$\sum_{i=1}^{n-1}(i+i-1)=n(n-1)-(n-1)=(n-1)^2,$$

де $(y(i)-\text{num})/\text{denom}$ – константа.

Множення константи на D потребує $i+1$ операцій, добуток D на $X-X(i)-i+1$ операцій множення. Додавання до G потребує нуль операцій. Тому на останній крок витрачають

$$\sum_{i=1}^{n-1}(2i+2)=n(n-1)+2(n-1)=(n-1)(n+2)$$

операцій, а отже, загалом процедура INTERP потребує $O(n^2)$ часу.

11.2.3. *Швидке перетворення Фур'є*

Перетворенння Фур'є для неперервної функції $a(t)$ задають формулою

$$A(f)=\int_{-\infty}^{+\infty}a(t)e^{2\pi ift}dt, \tag{11.9}$$

тоді як обернене перетворення має такий вигляд:

$$a(t)=\frac{1}{2\pi}\int_{-\infty}^{+\infty}A(f)e^{-2\pi ift}df. \tag{11.10}$$

Символ i в двох рівняннях є квадратним коренем з -1. Константа e – основа натурального логарифма. Змінну t часто трактують як час, f – як частоту. Тому перетворення Фур'є інтерпретують як взяття функції часу від функції частоти.

Відповідно до неперервного перетворення Фур'є існує *дискретне* перетворення Фур'є, яке оперує з образами точок $a(t)$, фактично з a_0, a_1, ..., a_{N-1}. Дискретне пряме перетворення Фур'є визначають як набір чисел $A_0, A_1, \ldots, A_{N-1}$

$$A_j=\sum_{k=0}^{N-1}a_k e^{\frac{2\pi ijk}{N}},\quad 0\le j\le n-1, \tag{11.11}$$

а обернене як

$$a_k=\frac{1}{N}\sum_{j=0}^{N-1}A_j e^{-\frac{2\pi ijk}{N}},\quad 0\le k\le n-1. \tag{11.12}$$

Якщо уявити собі многочлен типу (11.1) тоді елемент перетворення Фур'є A_j – це оцінка $a(x)$ в точці $x=\omega^j$, де $\omega=e^{\frac{2\pi i}{N}}$. Якщо розглядати многочлен з коефіцієнтів Фур'є

$$A(x)=A_{N-1}x^{N-1}+A_{N-2}x^{N-2}+\ldots+A_1x+A_0,$$

тоді кожне a_k відповідає оцінці $A(x)$ в $x=(\omega^{-1})^k$, де $\omega=e^{\frac{2\pi i}{N}}$. *Отже, дискретне перетворення Фур'є точно відповідає оцінці многочлена в n точках:* $\omega^0, \omega^1, \ldots, \omega^{N-1}$. *Швидке перетворення Фур'є* (скорочено FFT) є алгоритмом обчислення такої оцінки за час $O(N\log N)$.

Швидке обчислення перетворення Фур'є за правилом Горнера відбувається завдяки визначенню оцінки в специфічних точках: вони є показниками степеня ω^j для $0\le j\le N-1$, де $\omega=e^{\frac{2\pi i}{N}}$. Точка ω – це первинний N-й корінь з одиниці в полі комплексних чисел.

Елементом ω комутативного кільця [9] називають *первинний n-й корінь з одиниці*, якщо $\omega\ne 1$, $\omega^N=1$ і $\sum_{p=0}^{N-1}\omega^{jp}=0$, $1\le j\le N-1$.

Відомо [127]: якщо $2n$ і ω первинний N-й корінь з одиниці, тоді $-\omega^j=\omega^{j+n}$ і ω^2 – первинний n-й корінь з одиниці. З цього можна дійти висновку: якщо ω^j, $0<j\le N-1$ первинні N-ні корені з одиниці, $N=2n$, тоді ω^{2j}, $0<j\le n-1$ також первинні n-ні корені з одиниці. Ця ідея є базою застосування методу «розділяй і пануй» для реалізації перетворення Фур'є з часовою складністю алгоритму $O(N\log N)$.

Нехай знову $a_{N-1}, \ldots, a_0$ – коефіцієнти перетворення та $a(x)=a_{N-1}x^{N-1}+\ldots+a_1x+a_0$. Розділимо $a(x)$ на дві пари, одна з яких містить лише парні номери експонент, а інша – непарні. Отримаємо

$$a(x)=a_{N-1}x^{N-1}+a_{N-3}x^{N-3}+\ldots+a_1x+\ldots+a_{N-2}x^{N-2}+\ldots+a_2x^2+a_0.$$

Припустивши $y=x^2$, можна переписати $a(x)$ у вигляді суми двох многочленів:

$$a(x)=\left(a_{N-1}y^{\frac{N}{2}-1}+a_{N-3}y^{\frac{N}{2}-2}+\ldots+a_1\right)x+$$
$$+\left(a_{N-2}y^{\frac{N}{2}-1}+a_{N-4}y^{\frac{N}{2}-2}+\ldots+a_0\right)=c(y)x+b(y).$$

Пригадавши ціну перетворення Фур'є $a(\omega^j)$, $0\le j\le N-1$, значення $a(x)$ в точках ω^j, $0\le j\le \frac{N}{2}-1$ можна записати як $a(\omega^j)=c(\omega^{2j})\omega^j+b(\omega^{2j})$ і $a(\omega^{j+n})=-c(\omega^{2j})\omega^j+b(\omega^{2j})$.

Ці дві формули є основою використання методу «розділяй і пануй» для розв'язання задачі розміром N, перетворивши її на дві тотожні підзадачі розмірами $n=N/2$. Підзадачами у цьому разі будуть оцінки многочленів $b(y)$ та $c(y)$ степеня $n-1$ в точках $(\omega^2)^j$, $0\le j\le n-1$, а ці точки є первинними коренями n-го степеня. Тому бажано вибирати $N=2^m$. Тоді можемо продовжувати процедуру ділення доти, доки не отримаємо зовсім тривіальну задачу (в даному випадку тривіальною буде задача оцінки константного многочлена).

Процедура FFT використовує заздалегідь запропоновані ідеї та описує рекурсивний варіант швидкого перетворення Фур'є.

```
Procedure FFT(N: integer; aa: array of complex; w: complex; var A: array of complex);
  var bb, B, cc, C: array [0..m] of complex;
      wp: array [–1..m] of complex;
      w2: complex;
      i: integer;
begin
   if N = 1 then A[0] := aa[0]
     else
       begin
         N := round(N/2);
         for i := 0 to N – 1 do
           if odd(i) then
             begin
               C[i].re := aa[i].re;
               C[i].im := aa[i].im;
               cc[i].re := aa[i].re;
               cc[i].im := aa[i].im;
             end
           else
             begin
               B[i].re := aa[i].re;
               B[i].im := aa[i].im;
               bb[i].re := aa[i].re;
               bb[i].im := aa[i].im;
             end;
         w2.re := w.re * w.re – w.im * w.im;
         w2.im := 2 * w.re * w.im;
         FFT(N,bb,w2,B);
         FFT(N,cc,w2,C);
         wp[–1].re := w.re / (w.re * w.re + w.im * w.im);
         wp[–1].im := –w.im / (w.re * w.re + w.im * w.im);
         for i := 0 to N – 1 do
           begin
             wp[i].im := wp[i – 1].re * w.im + wp[i – 1].im * w.re;
             wp[i].re := wp[i – 1].re * w.re – wp[i – 1].im * w.im;
             A[i].re := B[i].re + wp[i].re * C[i].re – wp[i].im * C[i].im;
             A[i].im := B[i].im + wp[i].re * C[i].im + wp[i].im * C[i].re;
             A[i + N].re := B[i].re – wp[i].re * C[i].re + wp[i].im * C[i].im;
             A[i + N].im := B[i].im – wp[i].re * C[i]. im – wp[i].im * C[i].re;
           end;
       end;
end;
```

Обчислимо [127] часову оцінку процедури FFT. Нехай $T(N)$ – час алгоритму на N вхідних наборах. Тоді отримаємо $T(N) = 2T(N/2) + cN$, де c – константа і N – обмежене часом створення $b(x)$, $c(x)$, A і B. Оскільки $T(1) = d$, де d – інша константа, можна досить просто записати дану рекурсію. Отримаємо:

$$T(2^m) = 2T(2^{m-1}) + c2^m = \ldots = cm2^m + T(1)2^m = \\ = cN\log_2 N + dN = O(N\log_2 N).$$

Розглянемо ітераційну версію [127] реалізації FFT.

Нехай елементи вектора $(a_0, \ldots, a_{N-1})$ є коефіцієнтами многочлена $A(x)$, тоді перетворення Фур'є можна розглядати як обчислення $a(\omega^j)$ для $0 \le j < N$. Це перетворення також еквівалентне до обчислення остачі від ділення $A(x)$ на лінійний множник $x - \omega^j$, де $q(x)$ і c (частка і остача), такі, що $A(x) = (x - \omega^j)q(x) + c$, тоді $A(\omega^j) = 0 \cdot q(x) + c = c$. Ми можемо розділити $A(x)$ на N лінійних многочленів, витративши на це $O(N^2)$ операцій. Можна обчислити ці остачі за допомогою спеціалізованого процесу, що має структуру бінарного дерева.

Розглянемо добуток лінійних множників $(x - \omega^0)(x - \omega^1)\ldots(x - \omega^7) = x^8 - \omega^0$. Всі проміжні члени скорочуються, залишаються лише показники степеня 8 і 0 з ненульовими коефіцієнтами. Якщо ми виберемо з цих добутків члени з парним і непарним степенем, то: $(x - \omega^0)(x - \omega^2)(x - \omega^4)(x - \omega^6) = (x^4 - \omega^0)$ і $(x - \omega^1)(x - \omega^3)(x - \omega^5)(x - \omega^7) = x^4 - \omega^4$. Продовжуючи у такий самий спосіб, бачимо (рис. 11.1), що виділені добутки мають лише два ненульових члена; такий поділ можна продовжувати доти, доки не залишаться лише лінійні множники.

Нехай, наприклад, нам потрібно знайти остачу від розкладу многочлена $A(x)$ на 8 лінійних множників $(x - \omega^0)$, ..., $(x - \omega^7)$. Ми починаємо з обчислення остачі від ділення $A(x)$ на добуток $D(x) = (x - \omega^0) \ldots (x - \omega^7)$. Якщо $A(x) = Q(x)D(x) + R(x)$, тоді $A(\omega^j) = R(\omega^j)$, $0 \le j \le 7$, оскільки $D(\omega^j) = 0$ і степінь $R(x)$ менший за степінь $D(x)$, що дорівнює 8. Тепер ми ділимо $R(x)$ на $x^4 - \omega^0$ і отримуємо $S(x)$, поділивши на $x^4 - \omega^4$, отримаємо $T(x)$. $A(\omega^j) = R(\omega^j) = S(\omega^j)$ для $j = 0, 2, 4, 6$ і $A(\omega^j) = R(\omega^j) = T(\omega^j)$ для $j = 1, 3, 5, 7$ і степені S і T менші за 4. Потім ми ділимо $S(x)$ на $x^2 - \omega^0$ і $x^2 - \omega^4$ і отримуємо залишки $U(x)$ та $V(x)$, де $A(\omega^j) = U(\omega^j)$ для $j = 0, 4$ і $A(\omega^j) = V(\omega^j)$ для $j = 2, 6$.

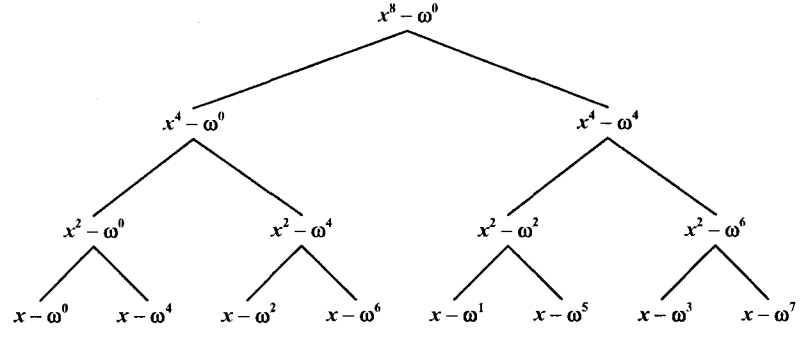


Рис. 11.1. Процес ділення в алгоритмі FFT для степеня 8

Зверніть увагу на те, що кожен дільник має лише два ненульові коефіцієнти, а тому процес поділу буде досить швидким. Продовжуючи у такий самий спосіб, ми нарешті отримаємо 8 значень $A(x) \bmod (x - \omega^j)$ для $j = 0, 1, ..., 7$.

Виконуючи успішні поділи, рухаємось вниз по бінарному дереву (рис. 11.1). В листках дерева отримаємо потрібні коефіцієнти перетворення Фур'є. Порядок їх буде збігатись з появою $x - \omega^j$ в листках дерева, проте їх можна буде впорядкувати в кінці алгоритму. Оскільки многочлени, що розміщені у вузлах дерева, на кожному рівні мають просту форму, ділення їх досить просте. Легко довести, що остаточна часова складність виконання алгоритму буде мати порядок $O(N \log N)$.

Доведіть останнє твердження.

```
Procedure NFFT(var a: array of complex; n: integer);
  var  r, s, t: complex;
       i, j, k, l, ndiv2, pow2, pow2m1, index: integer;
begin
    n div 2 := round(n / 2);
    j := 1;
    for i := 0 to n – 2 do
      begin
        if i < j then
          begin
            t.re := a[j].re; t.im := a[j].im;
            a[j].re := a[i].re;
            a[j].im := a[i].im;
            a[i].re := t.re;
            a[i].im := t.im;
          end;
        k := n div 2;
        while k < j do
          begin
            j := j – k;
            k := round(k / 2);
          end;
        j := j + k;
      end;
    for l := 0 to round(ln(n) / ln(2) – 1) do
      begin
        pow2 := pow(2, l);
        pow2m1 := round(pow2 / 2);
        r.re := 1;
        r.im := 0;
        s.re := cos(pi / pow2m1);
        s.im := sin(pi / pow2m1);
        for j := 0 to pow2m1 – 1 do
```



```
          begin
            i := j;
            while i <= n – 1 do
              begin
                index := i + pow2m1;
                t.re := a[index].re * r.re – a[index].im * r.im;
                t.im := a[index].re * r.im + a[index].im * r.re;
                a[index].re := a[i].re – t.re;
                a[index].im := a[i].im – t.im;
                a[i].re := a[i].re + t.re;
                a[i].im := a[i].re + t.im;
                i := i + pow2;
              end;
                r.re := r.re * s.re – r.im * s.im;
                r.im := s.re * r.im + s.im * r.re;
          end;
      end;
end;
```

У процедурі NFFT (ітераційна версія швидкого перетворення Фур'є) використовують комплексну арифметику, де $\omega = e^{2\pi i/j}$ представлено у тригонометричній формі. Тому всі обчислення в алгоритмі мають наближений характер з несуттєвою втратою точності. Для систем символьних обчислень цей факт не дозволяє використання швидкого перетворення Фур'є в такому вигляді. Похибки обчислень наближеної комплексної арифметики можна обійти, працюючи в скінченному полі.

Нехай p – просте число, менше за розмір машинного слова, і таке, що цілі 0, 1, ..., $p - 1$ містять первинні n-ні корені одиниці. Виконуючи всю арифметику перетворення Фур'є за модулем p, отримаємо результати простої точності. Якщо обрати p простим, тоді цілі 0, 1, ..., $p - 1$ утворять поле, в якому можна виконувати всі арифметичні операції (включаючи ділення). Якщо значення обчислень напрямлятимуться до $p - 1$, тоді точна відповідь буде, коли $x \bmod p = x$, якщо $0 \le x \le p$. У разі, коли одне або більше значень перевищуватимуть $p - 1$, точну відповідь можна отримати, використавши результат теореми про китайський залишок. Детальніше ділення за певним модулем розглядатиметься в підрозділі 11.4.

Потрібно розглянути ще одну задачу: чи можна знайти для заданого N достатню кількість простих чисел певного розміру, які містять N-ні корені. З теореми про скінченне поле $\{0, 1, ..., p - 1\}$ містить первинні N-ні корені, якщо N ділить $p - 1$. Щоб перетворити послідовність розміром $N = 2^m$, слід знайти прості форми $p = 2^e k + 1$, де $m \le e$. Такі прості числа називають *простими числами Фур'є*. Ліпсон довів, що їх існує більше за $\frac{x}{2^{e-1} \ln x}$ і менше за x^e. Тому простих чисел Фур'є достатньо для будь-якого задання в розумних межах.

11.3. Арифметичні операції над цілими числами і поліномами

11.3.1. *Множення і ділення цілих чисел*

Скористаємося такими позначеннями [9]: $M(n)$ – час множення двох цілих чисел розміру n; $D(n)$ – час ділення цілого числа розміру не більше ніж $2n$ на ціле число розміру n; $S(n)$ – час піднесення до квадрату цілого числа розміру n; $R(n)$ – час обернення цілого числа розміру n. Тут час вимірюють кількістю бітових операцій. Можна вважати, що для вищезазначених функцій виконуватиметься властивість, яку наведемо тільки для $M(n)$: $a^2M(n) \ge M(an) \ge aM(n)$ для $a \ge 1$.

«Оберненим» до числа i називатимемо наближення до дробу $1/i$, який має n значущих двійкових розрядів (бітів). Оскільки припускається, що на зсув двійкової коми час не витрачають, можна казати, що «обернене» число до числа i – це частка від ділення $2^{2n-1}/i$. Далі термін «обернене» використовуватимемо саме в цьому сенсі.

Розглянемо [9] пошук послідовних наближень до числа $A = 1/P$, де P ціле. Нехай A_i буде i-м наближенням до $1/P$. Тоді точне значення A можна зобразити у вигляді:

$$A = A_i + \frac{1}{P}\left(1 - A_i P\right). \qquad (11.13)$$

Якщо в (11.13) число A_i взяти як наближення до $1/P$, отримаємо формулу

$$A_{i+1} = A_i + A_i\left(1 - A_i P\right) = 2A_i - A_i^2 P, \qquad (11.14)$$

яку можна використовувати для знаходження $(i + 1)$-го наближення числа A за його i-м наближенням. Якщо ж $A_i P = 1 - S$, то

$$A_{i+1}P = 2A_iP - A_i^2P^2 = 2\left(1-S\right) - \left(1-S\right)^2 = 1 - S^2.$$

Тому ітерація за формулою (11.14) надає квадратичну швидкість збіжності. Якщо ж $S \le 1/2$, то число вірних двійкових розрядів подвоюється в кожній ітерації.

Оскільки число P, за припущенням, містить у собі багато двійкових розрядів, то для перших наближень не обов'язково враховувати всі його цифри, тому що на вірні цифри числа A_{i+1} впливають тільки старші розряди в P. Якщо в A_i перші k цифр праворуч від коми вірні, тоді $2k$ цифри числа A_{i+1} можна знайти за формулою (11.14). Подібне використання наближених значень може, звичайно, вплинути на збіжність, оскільки раніше припускалось, що число P точне.

Для обчислення частки $\lfloor 2^{2n-1}/P \rfloor$, де $P = p_1, p_2, \ldots, p_n$, n – розрядне двійкове ціле з $p_1 = 1$, що дорівнює цілому числу $A = [a_1, \ldots, a_n]$, $a[x]$ – ціле число з двійковою формою запису x (наприклад, [011] = 3), застосуємо формулу (11.14), до якої додамо перенос коми (щоб можна було працювати лише з цілими числами). Отримаємо алгоритм 11.1 обернення цілих чисел [9], який реалізує програма 11.2.


```
Function Inverted(p, k);
{Нехай на вхід алгоритму подають [p1p2... pk] – двійкове ціле число з p1=1;
для зручності вважатимемо, що k – степінь двійки}
begin
  if k = 1 then inverted := 10
    else
      begin
        c := inverted(p / (2^(k/2)), k / 2);
        d ← c(2^(3k/2)) – c^2p;
        a ← d / (2^(k-1));
        for x := 2 step –1 until 0 do
          begin
            p2 := 2^(2k-1);
            if (a + 2^x)p ≤ p2 then
              a := a + 2^x;
          end
      end
  inverted := a;
end;
```

Алгоритм 11.1. Обернення цілих чисел

```
Program Ex_11_2;

{$x+}
 uses crt;
 var  x, k: byte;                                  {k – визначає число бітів}
      r: word;

function BinDec(bin: string): word;    {Конвертує із двійкового задання в десяткове}
begin
  r := 0;
  k := length(bin);
  for x := 1 to k do
    if bin[k – x + 1] = '1' then r := r + (1 shl pred(x));
  BinDec := r;
end;

function DecBin(dec: word): string;    {Конвертує із десяткового задання в двійкове}
 var ch,st: string;
begin
  x := 0;
  st := '';
  repeat
    if (dec mod 2 = 1) then ch := '1' else ch := '0';
```

```
      st := ch + st; dec := dec shr 1;
    until dec = 0;
    DecBin := st;
end;

{Процедура для обернення цілих чисел. Реалізація}
{P = 'p1p2...pk' – послідовність бітів (тобто P = '1010..'), де k – число бітів,
k – степінь 2}
function Inverted(p: word; k: byte): word;
  var  c, d, a, p2: word;
       x: byte;
begin
{1} if k = 1 then
      begin
        Inverted := BinDec('10');
        k := 2
      end
    else
      begin
{2}     c := Inverted(p shr (k shr 1), (k shr 1));
{3}     d := c * (1 shl ((3 * k) shr 1)) - c * c * p;
{4}     a := d shr (k - 1);
{5}     for x := 2 downto 0 do
          begin
            p2 := 1 shl ((k * 2) - 1);
{6}         if (a + (1 shl x)) * p <= p2 then
{7}           a := a + (1 shl x);
          end;
{8}     Inverted := a;
      end;
end;

{Головна програма}
  var  result: word;
begin
    writeln('Input = ', 153, "", decbin(153), "");
    result := Inverted(153, 8);
    writeln('Output = ', result, "",decbin(result),"");
    readkey;
end.
```

Програма 11.2. Обернення цілих чисел

У праці [9] можна знайти доведення наступних теорем, що характеризують часові оцінки роботи алгоритму.

Теорема 11.1. Функція *Inverted* знаходить таке число $[a_0a_1...a_k]$, що $[a_0a_1...a_k] * [p_1p_2...p_k] = 2^{2k-1} - S$ і $0 \le S < [p_1p_2...p_k]$.

Теорема 11.2. Існує така стала c, що $R(n) \le cM(n)$.

Очевидно, що за допомогою функції *Inverted* можна обчислити $1/P$ з n значущими двійковими цифрами, якщо P має стільки ж цифр (у цьому разі не має значення, де розміщена кома).

Тепер наведемо алгоритм піднесення до квадрата за допомогою обернених величин [9], який, маючи на вході *N – розрядне ціле число P в двійковій формі запису*, отримує на виході *двійковий запис числа* P^2, і для якого час $S(n)$, потрібний для піднесення до квадрата цілого числа розміру n, не перевищує за порядком час $R(n)$ обернення цілого числа розміру n.

Метод ґрунтується на тотожності

$$P^2 = \frac{1}{(\frac{1}{P} - \frac{1}{P+1})} - P \qquad (11.15)$$

і містить наступну послідовність кроків алгоритму 11.2.

Begin

1. Застосовуємо функцію *Inverted* для обчислення $A = \lfloor 2^{4n-1}/P \rfloor$, модифікувавши її для довільних цілих чисел (довжина яких не обов'язково дорівнює степеню числа 2). Для цього додаємо $2n$ нулів до P, застосовуємо функцію *Inverted* для обчислення $\lfloor 2^{4n-1}/P2^{2n} \rfloor$ і зсуваємо результат.
2. Аналогічно обчислюємо $B = \lfloor 2^{4n-1}/(P+1) \rfloor$.
3. Покладемо $C = A - B$. Зазначимо, що $C = 2^{4n-1}/(P + P^2) + T$, де $|T| \le 1$. Складова T з'являється тому, що відкидання знаків під час обчислення A і B може призвести до помилки, навіть до 1. Оскільки $2^{2n-2} < P^2 + P < 2^{2n}$, то $2^{2n+1} \ge C \ge 2^{2n-1}$.
4. Обчислюємо $D = \lfloor 2^{4n-1}/C \rfloor - P$.
5. Нехай Q – чотири останніх біти числа P. Збільшуємо чи зменшуємо D на мінімально можливу величину так, щоб останні 4 біти результату дорівнювали останнім 4 бітам в Q^2.

End.

Алгоритм 11.1. Піднесення до квадрата за допомогою обернених величин

У праці [9] доведено, що:

1) *алгоритм 11.1 обчислює* P^2;

2) *знайдеться така стала* c, *що* $S(n) \le cR(n)$;

3) $M(n)$, $R(n)$, $D(n)$, $S(n)$ *дорівнюють з точністю до сталих множників;*

4) *ділення* $2n$*-розрядного двійкового цілого числа на* n*-розрядне можна виконати за час* $O_Б(n\log(n\log(\log n)))$.

11.3.2. *Множення і ділення поліномів*

Всю техніку роботи з числами у попередньому розділі можна перенести на реалізацію арифметичних дій з поліномами від однієї змінної. Тому в цьому розділі функції $M(n)$, $D(n)$, $R(n)$ і $S(n)$ також позначатимуть часові оцінки, які потрібні відповідно для множення, ділення, обернення та піднесення до квадрата поліномів степеня n. Як і раніше, можна вважати, що $a^2M(n) \geq M(an) \geq aM(n)$ для $a \geq 1$, а подібні нерівності справедливі й для інших функцій.

Під «оберненим» до полінома $p(x)$ степеня n розуміють поліном $\lfloor x^{2n}/p(x) \rfloor$. $D(n)$ – це час знаходження полінома $\lfloor s(x)/p(x) \rfloor$, де $p(x)$ має степінь n, а $s(x)$ – степінь, не більший за $2n$. Зазначимо, що можна «змінювати масштаб» поліномів, помноживши або поділивши їх на степені змінної x.

Розглянемо алгоритм обернення поліномів [9], аналогічний алгоритму обернення цілих чисел. Алгоритми для поліномів простіші від аналогічних алгоритмів для цілих чисел в тому розумінні, що в степеневих рядах, на відміну від цілих чисел, немає переносів, і тому не потрібно коригувати молодші значущі розряди.

Входом алгоритму є поліном $p(x)$ степеня $n - 1$, де n, для зручності, є степінь числа 2 (тобто $p(x)$ має $k = 2^t$ членів, де t – деяке ціле число). Основою алгоритму 11.2 обернення поліномів є функція $Inverted\left(\sum_{i=0}^{k-1} a_i x^i\right)$, $a_{k-1} \neq 0$, що обчислює $\left\lfloor \frac{x^{2k-2}}{\sum_{i=0}^{k-1} a_i x^i} \right\rfloor$.

Begin

function Inverted$\left(\sum_{i=0}^{k-1} a_i x^i\right)$;

begin

if k = 1 **then return** $1/a_0$

else

begin

$q(x) \leftarrow \text{Inverted}\left(\sum_{i=0}^{k-1} a_i x^{i-k/2}\right)$;

$r(x) \leftarrow 2 * q(x) * x^{(3/2)*k-2-q(x)*q(x)}\left(\sum_{i=0}^{k-1} a_i x^i\right)$;

return $\lfloor r(x) / \exp(x * \ln(k-2)) \rfloor$

end;

end;

call Inverted($p(x)$).

Алгоритм 11.2. Обернення поліномів

Очевидно, що час роботи алгоритмів обчислення як оберненого цілого, так і оберненого полінома оцінюють подібно до того, якщо розглядати дві міри складності: арифметичну й бітову. Аналогічно можна показати, що й інші оцінки часу, знайдені для алгоритму обчислення оберненого цілого, переносять на поліноми, якщо замість бітових кроків розглядати арифметичні. Отже, функції арифметичної складності $M(n)$ – множення, $D(n)$ – ділення, $R(n)$ – обернення і $S(n)$ – піднесення до квадрата поліномів від однієї змінної будуть рівними між собою з точністю до сталих множників. Звідки [9] поліном степеня $2n$ можна розділити на поліном степеня n за час $O_A(n \log n)$.

Реалізацію у Паскалі алгоритму обернення полінома наведено у програмі 11.3.

```
Program Ex_11_3;

{$A-, B-, D-, E+, F-, G+, K+, I-, L-, N+, O+, P-, R-, S-, Q-, T-, V-, X-, Y-, W+}
{$M 65520, 512, 655000}

 uses crt;
 const max = 128;                                              {max – ↓}
 typepolinom = array [0..max] of real ;
     poli=^polinom;

{Глобальні змінні}
 var p1: pointer;
     q: poli;              {Поліном Q(x), що використовують у функції Inverted}
     r: poli;              {Поліном R(x), що використовують у функції Inverted}
     rez1, rez2, rez3, rez4: poli;                          {Допоміжні поліноми}
     x, y: byte;                                          {Допоміжні змінні для циклів}
     tmp: poli;                                           {Тимчасовий Pointer-поліном}
     pp, pp2: poli;                                       {Вхідний та вихідний поліноми}
     tmpX: poli;                                {Допоміжний поліном для X^(3/2)k–2}
     StpX: byte;                      {Степінь tmpX полінома, тобто StpX = (3/2)k – 2}
     ii, kk: byte;              {Верхній, нижній індекси i та k для виклику в процедури}
     SzQ: byte;                                             {Розмір полінома – Q}
{Наступні 2 змінні необхідні для виправлення BUG'a в Pascal'i}
     PE: array [1..64, 0..max] of real ;
     pe1: byte;

function Neu(var pol: poli; k2: byte): poli;
begin
{Наступні 2 рядки необхідні для виправлення BUG'a в Pascal'i}
   inc(pe1);
   for x := 0 to k2 – 1 do pe[pe1, x] := pol^[x];
   y := 0;
   for x := (k2 shr 1) to pred(k2) do
```

```
        begin
          tmp^[y] := pol^[x];
          inc(y);
        end;
      Neu := tmp;
end;

procedure Umn(var pl1, pl2, rez: poli; k3, k4: byte);
  var  tmp3: polinom;
    begin
      for x := 0 to max do
        tmp3[x] := 0;                    {Очищення змінної для занесення результатів}
      for x := 0 to pred(k3) do
        for y := 0 to pred(k4) do
          tmp3[x + y] := tmp3[x + y] + pl1^[x] * pl2^[y]; {Безпосередньо множення}
      rez^ := tmp3;                      {Занесення результату в тимчасовий tmp поліном}
    end;

{!! Процедура обернення поліномів !!}
function Inverted(pol: poli; i, k: byte): poli;
begin
    if k=1 then
      Inverted^[0]:=1/pol^[0];
    else
      begin
        Q := Inverted(Neu(pol, k), i, (k shr 1));
        SzQ := (k shr 1) – i;                                  {Розмір полінома – Q}
{Наступні два рядки були б непотрібні.., але не тільки Microsoft робить помилки}
        for x := 0 to k – i – 1 do pol^[x] := pe[pe1, x];
        dec(pe1);

{Початок обчислення полінома – r(x)}
        StpX := ((3 * k) shr 1) – 2; {Степінь tmpX полінома, тобто StpX = (3/2)k – 2}
{Створення полінома 2 * X^(3/2)k – 2 + 0 * X + 0 * X + ...}
        for x := 0 to StpX do tmpX^[x] := 0;
        tmpX^[StpX] := 2;
{Множення tmpX полінома на Q}
        Umn(Q, tmpX, rez1, SzQ, StpX + 1);                     {Результат в rez1}
{Підведення Q до степеня 2, Q*Q}
        Umn(Q, Q, rez2, SzQ, SzQ);                             {Результат в rez2}
{Множення rez2 на поліном pol}
        Umn(rez2, pol, rez3, SzQ + SzQ – 1, k – i);            {Результат в rez3}
{Знаходження кінцевого r(x) = rez1 – rez3}
        for x := 0 to max do
          r^[x] := rez1^[x] – rez3^[x];
{Кінець обчислення r(x)}
```



```
{Ділення r(x) на x^(k – 2)}
        y := 0;
        for x := k – 2 to max do
          begin
            rez4^[y] := r^[x];
            inc(y);
          end;
        Inverted := rez4;
      end;
end;

{Головна програма}
begin
    Mark(p1);                  {Маркування початку для видалення динамічних змінних}
    new(pp); new(pp2); new(tmp); new(r); new(q);          {Створення всіх потрібних}
    new(tmpX); new(rez1); new(rez2); new(rez3); new(rez4);                {покажчиків}
    pp^[0] := 4; pp^[1] := 1; pp^[2] := –3; pp^[3] := –1;     {Приклад задання полінома}
    pp^[4] := 2; pp^[5] := 1; pp^[6] := –1; pp^[7] := 1;
    ii := 0;                                                    {Нижній індекс, від 0}
    kk := 8;                                  {Верхній індекс, кількість членів полінома}
    writeln('Вхідний поліном, ступінь ', kk – 1);
    for x := kk – 1 downto 0 do
      if pp^[x] <> 0 then write(pp^[x] : 4 : 0) else write('–' : 4);
    pp2 := Inverted(pp, ii, kk);
    writeln('Вихідний поліном');
    for x := kk – 1 downto 0 do
      if pp2^[x] <> 0 then write(pp2^[x] : 4 : 0) else write('*' : 4);
    release(p1);                                              {Знищити всі покажчики}
end.
```

Програма 11.3. Обернення поліномів

11.4. Модульна арифметика

Для роботи з великими числами в системах кодування, у символьних обчисленнях широко застосовують модульну арифметику. Розглянемо основні принципи таких обчислень згідно з працями [9, 127].

Головна перевага модульного задання чисел полягає в тому, що арифметичні операції можна реалізовувати паралельно з меншими апаратними витратами, оскільки обчислення виконують незалежно для кожного модуля і не потрібні жодні перенесення. Проблеми ефективного ділення цілих чисел і контролю за переповненням і в цьому разі залишаються нездоланними. Незважаючи на це ідеї модульного задання чисел застосовують переважно під час роботи з поліномами, оскільки ділити поліноми, найімовірніше, не знадобиться. Крім того, як ми побачимо в наступному

розділі, обчислення поліномів та їх лишків (за модулем інших поліномів) тісно пов'язані.

11.4.1. *Модульна арифметика цілих чисел*

Нагадаємо, що оператора mod визначають як $x \bmod y = x - y\lfloor x / y \rfloor$, якщо $y \neq 0$, $x \bmod 0 = x$.

Визначимо множину $\{0, 1, \ldots, p-1\}$, де p – просте із поля GF (поля Галояса з p елементами [9]), p є величиною простої точності обчислень. Множина $GF(p)$ утворює поле, в якому операції додавання, віднімання, множення та ділення визначають так:

якщо $a, b \in GF(p)$, тоді

$$(a+b) \bmod p = \begin{cases} a+b, & \text{якщо } a+b < p, \\ a+b-p, & \text{якщо } a+b \geq p; \end{cases}$$

$$(a-b) \bmod p = \begin{cases} a-b, & \text{якщо } a-b \geq 0, \\ a-b+p, & \text{якщо } a-b < 0; \end{cases}$$

$(ab) \bmod p = r$, де r – залишок від ділення добутку ab на p, $ab = qp + r$, якщо $0 \leq r < p$;

$(a/b) \bmod p = (ab^{-1}) \bmod p = r$ – єдиний залишок, коли ab^{-1} ділиться на p, $ab^{-1} = qp + r$, $0 \leq r < p$, де b^{-1} – мультиплікативне обернене до b в $GF(p)$ (для кожного елемента b в $GF(p)$, крім нуля, існує єдиний елемент b^{-1}, такий що $bb^{-1} \bmod p = 1$).

Зрозуміло, що час реалізації операцій додавання, віднімання та множення за mod p, описаних цими формулами, становить $O(1)$.

Для визначення часової оцінки ділення ми маємо вибрати алгоритм для обчислення оберненого множення елементу b з $GF(p)$. За визначенням, для знаходження $x = b^{-1}$ має існувати ціле k, $0 \leq k \leq p$, таке що $bx = kp + 1$.

Алгоритм знаходження інверсії b в $GF(p)$ випливає з узагальнення евклідового алгоритму знаходження найбільшого спільного дільника (НСД). Він використовує відомий факт: $a > b \geq 0$, тоді НСД(a,b) = НСД$(b, a \bmod b)$, інакше НСД$(a,0) = a$. Можливе також знаходження двох цілих x, y, таких що $ax + by$ = НСД(a, b). Нехай a – просте число, $p, b \in GF(p)$, тоді НСД$(p, b) = 1$ (дільники простого числа – 1 і воно саме) і узагальнення обмежується знаходженням цілих x, y, таких що $px + by = 1$. Це означає, що y – інверсне множення b mod p. Відповідний узагальнений евклідовий алгоритм описує функція *Exeuclid*.

```
Function Exeuclid(b, p: integer): integer;
{Для b із GP(p) та p повертає таке x, що bx + kp = 1}
  var w, q, c, d, x, y, e: integer;
```



```
begin
    c := p; d := b; x := 0; y := 1;
    while d <> 1 do
      begin
        q := trunc(c / d);
        e := c - d * q;
        w := x - y * q;
        c := d; d := e; x := y; y := w;
      end;
    if y < 0 then y := y + p;
    Exeuclid := y;
end;
```

Для дослідження часової складності алгоритму *Exeuclid* використовують результати теореми Ламе [9]. З неї випливає, що оцінка $O(\log_{10} p)$ буде часовою оцінкою як алгоритму Евкліда, так і модульного ділення.

Розглянемо детальніше виконання арифметичних операцій над цілими в модульному вигляді, тобто в заданні за допомогою системи класів лишків. Замість того щоб задавати ціле в системі числення з фіксованою основою, його задають лишками за модулями з множини попарно взаємно простих чисел [9].

Так, якщо $p_0, p_1, \ldots, p_{k-1}$ – попарно взаємно прості числа і $p = \prod_{i=0}^{k-1} p_i$, тоді будь-яке ціле число a, $0 \leq a < p$, можна однозначно зобразити множиною його лишків $r_0, r_1, \ldots, r_{k-1}$, де $r_i = a$ за модулем p_i, $0 \leq i < k$. Коли $p_0, p_1, \ldots, p_{k-1}$ – фіксовані, пишуть $a \leftrightarrow (r_0, r_1, \ldots, r_{k-1})$.

Нехай

$$u \leftrightarrow (u_0, u_1, \ldots, u_{k-1}), \quad v \leftrightarrow (v_0, v_1, \ldots, v_{k-1}),$$

тоді

$$u + v \leftrightarrow (w_0, w_1, \ldots, w_{k-1}), \quad \text{де } w_i = (u_i + v_i) \bmod p_i, \tag{11.16}$$

$$u - v \leftrightarrow (x_0, x_1, \ldots, x_{k-1}), \quad \text{де } x_i = (u_i - v_i) \bmod p_i, \tag{11.17}$$

$$uv \leftrightarrow (y_0, y_1, \ldots, y_{k-1}), \quad \text{де } w_i = (u_i v_i) \bmod p_i. \tag{11.18}$$

Оскільки обчислення для кожного модуля можна проводити незалежно, то очевидно, що модульну арифметику можна природно реалізувати у вигляді паралельних обчислень. У ній також ефективно реалізовують роботу з цілими довільної довжини і раціональними операторами. Тому за допомогою модульної арифметики ефективно реалізовують алгоритми маніпуляції з великими числами, що зумовлює її широке використання у теорії кодування.

Приклад 11.1. Нехай $p_0 = 5$, $p_1 = 3$ і $p_2 = 2$. Тоді $6 \leftrightarrow (1, 0, 0)$, оскільки $1 = 6 \bmod 5$, $0 = 6 \bmod 3$ і $0 = 6 \bmod 2$.

Аналогічно $11 \leftrightarrow (1, 2, 1)$.

Однак неясно, як у модульній арифметиці економно виконати ділення. Дійсно, якщо p_i не є простим числом, то між 0 та $p_i - 1$ може опинитись кілька цілих чисел w, що дорівнюють u_i/v_i за модулем p_i в тому розумінні, що $wv_i \equiv u_i \pmod{p_i}$.

У праці [9] можна знайти доведення, що співвідношення (11.16) – (11.18) виконуються і відповідність $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$ є взаємно однозначною. Зрозуміло, що для ефективного використання модульної арифметики потрібні ефективні алгоритми переходу від позиційного зображення до модульного і навпаки. Отже, розглянемо модульну арифметику як трансформаційну техніку спрощення роботи з цілими числами.

Один з методів переходу від позиційного задання цілого числа u до модульного полягає в тому, щоб поділити u на кожне з чисел p_i, $0 \le i < k$. Замість прямого ділення спочатку обчислюємо добутки $p_0, p_1, p_2, p_3, \ldots, p_{k-2}, p_{k-1}$, потім $p_0, p_1, p_2, p_3, \ldots, p_{k-4}, p_{k-3}, p_{k-2}, p_{k-1}$ і т. д. Потім обчислюють лишки за допомогою методу «розділяй і пануй», як ми розглядали раніше. Ділимо й отримуємо лишки u_1 і u_2 числа u за модулями $p_0, \ldots, p_{k/2-1}$ і $p_{k/2}, \ldots, p_{k-1}$ відповідно. Отже, задача обчислення $i \bmod p_i$, $0 \le i < k$ зведена до двох підзадач половинного розміру, а саме $u \bmod p_i = u_i \bmod p_i$ для $0 \le i < k/2$ і $u \bmod p_i = u_2 \bmod p_i$ для $k/2 \le i < k$. Детальніше процес обчислення лишків описує алгоритм 11.3, який реалізує програма 11.4.

```
Begin
  for i := 0 to k – 1 do q_i0 ← p_i;
    for j := 1 to t – 1 do
      begin
        for i := 0 to k – 1 step 2^j do
          q_ij ← q_i,j-1 * q_i+2^(j-1),j-1;
        u_0t ← u;
      end;
  for j := t to 1 step –1 do
    for i := 0 to k – 1 step 2^j do
      begin
        u_i,j-1 ← ОСТАЧА(u_ij / q_i,j-1);
        u_i+2^(j-1),j-1 ← ОСТАЧА(u_ij / q_i+2^(j-1),j-1);
      end;
  for i := 0 to k – 1 do u_i ← p_i0;
end.
```

Алгоритм 11.3. Обчислення лишків

```
Program Ex_11_4;
  const T = 3;
        K = (1 shl T);
  var  q, u: array [0..k, 0..t] of integer;
       p: array [0..k – 1] of integer;
       uu: array [0..k] of integer;
       i, j, u2: integer;
       a: word;

begin
  writeln('Знаходження u mod p(i) E.g. i від 0 до ',pred(k));
  for a := 0 to k – 1 do
    begin
      write('p(', a, ') – ');
      readln(p[a]);
    end;
  write('u = ');
  readln(u2);
  for i := 0 to k – 1 do q[i, 0] := p[i];                    {Початок обчислень}
    for j:=1 to t – 1 do
      begin
        i:=0;
        repeat                                               {q_ij ← q_ij-1 * q_i+2^j_j-1}
          q[i, j] := q[i, j – 1] * q[i + (1 shl (j – 1)), j – 1];
          i := i + (1 shl j);
        until i > k – 1;
      end;
  u[0, t] := u2;                                             {u_0t ← u}
  for j := t downto 1 do
    begin
      i := 0;
      repeat
        u[i, j – 1] := u[i, j] mod q[i, j – 1];              {u_i,j-1 ← ОСТАЧА(u_ij / q_i,j-1)}
                                                             {u_i+2^(j-1)_j-1 ← ОСТАЧА(u_ij / q_i+2^(j-1)_j-1)}
        u[i + (1 shl (j – 1)), j – 1] := u[i, j] mod q[i + (1 shl (j – 1)), j – 1];
        i := i + (1 shl j);
      until i > k – 1;
    end;
  for i := 0 to k – 1 do                                     {Результат}
    begin
      uu[i] := u[i, 0];
      writeln('u(', i, ') = ', u[i, 0], ' (', u2, ' mod ', p[i], ' = ', u[i, 0], ')');
    end;
end.
```

Програма 11.4. Обчислення лишків

На вхід програми подають модулі $p_0, p_1, ..., p_{k-1}$ і таке ціле число u, що

$$0 \le u < p = \prod_{i=0}^{k-1} p_i.$$

На виході отримуємо числа u_i, $0 \le i < k$, такі що $u_i = u \bmod p_i$ згідно з наступними припущеннями і роздумами.

Припустимо, що k – степінь числа 2, наприклад $k = 2^t$. Зрозуміло, як що це не так, до входу можна додати зайві модулі, що дорівнюють 1, щоб зробити k степенем числа 2. Починаємо з обрахування визначених модулів. Нехай $0 \le j < t$, число i кратне числу 2^j і $0 \le i < k$; вважатимемо $q_{ij} = \prod_{m=i}^{i+2^j-1} p_m$. У такий спосіб, $q_{i0} = p_i$ і $q_{ij} = q_{i,j-1}\, q_{i+2^{j-1},j-1}$.

Спочатку обрахуємо числа q_{ij}, потім знаходимо залишки u_{ij} від ділення u на кожне із чисел q_{ij}. Шуканою відповіддю будуть числа u_{i0}.

Можна довести [9], що алгоритм обрахунку лишків працює вірно і витрачає $O_Б(M(br)\log k)$ часу, якщо для зображення кожного з чисел p_i достатньо b бітів.

Розглянемо обернене перетворення. Нехай після виконання деяких потрібних нам арифметичних операцій з цілими r-кортежами отримаємо деякий r-місний кортеж $(c_1, ..., c_r)$. Далі потрібно перетворити його з модульної форми. Це можна зробити, використовуючи теорему Ейлера [124].

Теорема 11.3 (теорема китайського залишку). Нехай $p_1, ..., p_r$ – додатні цілі, такі що не існує двох цілих, які б мали спільний множник. Нехай $p = p_1, ..., p_r$ і нехай $b, a_1, ..., a_r$ – цілі. Тоді існує лише одне ціле a, що задовольняє умовам:

$$b \le a < b + p \text{ і } a = a_i \pmod{p_i} \text{ для } 1 \le i \le r.$$

Можна довести [9], що часова складність перетворення числа в модульну форму і обчислення r результатів становить $O(\log a)$. Тому цінність цього методу залежить від того, як швидко можна виконати обернене перетворення за допомогою алгоритму знаходження китайського залишку.

Припустимо, що маючи $a \bmod p$ і $c \bmod q$, потрібно знайти таке єдине c, щоб $c \bmod p = a$ і $c \bmod q = b$. Значення c, яке задовольняє цим вимогам, буде $c = (b - a)sp + a$, де s задовольняє умові $ps \bmod q = 1$. Функція *Onestepcra* використовує функцію *Exeuclid* і арифметичний модуль p для обчислення одного кроку алгоритму знаходження китайського залишку формули, що було щойно описано.

```
Function Onestepcra(a, p, b, q: integer): integer;
  var s, t, pb, r, u: integer;
begin
    t := a mod q;
    pb := p mod q;
    s := Exeuclid(pb, q);
    u := ((b – t) * s) mod q;
    Onestepcra := u * p + a;
end;
```


Часову оцінку функції *Onestepcra* визначають викликом функції *Exeuclid*, що потребує $O(\log q)$ дій.

Для реалізації теореми китайського залишку з r модулями слід використати функцію *Onestepcra* $r - 1$ раз в описаний спосіб. Тому загальний час обчислення буде $O(r\log q) = O(r^2)$.

Розглянемо ще один підхід до розв'язання задачі перетворення модульного цілого числа в його позиційне задання – використання цілочислового аналога інтерполяційної формули Лагранжа для поліномів. Нехай дано попарно взаємно прості модулі $p_0, p_1, ..., p_{k-1}$ і лишки $u_0, u_1, ..., u_{k-1}$, де $k = 2^t$; треба знайти таке ціле число u, що $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$.

Відомо [127], що $c_i = p/p_i$, де $p = \prod p_i$, і якщо припустити, що $d_i = c_i^{-1} \bmod p_i$, тобто $d_i c_i \equiv 1 (\bmod p_i)$ і $0 \le d_i < p_i$, тоді

$$u \equiv \sum_{i=0}^{k-1} c_i d_i u_i \pmod{p}. \qquad (11.19)$$

Звідки нашу задачу перетворення можна трансформувати в задачу організації ефективного обчислення (11.19).

Пропонують кілька шляхів такої організації.

Спочатку розглянемо варіант китайської задачі про залишки з «попередньою обробкою». Якщо частина вхідних даних фіксована для декількох задач, то всі величини, залежні лише від цієї фіксованої частини, можна обчислити заздалегідь і вважати частиною входу. Алгоритм, в якому зроблено такі попередні обчислення, називають алгоритмом з попередньою обробкою даних.

Можна довести, що на часову складність алгоритму попередня обробка суттєво не впливає.

Доведіть цей факт самостійно.

Для китайського алгоритму з попередньою обробкою даних входом слугуватимуть лишки, модулі p_i й обернені до них числа d_i. Аналізуючи вираз (11.19), помічаємо, що складові $c_i d_i u_i$ за різних i мають багато спільних співмножників. Наприклад, $c_i d_i u_i$ має співмножником число $p_0 p_1 \ldots p_{k/2-1}$, якщо $i \ge k/2$, і число $p_{k/2} p_{k/2+1} \ldots p_{k-1}$, якщо $i < k/2$. Тому (11.19) можна записати у вигляді:

$$u = \left(\sum_{i=0}^{k/(2-1)} c_i' d_i u_i \right) \times \prod_{i=k/(2+1)}^{k-1} p_i + \left(\sum_{i=k/(2+1)}^{k-1} c_i'' d_i u_i \right) \times \prod_{i=0}^{k/(2-1)} p_i,$$

де c_i' – добуток $p_0 p_1 \ldots p_{k/2-1}$ без p_i, а c'' – добуток $p_{k/2} p_{k/2+1} \ldots p_{k-1}$ без p_i. Це спостереження підводить до застосування методу «розділяй і пануй».

Знаходимо добуток $q_{ij} = \prod_{m=i}^{i+2^j-1} p_m$ (як в алгоритмі обчислення лишків),

а потім – цілі числа $s_{ij} = \prod_{m=i}^{i+2^j+1} \frac{q_{ij} d_m u_m}{p_m}$. Якщо $j = 0$, то $s_{i0} = d_i u_i$. Якщо $j > 0$, тоді обчислюємо s_{ij} за формулою $s_{ij} = s_{i,j-1} q_{i+2,j-1} + s_{i+2,j-1} q_{i,j-1}$.

Нарешті, отримуємо $s_{0t} = u$, що і треба для (11.19).

Отже, якщо на вхід подати попарно взаємно прості модулі $p_0, p_1, ..., p_{k-1}$, де $k = 2^t$ для деякого t, множину «обернених» до них чисел $d_0, d_1, ..., d_{k-1}$, де $d_i = (p/p_i)^{-1} \bmod p_i$, де $p = \prod_{i=0}^{k-1} p_i$, послідовність лишків $(u_0, u_1, ..., u_{k-1})$ і провести обчислення згідно з алгоритмом 11.4, тоді на виході матимемо єдине число u, $0 \le u < p$, таке що $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$.

```
Begin
   for i := 0 to k – 1 do q[i0] ← p[i];
   for j := 1 to t – 1 do
     for i := 0 to k – 1 step 2^j do
        q[ij] ← q[i,j–1] * q[i+2^j,j–1];
   for i := 0 to k – 1 do s[i0] ← d[i] * u[i];
   for j := 1 to t do
     for i := 0 to k – 1 step 2^j do
        s[ij] ← s[i,j–1] * q[i+2^j,j–1] + s[i+2^(j–1),j–1] * q[i,j–1];
   writeln(s[ot] mod q[ot])                    {Остачу від ділення s[ot] на q[ot]}
end.
```

Алгоритм 11.4. Обчислення цілого числа за його модульним зображенням

Реалізацію у Паскалі швидкого китайського алгоритму з попередньою обробкою даних (алгоритм 11.4) наведено в програмі 11.5.

```
Program Ex_11_5;
  uses Crt;
  const T2;
        K(1 shl T);
  var  q, u, s: array [0..k,0..t] of integer;
       p, d: array [0..k – 1] of integer;
       uu: array [0..k] of integer;
       i, j, u2: integer;
       e: longint;
       a: word;

begin
   writeln('K є ',k);
   writeln('Введіть p0, p1, ..., p(k – 1)');
   for a := 0 to k – 1 do
      begin
         write(' p(', a, ') – ');
         readln(p[a]);
      end;
   writeln('Введіть числа інвертування d0, d1, ..., d(k – 1)');
   for a := 0 to k – 1 do
      begin
         write(' d(', a, ') – ');
         readln(d[a]);
      end;
   writeln('Введіть u0, u1, ..., u(k – 1)');
   for a := 0 to k – 1 do
      begin
         write(' u(', a, ') – ');
         readln(uu[a]);
      end;
{SIDE I – Використано модуль з попередньої задачі}
   for i := 0 to k – 1 do q[i,0] := p[i];
   for j: = 1 to t do
      begin
         i := 0;
         repeat
            q[i,j] := q[i,j – 1] * q[i + (1 shl (j – 1)), j – 1];
            i := i + (1 shl j);
         until i > k – 1;
      end;
{Кінець SIDE I}
{Нова частина !!!}
   for i := 0 to k – 1 do s[i,0] := d[i] * uu[i];
   for j := 1 to t do
    begin
     i := 0;
     repeat                                  {s[ij] ← s[i,j–1] * q[i+2^j,j–1] + s[i+2^(j–1),j–1] * q[i,j–1]}
        s[i,j] := s[i,j – 1] * q[i + (1 shl (j – 1)),j – 1] + s[i + (1 shl(j – 1)),j – 1] * q[i,j – 1];
        i := i + (1 shl j);
     until i >= k – 1;
    end;
   writeln(s[0,t], ' mod ', q[0,t], ' = ', s[0,t] mod q[0,t]);
end.
```

Програма 11.5. Реалізація швидкого китайського алгоритму з попередньою обробкою даних

Методом математичної індукції [9] можна показати, що алгоритм працює правильно. Там же можна знайти і доведення теореми.

Теорема 11.4. Нехай маємо k попарно взаємно простих цілочислових модулів $p_0, p_1, ..., p_{k-1}$ і лишки $(u_0, u_1, ..., u_{k-1})$.

Якщо кожне з чисел p_i містить не більше ніж b бітів, то існує алгоритм з попередньою обробкою даних обрахування числа u, для якого $0 \le u < p = \prod_{i=0}^{k-1} p_i$ і $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$, за час $O_Б(M(bk)\log k)$, де $M(n)$ час множення двох n-бітових чисел.

З цієї теореми безпосередньо випливає, що китайський алгоритм з попередньою обробкою даних, застосований до k модулів, де по b бітів у кожному, працює не більше ніж $O_Б(bk\log(k\log(bk\log(\log bk))))$ часу.

11.4.2. *Модульна арифметика поліномів*

Результати, аналогічні результатам для цілих чисел, дійсні й для поліномів [9]. Нехай $p_0, ..., p_{k-1}$ – поліноми і $p = \prod_{i=0}^{k-1} p_i$. Тоді кожен поліном u можна представити послідовністю $u_0, u_1, ..., u_{k-1}$ залишків від ділення u на кожен поліном p_i. Поліном u_i – єдиний, для якого $CT(u_i) < CT(p_i)$ і $u = p_i q_i + u_i$ для деякого полінома q_i. У такій ситуації ми напишемо $u_i = u \bmod p_i$, що абсолютно аналогічно модульному запису цілих чисел.

Можна показати [127], що відповідність $u \leftrightarrow (u_0, u_1, ..., u_{k-1})$ взаємно однозначна, якщо поліноми p_i попарно взаємно прості і степінь полінома u менший за степінь полінома p, тобто SIZE(u) < SIZE(p). Описаний в розділі алгоритм обчислення лишків можна застосувати і до поліномів, якщо як p_i взяти поліноми, а не цілі числа. Замість b (кількість бітів у кожному p_i) слід розглядати найбільший степінь поліномів p_i. Очевидно, складність вимірюють числом арифметичних, а не бітових операцій. Доведено [9], що в цьому разі можна знайти лишки полінома $u(x)$ відносно поліномів $p_0, p_1, ..., p_{k-1}$ за час $O_A(M(dk)\log k)$, де d – верхня межа степенів поліномів p_i і степінь полінома u менший степеня полінома $\prod_{i=0}^{k-1} p_i$.

Доведіть, що для знаходження лишків полінома відносно поліномів $p_0, p_1, ..., p_{k-1}$ необхідно не більше $O_A(dr\log(k\log dr))$ часу, де d – верхня межа степенів поліномів p_i і степінь полінома u менший за степінь полінома $\prod_{i=0}^{k-1} p_i$.

11.5. Найбільші спільні дільники

Нагадаємо визначення найбільшого спільного дільника. Нехай a_0 і a_1 – додатні цілі числа. Додатне ціле число g називають *найбільшим спільним дільником* чисел a_0 і a_1 (його часто позначають через НСД(a_0, a_1), якщо g ділить a_0, a_1, а довільний спільний дільник чисел a_0, a_1 ділить g.

Зрозуміло, що для додатних цілих чисел a_0 і a_1 таке число g єдине. Наприклад, НСД(54, 36) = 18.

Ми вже розглядали алгоритм Евкліда для обчислення НСД(a_0, a_1). Ідея алгоритму полягає в обчисленні послідовності залишків $a_0, a_1, ..., a_k$, де $i < k$ – ненульовий залишок від ділення a_{i-2} на a_{i-1} і a_k ділить без остачі a_{k-1} (тобто $a_{k+1} = 0$). За цих умов НСД(a_0, a_1) = a_k. Ми також розглядали розширений алгоритм Евкліда, який знаходив не тільки найбільший спільний дільник чисел a_0 і a_1, але також і цілі числа x і y, для яких $a_0x + a_1y$ = НСД(a_0, a_1). Ці алгоритми мають непогані часові характеристики, але хотілося б мати ще ліпші. Тому розглянемо ще один відомий підхід [9] до знаходження найбільшого спільного дільника для поліномів – асимптотично швидкий алгоритм.

Нехай a_0 і a_1 – цілі числа і $a_0, a_1, \ldots, a_k$ – відповідна послідовність залишків. Нехай $q_i = \lfloor a_{i-1} / a_i \rfloor$, $1 \le i \le k$. Визначимо (2 × 2)-матрицю $R_{ij}^{(a_0, a_1)}$ (або R_{ij}, якщо зрозуміло, які a_0 і a_1 використовують) для $0 \le i \le j \le k$ такими умовами:

1) $R_{ij} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ для $i \ge 0$,

2) якщо $i > j$, то $R_{ij} = \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & -q_{i-1} \end{bmatrix} \cdot \ldots \cdot \begin{bmatrix} 0 & 1 \\ 1 & -q_{i+1} \end{bmatrix}$.

Методом математичної індукції можна довести дві цікаві властивості цих матриць:

$$\text{(а)} \quad \begin{bmatrix} a_j \\ a_{j+1} \end{bmatrix} = R_{ij} \begin{bmatrix} a_j \\ a_{j+1} \end{bmatrix} \qquad \text{для } i < j < k,$$

$$\text{(б)} \quad R_{0j} = \begin{bmatrix} x_j & y_j \\ x_{j+1} & y_{j+1} \end{bmatrix} \qquad \text{для } 0 \le j < k,$$

де числа a_i, x_i, y_i визначають розширеним алгоритмом Евкліда.

Для однозначності домовимося, якщо $a_0(x)$ та $a_1(x)$ – два полінома, тоді $CT(a_1(x)) < CT(a_0(x))$. Виконання цієї умови легко домогтися. Якщо $CT(a_0) = CT(a_1)$, то замінюємо поліноми a_0 і a_1 поліномами a_1 і $a_0 \bmod a_1$, тобто другим і третім членами послідовності залишків, і працюємо з ними.

Спочатку побудуємо алгоритм, що знаходить останній член послідовності залишків, степінь яких більший ніж половина степеня полінома a_0.

Нехай $l(i)$ – єдине ціле число, для якого $CT(a_{l(i)}) > i$ і $CT(a_{l(i)+1}) \le i$. Якщо a_0 має степінь n, то $l(i) \le n - i - 1$, за припущенням, що $CT(a_1) < CT(a_0)$, оскільки $CT(a_i) \le CT(a_{i-1}) - 1$ для всіх $i \le 1$. Для цього вводять рекурсивний алгоритм ННСД (напівНСД), який за поліномами a_0 і a_1, такими що $CT(a_0) < CT(a_1)$, формує матрицю R_{0j}, де $j = l(n/2)$, тобто a_0 – останній член послідовності залишків, степінь якого більший ніж половина степеня полінома a_0.

В основу алгоритму 11.5 знаходження ННСД покладено той факт, що частки від ділення поліномів степенів d_1 і d_2, $d_1 > d_2$ залежать тільки від $2(d_1 - d_2) + 1$ старших членів діленого і $d_1 - d_2 + 1$ старших членів дільника.

Процедура ННСД (a_0, a_1);

Якщо $CT(a_0)/2 \ge CT(a_1)$, **тоді повернути відповідь** $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

інакше

почати

нехай $a_0 = b_0 x^m + c_0$, де $m = \lfloor CT(a_0)/2 \rfloor$ і $CT(c_0) < m$;

нехай $a_1 = b_1 x^m + c_1$, де $CT(c_1) < m$;

{b_0 і b_1 – старші члени поліномів a_0, a_1}

Маємо: $CT(b_0) = \lceil CT(a_0)/2 \rceil$ *і* $CT(b_0) - CT(b_1) = CT(a_0) - CT(a_1)$*}*

$R \leftarrow$ ННСД(b_0, b_1);

$\begin{bmatrix} d \\ e \end{bmatrix} \leftarrow R \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}$;

$f \leftarrow d$ **mod** e;

{e і f – сусідні члени послідовності залишків, їх степінь не перевищує $\lceil 3m/2 \rceil$*, тобто 3/4 степеня полінома* a_0*}*

нехай $e = g_0 x^{\lfloor m/2 \rfloor} + h_0$, де $CT(h_0) < \lfloor m/2 \rfloor$;

нехай $f = g_1 x^{\lfloor m/2 \rfloor} + h_1$, де $CT(h_1) < \lfloor m/2 \rfloor$;

{степені поліномів g_0 і g_1 не перевищують $m + 1$}

$S \leftarrow$ ННСД(g_0, g_1);

$q \leftarrow \lfloor d/e \rfloor$;

повернути відповідь $S * \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} * R$

закінчити

закінчити.

Алгоритм 11.5. Знаходження ННСД

У праці [9] доведено наступні цікаві факти щодо розглянутого нами алгоритму ННСД.

1) Якщо $a_0(x)$ і $a_1(x)$ – такі поліноми, що $CT(a_0) = n$ і $CT(a_1) < n$, тоді

$$\text{ННСД}(a_0, a_1) = R_{0,\, l(n/2)};$$

2) Процедура ННСД виконується за $O_A(M(n)\log n)$ кроків, якщо її аргументи мають степені не більші за n; тут $M(n)$ – час множення двох поліномів степеня n.

Один із варіантів реалізації алгоритму знаходження ННСД наведено в програмі 11.6.

```
Program Ex_11_6;
  uses crt;
  const stupin = 5;
  type polinom = array [-1..stupin] of real;
       matr = array [1..2, 1..2] of polinom;
       vect = array [1..2] of polinom;
  var  p1, p2, p, tempo: polinom;
       norec, i, j, k: shortint;
       t: matr;

{Початок розділу допоміжних процедур}
function CT(a: polinom): shortint;          {Процедура визначення степеня полінома}
  var  i: shortint;
begin
    a[-1] := 1;
    i := stupin;
    while (a[i] = 0) do dec(i);
    CT := i;
end;

procedure Deviz(p1, p2: polinom;var p, pl: polinom);          {Ділення полінома p1}
  var  i, j, k, st1, st2: shortint;                           {за модулем p2}
       otn: real;
begin
    st1 := CT(p1);
    st2 := CT(p2);
    if st2 > st1 then
      begin
        writeln('некоректне ділення ');
        halt(0);
      end
    else
      begin
        for i := 0 to stupin do pl[i] := p1[i];
        for i := 0 to stupin do p[i] := 0;
        for i := st1 downto st2 do
          begin
            otn := pl[i] / p2[st2];
            p[i - st2] := otn;
            for j := st2 downto 0 do
```

```
                pl[i + j – st2] := pl[i + j – st2] – p2[j] * otn;
              end;
          end;
end;

procedure mnozhpol(p1, p2: polinom; var p: polinom);      {Множення двох поліномів}
  var  stp2, stp1, i, j: shortint;
begin
    stp1 := CT(p1);
    stp2 := CT(p2);
    if (stp1 + stp2 > stupin) then
      writeln('Некоректне множення 1');
    for i := 0 to stupin do p[i] := 0;
    if (stp1 + stp2 = 0) then p[0] := p1[0] * p2[0];
    if (stp1 + stp2 > 0) then
      for i := 0 to stp1 do
        for j := 0 to stp2 do
          p[i + j] := p[i + j] + p1[i] * p2[j];
end;

procedure mnozhvect(r: matr; p: vect; var rez: vect);          {Множення матриці 2×2}
  var  i, j, k: shortint;                                      {на вектор 1×2}
       tmp: polinom;
begin
    for i := 1 to 2 do
      for k := 0 to stupin do
        rez[i,k] := 0;
    for i := 1 to 2 do
      for j := 1 to 2 do
        begin
          mnozhpol(r[i,j], p[j], tmp);
          for k := 0 to stupin do
            rez[i,k] := rez[i,k] + tmp[k]
        end;
end;

procedure mnozhmatr(r, p: matr; var rez: matr);          {Множення двох матриць 2×2}
  var  i, j, k, t: shortint;
       tmp: polinom;
begin
    for i := 1 to 2 do
      for j := 1 to 2 do
        for t := 0 to stupin do
          rez[i,j,t] := 0;
    for i := 1 to 2 do
      for j := 1 to 2 do
```



```
        for k := 1 to 2 do
          begin
            mnozhpol(r[i, k], p[k, j], tmp);
            for t := 0 to stupin do
              rez[i, j, t] := rez[i, j, t] + tmp[t];
          end;
end;
{Кінець розділу допоміжних процедур}

{Початок головної процедури}
procedure nnsd(a0, a1: polinom; var vidp: matr);
  var  m, i, j, k, stb0, stb1, sta0, sta1, stg1, stg2: shortint;
       b0, b1, d, e, f, q, tempo: polinom;
       tmp, tmp1, g: vect;
       r, s, temp, temp1: matr;
begin
    for i := 1 to 2 do
      for j := 1 to 2 do
        for k := 0 to stupin do
          vidp[i, j, k] := 0;
    sta0 := CT(a0);
    sta1 := CT(a1);
    if sta1 <= sta0/2 then {Якщо CT(a0)/2 ≥ CT(a1), тоді повернути відповідь [1 0; 0 1]...}
      for i := 1 to 2 do
        vidp[i, i, 0] := 1
    else
      begin
        m := sta0 div 2;          {Нехай a0 = b0x^m + c0, де m = ⌊CT(a0)/2⌋ і CT(c0) < m}
        stb0 := (sta0 mod 2) + (sta0 div 2);
        for i := 0 to stb0 do
          b0[i] := a0[i + sta0 – stb0];
        for i := stb0 + 1 to stupin do
          b0[i] := 0;
        stb1 := stb0 – (sta0 – sta1);          {Нехай a1 = b1x^m + c1, де CT(c1) < m}
        for i := 0 to stb1 do
          b1[i] := a1[i + sta1 – stb1];
        for i := stb1 + 1 to stupin do
          b1[i] := 0;
        nnsd(b0, b1, r);                                {R ← НСД(b0, b1)}
        for i := 0 to stupin do
          tmp1[1,i] := a0[i];
        for i := 0 to stupin do
          tmp1[2, i] := a1[i];
```

```
      mnozhvect(r, tmp1, tmp);                          {[d; e] ← R[a0; a1]}

      for i := 0 to stupin do
        d[i] := tmp[1, i];
      for i := 0 to stupin do
        e[i] := tmp[2, i];
      deviz(d, e, tempo, f);                            {f ← d mod e;}
      stg1 := CT(e) – (m div 2);
      for i := 0 to stg1 do                             {Нехай e = g0x^⌊m/2⌋ + h0, де CT(h0) < ⌊m/2⌋}
        g[1, i] := e[i + m div 2];
      for i := stg1 + 1 to stupin do
        g[1, i] := 0;
      stg2 := CT(f) – (m div 2);
      for i:=0 to stg2 do                               {Нехай f = g1x^⌊m/2⌋ + h1, де CT(h1) < ⌊m/2⌋}
        g[2, i] := f[i + m div 2];
      for i := stg2 + 1 to stupin do
        g[2, i] := 0;
      nnsd(g[1], g[2], s);                              {S ← ННСД(g0, g1)}
      deviz(d, e, q, tempo);                            {q ← ⌊d / e⌋;}
      for i := 1 to 2 do
        for j := 1 to 2 do
          temp1[1, 1, 0] := 0;
      for k := 1 to stupin do
        temp1[1, 1, k] := 0;
      temp1[1, 2, 0] := 1;
      for k := 1 to stupin do
        temp1[1, 2, k] := 0;
      temp1[2, 1, 0] := 1;
      for k := 1 to stupin do
        temp1[2, 1, k] := 0;
      for i := 0 to stupin do
        temp1[2, 2, i] := q[i] * (–1);

      mnozhmatr(s, temp1, temp);        {Повернути відповідь S·[0 1; 0 1]·R}

      mnozhmatr(temp, r, vidp);
    end;
end;
{Кінець розділу процедур}

begin
  clrscr;
  norec := 0;
  for i := 0 to 5 do
    p1[i] := 1;
  p2[0] := –7; p2[1] := –1; p2[2] := 3; p2[3] := –2; p2[4] := 1;
  nnsd(p1, p2, t);
  writeln('Відповідь');
  for i := 1 to 2 do
    for j := 1 to 2 do
      begin
        for k := 3 downto 0 do
          write(t[i, j, k] : 10 : 6);
        writeln;
      end;
  repeat until keypressed;
end.
```

Програма 11.6. Алгоритм знаходження ННСД

У повному алгоритмі знаходження найбільших дільників використовують процедуру ННСД, яка обчислює $R_{0,n/2}$ потім $R_{0,3n/4}$, $R_{0,7n/8}$ і т. д., де n – степінь входу. Не важко довести [9], що НСД двох поліномів степеня, не вищого за n, можна обчислити за час $O_A(n\log^2 n)$. Детальний опис цього процесу наведено в програмі 11.7.

```
program Ex_11_7;
  uses crt;
  const stupin = 6;
  type polinom = array [–1..stupin] of real;
       matr = array [1..2,1..2] of polinom;
       vect = array [1..2] of polinom;
  var  p1, p2, p, tempo: polinom;
       norec, i, j, k: shortint;
       t: matr;

{Початок розділу допоміжних процедур}
function CT(a: polinom): shortint;          {Процедура визначення степеня полінома}
  var  i: shortint;
begin
    a[–1] := 1;
    i := stupin;
    while (a[i] = 0) do dec(i);
    CT := i;
end;

procedure Deviz(p1, p2: polinom; var p, pl: polinom);     {Ділення полінома p1}
  var  i, j, k, st1, st2: shortint;                       {за модулем p2}
       otn: real;
begin
    st1 := CT(p1);
    st2 := CT(p2);
```

```
    if st2 > st1 then
      begin
        writeln('incorrect devizion');
        for k := 0 to stupin do
          pl[k] := 0;
      end
    else
      begin
        for i := 0 to stupin do
          begin
            pl[i] := p1[i];
            p[i] := 0;
          end
        for i := st1 downto st2 do
          begin
            otn := pl[i] / p2[st2];
            p[i – st2] := otn;
            for j := st2 downto 0 do
            pl[i + j – st2] := pl[i + j – st2] – p2[j] * otn;
          end;
        for i := 0 to stupin do if pl[i] < 1E–9 then pl[i] := 0;
      end;
end;

procedure mnozhpol(p1, p2: polinom; var p: polinom);      {Множення двох поліномів}
  var  stp2, stp1, i, j: shortint;
begin
    stp1 := CT(p1);
    stp2 := CT(p2);
    if (stp1 + stp2 > stupin) then
      writeln('incorrect multiplication 1');
    for i := 0 to stupin do
      p[i] := 0;
    if (stp1 + stp2 = 0) then
      p[0] := p1[0] * p2[0];
    if (stp1 + stp2 > 0) then
      for i := 0 to stp1 do
        for j := 0 to stp2 do
          p[i + j] := p[i + j] + p1[i] * p2[j];
end;

procedure mnozhvect(r: matr; p: vect; var rez: vect);      {Множення матриці 2×2}
  var  i, j, k: shortint;                                  {на вектор 1×2}
       tmp: polinom;
begin
    for i := 1 to 2 do
```



```
      for k := 0 to stupin do
        rez[i,k] := 0;
    for i := 1 to 2 do
      for j := 1 to 2 do
        begin
          mnozhpol(r[i,j], p[j], tmp);
          for k := 0 to stupin do
            rez[i,k] := rez[i,k] + tmp[k]
        end;
end;

procedure mnozhmatr(r, p: matr; var rez: matr);      {Множення двох матриць 2×2}
  var  i, j, k, t: shortint;
       tmp: polinom;
begin
    for i := 1 to 2 do
      for j := 1 to 2 do
        for t := 0 to stupin do
          rez[i,j,t] := 0;
    for i := 1 to 2 do
      for j := 1 to 2 do
        for k := 1 to 2 do
          begin
            mnozhpol(r[i,k], p[k,j], tmp);
            for t := 0 to stupin do
              rez[i,j,t] := rez[i,j,t] + tmp[t];
          end;
end;
{Кінець розділу допоміжних процедур}

procedure nnsd(a0, a1: polinom; var vidp: matr);      {процедура ННСД}
  var  m, i, j, k, stb0, stb1, sta0, sta1, stg1, stg2: shortint;
       b0, b1, d, e, f, q, tempo: polinom;
       tmp, tmp1, g: vect;
       r, s, temp, temp1: matr;
begin
    for i := 1 to 2 do
      for j := 1 to 2 do
        for k := 0 to stupin do
          vidp[i,j,k] := 0;
    sta0 := CT(a0);
    sta1 := CT(a1);
{1} if sta1 <= sta0 / 2 then
      for i := 1 to 2 do
        vidp[i,i,0] := 1
    else
```

```
    begin
      m := sta0 div 2;
{2}   stb0 := (sta0 mod 2) + (sta0 div 2);
      for i := 0 to stb0 do
        b0[i] := a0[i + sta0 – stb0];
      for i := stb0 + 1 to stupin do
        b0[i] := 0;
{3}   stb1 := stb0 – (sta0 – sta1);
      for i := 0 to stb1 do
        b1[i] := a1[i + sta1 – stb1];
      for i := stb1 + 1 to stupin do
        b1[i] := 0;
{4}   nnsd(b0, b1, r);
      for i := 0 to stupin do
        tmp1[1,i] := a0[i];
      for i := 0 to stupin do
        tmp1[2,i] := a1[i];
{5}   mnozhvect(r, tmp1, tmp);
      for i := 0 to stupin do
        d[i] := tmp[1,i];
      for i := 0 to stupin do
        e[i] := tmp[2,i];
{6}   deviz(d, e, tempo, f);
      stg1 := CT(e) – (m div 2);
{7}   for i := 0 to stg1 do
        g[1,i] := e[i + m div 2];
      for i := stg1 + 1 to stupin do
        g[1,i] := 0;
      stg2 := CT(f) – (m div 2);
{8}   for i := 0 to stg2 do
        g[2,i] := f[i + m div 2];
      for i := stg2 + 1 to stupin do
        g[2,i] := 0;
{9}   nnsd(g[1], g[2], s);
{10}  deviz(d, e, q, tempo);
      for i := 1 to 2 do
        for j := 1 to 2 do
          temp1[1, 1, 0] := 0;
      for k := 1 to stupin do
        temp1[1, 1, k] := 0;
      temp1[1, 2, 0] := 1;
      for k := 1 to stupin do
        temp1[1, 2, k] := 0;
      temp1[2, 1, 0] := 1;
      for k := 1 to stupin do
        temp1[2, 1, k] := 0;
```



```
      for i := 0 to stupin do
        temp1[2, 2, i] := q[i] * (–1);
{11}  mnozhmatr(s, temp1, temp);
      mnozhmatr(temp, r, vidp);
    end;
end;

{Початок головної процедури}
procedure nsd(a0, a1: polinom; var vidp: polinom);
  var i, j, k, st: shortint;
      c, b0, b1, q, tempo, tempo1: polinom;
      tmp, tmp1: vect;
      r: matr;
      ch: char;
begin
    deviz(a0, a1, tempo1, tempo);
    st := CT(tempo);
{1} if st < 0 then
      for i := 0 to stupin do
        vidp[i] := a1[i]
    else
      begin
{2}     nnsd(a0, a1, r);
        for i := 0 to stupin do
          begin
            tmp1[1,i] := a0[i];
            tmp1[2,i] := a1[i];
          end;
{3}     mnozhvect(r, tmp1, tmp);
        for i := 0 to stupin do
          begin
            b0[i] := tmp[1,i];
            b1[i] := tmp[2,i];
          end;
{4}     deviz(b0, b1, tempo1, c);
        st := CT(c);
        if st < 0 then
          for i := 0 to stupin do
            vidp[i] := b1[i]
        else
{5}
{6}       nsd(b1,c,vidp);
      end;
end;
{Кінець розділу процедур}
```

```
begin
    clrscr;
    norec := 0;
    for i := 0 to 5 do
      p1[i] := 1;
    p2[0] := -7;
    p2[1] := -1;
    p2[2] := 3;
    p2[3] := -2;
    p2[4] := 1;
    writeln('Найбільшим спільним дільником поліномів з коефіцієнтами');
    writeln('N поліном 1 поліном 2');
    for i := 0 to stupin do
      writeln(i, '. ', p1[i] : 9 : 4, p2[i] : 9 : 4);
    writeln('є поліном з коефіцієнтами');
    nsd(p1, p2, p);
    for i := 0 to CT(p) do
      writeln(i, '. ', p[i] : 9 : 4);
    repeat until keypressed;
end.
```

Програма 11.7. Алгоритм знаходження НСД

1. Спробуйте довести останнє твердження самостійно.
2. Внесіть потрібні зміни до процедур ННСД та НСД, щоб вони працювали й для цілих чисел.
3. Отримайте часову оцінку розробленого вами алгоритму знаходження НСД.

11.6. Організація ефективних обчислень розріджених поліномів

До цього моменту, розглядаючи поліном від однієї змінної у вигляді $\sum_{i=0}^{n-1} a_i x^i$, здебільшого, за винятком особливих обговорень, ми припускали, що він щільний. На практиці у багатьох прикладних задачах поліном є розрідженим (число ненульових коефіцієнтів набагато менше за його степінь). Нагадаємо, що одним із варіантів задання такого полінома є використання списку пар (a_i, j_i), що складаються з ненульового коефіцієнта і відповідного йому показника степеня змінної x.

Відома різноманітна техніка виконання арифметичних операцій над розрідженими поліномами [9]. Всю гаму нюансів використання такого

задання поліномів ми розглянути не в змозі. Покажемо лише один із методів реалізації арифметичних операцій над розрідженими поліномами – множення.

Поліном $\sum_{i=0}^{n-1} a_i x^j$ задамо списком пар (a_1, j_1), (a_2, j_2), …, (a_n, j_n), де усі j_i – відмінні й розташовані за спаданням, тобто $j_i > j_{i+1}$ для $1 \le i < n$. Для множення поліномів знайдемо добутки пар і розташуємо їх згідно із значеннями показників (за другими компонентами пар), об'єднуючи усі члени з однаковими показниками. Складність обчислення почне значно перебільшувати ту, за якої приведення подібних членів виконувалося б на кожному кроці. Для усунення цієї вади можна скористатися впорядкованістю пар початкового задання многочлена [127].

Вважатимемо, що на вхід алгоритму подають два поліноми $f(x) = \sum_{i=1}^{m} a_i x^{j_i}$ і $g(x) \sum_{i=1}^{m} b_i x^{j_i}$, зображені списками пар (a_1, j_1), (a_2, j_2), …, (a_m, j_m) і (b_1, j_1), (b_2, j_2), …, (b_m, j_n) відповідно. Послідовності j_1, …, j_m і k_1, …, k_n монотонно спадають. Нехай $m \ge n$. Тоді основою алгоритму множення будуть два основні кроки:

1. Побудова послідовності S_i добутків полінома $f(x)$ на j-й член полінома $g(x)$, $1 \le i \le n/2$, у яких r-й член, $1 \le r \le m$, дорівнює $(a_r b_i, j_r + k_i)$.

2. Об'єднання S_{2i-1} з S_{2i} для $1 \le i \le n/2$ шляхом приведення подібних членів і повторенням процесу, доки не отримаємо одну впорядковану послідовність.

На виході матимемо поліном $\sum_{i=1}^{p} c_i x^{j_i} = f(x) g(x)$, зображений списком пар, в якому послідовність l_1, …, l_p монотонно спадає.

З попередніх роздумів стає зрозуміло, що часова оцінка $O(mn \log n)$ справджуватиметься за припущення, що $m \ge n$.

Реалізацію алгоритму у Паскалі наведено в програмі 11.8.

```
Program Ex_11_8;

{Вхідний файл містить значення двох поліномів у форматі:
<число пар> <степінь змінної> <коефіцієнт> ...
<число пар> <степінь змінної> <коефіцієнт> ... }

  const max_size = 200;                    {Максимальна кількість доданків полінома}
  type polinom = record
                   n: integer;                                  {Кількість членів}
                   power: array [1..max_size] of integer;       {Пари: степінь змінної –}
                   coef: array [1..max_size] of real;           {коефіцієнт}
                 end;
  var p1, p2, res: polinom;
```

```
procedure ReadPolinoms(fname: string);
  var inf: text;
      i: integer;
begin
    assign(inf, fname);
    reset(inf);
    readln(inf, p1.n);                                  {Перший поліном}
    for i := 1 to p1.n do
      readln(inf, p1.power[i], p1.coef[i]);
    readln(inf, p2.n);                                  {Другий поліном}
    for i := 1 to p2.n do
      readln(inf, p2.power[i], p2.coef[i]);
    close(inf);
end;

procedure AddPolinoms(arg: polinom; var res: polinom);
  var i, j, k: integer;
begin
    for i := 1 to arg.n do                  {Об'єднання двох поліномів (додавання)}
      begin
        k := -1;
        for j := 1 to res.n do
          if arg.power[i] = res.power[j] then k := j;
        if k > 0 then
          res.coef[k] := res.coef[k] + arg.coef[i];
        else
          begin
            inc(res.n);                                 {Такого степеня немає}
            res.power[res.n] := arg.power[i];           {Збільшення кількості доданків}
            res.coef[res.n] := arg.coef[i];
          end;
      end;
end;

procedure MulPolinoms(arg1, arg2: polinom; var res: polinom);
  var tmp: polinom;
      i, j: integer;
begin
    res.n := 0;
    for i := 1 to arg2.n do
      begin
        tmp := arg1;
        for j:=1 to tmp.n do                    {Множення полінома на доданок}
          begin
            tmp.coef[j] := tmp.coef[j] * arg2.coef[i];
            tmp.power[j] := tmp.power[j] + arg2.power[i];
          end;
```



```
        AddPolinoms(tmp, res);                  {Додавання та спрощення}
      end;
end;

procedure PrintPolinom(arg:polinom);            {Виведення полінома за спаданням}
  var i, k, max: integer;                       {степеня змінної}
      t: polinom;
begin
    t := arg; max := 1; k := 1;
    repeat
      for i := 1 to t.n do
        if t.power[i] > t.power[max] then max := i;
      if (t.power[max] >= 0) then
        begin
          write(t.coef[max] : 3 : 0, '*[', t.power[max], '] ');
          t.power[max] := -1;
        end
      else k := 0;
    until k = 0;
    writeln;
end;

procedure WritePolinoms;
begin
    write('Перший поліном: '); PrintPolinom(p1);
    write('Другий поліном: '); PrintPolinom(p2);
    write('Результат множення: '); PrintPolinom(res);
end;

begin
    ReadPolinoms('input.txt');
    MulPolinoms(p1, p2, res);
    WritePolinoms;
end.
```

Програма 11.8. Множення розріджених поліномів

Задачі та вправи до розділу 11

1. Переведіть з десяткової системи числення в двійкову число 118.
2. Переведіть з двійкової системи числення в десяткову число 110 011 001.
3. Знайдіть за допомогою якогось з алгоритмів число, «обернене» до 384.
4. Обчисліть за допомогою розглянутого алгоритму число 384^2.
5. Використовуючи розглянутий алгоритм знайдіть поліном, «обернений» до $x^7 - 2x^6 + x^5 - x^4 + x^3 - x^2 + 4x + 1$.
6. Опишіть алгоритм для обчислення квадратів поліномів і за його допомогою обчисліть $(x^3 - 6x^2 + 4x - 2)^2$.

7. Знайдіть задання числа 30 000 за модулями 2, 3, 5, 7, 11.
8. Напишіть алгоритм знаходження лишків полінома і використайте його для знаходження лишків полінома $x^7 + 2x^6 + x^5 + 5x^4 + x^2 + 4$ за модулями $x + 1$, $x^3 - 2x + 1$, $x^2 + 3x - 2$ і $x^2 - 1$.
9. Нехай 5, 9, 7, 13 – чотири модулі. Знайдіть таке число u, яке менше їх добутку і $u \leftrightarrow (1, 2, 3, 4)$.
10. Знайдіть поліном, значення якого в точках 0, 1, 2, 3 дорівнюють відповідно 1, 2, 3, 4.
11. Використайте асимптотично швидкий алгоритм для знаходження НСД(368,246).
12. Напишіть програму реалізації методу Горнера обчислення многочлена в розрідженому заданні.
13. Напишіть програму додавання впорядкованих розріджених поліномів.

Розділ 12

ЕВРИСТИЧНІ АЛГОРИТМИ

12.1. Перші спроби

Евристичні підходи до розв'язання задач людина використовувала з давніх часів. Короткий оксфордський словник англійської мови трактує: «Евристика – мистецтво знаходження істини». У давньогрецькій мові слово евристика (heuristic) має ті самі джерела що й слово еврика. Прикладом ілюстрації цього методу в загальних рисах може слугувати класична задача про мавпу і банани. Мавпа знаходиться у порожній кімнаті, де до стелі підвішено низку бананів і біля стіни стоїть скриня. Мавпа може переміщуватися по кімнаті, пересувати скриню та залазити на неї. Але дістати банани вона зможе тільки тоді, коли залізе на скриню, розміщену точно під бананами. Слід примусити мавпу виробити варіант розв'язання задачі діставання бананів шляхом потрібного зсуву скрині.

Під час розгляду задач, що належать до класу *NP*, важливе місце займає знаходження варіантів розв'язання цих задач для певних наборів даних (а не для всіх даних) за поліноміальний час. Для цього досліднику бажано підказати шлях, який позбавляє від необхідності розгляду повного перебору, виходячи з певних особливостей задачі розгляду. Ці особливості й називають евристиками, а правила їх використання – евристичними правилами. Евристичні правила можуть бути дуже простими і досить складними. Прикладом простої евристики слугують прислів'я. Багато з них можна розглядати як керівництво до дії в повсякденному житті. Оскільки евристичні правила мають рекомендаційний характер, то евристичні методи не завжди приводять до бажаних результатів розв'язання задачі. Отже, бажано мати якнайбільший набір евристик.

Для більш глибокого розуміння евристичного підходу розглянемо два типи задач: *добре* і *погано* визначені задачі [52].

Добре визначеною називають таку задачу, для якої за використання заданого можливого розв'язку можна застосувати алгоритмічний метод визначення *чи буде розв'язок шуканим*. Прикладом такої задачі може слугувати задача інтегрування алгебричного виразу відносно однієї із змінних, що входять в нього. Процес інтегрування не є, в загальному випадку, алгоритмічним. Перевірку інтегрування можна провести шляхом диференціювання за тією ж змінною. Оскільки останнє – алгоритмічний процес, то інтегрування відносять до добре визначених задач.

Більшість задач повсякденного життя є погано визначеною; вибирають деяку послідовність дій, для якої немає впевненості, гарантії, що вона,

навіть у перспективі, буде найоптимальнішою за даних обставин. Наприклад, задачу вибору ходу в шахах, незважаючи на чітко виражений кількісний характер вхідних даних, слід розглядати як погано визначену задачу.

Для добре визначених задач традиційно існує деякий алгоритмічний метод їх розв'язання (за винятком алгоритмічно нерозв'язних задач). Для них можна визначити *простір розв'язків*, що містить істинний розв'язок. Так, у разі задачі інтегрування таким буде простір розв'язків, що містить всі можливі алгебричні вирази наперед визначеної довжини. Коли є змога визначити спосіб його перегляду, кажуть, що добре визначену задачу можна, в принципі, розв'язати алгоритмічно. Складність полягає в тому, що за повного перебору варіантів розв'язку розв'язок стає нереальним внаслідок великого розміру простору розв'язків, звідки отримуємо ще одне визначення евристики [21]:

Евристика (евристичне правило, евристичний метод) є довільним правилом, стратегією, маневром, спрощенням або якимось іншим засобом істотного зменшення, обмеження об'єму пошуку розв'язку в значних просторах розв'язку.

Згідно з цим визначенням необхідним є вірне адміністрування побудови розв'язку. Під останнім розуміють метод, який за деяким евристичним критерієм серед декількох можливих підходів до розв'язку вибирає найоптимальніший на даному кроці. Інакше кажучи, адміністративна частина алгоритму має бути в змозі дати оцінки складності підзадач та їх корисності, а також використовувати ці оцінки для планування процесу розв'язання задач. Отже, евристичний метод є корисним у разі встановлення порядку виконання дій розв'язання задачі, який є не обов'язково оптимальним. Цей порядок, з великою ймовірністю, має виявитися набагато ліпшим за випадково вибраний порядок.

Спробуйте навести свої приклади добре визначених і погано визначених задач.

Серед загального огляду евристик важливе місце займає основна евристика, евристика-навчання Мінського і Селфріджа [136]. Її формулюють так: «У *новій ситуації спробуйте використати методи, що найліпше працювали в аналогічних заздалегідь відомих ситуаціях*». У цьому разі важливим є вибір критерію подібності задач, ситуацій, розв'язків. Така проблема є досить складною. Для визначення міри подібності задач Мінським було введено поняття евристичного зв'язку задачі [139]. Ця ідея може зумовити важливі висновки щодо природи інтелекту і має дуже широке застосування під час розв'язання задач штучного інтелекту.

Наприкінці 1950-х і на початку 1960-х років Ньюел, Саймон і Шоу опублікували піонерські роботи, які заклали початок класичного евристичного підходу до розв'язання інтелектуальних задач; основний акцент був зроблений на планування цілеспрямованих дій. Першою інтелектуальною програмою стала розроблена ними програма *Логік–Теоретик*, за

допомогою якої можна було доводити теореми математичної логіки. Ідеологія нового підходу полягала в породженні різноманітних здогадок та припущень з подальшою перевіркою їх справедливості.

Розвиток цього напряму був пов'язаний з програмою *«Загальний вирішувач задач»* (GPS – General Problem Solver), яку ще іноді називають *«Універсальний вирішувач задач»*. Були запропоновані деякі загальні універсальні принципи розв'язання задач з довільної предметної області. Підхід ґрунтується на розгляді ситуацій, що виникають під час розв'язання задач, та операторів, які можуть змінювати ці ситуації. Методика GPS передбачає аналіз цілей та засобів їх досягнення. Аналіз цілей випливає з аналізу розбіжностей між поточною та бажаною ситуаціями. Схему прийняття рішення можна представити у такому вигляді [78]:

1. *Проаналізувати поточну ситуацію.*
2. *Порівняти ситуацію з бажаною; якщо відмінностей немає – кінець роботи.*
3. *З'ясувати, який оператор або оператори можна застосувати для зменшення існуючої різниці.*
4. *Послідовно застосовувати оператори, знайдені на кроці 3, поки один з них не спрацює.*
5. *Повернутися на крок 1.*

Ця проста схема, очевидно, не залежить від конкретної предметної області – в принципі, її можна застосувати для розв'язання будь-якої задачі. На попередньому етапі необхідно зафіксувати *перелік можливих відмінностей між поточною та бажаною ситуаціями та перелік операторів, що можуть ліквідовувати ці відмінності.*

Ключовим поняттям за використання «Універсального вирішувача задач» є *евристична таблиця, або таблиця відмінностей*. Її рядки відповідають типам відмінностей, а стовпці – різним операторам. Якщо даний оператор можна застосовувати для зняття даного типу розбіжностей, в клітині перетину рядка і стовпця ставлять одиницю, в іншому разі – нуль.

Під час розвитку і вдосконалення ідеї GPS стало зрозуміло, що, приймаючи рішення, слід також брати до уваги інші характеристики конкретної задачі, наприклад причинно-наслідкові зв'язки. У праці [79] наведено ще один приклад на цю тему. Є дворукий нерухомий робот-маніпулятор, призначений для переміщення предметів. Є два оператори: *«взяти лівою рукою»* та *«взяти правою рукою»*. Нехай спочатку робот спробував узяти предмет правою рукою і зазнав невдачі. Тут важливою є причина невдачі. Якщо вона полягає у тому, що предмет неможливо дістати, слід спробувати інший оператор – *«взяти лівою рукою»*, оскільки ліва рука розташована в іншій точці простору. Якщо ж причиною невдачі була надто висока температура предмета, то невдача чекатиме і у разі спроби взяти його лівою рукою, тому цю спробу робити не варто.

Перевірте застосування GPS на грі в хрестики-нулики на обмеженому полі.

12.2. Евристичний пошук

Широке коло задач прийняття рішень можна сформулювати як класичну оптимізаційну задачу: знайти розв'язок, на якому деяка цільова функція досягає свого максимуму за заданих обмежень. Так, наприклад, керівник фірми бажає максимізувати свій зиск, не порушуючи закони, або потрібно якнайшвидше посадити космічний корабель на Марсі так, щоб сила удару була не надто великою. Подібні задачі вивчає розділ науки, який називають *дослідженням операцій*. Основні підходи до розв'язання оптимізаційних задач розглянуто в розділі 6.

Існує очевидний і досить універсальний метод розв'язання оптимізаційних задач, який можна застосовувати, якщо множина припустимих рішень M обмежена (компактність є сильнішою вимогою, але для більшості практичних ситуацій вона не є необхідною). Це метод *повного перебору*, який полягає у переборі всіх можливих варіантів. Метод дає гарантований розв'язок, якщо множина M скінченна (ситуація, характерна для дискретного програмування), і існує ефективний алгоритм породження будь-якого елементу з M та обчислення на цьому елементі цільової функції. Якщо ж множина припустимих рішень є континуумом, слід використовувати мережну апроксимацію.

Компактом називають компактний метричний простір, зокрема будь-яку обмежену замкнену множину евклідового простору. Компактність – це термін топології. Множину M топологічного простору називають зліченно-компактною, якщо будь-яка нескінченна послідовність точок цієї множини має межову точку, що належить множині M. Множину M називають компактною, якщо з будь-якого її відкритого покриття можна вибрати скінченне підпокриття. Якщо множина M міститься в скінченновимірному евклідовому просторі, то вона компактна тоді й тільки тоді, коли обмежена і замкнена. На компактній множині будь-яка неперервна функція досягає свого мінімального і максимального значення. Прикладом компактної множини можуть бути відрізок $[a, b]$ на числовій прямій або коло в евклідовій площині.

Повернемося до визначення евристичного пошуку. Отже, якщо система потрапляє в нову ситуацію і намагається планувати подальші дії, вона може спробувати звести задачу планування цілеспрямованих дій до оптимізаційної задачі (для цього, очевидно, достатньо визначити множину припустимих рішень M і цільову функцію $f(x)$). Якщо це вдасться, повний перебір варіантів у більшості випадків дасть змогу отримати оптимальне рішення.

Повний перебір має очевидний недолік: для більшості практичних ситуацій кількість варіантів, які доводиться перебирати, надто велика, і реалізувати метод за прийнятний час неможливо. Тому, враховуючи, що не завжди слід шукати саме оптимальний розв'язок, часто достатнім є субоптимальний розв'язок або навіть просто допустимий розв'язок, тобто слід виробити правило, яке б обмежувало проведення повного перебору і для більшості вхідних даних давало прийнятний результат.

Багато практичних задач можна розв'язати, використовуючи процедуру, що має назву ***евристичного пошуку***. Евристичним пошуком прийнято називати процедуру систематизованого перебору на основі послідовного прийняття рішень. Можна навести таку загальну схему евристичного пошуку:

1. *Вибрати деяку дію з області можливих дій.*
2. *Здійснити дію. Це призведе до зміни ситуації.*
3. *Оцінити нову ситуацію.*
4. *Якщо досягнуто успіху – кінець, якщо ні – повернутися на крок 1 і почати спочатку.*

«Універсальний вирішувач задач» можна розглядати як одну з можливих реалізацій евристичного пошуку.

Евристичний пошук часто розглядають як *пошук шляху на дереві або графі*. Дійсно, розглянемо типову схему розв'язання задачі, згідно з якою на кожному кроці можна вибирати одну з можливих дій. Тоді можна говорити про породження *дерева можливостей*, вузли якого відповідають ситуаціям проблемної області, а дуги – можливим діям. Тоді задача переходу від початкової ситуації до бажаної зводиться до задачі пошуку шляху на дереві.

Аналогічно виникає і графова інтерпретація.

Модифікуйте програму пошуку в глибину або в ширину на графі для реалізації евристичного пошуку.

У розділах 8 та 9 зазначалося, що повний перебір або перебір з поверненням дає змогу в низці випадків отримати бажаний розв'язок, але часто реалізація таких методів забирає дуже багато часу і тому є неможливою чи недоцільною. Отже, в таких ситуаціях звертаються до *обмежувальних правил* та *евристик* як до типових засобів скорочення перебору. Саме так у більшості випадків і діє людина під час планування своїх дій.

Обмежувальним під час планування цілеспрямованих дій називають правило, якому мають бути підпорядковані альтернативні дії розгляду.

Інакше кажучи, застосування обмежувальних правил дає змогу включати до перебору не всі можливі дії, а лише ті, які не суперечать цим правилам. Це іноді дає змогу різко скоротити перебір, а інколи навіть звести задачу до поліноміальної. Зокрема, описані вище *жадібні алгоритми* можна розглядати як застосування обмежувальних правил.

Природа обмежувальних правил може бути різноманітною. Можна довести, що обмежувальне правило дає змогу відкинути явно безперспективні гілки і досягти того ж результату, що й повний перебір без застосування цього правила. Такі правила є *теоретично обґрунтованими* і мають безумовно застосовуватися.

Частіше трапляється ситуація, коли обмежувальні правила спираються на певний апріорний досвід, але теоретично не обґрунтовані, і не

гарантують або просто не дають змоги отримати оптимальний розв'язок. Застосування цих правил дозволяє скоротити перебір, але за рахунок втрати гарантованої оптимальності (власне, у багатьох випадках нічого ліпшого і не залишається). Отримуємо ще одне визначення евристики.

Евристикою під час планування цілеспрямованих дій називають обмежувальне правило, яке спирається на певний досвід і не гарантує оптимальності рішення.

Часто буває і так, що існує не одна евристика, а кілька, і вони повністю або частково суперечать одна одній. Тоді окремою проблемою стає проблема застосування евристик.

Для подальшого уточнення поняття евристики та евристичних алгоритмів звернемось до особливостей їх застосування у разі розв'язання конкретних задач.

12.3. Задача розфарбування графа

У XIX ст. вперше постало питання, якою мінімальною кількістю фарб можна розфарбувати топологічну карту. Тоді з'явилася ідея розглядати карту як планарний граф; так з'явилася проблема розфарбування графа мінімально можливою кількістю фарб [64, 97]. Потім виявилося, що проблема розфарбування не лише планарного, а й довільного графа (звичайно, з деякими обмеженнями) є надзвичайно актуальною. Задача розфарбування графа має безліч застосувань у комбінаториці, фізиці, економіці, програмуванні. До задачі розфарбування графа можна звести такі задачі, як класифікація об'єктів, розподіл ресурсів, складання розкладу, розподіл пам'яті. Розглянемо ж детальніше цю проблему та методи її вирішення [3, 6, 10, 43, 45, 46, 64]. У своєму дослідженні розглядатимемо лише зв'язні неорієнтовані графи без петель та паралельних ребер.

12.3.1. *Базові поняття*

Нехай маємо $G = (V, E)$ – неорієнтований граф, де $V = \{v_1, \ldots, v_n\}$ – множина вершин графа. Число n далі називатимемо кількістю вершин графа. *Розфарбуванням вершин* графа G називатимемо відображення φ множини V на множину C, таку що $C = (c_1, \ldots, c_k)$ і $k \le n$. Елементи множини C називатимемо *фарбами,* а число k визначатиме *кількість фарб*, використаних під час розфарбування графа G.

Вірним розфарбуванням графа G називатимемо таке розфарбування вершин, для якого завжди виконується $(x_i, x_j) \in E \Rightarrow \varphi(x_i) \neq \varphi(x_j)$, тобто якщо дві вершини суміжні, тоді вони пофарбовані різними фарбами.

Вірне m-розфарбування розбиває множину V вершин графа на m підмножин $V_1, \ldots, V_m$ (або класів), що не перетинаються та жодна з яких не містить суміжних вершин. Очевидно, що ці множини будуть незалежними. Кожна з них містить вершини, розфарбовані певною фарбою. Такі множини називають *хроматичними класами* графа G.

Мінімальним розфарбуванням називатимемо будь-яке вірне розфарбування, кількість фарб у якому не більша за будь-яке інше вірне розфарбування.

З цього визначення випливає, що кількість фарб для всіх мінімальних розфарбувань є однаковою. Таку найменшу кількість фарб, за якої граф має мінімальне розфарбування, називають *хроматичним числом* графа, яке є характеристикою графа і позначається $\chi(G)$, або ж просто χ.

Називатимемо граф *k-розфарбованим*, якщо $\chi(G) \le k$, та *k-хроматичним*, якщо $\chi(G) = k$.

Для знаходження хроматичного числа графа недостатньо знати кількість його вершин і ребер, розподіл степенів вершин. Однак за цими характеристиками можна знайти верхню і нижню оцінку хроматичного числа графа. Постає питання, як саме слід діяти, щоб отримати мінімальне розфарбування графа та визначити його хроматичне число [43, 45, 64]?

Знайдіть за цими характеристиками верхню й нижню оцінки хроматичного числа графа.

Отже, маємо окреслене коло проблем, пов'язаних з поняттям розфарбування графа. Задачу розфарбування графа можна природно розкласти на дві складові. По-перше, її можна трактувати як задачу знаходження хроматичного числа графа, по-друге, задачу розфарбування графа можна поставити як знаходження хоча б одного мінімального розфарбування графа. Власне, ці дві підзадачі і формують задачу розфарбування графа. Вони пов'язані між собою, оскільки, якщо маємо гарантовано мінімальне розфарбування графа G, тоді можна дуже просто знайти його хроматичне число $\chi(G)$. Якщо ж ми у якийсь спосіб знайшли хроматичне число графа $\chi(G)$, тоді, ймовірно, можемо знайти й правильне розфарбування графа G за допомогою $\chi(G)$ фарб.

Іноді розглядають і часткові випадки задачі розфарбування графа, наприклад, розфарбування графа у наперед визначені кольори. Для кожної вершини відомі ті кольори, в які можна її розфарбувати, і ті, в які її розфарбувати не можна. Тобто до класичної задачі розфарбування додають певні обмеження щодо співвідношення кольорів. Її вирішення можна природно отримати із розв'язку базової задачі.

Планарний граф – це граф, який можна зобразити на площині так, що різним вершинам відповідають різні точки площини, ребрам – дуги (без самоперетинань), і жодні два ребра не мають спільних точок, крім інцидентної їм обом вершини.

Постає питання, якою мінімальною має бути кількість фарб розфарбування такого графа. Ще у XIX ст. було висунуто знамениту *гіпотезу 4 фарб* – будь-який планарний граф можна правильно розфарбувати 4 фарбами, тобто $\chi(G) \le 4$, якщо G – планарний граф. Її вдалося підтвердити на практиці лише у XX ст. за допомогою ЕОМ.

Також іноді кажуть про *реберне розфарбування* графа. У цьому разі мають справу з відображенням множини ребер на множину фарб, а правильним розфарбуванням називають розфарбування, за якого будь-які два ребра, що інцидентні одній вершині, пофарбовані різними фарбами.

Далі буде проаналізовано точні та евристичні методи вершинного розфарбування графа. Зауважимо лише те, що точні методи розфарбування графа дають в результаті точне хроматичне число, але вони не є ефективними. Якщо ж користуватися евристичними методами розфарбування, то не можна сподіватися на ідеально точне знаходження хроматичного числа, але можна знехтувати похибкою обчислень на користь більш ефективного алгоритму.

12.3.2. *Аналіз хроматичного числа графа*

Якщо граф G має n вершин, то очевидно, що він має n-розфарбування, $\chi(G)$-розфарбування і, крім того, має k-розфарбування для будь-якого k, такого що $\chi(G) \le k \le n$.

Звідси видно, що хроматичне число повного графа з n вершинами дорівнює n. Хроматичне число парного циклу дорівнює 2, непарного циклу – 3. Хроматичне число пустого графа (в якого множина ребер E пуста) дорівнює 1. Зауважимо, що граф є 1-хроматичним тоді і тільки тоді, коли він пустий. Ці очевидні закономірності не дають змоги оцінити, яким буде хроматичне число, але вони є базою відомих теоретичних досліджень для оцінки хроматичного числа різних типів графів [43, 45]. Прикладом таких досліджень може бути теорема Кенінга.

Теорема 12.1 (теорема Кенінга). Граф є біхроматичним (2-хроматичним) тоді і тільки тоді, коли він не містить непарних простих циклів.

Звідси випливає, що будь-яке дерево є біхроматичним і будь-який дводольний граф є біхроматичним. Проте невідомо, як характеризувати k-хроматичні графи, якщо $k \ge 3$. Такий критерій навіть для $k = 3$ допоміг би вирішити проблему 4 фарб.

Спробуйте довести:

- *якщо граф G має у своєму складі повний підграф (кліку) з t вершинами, то його хроматичне число не може бути меншим за t і задовольняє нерівності $t \le \chi(G) \le n$;*
- *якщо граф має вершину з найбільшим степенем m, то цей граф $(m+1)$-розфарбовуваний;*
- *якщо найбільший степінь вершин графа дорівнює m, то граф є m-розфарбовуваним, крім тих випадків, коли граф має у своєму складі повний $(m+1)$-вершинник, або ж коли $m = 2$ і одна з компонент графа є непарним циклом.*

Користуватися цими фактами зручно, коли степені більшості вершин графа приблизно однакові. Якщо ж граф має вершини зі степенями, що сильно відрізняються, та якщо вершин з великими степенями небагато, така оцінка буде надто завищеною.

Дослідниками [97] також було визначено кілька верхніх оцінок хроматичного числа графа.

Теорема 12.2 (теорема Секереша–Вільфа). Для будь-якого графа G

$$\chi(G) \le 1 + \max_{\bar{H} \subset G} \min_{v \in H} \deg_H v,$$

де максимум беруть із всіх власних підграфів $\bar{H}$ графа G.

Теорема 12.3 (теорема Уелша–Пауела). Нехай у графі вершини впорядковані за спаданням степенів. Тоді $\chi(G) \le \max_{1 \le i \le n} \min\{i, 1 + \deg v_i\}$.

Використання цих оцінок буде розглянуто нижче при дослідженні наближених методів розфарбування графа.

Задача мінімального розфарбування графа, як і багато інших комбінаторних задач, є *NP*-повною [39]. Це означає, що її точний розв'язок потребує експоненційного часу розв'язання, тобто число кроків перебору буде експоненційно зростати залежно від розмірності задачі.

До того ж розв'язання задачі розфарбування графа шляхом безпосереднього перебору варіантів передбачає побудову всіх можливих варіантів розфарбування і вибір з них мінімального вірного розфарбування. При цьому породжуватимуться не лише правильні розфарбування, а також безліч розфарбувань, які можна перетворити одне в одне за допомогою простого перефарбування вершин, тобто, по суті, однакових. У такий спосіб, ми бачимо, що алгоритм прямого перебору не дає нам змоги розв'язати цю задачу за реальний час за великої розмірності вхідних даних.

Проаналізувавши тривіальне рішення, дійдемо висновку, що слід шукати одразу лише правильні розфарбування графа і намагатися відразу ж знайти мінімальне розфарбування. Шукаючи алгоритм розв'язання задачі, ми відмовимося від можливості знаходження оптимального розв'язку і будемо знаходити розв'язок, близький до оптимального за прийнятний час. Алгоритми, засновані на такому обмеженні, називають евристичними, бо вони використовують різноманітні вдалі міркування без строгого обґрунтування.

12.3.3. *Точні алгоритми розфарбування графа*

Точні алгоритми розфарбування знаходять точне хроматичне число графа і мінімальне розфарбування. Звичайно, вони мають більшу складність, ніж наближені алгоритми, але меншу, ніж мав би простий перебір.

Існує багато підходів скорочення об'ємів перебору, хоча при цьому і не заперечується експоненційна часова складність алгоритмів. До таких підходів відносять, наприклад, метод гілок і границь, або метод неявного перебору, які полягають у побудові часткових розв'язків, зображених у вигляді дерева пошуку, і застосуванні потужних методів побудови оцінок, що дають змогу розпізнати безперспективні часткові розв'язки, в результаті чого на одному кроці від дерева пошуку відтинається ціла гілка. До інших методів належать метод динамічного програмування, розбиття графа на максимальні r-підграфи. Нижче наведено один з точних алгоритмів розфарбування.

На першому кроці визначають нижню оцінку хроматичного числа, що дорівнює кількості вершин в одній з клік графа, верхню оцінку хроматичного числа шляхом розфарбування вершин наближеним методом розфарбування (або ж визначають за однією з формул знаходження верхніх оцінок).

Якщо верхня оцінка дорівнює нижній, то точне розфарбування знайдене, якщо ні, то роблять спробу розфарбувати вершини графа у задану кількість кольорів $\chi_я = \chi_н + [(\chi_в - \chi_н) / 2]$ (де $\chi_н$ – нижня оцінка, $\chi_в$ – верхня оцінка) за допомогою наступного наближеного алгоритму.

Нехай k_i – номер кольору i-ї вершини, $N_n = \overline{k_1, k_2, ..., k_n}$ – n-значне число, що відповідає розфарбуванню вершин графа. Алгоритм побудований на монотонному обчисленні N_s з N_{s-1}. Алгоритм дає або правильне розфарбування у M кольорів, або інформацію про неможливість такого розфарбування.

Якщо спроба вдалася, то змінюється значення верхньої оцінки $\chi_в = \chi_я$, інакше змінюється нижня оцінка $\chi_н = \chi_я + 1$. На кожній ітерації другого кроку алгоритму інтервал між нижньою та верхньою оцінками ділять навпіл і в цій точці знаходять нові оцінки.

Напишіть програму реалізації такого алгоритму та побудуйте часову оцінку алгоритму.

Однак існують наближені алгоритми розв'язання цієї задачі, що мають не більш як квадратичну часову оцінку, але не гарантують оптимального розв'язання задачі. Кожен з таких алгоритмів базується на певній евристиці, і будь-яке додавання евристики до алгоритму може його або ускладнити, але наблизити до оптимальності, або ж зробити більш ефективним за рахунок пошуку рішення, близького до оптимального.

12.3.4. *Евристичні алгоритми розфарбування*

Намагаючись знайти алгоритм з не більш як квадратичною часовою оцінкою, ми приходимо до висновку, що на розфарбування кожної з n вершин графа ми можемо витратити не більш як n кроків.

Так ми приходимо до ідеї організації алгоритму без перефарбовування вершин.

Прийнявши таке рішення ми маємо дві альтернативи – починати з вершин або з фарб. Перший шлях – брати певну фарбу і фарбувати нею вершини за якимось певним порядком. Коли не залишиться вершин, які можна фарбувати цією фарбою, брати наступну – доти, доки не будуть розфарбовані всі вершини в графі. Другий шлях – брати по черзі вершину і вибирати за якимось критерієм для неї фарбу, якою її можна розфарбувати.

Придивимося уважніше до цих варіантів [45]. Йдучи першим шляхом, ми на кожному кроці маємо множину розфарбованих вершин V_c (на першому кроці ця множина буде пустою) і множину нерозфарбованих вершин, з яких ми і шукаємо наступну вершину – кандидата на розфарбування поточною фарбою α. Множина розфарбованих вершин має підмножину вершин $V(\alpha)$, розфарбованих фарбою α. Множина нерозфарбованих вершин, у свою чергу, містить множину тих вершин, які можна розфарбувати фарбою α (умовно позначимо її $V^д$), і множину тих вершин, які не можна розфарбувати цією фарбою (позначимо цю множину $V^з$). Причому ця остання множина, очевидно, містить вершини, суміжні хоча б з однією вершиною з $V(\alpha)$. Тобто на черговому кроці ми або беремо вершину з $V^д$ і розфарбовуємо її фарбою α, або ж, якщо множина $V^д$ дозволених для розфарбування фарбою α вершин пуста, переходимо до наступної фарби. Алгоритм завершується, коли всі вершини графа будуть розфарбовані.

Однак такий підхід має певні недоліки. Кожного разу ми маємо маніпулювати із множиною $V(\alpha)$ і з усіма її околами першого порядку, з множинами V і V_c.

У разі другого підходу, розфарбувавши вершину v, ми вже не можемо розфарбовувати вершини з її околу першого порядку $R_1(v)$. Аналізуючи вершину, ми шукаємо фарбу, якою можемо розфарбувати цю вершину, і якщо не знаходимо її, то використовуємо нову фарбу. Очевидно, користуючись таким методом, маємо вибирати вершини за певним порядком. Визначення його дає нам ще декілька альтернатив організації алгоритмів.

У такий спосіб, ми вже маємо два різні підходи до розфарбування і можемо їх використати для розв'язання нашої задачі.

Спробуйте побудувати свій евристичний алгоритм розфарбування.

Розфарбування як склеювання вершин

Можна за допомогою подальших роздумів додати ще ліпші евристики [45]. Уважніше придивимося до правил формування множини $V^з$. Вона складається з нерозфарбованих вершин, що належать околам першого порядку вершин з множини $V(\alpha)$. Оскільки всі вершини з $V(\alpha)$ є вже

розфарбованими, в нашому подальшому аналізі ми скористаємося ними лише для визначення суміжних з ними вершин. Якщо розглядати всі вершини з $V(\alpha)$ як одну деяку вершину w, то матимемо $V^3 = R_1(w)$, а це вже є значним спрощенням процедури визначення обмежень вибору наступної розфарбовуваної вершини. У такий спосіб, ми нібито «склеюємо» всі вершини з $V(\alpha)$. Робитимемо це послідовно: маючи вершину v, пофарбовану фарбою α, знаходимо у множині V^R вершину v' і склеюємо вершини v та v', заміняючи процедуру призначення вершині певної фарби попарним склеюванням несуміжних вершин.

Склеювання двох вершин v_1 та v_2 полягатиме у вилученні з графа вершин v_1 та v_2 разом з інцидентними їм ребрами та додаванні вершини v та інцидентних їй ребер так, щоб її окіл першого порядку став об'єднанням околів першого порядку вершин v_1 та v_2, тобто $R_1(v) = R_1(v_1) \cup R_1(v_2)$. У результаті отримаємо граф $G' = (V', E')$, де потужність множини V' є меншою на 1, ніж у попередньому графі; ребер також може бути менше за рахунок взаємного поглинання ребер. Поглинання ребер відчуватиметься, якщо якась вершина u була суміжною з обома вершинами, які склеюють, і в результуючому графі замість двох ребер (v_1, u) та (v_2, u) залишиться одне ребро (v, u). Подивившись більш прискіпливо на процес поглинання, бачимо, що воно обов'язково відбуватиметься для вершин, які знаходяться одна від одної на відстані 2, тобто коли $v_1 \in R_2(v_2)$ та $v_2 \in R_2(v_1)$. Тут під околом другого порядку певної вершини розумітимемо множину всіх вершин, які знаходяться від неї на відстані 2. Таку вершину u називатимемо *відокремлюючою* вершиною. Кількість ребер, що зникають, дорівнюватиме кількості вершин відокремлення.

Перетворення кожного чергового графа G відбуваються доти, доки в ньому не залишиться жодної пари несуміжних вершин, тобто поки граф G не виявиться деяким k-повним графом. Як зазначалось, k-повний граф має лише одне тривіальне розфарбування: кожній з k вершин приписують одну з k фарб; ця фарба буде фарбою для всіх тих вершин початкового графа, які було вклеєно у дану вершину. Оскільки всі вершини, що вклеєні в певну вершину результуючого повного графа, є попарно несуміжними, то отримане розфарбування є допустимим.

Тепер слід довести, що зводячи розфарбування вершин до їх послідовного склеювання, ми не втрачаємо можливості отримати мінімальне розфарбування графа [46].

Теорема 12.4. Для будь-якого графа з хроматичним числом χ існує послідовність попарних склеювань вершин, що приводить до χ-повного графа.

Дві вершини графа G називають *суцвітними*, якщо існує мінімальне розфарбування графа G, при якому ці вершини пофарбовані однією фарбою.

Відомо, що в будь-якому неповному графі G існує принаймні пара суцвітних вершин, і якщо граф G' отримано з графа G склеюванням пари суцвітних вершин, то $\chi(G') = \chi(G)$.

На основі цих ідей маємо право склеювати вершини, зберігаючи хроматичне число початкового графа, і проводити склеювання, доки граф не стане повним, причому в ньому буде $\chi(G)$ вершин.

Один з найпоширеніших алгоритмів розфарбування першого типу – алгоритм Єршова–Кожухіна [45] базується на евристиках склеювання. Залишається лише вирішити проблему знаходження суцвітних вершин у графі.

Теорема 12.5 (теорема Єршова–Кожухіна про суцвітні вершини). Для будь-якої вершини v графа G, для якої її окіл першого порядку $R_1(v)$ не збігається з множиною $X \setminus \{v\}$, в околі другого порядку $R_2(v)$ існує принаймні одна суцвітна вершина.

Ця теорема дає нам евристику істотного обмеження простору пошуку суцвітних вершин, гарантуючи їхнє місцезнаходження в околі другого порядку вершини розгляду.

Отже, знайшли ще одну евристику уточнення, поліпшення алгоритму. Тоді алгоритм набуде такого вигляду.

Вважатимемо, що вершини графа якимось способом пронумеровані. На черговому кроці беруть вершину v та в її околі другого порядку шукають вершину w з максимальною кількістю вершин відокремлення. Вершини v та w склеюють, отриману вершину вважають черговою. Процес продовжують доти, доки чергова вершина не стане зіркою, після чого переходять до наступної вершини-незірки тощо. Процес склеювання вершин закінчується, коли граф стає повним.

Попереднє впорядкування вершин

Розглянемо два підходи побудови евристичних алгоритмів розфарбування другого типу. Згадаємо теорему 12.2. Ця теорема, даючи верхню оцінку хроматичного числа графа через максимальний степінь вершини, наштовхує на евристику, що треба було б починати розфарбовувати вершини з вершин з більшими степенями. Якщо таких вершин мало, тоді можна обійтися меншою кількістю фарб.

Якщо впорядкувати вершини графа за спаданням їхніх степенів і для кожної чергової вершини розфарбування вибирати фарбу з найменшим можливим номером, то отримаємо ще один евристичний алгоритм розв'язання задачі розфарбування графа. Після того як буде пофарбовано i вершин, буде витрачено, максимум, i кольорів, а для розфарбування решти вершин буде потрібно не більше за $1 + \max_i \deg v_i$ кольорів, звідки й отримаємо оцінку з теореми 12.3:

$$\chi(G) \le \max_{1 \le i \le n} \min\{i, 1 + \deg v_i\}.$$

Упорядкування вершин за їхнім спаданням називають ПН-упорядкуванням (перші найбільші), а алгоритми, засновані на такому впорядкуванні – ПН-алгоритмами. Можна відзначити, що кількість фарб, що її потребує

ПН-алгоритм, може змінюватися через неоднозначність ПН-упорядкування, але вона ніколи не перевищуватиме числа, визначеного верхньою оцінкою.

Розглянемо ще одну евристику розфарбування, яка намагається зменшити кількість фарб у попередньому алгоритмі розфарбування графа. Назвемо *степенем насиченості* вершини v у частково розфарбованому графі кількість різних кольорів на суміжних з v вершинах.

На першому кроці впорядковуємо вершини за спаданням (ПН-упорядкування), і першу вершину фарбуємо першою фарбою. Після цього на кожному кроці формуємо множину вершин K з найбільшим степенем насиченості, вибираємо з неї вершину з найбільшим степенем в нерозфарбованому підграфі і розфарбовуємо її найменшим можливим кольором. Діємо так доти, доки в графі не будуть пофарбовані всі вершини.

Для побудови евристичних алгоритмів мінімального розфарбування можуть використовувати й інші впорядкування [137].

Напишіть програму реалізації евристичного алгоритму розфарбування, що базується на таких двох правилах:

- *беремо вершину з мінімальним степенем у графі та вилучаємо її з графа разом з інцидентними їй ребрами;*
- *у графі, що залишився, знову шукаємо вершину з мінімальним степенем, і робимо так, доки в графі не залишиться жодної вершини.*

Порядок, в якому ми розфарбовуватимемо вершини, буде оберненим до того порядку, в якому ми їх вилучали. Таке впорядкування називають НО-впорядкуванням (найменші – останніми). Слід зазначити, що це впорядкування не є еквівалентним ПН-упорядкуванню, подумайте чому?

Окільний алгоритм

Для кожного евристичного алгоритму розфарбування існує клас графів, для яких отримане розфарбування не буде мінімальним. Це твердження ґрунтується на теоремі Гері–Джонсона [137].

Теорема 12.6 (теорема Гері–Джонсона). Для будь-якого евристичного алгоритму знайдеться такий граф G, що $A(G) / \chi(G) \geq 2$, де $A(G)$ кількість фарб, витрачених на розфарбування графа G евристичним алгоритмом, а $\chi(G)$ – хроматичне число графа G. Тому бажано мати більше евристичних алгоритмів з визначенням класів графів, на яких вони дають добрі оцінки затраченої кількості фарб. У праці [6] був введений ще один клас евристичних алгоритмів – клас окільних алгоритмів. Для алгоритмів з цього класу число затрачених фарб розфарбування будь-якого графа задовольняє оцінці Єршова–Кожухіна [45]. Під час практичного тестування вони також показали добрі результати (число фарб наближалось до хроматичного числа) на графах із тих класів, на яких заздалегідь розглянуті евристичні алгоритми давали погані результати.

Основною евристикою окільних алгоритмів розфарбування є розфарбування графа згідно з другим типом розфарбування відносно певного порядку околів, на які розбивають множину вершин графа. Задаючи певний порядок вибору цих околів і порядок розфарбування вершин околу, отримують конкретні алгоритми цього класу. Нагадаємо, що околом першого порядку вершини називають множину, що містить цю вершину і всі вершини, суміжні з нею.

Першим центром околу вибирають вершину з максимальним степенем у графі. Вершини околу розфарбовують за таким порядком: кожен раз з околу вибирають вершину з максимальною кількістю розфарбованих суміжних вершин. Центром наступного околу розфарбування вибирають нерозфарбовану вершину в графі з максимальною кількістю розфарбованих суміжних вершин та її окіл і розфарбовують за порядком, зазначеним вище. Процедуру повторюють доти, доки всі вершини в графі не будуть розфарбовані. Вершину розфарбування фарбують фарбою з мінімальним незадіяним номером серед вже розфарбованих суміжних їй вершин.

Реалізацію алгоритму в Паскалі наведено у програмі 12.1.

```
Program Ex_12_1;
  uses crt;
  const maxvert = 30;                          {Максимальна кількість вершин графа}
        mycol: array [0..14] of byte = (7, 9, 10, 12, 13, 14, 11, 15, 1, 2, 4, 5, 6, 3, 8);
  type pNode = ^Node;
       Node = record
                num: byte;
                next: pNode;
              end;
       pList = ^List;
       List = record
                num: byte;
                next: pNode;
                nextl: pList;
              end;
       vectval = array [0..maxvert] of byte;

{*************************************************************************}
{****************** Читати початкові дані із файла ***********************}
{*************************************************************************}

function SrcDataFL(var n: byte): pList;
  var infile: text;
      filename: string[12];
      ch: char;
```

```
    num: byte;
    top, tail, tmp: pList;
    t, tfst: pNode;
begin
  writeln('Ви можете ввести початкові дані або з файла, або з клавіатури.');
  writeln('Введіть ім'я файла (натисніть ENTER для вводу з клавіатури) : ');
  readln(filename);
  assign(infile,filename);
  reset(infile);
  num := 0; n := 0;
  top := nil;
  tail := nil;
  new(t);
  tfst := t;
  while not eof(infile) do
    begin
      while not eoln(infile) do
        begin
          read(infile, ch);
          if (ch >= '0') and (ch <= '9') then num := num * 10 + ord(ch) - 48
            else
              if ch = ' ' then
                begin
                  t^.num := num;
                  new(t^.next);
                  t := t^.next;
                  t^.next := nil;
                  num := 0;
                end
              else writeln('Помилка в даних файла');
        end;
      readln(infile);
      t^.num := num;
      num := 0;
      t^.next := nil;
      new(tmp);
      inc(n);
      tmp^.num := n;
      tmp^.next := tfst;
      tmp^.nextl := nil;
      if top = nil then
        begin
          top := tmp;
          tail := tmp;
        end
```



```
        else
          begin
            tail^.nextl := tmp;
            tail := tmp;
          end;
        new(t);
        tfst := t;
      end;
  close(infile);
  SrcDataFL := top;
end;

{*************************************************************************}
{***************************** Ініціалізація *****************************}
{*************************************************************************}
procedure Init(n: byte; var colors, nc: vectval; var colored: pList );
 var  i: byte;
      tmp: pList;
begin
  for i := 1 to n do
    begin
      colors[i] := 0;
      nc[i] := 0;
    end;
  colors[0] := 0;      {В нульовий елемент заносимо кількість використаних фарб}
  new(colored);
  colored^.num := 1;
  colored^.next := nil;
  tmp := colored;
  for i := 2 to n do
    begin
      new(tmp^.nextl);
      tmp := tmp^.nextl;
      tmp^.num := i;
      tmp^.next := nil;
    end;
  tmp^.nextl := nil;
end;

{*************************************************************************}
{********** Знаходження в головному списку типу pList елемент,**********}
{ ** що відповідає вершині num і повертає покажчик на цей елемент типу pList  **}
{*************************************************************************}
function FindList(top: pList; num: byte): pList;
 var  tl: pList;
```

```
begin
    tl := top;
    while (tl <> nil) and (tl^.num <> num) do
      tl := tl^.nextl;
      FindList := tl;
    end;

procedure InsColnode(rvert: pList; v: byte; color: vectval);
  var  tmp, tn: pNode;
begin                                   {Занесення вершини у список відповідних кольорів}
    if rvert^.next = nil then
      begin
        new(rvert^.next);
        rvert^.next^.num := v;
        rvert^.next^.next := nil;
      end
    else
      if color[v] < color[rvert^.next^.num] then
        begin
          new(tn);
          tn^.num := v;
          tn^.next := rvert^.next;
          rvert^.next := tn
        end
      else
        if color[v] > color[rvert^.next^.num] then
          begin
            tmp := rvert^.next;
            while (tmp^.next <> nil) and (color[tmp^.next^.num] <= color[v]) do
              tmp:= tmp^.next;
            if color[tmp^.num] < color[v] then
              begin
                new(tn);
                tn^.num := v;
                tn^.next := tmp^.next;
                tmp^.next := tn;
              end;
          end;
end;

procedure RemNode(rvert: pList; v: byte);
  var  tmp: pNode;
begin
    if rvert^.next^.num = v then
        rvert^.next := rvert^.next^.next
      else
        begin
          tmp := rvert^.next;
          while (tmp^.next <> nil) and (tmp^.next^.num <> v) do
            tmp := tmp^.next;
          if tmp^.next <> nil then
            tmp^.next := tmp^.next^.next;
        end;
end;

procedure RemList(v: byte; var top: pList);
  var  prev: pList;
begin
    if v = top^.num then
        top := top^.nextl
      else
        begin
          prev := top;
          while (prev^.nextl <> nil) and (prev^.nextl^.num <> v) do
            prev:=prev^.nextl;
          if prev^.nextl <> nil then prev^.nextl := prev^.nextl^.nextl;
        end;
end;

{*************************************************************************}
{************************ Розфарбування вершини ************************}
{*************************************************************************}
procedure ColorTop(vertnc: pList; color: byte; topnc, topc: pList;
                   var colored, neighc: vectval; dotab: boolean);
  var  vertpr: pList;
       num: byte;
       tmp: pNode;
       tls: pList;
       tlsc: pList;
begin
    num := vertnc^.num;
    colored[num] := color;
    if color > colored[0] then
      colored[0] := color;
    tmp := vertnc^.next;
    while tmp <> nil do
      begin
        inc(neighc[tmp^.num]);
        tls := FindList(topnc,tmp^.num);
        RemNode(tls, num);
        tlsc := FindList(topc, tmp^.num);
        InsColnode(tlsc, num, colored);
```

```
            tmp := tmp^.next;
          end;
      if dotab then write(' ') else write('--');
      write('Vertice ', num: 2, ' colored with paint ');
      textcolor(mycol[color]);
      write(color: 2);
      textcolor(mycol[0]);
      writeln('.');
end;
{************************************************************************}
{******************** Процедура розфарбування ***************************}
{************************************************************************}
function ChooseCen(topnc: pList; nc: vectval): pList;
  var  maxl: pList;
       tl: pList;
begin
      tl := topnc;
      maxl := tl;
      tl := tl^.nextl;
      while tl <> nil do
        begin
          if nc[tl^.num] > nc[maxl^.num] then maxl := tl;
          tl := tl^.nextl;
        end;
      ChooseCen := maxl;
end;

function ChooseNeigh(centre: pList; nc: vectval): byte;
  var  tmp: pNode;
       maxnum: byte;
begin
      tmp := centre^.next;
      maxnum := tmp^.num;
      tmp := tmp^.next;
      while tmp <> nil do
        begin
          if nc[tmp^.num] > nc[maxnum] then maxnum:=tmp^.num;
          tmp := tmp^.next;
        end;
      ChooseNeigh := maxnum;
end;

function ChoosePaint(v: byte; topc: pList; color: vectval): byte;
  var  paint: byte;
       tl: pList;
       tmp: pNode;
```



```
begin
      paint := 1;
      tl := FindList(topc, v);
      tmp := tl^.next;
      while (tmp <> nil) and (paint = color[tmp^.num]) do
        begin
          inc(paint);
          tmp := tmp^.next;
        end;
      ChoosePaint := paint;
end;

procedure ColorNeigh(centre: pList; var topnc, topc: pList; var colored, nc: vectval);
  var  num: byte;
       vertn: pList;
       col: byte;
begin
      while centre^.next <> nil do
        begin
          num := ChooseNeigh(centre, nc);
          col := ChoosePaint(num, topc, colored);
          vertn := FindList(topnc, num);
          ColorTop(vertn, col, topnc, topc, colored, nc, true);
          RemNode(centre, num);
          RemList(num, topnc);
        end;
      RemList(centre^.num, topnc);
end;

function MaxPow(top: pList): pList;
  var  count: byte;
       max: byte;
       maxp: pList;
       curm: pNode;
       currl: pList;
begin
      currn := top^.next;
      count := 0;
      while currn <> nil do
        begin
          inc(count);
          currn := currn^.next;
        end;
      max := count;
      maxp := top;
      currl := top^.nextl;
```

```
    while currl <> nil do
      begin
        currn := currl^.next;
        count := 0;
        while currn <> nil do
          begin
            inc(count);
            currn := currn^.next;
          end;
        if count > max then
          begin
            max := count;
            maxp := currl;
          end;
        currl := currl^.nextl;
      end;
    MaxPow := maxp;
end;

{************************************************************************}
{******************************* Фарбування *****************************}
{************************************************************************}
procedure ColorGraph(noncol: pList; n: byte);
  var
    colors: vectval;
    neighbc: vectval;
    topcol: pList;
    centre: pList;
    paint: byte;
begin
    Init(n, colors, neighbc, topcol);
    centre := MaxPow(noncol);
    paint := 1;
    ColorTop(centre, paint, noncol, topcol, colors, neighbc, false);
    ColorNeigh(centre, noncol, topcol, colors, neighbc);
    while noncol <> nil do
      begin
        centre := ChooseCen(noncol, neighbc);
        paint := ChoosePaint(centre^.num, topcol, colors);
        ColorTop(centre, paint, noncol, topcol, colors, neighbc, false);
        ColorNeigh(centre, noncol, topcol, colors, neighbc);
      end;
    writeln;
    writeln('Число кольорів становить: ',colors[0],'.');
end;
```



```
{************************************************************************}
{************************** Головна програма ****************************}
{************************************************************************}
  var n: byte;
      graf: pList;
begin
    clrscr;
    graf := SrcDataFl(n);
    writeln('Розфарбований граф');
    writeln('Центри околів розфарбування помічені "--" . ');
    writeln;
    ColorGraph(graf, n);
    writeln;
    writeln('Для закінчення натисніть будь-яку клавішу...');
    while keypressed do readkey;
    repeat until keypressed;
end.
```

Програма 12.1. Реалізація окільного алгоритму

У поданій реалізації алгоритму граф задають списками суміжності. Для кожної вершини формують лінійний однозв'язний список вершин, що суміжні з нею, причому для полегшення пошуку вершини в такому списку впорядковано за зростанням номерів. Ці списки суміжності зв'язані між собою ще в один динамічний список, який і зображує собою граф.

Для машинної реалізації такої структури даних в розділі **type** оголошують такі типи даних.

```
pNode = ^Node;
Node = record
         num: byte;
         next: pNode;
       end;
```

Змінна типу *Node* є елементом списку суміжності для вершини з полями *num* для номера вершини та *next* – покажчиком на наступний вузол у списку суміжності. Змінна типу *pNode* є покажчиком на тип *Node*.

```
pList = ^List;
List = record
         num: byte;
         next: pNode;
         nextl: pList;
       end;
```

Змінна типу *List* є елементом головного списку з полями *num* для номера вершини, *next* – покажчиком на список суміжності для цієї вершини

і *nextl* – покажчиком на наступний елемент головного списку або на *nil* для закінчення списку. Змінна типу *pList* є покажчиком на тип *List*.

Вхідні дані вводять з файла, який має наступну структуру – стільки рядків, скільки граф має вершин, і в кожному рядку перелічені за порядком ті вершини, з якими суміжна відповідна вершина (номер самої вершини не зазначають).

Для зчитування вхідних даних використано функцію **SrcDataFL**. Функція повертає значення типу *pList*, що є покажчиком на вершину списку задання графа, а також повертає як параметр значення *n* типу *byte*, яке є кількістю вершин у графі.

Для зручності реалізації алгоритму також вводять ще одну додаткову структуру даних – список з такою самою структурою, як і зчитаний граф, який призначений для зберігання списків вже розфарбованих суміжних вершин для кожної вершини. Список, яким представлено граф, у процесі роботи програми перетворюють на список нерозфарбованих вершин, елементи якого вказують на списки нерозфарбованих суміжних вершин для кожної вершини. Отже, на кожному кроці алгоритму ми маємо для кожної вершини відповідні списки розфарбованих і нерозфарбованих суміжних з нею вершин окремо.

Списки розфарбованих вершин упорядковані за зростанням значень кольорів відповідних вершин, що значно спрощує процес вибору фарби при розфарбуванні вершини. Список нерозфарбованих вершин використовують для адекватних перетворень при розфарбуванні вершини, а також для визначення наступного кандидата на розфарбування. Перевагою використання такого списку є те, що при розфарбуванні околу вершини ми маємо вже список, який включає лише нерозфарбовані вершини. Звідси видно, що якщо ми розфарбували вершину, то нас вже не цікавить жоден з таких списків для цієї вершини, і вона нас цікавить хіба що у контексті ще не розфарбованих суміжних з нею вершин для вибору фарби. Тому внесення до списку розфарбованих вершин йде лише для ще не розфарбованих вершин, що значно поліпшує швидкість виконання алгоритму шляхом зменшення глобальних операцій над списками, що задають граф.

У програмі використовують такі процедури та функції.

Процедура **Init** ініціалізує масиви *colors* та *nc* (обнуляє), що зберігають кольори розфарбування вершини, та кількість розфарбованих суміжних вершин для вершини відповідно, а також створює список *colored* типу *pList*, в який поступово заноситимуть уже розфарбовані вершини.

Функція **FindList** знаходить в головному списку типу *pList* елемент, що відповідає вершині *num* і повертає покажчик на цей елемент типу *pList*.

Процедура **InsColnode** вставляє в список суміжності, на який вказує елемент головного списку розфарбованих вершин *rvert*, вершину з номером *v* у такий спосіб, щоб вершини в цьому списку були впорядковані за значеннями кольорів, у які вони розфарбовані (для цього в процедуру передають масив кольорів *color*).

Процедура **RemNode** видаляє зі списку суміжності, на який вказує елемент головного списку *rvert*, вершину з номером *v*.

Процедура **RemList** вставляє у головний список елемент для вершини з номером *v* і повертає покажчик на вершину списку *top* (для того випадку, якщо вставлений елемент буде першим).

Сам алгоритм працює так. Перший центр околу вибирають функцією **MaxPow**, в яку передають покажчик на вершину списку, що являє собою граф. Ця функція підраховує степінь кожної вершини в іще не розфарбованому графі, використовуючи для цього список вершин, що нерозфарбовані, і повертає покажчик на елемент цього списку, який відповідає вершині, що має максимальний степінь у графі.

Кожен наступний окіл розфарбування вибирають функцією **ChooseCen**, в яку передають вершину списку нерозфарбованих вершин та масив, елементи якого є числами, що характеризують кількість розфарбованих суміжних вершин. Оскільки в цьому списку є лише елементи, які відповідають нерозфарбованим вершинам, то наступним центром околу може бути будь-яка з вершин, зображених елементами цього списку. З них вибирають вершину з максимальним значенням кількості суміжних розфарбованих вершин. Функція повертає покажчик на відповідний елемент цього списку.

Окіл вершини розфарбовують за допомогою функції **ChooseNeigh** та процедури **ColorNeigh**. Функція *ChooseNeigh* вибирає з околу вершини номер нерозфарбованої вершини з максимальним значенням кількості розфарбованих суміжних вершин. Процедура *ColorNeigh* виконує весь процес розфарбування околу вершини: доки в околі вершини залишилася хоча б одна нерозфарбована вершина (тобто доки елемент, що відповідає центру околу, у списку нерозфарбованих вершин не вказує на *nil*), з околу вибирають вершину – кандидата на розфарбування за допомогою функції *ChooseNeigh*, і розфарбовують фарбою, вибраною функцією *ChoosePaint* (див. далі). Потім з околу видаляють цю розфарбовану вершину та елемент, відповідний до неї у списку нерозфарбованих вершин. Коли всі вершини в околі розфарбовані, з головного списку видаляють елемент, що відповідає центру околу.

Функція **ChoosePaint** вибирає для вершини фарбу, якою її можна розфарбувати. Для цього простежується список суміжних їй розфарбованих вершин (упорядкований, як зазначалося вище), доки не буде знайдено фарбу, яку ще не використано при розфарбуванні суміжних з нею вершин. Функція повертає номер фарби, що має тип *byte*.

Процедура **ColorTop** виконує розфарбування вершини вибраною фарбою, а також необхідні перетворення даних, що його супроводжують, а саме: присвоєння вершині певного кольору – занесення номера кольору у відповідний елемент масиву *colored*; коли використано новий колір – збільшення кількості використаних фарб, що розміщується в елементі 0 масиву *colored*; для всіх ще не розфарбованих вершин збільшення кількості розфарбованих суміжних з ними вершин на 1 (для розфарбованих вер-

шин нас цей показник вже не цікавить), для цього проходимо по всьому списку суміжності цієї вершини у списку нерозфарбованих вершин; для всіх ще не розфарбованих вершин видаляємо цю вершину з їх списків суміжності у списку нерозфарбованих вершин; для всіх ще не розфарбованих вершин вставляємо цю вершину у список розфарбованих суміжних вершин (для подальшого вибору потрібної фарби); виведення на екран повідомлення, що подану вершину розфарбовано даною фарбою.

Нарешті, процедура **ColorGraph** є головною процедурою, що реалізовує окільний алгоритм розфарбування графа. Крім ініціалізації даних та остаточного виведення кількості використаних фарб, вона виконує наступні дії: вибирає центр першого околу за допомогою функції *MaxPow* і фарбує його першою фарбою, викликаючи функцію *ColorTop*, а також його окіл. Потім повторює процедуру вибору вершини – центру околу, її розфарбування та розфарбування її околу доти, доки список нерозфарбованих вершин не виявиться пустим, тобто не стане вказувати на *nil*.

12.4. Задача виконання робіт

Серед багатьох проблем теорії графів, які є актуальними і потребують вирішення евристичними методами, виділяють також проблему побудови розкладу виконання робіт [27, 92]. Її часто ототожнюють з проблемою побудови розкладу функціонування паралельних обчислювальних систем.

12.4.1. *Паралельні обчислювальні системи*

Ідеї побудови обчислювальних систем з великою кількістю процесорів для досягнення високої продуктивності виникли ще у 1950-ті роки. Незважаючи на намагання, ці ідеї досить довгий час не могли бути реалізовані з достатньою повнотою через відсутність потрібної елементної та програмної бази.

Становище поліпшилось зі створенням великих та надвеликих інтегральних схем, мікропроцесорів та мікро-ЕОМ, а також із розвитком паралельного програмування і паралельних обчислювальних систем. На їх основі було створено такі багатопроцесорні системи, як МАРС, ЗМ, МВК, ILLIAC-4, що послугувало поштовхом для розвитку мов паралельного програмування типу АДА, МАЯК, ОККАМ, ПАРУС, матрична мова потоків даних [27, 34, 35, 37, 55].

Отже, з накопиченням досвіду почали розробляти все більше і більше систем зі зростаючою кількістю процесорів, виявляючи їхні переваги і вади, визначаючи найперспективніші архітектури. Нарешті, в середині 1980-х років паралельні архітектури подолали межу практичного застосування і вийшли на ринок, що надало їхньому розвитку нового прискорення, яке підтримується все досконалішими мікропроцесорами і мікро-ЕОМ [36].

Обчислювальні системи з великою кількістю процесорів характеризуються підвищеною надійністю, стійкістю, готовністю і продуктивністю, можливістю мати значні обчислювальні ресурси.

Паралельна програма має такі частини, що можуть працювати одночасно (паралельно).

Під паралельною обробкою інформації розуміють одночасне виконання двох або більше частин однієї і тієї ж програми двома або більше основними процесорами обчислювальної системи (в принципі, обробку можна здійснювати як в однопрограмному, так і в мультипрограмному режимі, але ми розглядатимемо тільки однопрограмний). Обчислювальні системи та програми для паралельної обробки інформації називають, в свою чергу, паралельними. Паралельна обчислювальна система має не менше двох процесорів, що можуть одночасно й узгоджено здійснювати обробку інформації.

Паралельні обчислювальні системи можна поділити на такі: багатопроцесорні, багатомашинні, конвеєрні, матричні, асоціативні, з комбінованою структурою, інші. Багатопроцесорна система має не менше двох процесорів, загальну оперативну пам'ять, загальну зовнішню пам'ять і периферійні пристрої.

Обчислювальні системи з багатьма процесорами за їх ознаками можна поділити на однорідні та неоднорідні; процесори, що входять до складу таких систем, можуть бути універсальними і спеціалізованими, мати різну продуктивність (продуктивність процесорів вважають відомою).

Кожен процесор може мати власну оперативну пам'ять. Процесори пов'язані із загальною оперативною пам'яттю через комутатор (перехресний, матричний тощо) або шину (магістралі). Загальне керування всіма апаратними і програмними засобами обчислювальної системи здійснює операційна система. Основна відмінність від багатомашинних комплексів полягає в тому, що в останніх розділена пам'ять і кожна машина керована своєю операційною системою; до багатомашинних систем можуть входити і багатопроцесорні.

Враховуючи хоч і великий, проте обмежений процесорний ресурс паралельних обчислювальних систем, на практиці першочергове значення приділяють побудові ефективного обчислювального алгоритму.

Програмування для багатопроцесорних і багатомашинних систем пов'язано з розпаралелюванням і синхронізацією обчислень, а також з організацією виконання паралельних обчислювальних процесів. А такі дії мають своїм підґрунтям низку досить складних задач, серед яких важливими є розрахунки часових характеристик і кількості операцій, побудова розкладів (планів) виконання паралельних програм на таких системах.

Розповсюдженим унаслідок своєї зручності є підхід до розрахунку цих характеристик і побудови розкладу на основі задання програм як графових моделей (графи з імовірнісними і детермінованими парамет-

рами, в яких відображена структура програм і враховані обчислювальні параметри ділянок програм та умови переходу між ними).

Для простоти розглянемо моделі програм без циклів, отже відповідні графи не матимуть контурів.

12.4.2. *Задача побудови розкладу виконання робіт*

Розглянемо загальну задачу побудови розкладу виконання робіт. Нехай є: набір робіт $W = \{w_1, w_2, ..., w_n\}$; множина $P = \{P_1, ..., P_m\}$ ідентичних процесорів; з кожною w_i пов'язують час її обробки t_i, $i = 1, 2, ..., n$, якимось із процесорів із P; відношення часткового порядку < на W – залежність робіт одна від одної; заздалегідь визначений межовий (директивний) час закінчення виконання всіх робіт d.

Потрібно відповісти на питання: чи існує m-процесорний розклад для завдань з W, при якому всі завдання виконають за час, менший або рівний d (*задача 1*).

Інакше кажучи, чи існує функція f: $W \to \{0, 1, ..., d\}$, така що

$$\forall I \in \{0, 1, ..., d\} \text{ і } \forall w \in W \Rightarrow f(w) = I,$$

і якщо $w < w'$, де $w' \in W$, тоді $f(w) < f(w')$.

Задачі, що зводяться до задачі 1:

- мінімізувати загальний час виконання всіх робіт;
- якщо всі роботи неможливо виконати за директивний час d, тоді слід мінімізувати кількість невиконаних робіт.

Карпом [55] було показано, що задачу 1 розв'язують за поліноміальний час, якщо дотримується хоча б одна з умов: m – фіксоване, а < – ліс або має доповненням тріангульований граф (на практиці малоймовірний випадок); $m = 2$, задача 1 є *NP*-повною, навіть у випадках: $d = 3$; час виконання кожної роботи дорівнює 1; кількість процесорів дорівнює 2, а час виконання довільної роботи – рівний 1 або 2.

На множині робіт W, з урахуванням часткового порядку виконання робіт <, побудуємо орієнтований граф залежності виконання робіт $G = (W, <)$; d визначатиме мінімально можливий час виконання всіх елементів із W. Кількість процесорів із P, які задовольнятимуть такому розкладу, також намагатимемось мінімізувати, де це можливо. При цьому вважатимемо, що обмін даними між процесорами проходить за константний час (або час = 1), незалежно від розміру даних. Вважатимемо, що кожен процесор в будь-який момент часу виконує не більше однієї роботи.

Оптимальні розклади виконання робіт будують за допомогою методів динамічного програмування або методу гілок і границь (множину всіх можливих розв'язків розбивають на класи, які потім аналізують) або методів оптимізаційного типу. Оптимальні розклади використовують, звичайно, для порівняльного аналізу наближених розкладів і оцінки потенційних можливостей диспетчеризації. На практиці побудова оптимальних розкладів істотно обмежується розмірністю початкової задачі. За даними праці [38], за допомогою методу динамічного програмування вдалося побудувати оптимальний розклад для графів з максимальною кількістю вершин 24.

Побудуйте алгоритм розкладу виконання робіт методом динамічного програмування.

Це дало поштовх до розробки великої кількості евристичних алгоритмів побудови наближених розкладів, які можна поділити на дві групи, що за деяких умов можуть збігатися: локально-оптимальні (забезпечують оптимальність на окремих етапах обчислень і не гарантують оптимальність загалом) і евристичні.

Методи побудови розкладів можна поділити на два великих класи: неітераційні (розклад отримують один раз і не коригують) та ітераційні (отриманий початковий розклад коригують за допомогою спеціалізованої процедури поліпшення, яка все більше наближає його до оптимального).

Операційні проблеми паралельного програмування (програмування для паралельних систем) містять задачі апріорного оцінювання кількісних характеристик процесорів, таких як очікуваний час виконання процесів, очікувана кількість потрібних операцій, необхідна кількість процесорів. Ці характеристики, а також задачі побудови прийнятних розкладів виконання паралельних програм називають операційними.

У процедурі побудови розкладу можна виділити три основні операції: вибір операторів (робіт) для їх призначення на процесори; вибір процесорів для призначення на них операторів; призначення.

Методи вибору роботи (оператора) можна поділити за єдиним некомбінованим критерієм і за комбінованим (складеним) критерієм. Якщо під критерій підпало кілька кандидатів, то вибір здійснюють, як правило, довільно. У всіх випадках при виборі операторів використовують їх впорядкованість (залежність одного від одного, послідовність виконання). В низці розкладів використовують послідовність надходження операторів у чергу готових до виконання і здійснюють вибір того оператора, який надійшов у чергу раніше других.

Інші характеристики, що впливають на вибір оператора [36], можна поділити на локальні структурні та часові й глобальні структурні та часові (табл. 12.1.).

Для характеристик 3–6 вибрано назви, в основу яких покладено фундаментальне поняття довжини критичного шляху – ДКШ (шляху найбільшої довжини), що становить єдину змістову базу цих характеристик. За довжину шляху прийнято число вершин у шляху (ДКШ за кількістю операторів), або (ДКШ за часом) час, витрачений на виконання операторів шляху.

Структуру графів враховують також у характеристиках КР ЧАС ШЛЯХ ПОЧ та КР ЧАС ШЛЯХ КІН, але у них час виконання одного завдання

Таблиця 12.1. Характеристики вибору оператора

Характеристика	Структурна	Часова
ЛОКАЛЬНА	1) Зв'язність на глибину 1 (кількість нащадків) – ЗВ'ЯЗНІСТЬ	2) Час виконання оператора – ЧАС
	3) Довжина критичного шляху по кількості операторів від початкового оператора до даного – КР ОП ШЛЯХ ПОЧ	4) Довжина критичного шляху по часу від початкового оператора до даного – КР ЧАС ШЛЯХ ПОЧ
ГЛОБАЛЬНА	5) Довжина критичного шляху по кількості операторів від даного оператора до кінцевого – КР ОП ШЛЯХ КІН	6) Довжина критичного шляху по часу від даного оператора до кінцевого – КР ЧАС ШЛЯХ КІН

може виявитися вирішальним фактором. Ці характеристики є узагальненням структурних характеристик КР ОП ШЛЯХ ПОЧ та КР ОП ШЛЯХ КІН. Перші та другі збігаються у разі одиничного часу виконання операторів.

Характеристики з табл. 12.1 використовують в алгоритмах вибору операторів у низці відомих спискових розкладів, наприклад:

- найбільша зв'язність;
- найбільший або найменший час виконання;
- найменший критичний за кількістю операторів або за часом шлях від початкового оператора;
- найбільший критичний за кількістю операторів або за часом шлях до останнього оператора.

Напишіть програмну реалізацію одного із запропонованих алгоритмів.

Розглянемо питання вибору процесорів. У разі ідентичних процесорів враховують їх зайнятість і вибирають будь-який вільний процесор. В інших випадках доведеться враховувати можливості та продуктивність процесорів (або вибирати процесор з найбільшою продуктивністю, або підбирати такий процесор і такий оператор, що процесор вивільниться раніше, ніж за інших комбінацій зіставлення процесора і оператора тощо).

Також можна ввести обмеження у вигляді закріплення операторів за окремими процесорами (наприклад, у системах зі спеціалізованими складовими). Після вибору оператора і процесора слід призначити вибраний оператор на виконання на вибраному процесорі.

Варіанти призначення можуть бути *з перериваннями*, коли процес виконання обчислень може бути перерваний для виконання нових операторів, та *без переривань*.

Призначення операторів може бути *незатримним*, якщо їх виконання починається відразу після вивільнення будь-якого процесора. Цей принцип отримав відповідно назву «не звільнювати процесори», а розклади, побудовані за таким алгоритмом – *незатримними*. Такий підхід застосовують найчастіше для систем з великою кількістю ідентичних процесорів.

Штучний простій процесорів і відповідні затримки у призначенні готових до виконання операторів на вільні процесори буває виправданий, наприклад, чеканням вивільнення процесорів з більш високою продуктивністю (для систем з неідентичними процесорами).

12.4.3. *Принципи побудови і критерії оптимальності розкладів*

Вважатимемо, що паралельні програми розділені на відповідні програмні елементи (оператори або задачі, що визначають завдання на виконання обчислювальних робіт).

Вважаємо також, що незалежні процесори обчислювальних систем працюють в однопрограмному режимі; в динаміці виконання програми утворюються черги готових до виконання операторів (операторів з уже обчисленими аргументами). Розклади паралельних обчислювальних комплексів можна класифікувати [37] за принципами їх побудови, критеріями оптимальності, ознакою виконання цих критеріїв, за методами вибору операторів і процесорів і призначення завдань на виконання.

Залежно від принципу побудови можна виділити статичні (апріорні) та динамічні розклади (перші будують під час підготовки до розв'язання задачі, а другі – в процесі розв'язання). При побудові динамічних розкладів не виключена можливість виконання деяких попередніх робіт у період підготовки до розв'язання задачі (наприклад, визначення характеристики і пріоритетів операторів).

Основними *критеріями оптимальності* розкладів для детермінованих моделей паралельних програм є такі: мінімізація часу виконання програми; мінімізація кількості процесорів, потрібних для виконання; мінімізація середнього (зваженого) часу закінчення виконання завдань; мінімізація часу бездіяльності процесорів; максимізація завантаження процесорів.

Перші два критерії називають мінімаксними. У першому потрібно мінімізувати максимальний час закінчення обробки операторів за заданої кількості доступних процесорів, тобто максимізувати продуктивність під час виконання даної програми. У другому випадку потрібно мінімізувати кількість використовуваних процесорів за умови виконання програми за час, що не перевищує заданий (якщо це можливо). Критерій мінімізації середнього часу закінчення виконання завдань прямо враховує моменти закінчення кожного завдання, він орієнтований на найшвидше вивільнення ресурсів. Останні два критерії орієнтовані на найповніше використання процесорного часу.

При побудові розкладів, особливо для систем реального часу, можна встановити директивні (критичні) терміни. Якщо використовувати моделі паралельних програм з імовірнісними характеристиками, показники ефективності і характеристики оптимальності формулюють, звичайно,

у вигляді ймовірнісних аналогів вищезазначених критеріїв. Якщо розклад повністю задовольняє критерій, його називають оптимальним, якщо ні – наближеним.

12.4.4. *Спискові розклади*

Серед евристичних розкладів дуже популярні спискові (пріоритетні) розклади (що є підмножиною розкладів без переривань) унаслідок їх ефективності та простоти. Зміст спискових розкладів полягає у наступному: *роботи впорядковують в лінійний список згідно з деякими пріоритетами, виробленими заздалегідь і приписаними кожній роботі.* Пріоритети будують на основі певних евристик, за основу вибору яких покладено поняття критичного шляху.

Оператори впорядковують у вигляді лінійного списку за спаданням пріоритетів. У процесі виконання програм здійснюється динамічне призначення операторів відповідно до їх пріоритетів на процесори для виконання. Пріоритети можуть бути змінними, а списки операторів – статичними і динамічними.

У міру того як у динаміці обчислювального процесу оператори паралельної програми стають готовими до виконання, а процесори вивільняються, здійснюється вибір нових операторів і розподіл їх по процесорах у такий спосіб, щоб зменшити затримки в обчислювальному процесі.

Реалізація спискових розкладів грунтується на застосуванні лінійних списків операторів, упорядкованих за спаданням пріоритетів відповідно до прийнятого алгоритму вибору операторів. Списки можна скласти відразу або поновити під час виконання. У динаміці здійснюється вибір із списку операторів, готових до виконання операторів з найвищим пріоритетом. Алгоритми вибору операторів і алгоритми призначення операторів на процесори в спискових розкладах практично незалежні одні від інших і є неітераційними. Декілька алгоритмів вибору операторів зображено у табл. 12.2, що взято з праці [36]. Більшість алгоритмів розглядають детерміновані графові моделі з частково впорядкованими вершинами (без циклів та розгалужень), що мають одну вхідну та одну вихідну вершину.

Вибір операторів за допомогою алгоритмів може бути неоднозначним. У разі отримання в результаті дії алгоритму невеликої множини операторів вибір роблять випадково, якщо ж множина операторів після дії алгоритму недостатньо мала – алгоритм неефективний.

Алгоритми в табл. 12.2 розділено на 5 груп.

Найпростіші алгоритми: довільний вибір операторів з множини готових до виконання, використовуються тільки дані про послідовність виконання (вважається, що пріоритети всіх операторів однакові). В алгоритмах другої групи (3–5) використовують (4–9) локальні, а в алгоритмах третьої групи – глобальні характеристики відповідних моделей програм, включаючи структурні та часові характеристики.

Таблиця 12.2. Кодування алгоритмів

№	Алгоритм вибору операторів	Код
1	а) Послідовний вибір із довільного лінійного списку операторів, що відповідає їх упорядкуванню б) Рівноймовірний випадковий вибір оператора	ДОВІЛЬНИЙ ВИБІР
2	Вибір оператора, що став готовим до виконання раніше інших	ГОТОВИЙ РАНІШЕ ІНШИХ
3	Вибір оператора, що має найменший час виконання	НАЙМ ЧАС
4	Вибір оператора, що має найбільший час виконання	НАЙБ ЧАС
5	Вибір оператора з найбільшою зв'язністю на глибину 1	НАЙБ ЗВ'ЯЗНІСТЬ
6	Вибір оператора з найменшим критичним шляхом за кількістю операторів від початкового оператора	НАЙМ КР ОП ШЛЯХ ПОЧ
7	Вибір оператора з найбільшим критичним шляхом за кількістю операторів до кінцевого оператора	НАЙБ КР ОП ШЛЯХ КІН
8	Вибір оператора з найменшим критичним шляхом за часом від початкового оператора	НАЙМ КР ЧАС ШЛЯХ ПОЧ
9	Вибір оператора з найбільшим критичним за часом шляхом до кінцевого оператора	НАЙБ КР ЧАС ШЛЯХ КІН
10	Найбільша сума нормованих критичних за часом і за кількістю операторів шляхів до кінцевого оператора	НАЙБ КР ОП ЧАС ШЛЯХ КІН
11	Вибір оператора з найбільшою зв'язністю на глибину 1, коли він має найменший критичний за кількістю операторів шлях від початкового оператора	НАЙБ ЗВ'ЯЗНІСТЬ / НАЙМ КР ОП ШЛЯХ ПОЧ
12	Вибір оператора з найбільшою зв'язністю на глибину 1, коли він має найбільший критичний за кількістю операторів шлях від початкового оператора	НАЙБ ЗВ'ЯЗНІСТЬ / НАЙБ КР ОП ШЛЯХ ПОЧ
13	Вибір оператора з найменшим критичним за кількістю операторів шляхом від початкового оператора, коли він має найбільший час виконання	НАЙМ КР ОП ШЛЯХ ПОЧ / НАЙБ ЧАС
14	Вибір оператора з найменшим критичним за кількістю операторів шляхом від початкового оператора, коли він має найбільшу зв'язність на глибину 1	НАЙМ КР ОП ШЛЯХ ПОЧ / НАЙБ ЗВ'ЯЗНІСТЬ
15	Вибір оператора з найбільшим критичним за кількістю операторів шляхом до кінцевого оператора, коли він має найбільшу зв'язність	НАЙМ КР ОП ШЛЯХ КІН / НАЙБ ЗВ'ЯЗНІСТЬ
16	Вибір оператора з найбільшим критичним за кількістю операторів шляхом до кінцевого оператора і, в першу чергу, оператора, що знаходиться на критичному за кількістю операторів шляху графа загалом	НАЙБ КР ОП ШЛЯХ КІН / КР ОП ШЛЯХ ГРАФА
17	Оптимальний оператор, в тому числі за допомогою методів динамічного програмування і гілок та границь	ОПТИМАЛЬНИЙ

В алгоритмах (4–9) реалізовано принцип, згідно з яким передусім слід виконувати ті завдання, які могли б надалі затримати хід паралельного обчислювального процесу загалом більшою мірою. У списках для розкладів, основаних на алгоритмах (4, 5, 7, 9, 10), вищий пріоритет мають ті оператори, відповідні характеристики яких більші за величиною; у списках для розкладів, основаних на алгоритмах (3, 6, 8), більш

високий пріоритет мають оператори, відповідні характеристики яких менші.

В алгоритмі (10) враховують глобальні часові і глобальні структурні характеристики, що роблять цей розклад стійкішим до зміни часових і структурних характеристик. Пріоритет кожного оператора-вершини визначається величиною зваженої суми значень двох вказаних характеристик для даного оператора-вершини; залежно від специфіки коефіцієнти нормування можуть бути різними.

Четверту групу (11–16) становлять алгоритми, в яких послідовно застосовують два тих або інших початкових алгоритма з метою поліпшення розкладу.

П'ята група – це оптимальні алгоритми (17).

Розгляньте застосування описаних алгоритмів на конкретному прикладі.

Алгоритми вибору операторів можна зобразити в різних формах, наприклад, алгоритми НАЙМ КР ОП ШЛЯХ ПОЧ та НАЙБ КР ОП ШЛЯХ КІН на основі поняття розділення графів на рівні (багатоступеневі графи), у яких лежать вершини з найбільш ранніми або найбільш пізніми термінами виконання відповідно.

Кожен q-й рівень з розділення графа, рахуючи від початкової вершини, в першому випадку (від кінцевої вершини у другому випадку) містить всі вершини, для яких довжина критичного шляху від початкової (кінцевої) вершини дорівнює q.

Відповідні алгоритми вибору операторів, як і характеристики, що лежать у їх основі, збігаються за певних умов (наприклад, за одиничних значень часу виконання операторів). Якщо графова модель має структуру дерева з однією вихідною вершиною, то алгоритм НАЙМ КР ОП ШЛЯХ ПОЧ = НАЙМ КР ЧАС ШЛЯХ ПОЧ = НАЙБ КР ОП ШЛЯХ КІН = НАЙБ КР ЧАС ШЛЯХ КІН = «Алгоритм Ху» – оптимальний алгоритм.

Алгоритм НАЙМ КР ЧАС ШЛЯХ ПОЧ стає алгоритмом «найменший час», а алгоритм НАЙБ КР ЧАС ШЛЯХ КІН стає алгоритмом «найбільший час» для випадку незалежних одна від іншої обчислювальних робіт (графи з ізольованими вершинами).

12.4.5. *Показники ефективності розкладів*

Розглянемо деякі основні показники ефективності розкладів паралельних програм і розпаралелювання обчислювальних процесів, що ними реалізуються [37]. Ці показники розраховані на розклади, метою яких є мінімізація часу виконання паралельних програм. Як вихідні дані для оцінки показників ефективності розкладів можна використовувати такі величини: $T_i(m)$ – час від моменту початку до моменту завершення виконання паралельної програми на m-процесорній системі за використання i-го розкладу; $\tau(m)$ – мінімальний час виконання паралельної програми на m-процесорній системі (досягають за використання оптимального розкладу); $t(m)$ – час виконання програми на однопроцесорній системі з продуктивністю, що дорівнює сумарній продуктивності всіх процесорів m-процесорної системи.

Кількість процесорів виконання і довжину розкладу також можна розцінювати як критерії ефективності, тоді перевага в оцінці надається середньому часу закінчення виконання завдань.

Покладемо, що m-процесорна система ($m \geq 2$) може мати процесори як з однаковою, так і з різною продуктивністю, що виконання програм здійснюється в однопрограмному режимі (це дає змогу відділити вплив розкладу на характеристики обчислювальних процесів від впливу інших факторів) і що показники ефективності врешті-решт усереднюються за множиною одних і тих самих програм і вхідних даних. Для конкретних і середніх значень відповідних величин скористаємося одними й тими самими позначеннями, якщо це не призводить до неоднозначності.

Зазначимо, що у величинах $T_i(m)$, $\tau(m)$, $t(m)$ час на роботу операційної системи не враховано. З означень перелічених величин видно: $t(m) \leq \tau(m) \leq T_i(m)$ для обчислень за однією і тією самою програмою з однаковими вхідними даними. Ефективність m-процесорної системи щодо забезпечення розпаралелювання обчислювального процесу оцінюватимемо величиною $e(m) = t(m)/\tau(m)$ $(0 < e(m) \leq 1)$. Вона показує (за інших рівних умов) зменшення мінімально можливого часу виконання паралельної програми m-процесорною системою при переході до однопроцесорної системи з продуктивністю процесора, яка дорівнює сумарній продуктивності m процесорів багатопроцесорної системи.

Величина $e(m)$ характеризує потенційні можливості розпаралелювання обчислювальних процесів унаслідок застосування оптимального розкладу, з поліпшенням яких вона зростає. Визначення цієї величини, звичайно, спричинює ускладнення, оскільки для знаходження $\tau(m)$ слід побудувати оптимальний розклад.

Ефективність деякого i-го розкладу виконання паралельної програми на m-процесорній системі традиційно оцінюють величиною $f_i(m) = \dfrac{\tau(m)}{T_i(m)}$, $0 < f_i(m) \leq 1$, що показує (за інших рівних умов) зменшення часу виконання паралельної програми на m-процесорній системі при переході від деякого i-го до оптимального розкладу. Чим ліпше i-й розклад, тим ближче він до оптимального у цьому смислі, і тим більше значення матиме величина $f_i(m)$. Знайти $f_i(m)$ складно з тієї самої причини, що і $e(m)$.

Ефективність розпаралелювання обчислювального процесу, що забезпечується деяким i-м розкладом при виконанні паралельної програми на m-процесорній системі оцінюють величиною $g_i(m) = \dfrac{t(m)}{T_i(m)}$, $0 < g_i(m) \leq 1$,

що показує (за інших рівних умов) зменшення часу виконання паралельної програми (у разі використання i-го розкладу) при переході до однопроцесорної системи з продуктивністю, що дорівнює сумарній продуктивності m процесорів багатопроцесорної системи. Величина $g_i(m)$ характеризує ефективність m-процесорної системи щодо забезпечення розпаралелювання обчислювального процесу за умови застосування i-го розкладу. З вищенаведених формул видно, що $g_i(m) = e(m) * f_i(m)$, тобто будь-які два з трьох показників ефективності $e(m)$, $f_i(m)$ та $g_i(m)$ визначають третій показник, а також видно, що $g_i(m) \le e(m)$ та $g_i(m) f_i(m)$.

Для ліпших розкладів у цьому смислі величина $g_i(m)$ набуває більших значень. При визначенні значень $g_i(m)$, на відміну від $e(m)$ та $f_i(m)$, не потрібно знати оптимального розкладу. Крім цього, оскільки $g_i(m) \le f_i(m) \le 1$, то $g_i(m)$ можна використовувати як нижню межу для $f_i(m)$. Якщо величина $g_i(m)$ виявляється досить близькою до 1, то можна дійти висновку, що і $f_i(m)$ не менш близька до 1. Останнє означає, що i-й розклад розгляду досить близький до оптимального за ефективністю і що цю наближеність можна кількісно оцінити за величиною, не більшою за $1 - g_i(m)$. Наголосимо, що для таких висновків не потрібна побудова оптимального розкладу.

Наведена інтерпретація показників ефективності $e(m)$, $f_i(m)$, та $g_i(m)$ не є єдиною. Так, наприклад, можна говорити про точність розв'язання задачі планування паралельних обчислювальних процесів за допомогою деякого i-го розкладу, маючи на увазі його наближеність до оптимального розкладу, що дає точний розв'язок задачі. Можна говорити також про завантаження множини процесорів у разі використання i-го розкладу і характеризувати її величиною $g_i(m)$. Ступінь найвищого завантаження множини процесорів можна характеризувати величиною $e(m)$.

Під час дослідження розкладів досить часто виявляється необхідним порівнювати їх між собою. Ефективність j-го розкладу у порівнянні з ефективністю i-го можна оцінити відношенням часу виконання програм, що відповідають цим розкладам $h_{ij} = T_i(m) / T_j(m)$.

Наголосимо, що при порівнянні розкладів необхідно користуватися не тільки показниками їх ефективності вищезазначеного типу, що засновані на порівнянні забезпечуваних часів виконання паралельних програм (точностей розв'язання задачі планування паралельних обчислювальних процесів), а й враховувати затрати обчислювальних ресурсів, що потрібні для побудови і реалізації розкладів.

12.4.6. *Евристичний алгоритм побудови розкладу*

Опишемо евристичний списковий алгоритм побудови розкладу роботи багатопроцесорного паралельного комплексу, який забезпечує виконання всіх робіт за мінімально можливий час у разі достатньої кількості процесорів для максимального розпаралелювання без переназначень робіт [27].

Нехай є деяка багатопроцесорна система $P = \{p_1, p_2, ..., p_m\}$ однотипних процесорів, де m – таке число, що дозволяє максимально можливе розпаралелювання алгоритму розв'язання певного класу задач розгляду. На множині робіт W (послідовні підблоки паралельного алгоритму w_i, $i = 1, 2, ..., n$), з урахуванням часткового порядку $<$ виконання робіт, побудуємо орієнтований граф залежності виконання робіт $G = (W, <)$. Враховуючи можливість зведення циклічного графа до ациклічного шляхом заміни робіт, які утворюють цикл, однією роботою вищого рівня абстракції, вважатимемо, що побудований граф не має контурів. З кожним елементом w_i пов'язують час його обробки t_i процесором із P, $i = 1, 2, ..., n$. Час виконання робіт може бути різний. Потрібно знайти розклад виконання W на P f: $W \to \{0, 1, ..., d\}$, такий що для $\forall\, I \in \{0, 1, ..., d\}$ і $\forall\, w \in W \Rightarrow$ $\Rightarrow f(w) = I$, і якщо $w < w'$, де $w' \in W$, тоді $f(w) < f(w')$. Тут d визначатиме мінімально можливий час виконання всіх елементів із W. Кількість процесорів із P, які задовольнятимуть такому розкладу, також намагатимемось мінімізувати. При цьому вважатимемо, що час обміну даними між процесорами проходить за одиницю часу, незалежно від розміру даних обміну.

Для розв'язання цієї задачі пропонується використовувати евристичний алгоритм 12.1, який належить класу ітераційних спискових алгоритмів. На вхід алгоритму подають граф $G = (W, <)$, кожній вершині w_i графа G приписано мітку вартості t_i, час виконання відповідного підблока, $i = 1, 2, ..., n$, і множину однотипних процесорів $P = \{p_1, p_2, ..., p_m\}$. На виході алгоритму матимемо розклад f у вигляді масиву списків $A[0..d]$. Елементами списку $A[j]$, $0 \le j \le d$ будуть цілі числа від 1 до n. Наприклад, якщо $A[3] = \{1, 4, 7, 9\}$, тоді такий запис говоритиме, що в третій такт часу виконання деякі чотири процесори із P оброблятимуть відповідно 1, 4, 7 і 9-ту роботу (оператор).

Список пріоритетів будують згідно з такою евристикою. Кожній вершині графа ставлять у відповідність двомісний кортеж елементів. Перший елемент кортежу – мінімально можливий час початку виконання відповідної роботи із W, другий елемент – мінімально можливий час закінчення виконання відповідної роботи із W. Використовуючи алгоритм лексикографічного впорядкування [9], розміщуємо кортежі за зростанням у лексикографічному порядку та аналізуємо цю послідовність для призначення робіт на процесори. Потім за допомогою спеціальної процедури робимо спробу мінімізувати кількість процесорів, які забезпечать виконання всіх робіт за мінімально можливий час.

Початок

1. (*Початкова ініціалізація*). Будуємо граф $G' = (W', <')$. $W' = W \cup \{w_0\}$, де $<' = < \cup \{w_0, w_j\}$, $j = 1, 2, ..., k$, де k – кількість вершин графа G, які не мають жодного вхідного ребра, а w_j – одна з цих вершин.
2. Для всіх $w \in W'$ обчислимо згідно з $<'$:

а) мінімально можливий час початку виконання роботи φ_w дорівнює вартості максимального шляху від w_0 до w, визначений за допомогою алгоритму Дейкстри;

б) час закінчення обробки роботи $\Psi_w = \varphi_w + t_w$.

3. Будуємо послідовність $B[n]$ упорядкованих за зростанням у лексикографічному порядку кортежів $\{\varphi_i, \Psi_i\}$, $i = 1, 2, ..., n$.
4. (*Формування розкладу f*). Переглядаючи масив $B[n]$, визначаємо рівні паралельного виконання $A[i]$ у наступний спосіб. В $A[1]$ заносимо роботу із $B[1]$. Продовжуємо наповнювати його із масиву B роботами, час початку виконання яких φ_w не більший за мінімальний час закінчення виконання вже занесених у рівень робіт. Перший елемент із B, який не задовольняє умові поповнення рівня, визначає першу роботу наступного рівня.

   Процес побудови рівнів продовжують доти, доки не буде вичерпана послідовність B.

   Кількість рівнів виконання h визначатиме кількість призначень робіт на процесори. Максимальний час закінчення робіт останнього рівня h визначає мінімально можливий час d виконання всіх робіт.
5. Визначимо кількість процесорів m, які потрібні алгоритму для виконання всіх робіт, визначивши максимальну ширину рівня серед h рівнів розкладу. Якщо $m \le 2$, тоді перейти на *крок 6*, в іншому разі обчислимо початкове значення кількості процесорів, до якого оптимізуватимемо отриманий розклад $A[h]$:

   $m' = \max\{\lceil n/h \rceil, l\}$, де l – ширина останнього рівня розкладу виконання робіт $A[h]$.

   Викличемо процедуру оптимізації OPTIM (G, A, h, m, m').
6. Вивести до друку розклад f, надрукувавши значення масиву рівнів $A[h]$ і кількість потрібних процесорів m.

Кінець.

Процедура **OPTIM**

1. Початкову кількість потрібних процесорів визначають **max** := ***m***;
2. *доки* **max** ≠ ***m'***, *робити*
   початок
   повторити
   для всіх ***k*** від ***h*** – **1** до **1**
   зробити
   початок
   l := ***k*** – **1**;
   нехай **r** ширина рівня ***A*[*k*]**;
   доки ***r*** < ***m'*** *робити*
   початок
   якщо для якоїсь роботи ***j*** рівня ***A*[*l*]** виконуються умови:
   1) серед робіт рівня ***A*[*k*]** немає спадкоємців роботи ***j***;
   2) $\Psi_j \le$ всіх φ_i, ***i*** = **1**, **2**, …, ***q***, де ***q*** ширина рівня ***l***.

тоді
початок
перенести роботу ***j*** з рівня ***A*[*l*]** в рівень ***A*[*k*]**;
q := ***q*** – **1**;
r := ***r*** + **1**
кінець
кінець
кінець
поки не було зроблене перенесення роботи;
якщо **max** ≠ ***m'*** *тоді*
початок
max присвоїти максимальне значення ширини ***A*(*i*)**, ***i*** = **1**, **2**, …, ***h***;
m' := ***m'*** + **1**
кінець
кінець.

3. Кількість потрібних процесорів алгоритму ***m*** := ***m'***.
4. Кінець.

Алгоритм 12.1. Евристична побудова розкладу виконання робіт

Теорема 12.6. Розклад f виконання робіт багатопроцесорної системи P з обробки паралельного алгоритму, структуру якого задає граф залежності $G = (W, <)$, визначається в алгоритмі масивом рівнів $A[d]$ коректно, а значення d є мінімально можливим. Часова оцінка алгоритму буде $O(n^2)$, де n – кількість робіт.

Доведення. Коректність алгоритму випливає із правильності використаних алгоритмів Дейкстри і лексикографічного впорядкування, структури алгоритму та скінченності всіх циклів алгоритму.

Визначимо часову оцінку роботи алгоритму.

Крок 1 потребує константного часу. Часову оцінку кроку 2 визначає часова оцінка алгоритму Дейкстри – $O(n^2)$. Сформувати впорядковану в лексикографічному порядку послідовність двомісних кортежів (крок 3) можна за час $O(2(t + n))$, де $t = \max\{t_i\}$, $i = 1, 2, ..., n$. На виконання кроку 4 потрібно $O(n)$. Часову оцінку кроку 5 визначають часом роботи процедури OPTIM.

Розглянемо найгірший варіант початку роботи процедури. Нехай маємо таку структуру рівнів виконання. На першому рівні $A[1]$ знаходиться $(n - h + 1)$ роботи, а на рівнях $A[2]$, $A[3]$, …, $A[n]$ по одній роботі, $(n - h)$ робіт першого рівня можна перемістити в довільний із рівнів. Тоді оптимальна кількість процесорів визначатиметься: $(n - h)/h + 1$. Отже, можна перенести по $(n - h)/h$ робіт на вільні рівні. В $A[h]$ рівень можна зробити потрібний перенос, виконавши $(n - h)(h - 1)/h$ дій; в $A[h - 1]$ рівень: $(n - h)(h - 2)h$ дій; в $A[h - 2]$ рівень: $(n - h)(h - 3)h$; в $A[2]$ рівень: $(n - h)h$. Визначимо сумарну оцінку кількості витрачених переносів:

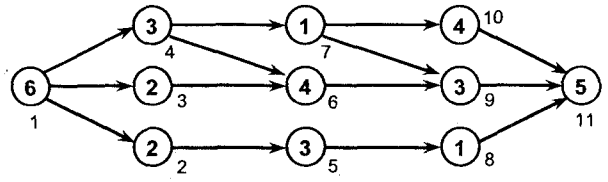


Рис. 12.1. Граф залежності робіт

$(n-h)h((h-1)+(h-2)+(h-3)+\ldots+1)=(n-h)(h-1)/2$. Вона становитиме $O(nh)$.

Враховуючи те, що початковий граф залежності робіт можна задати матрицею суміжності за $O(n^2)$ і виведення результату (крок 6) за $O(n)$, загальна часова оцінка алгоритму визначатиметься:

$$O(n^2)+O(n^2)+O(2(t+n))+O(n)+O(nh)+O(n)=O(n^2).$$

Розглянемо на конкретному прикладі використання запропонованого алгоритму побудови розкладу. Нехай структуру паралельного алгоритму, який буде реалізовано багатопроцесорною системою P, задає граф $G=(W, <)$, зображений на рис. 12.1. Тут у вершинах графа стоять числа, які характеризують час обробки відповідного підблока алгоритму, а під вершиною – номер відповідного підблока (роботи).

Табл. 12.3 ілюструє побудовану базу евристики. Упорядковану у лексикографічному порядку послідовність $B[n]$ представлено в табл. 12.4. Початковий розклад описує табл. 12.5. Остаточний розклад (після застосування процедури оптимізації) наведено в табл. 12.6.

Експериментальне дослідження алгоритму, реалізованого в Паскалі, підтвердило добре практичне застосування алгоритму.

Таблиця 12.3. База евристики

N	1	2	3	4	5	6	7	8	9	10	11
φ	0	6	6	6	8	11	9	11	15	10	18
ψ	6	8	8	9	11	15	10	12	18	14	23

Таблиця 12.4. Упорядкована послідовність $B[n]$

1	2	3	4	5	6	7	8	9	10	11
0, 6	6, 8	6, 8	6, 9	8, 11	9, 1	10, 14	11, 12	11, 15	15, 18	18, 23

Таблиця 12.5. Початковий розклад

$S(1)$	$S(2)$	$S(3)$	$S(4)$	$S(5)$	$S(6)$	$S(7)$	$S(8)$
1 (0, 6)	2: (6, 8) 3: (6, 8) 4: (6, 9)	5: (8, 11)	7: (9, 10)	10: (10, 14)	8: (11, 12) 6: (11, 15)	9: (15, 18)	11: (18, 23)

Таблиця 12.6. Остаточний розклад

$S(1)$	$S(2)$	$S(3)$	$S(4)$	$S(5)$	$S(6)$	$S(7)$	$S(8)$
1: (0, 6)	2: (6, 8)	5: (8, 11) 4: (6, 9)	7: (9, 10)	10: (10, 14) 3: (6, 8)	6: (11, 15)	9: (15, 18) 8: (11, 12)	11: (18, 23)

12.5. Задача ізоморфізму графів

Розглянемо евристичний поліноміальний алгоритм [32] визначення ізоморфізму графів, який можна віднести до групи методів, що використовують локальні характеристичні інваріанти графа.

Задача розпізнання ізоморфізму графів (РІГ) є однією з головних комбінаторних задач теорії графів. Вона займає особливе місце в теорії алгоритмів – невідомо, чи вона належить класу P, чи є NP-повною задачею.

Два графи називають ізоморфними, якщо можна відобразити один з них в інший з точністю до перенумерації вузлів, тобто вони відрізняються лише ідентифікацією своїх вершин. Ізоморфне відображення φ графа G_1 на граф G_2 задають підстановкою

$$\varphi=\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & \ldots & x_k & \ldots & x_n \\ \varphi(x_1) & \varphi(x_2) & \varphi(x_3) & \varphi(x_4) & \ldots & \varphi(x_k) & \ldots & \varphi(x_n) \end{pmatrix}.$$

Отже, у разі розпізнання ізоморфізму графів $G_1=(X_1, R_1)$ і $G_2=(X_2, R_2)$, потрібно відповісти на питання, ізоморфні вони чи ні, і, у разі ізоморфізму, вказати ізоморфну підстановку [81].

Зрозуміло, що існує багато перебірних алгоритмів перевірки ізоморфності графів. Наприклад, у разі задання графів матрицею суміжності можна переставляти рядки і відповідні стовпці однієї з матриць суміжності до того часу, поки вона не перетвориться в іншу; або зупинимося після $n!$ перестановок, якщо графи не ізоморфні. Цей підхід має істотну ваду – він практично незастосовний.

Алгоритми РІГ широко застосовують як під час розв'язання багатьох прикладних задач (автоматизація контролю в САПР БІС, синтаксичне розпізнання образів, автоматизація проектування дискретних пристроїв ЕОМ на основі уніфікованого набору блоків, аналіз і синтез структур обчислювальних систем, дослідження молекулярних структур хімічних сполук), так і низки теоретичних задач (розпізнання ізоморфізму булевих функцій, аналіз і конструктивне перелічення графів із заданими обмеженнями на їх групи автоморфізмів, дослідження властивостей симетрії графів, характеризація алгебричних структур – груп, підгруп, кілець тощо та їх розпізнання, дослідження чутливості – (повноти) інваріантів графів, розробка методу скорочення перебору на основі врахування симетрії графа у розв'язанні NP-повних задач на графах і

мережах). Ці приклади не вичерпують усіх застосувань алгоритмів РІГ, однак вони показують різноманітність сфер їх застосування [2, 3, 50, 58, 94].

12.5.1. *Класифікація алгоритмів розпізнання ізоморфізму*

Можна виділити два основні підходи до побудови алгоритмів РІГ.

Перший підхід пов'язаний з реалізацією принципу ієрархічної побудови перебірних алгоритмів, що рекурсивно поліпшують свою ефективність щодо повноти (чутливості) використаних характеристик вершин (ребер), що є інваріантними відносно ізоморфізму графів і називаються інваріантами. Основне положення, на якому ґрунтується вибір даного принципу, полягає в тому, що для переважної кількості класів графів прості й ефективні процедури дають змогу розрізняти графи на основі інваріантів або виділяти ізоморфну підстановку.

Найбільш вживані типи інваріантів графа такі:

а) *повний інваріант* – характеристика, що задає граф з точністю до ізоморфізму;

б) *характеристичний інваріант* – специфікація множини однотипних фрагментів графа за характеристикою, яка інваріантна щодо ізоморфізму графів;

в) *локальний характеристичний інваріант* – специфікація множини вершин графа за характеристикою, яка інваріантна щодо ізоморфізму графів;

г) *квазіглобальний характеристичний інваріант* – специфікація множини усіх n вершин графа за характеристикою, яка інваріантна щодо ізоморфізму графів, де $2 \le n < p = |X|$;

д) *глобальний характеристичний інваріант* – специфікація множини усіх n вершин графа, де $n = p = |X|$ за характеристикою, яка інваріантна щодо ізоморфізму графів [3].

Уточнимо поняття «інваріант графа» і метод побудови різних класів інваріантів. Для простоти викладу, не втрачаючи повноти, всі огляди проводитимемо у класі звичайних графів, множину яких позначимо як M. Нехай Q_τ – непуста множина елементів $q \in Q$ з визначеною на ній еквівалентністю τ, а R – відношення «бути ізоморфними графами». Тоді функцію Inv, що задана на M_R і набуває значення в Q_τ, називають *інваріантом* графа, якщо з умови $G_i(R)G_j$ випливає, що $\forall\ G_i, G_j \in M[Inv(G_i)\ (\tau)\ Inv(G_j)]$.

Графи G_i, G_j називають Inv-еквівалентними, якщо справедливо $Inv(G_i)\ (\tau)\ Inv(G_j)$. Очевидно, що ізоморфні графи складають підклас будь-якого класу Inv-еквівалентності. Позначимо через $|T(Inv)|$ потужність множини класів Inv-еквівалентності. Тоді чутливістю інваріанта Inv, заданого на M_R, називаються відношення типу $\alpha(Inv, R) = |T(Inv)| / |T(R)|$, де $|T(R)|$ означає потужність множини класів R-еквівалентності. Очевидно,

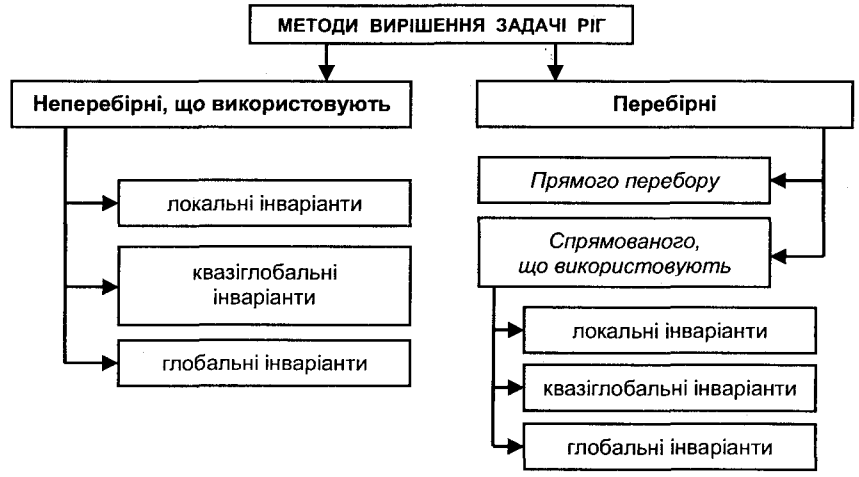


Рис. 12.2. Схема класифікації методів задачі РІГ

що $0 < \alpha(Inv, R) \le 1$. Величина $\alpha(Inv, R)$ характеризує здатність інваріанта Inv розв'язувати задачу P_i у множині M_R. Якщо $\alpha(Inv, R) = 1$, то інваріант Inv називають точним і повним. Тоді з умови $Inv(G_i)\ (\tau)\ Inv(G_j)$ випливає $G_i\ (R)\ G_j$ і точний інваріант повністю розв'язує задачу розпізнання ізоморфізму графів у M_R, якщо ця задача розв'язана в Q_τ [81].

Класифікація методів (рис. 12.2) обґрунтована різноманітністю класів інваріантів, які використовують в алгоритмах. Найбільшої глибини розгляду досягли методи, засновані на застосуванні локальних інваріантів. Методи, засновані на використанні квазіглобальних і глобальних інваріантів, мають теоретичну спрямованість і орієнтовані на дослідження сильно регулярних, дистанційно регулярних і дистанційно транзитивних графів. Всі методи канонізації графів, що засновані на різному заданні матриць графа, відносять до методів, які використовують глобальні інваріанти.

12.5.2. *Евристичний алгоритм розпізнання ізоморфізму графів*

Повернемось до постановки задачі. Нехай є два графи $G_1(U_1, V_1)$ та $G_2(U_2, V_2)$. Потрібно визначити, чи ізоморфні ці графи, тобто чи існує така комбінаторна функція перестановки вершин $f(V_2)$ для другого графа, застосувавши яку, ми отримаємо рівні між собою графи $G_1(U_1, V_1)$ та $G'_2(U_2, f(V_2))$.

Для будь-якої комбінаторної функції перестановки $f(X)$ скінченної послідовності X існує обернена функція $f^{-1}(X)$, що повертає початковий порядок елементів послідовності, який був до дії $f(X)$. Зрозуміло, що якщо взяти будь-який граф T і здійснити всі можливі комбінаторні перестановки його вершин, то отримаємо певну множину графів T^*, що будуть

ізоморфні графу T. Очевидно, що всі графи множини T^* ізоморфні між собою.

Отже, задачу ізоморфізму двох графів G_1 і G_2 можна звести до задачі перевірки належності: $G_1 \in G^* \;\&\; G_2 \in G^*$. Тому одним із можливих шляхів розв'язання нашої задачі є знаходження такого способу обробки графів, який для $\forall G_i \in G^*$ давав би таку комбінаторну перестановку вершин, в результаті дії якої ми отримували б щоразу один-єдиний граф $G_0(U_0, V_0)$. Назвемо такий граф *класичним*. Тоді, застосувавши цей спосіб обробки для будь-яких двох графів G_1 і G_2 по черзі, ми можемо порівняти отримані графи, і якщо ці останні рівні – визначити наявність ізоморфізму між G_1 і G_2. Це є необхідна і достатня умова, тому що: $\forall T \notin G^* \Rightarrow T\urcorner(\text{izom}(\forall G \in G^*))$.

Два графи вважають *рівними*, якщо рівні їх матриці суміжностей.

Розглянемо ще один важливий клас графів – *еквівалентні графи*.

Функцію перестановки елементів послідовності X називатимемо *тривіальною*, якщо в результаті її дії всі елементи послідовності X залишаються на своїх місцях.

Всі графи з множини G^*, які дорівнюють деякому графу G_i, утворюють підмножину *еквівалентних* графів G_{ie}. Тоді очевидно, що два графи будуть еквівалентними, якщо $G_1(U_1, V_1) = G_2(U_2, V_2)$ і $\exists f(V)$ – така нетривіальна функція перестановки вершин, що $G_1(U_1, V_1) = G_2(U_2, f(V_2))$. Зауважимо, що при такому визначенні поняття еквівалентності не всі графи еквівалентні самі собі.

Для подальшої роботи перевизначимо поняття інваріанта. *Інваріант* – це вершина (з певним номером) класичного графа; *n-інваріант* – це така вершина v_{ik} з підмножини вершин V_i графа G, для якої існує підмножина вершин V_j, така що коли зробити перестановку V_i та V_j (утворити новий граф G') , то G і G' – еквівалентні графи.

Підмножини вершин V_i, V_j назвемо *симетричними*, а графи, що мають n-інваріанти, називатимемо *автоеквівалентними*.

Якщо f – така комбінаторна функція перестановки, що $f(V_i) = V_j$, тоді вершини V_{i_k} з V_i та вершини V_{j_k} з V_j, які матимуть однаковий порядковий номер після дії f, назвемо *симетричними вершинами*.

Теорема 12.7. Якщо граф не має n-інваріантів (не є автоеквівалентним), то існує поліноміальний спосіб Q перетворення його в класичний граф.

Доведення. Для доведення існування Q використаємо конструктивний підхід.

Спосіб Q:

Крок 1. Спочатку вершини нумерують за зростанням у лексикографічному порядку кортежів ⟨*«наявність петлі»*, *«напівстепінь вхідних ребер»*, *«напівстепінь вихідних ребер»*⟩, а нумерацію вершин здійснюють за наступним принципом.

Принцип А: вершини з рівними кортежами отримують однакові номери, а наступний номер визначають як суму значення попереднього номера і кількості вершин, яким цей номер був призначений.

У такий спосіб утворяться підмножини вершин з однаковими номерами $V(i)$. Якщо $V(i)$ належить тільки одна вершина, то вона є інваріантом, оскільки спосіб її визначення не залежить від заданої нумерації вершин графа.

Вершини, які належать до підмножини $V(i)$ з кількістю елементів, більшою ніж 1, назвемо *багатозначними*, а самі такі підмножини – *переповненими*.

Крок 2. Якщо в графі ще існують переповнені підмножини, розділити їх згідно з лексикографічним порядком кортежів ⟨*«наявність петлі»*, *«впорядкована за зростанням послідовність номерів вершин, суміжних із вхідними ребрами»*, *«впорядкована за зростанням послідовність номерів вершин, суміжних із вихідними ребрами»*⟩ вершин кожної підмножини, надаючи їм нові номери за принципом *А*; інакше перейти на *крок 4*.

Крок 3. Перейти на *крок 2*.

Крок 4. Кінець.

Оскільки перенумерація вершин на *кроці 2* не залежить від заданої нумерації вершин графа і не порушує порядок нумерації, встановлений на попередніх ітераціях, отримаємо граф з інваріантами (тобто з вершинами, що мають унікальний номер кожна, і порядок нумерації для будь-якого графа з класу G^* буде єдиним), отже отримаємо класичний граф.

Залишається довести, що спосіб Q поліноміальний. Очевидно, коли спосіб Q не зациклиться на *кроці 2*, він повторить його не більше $|V|$ раз.

Припустимо, що на певній ітерації *кроку 2* у графі виявилася принаймні одна переповнена підмножина V_{i_k}, яка складається хоча б з двох вершин v_i і v_j, і її розділення більше неможливе. Сусідні цим двом вершинам мають або однакові номери, або є спільними з ними, інакше підмножина цих вершин була б розділена на другому кроці Q. Аналогічне можна сказати і про сусідні вершини. Іншими словами, яку б послідовність номерів суміжних вершин $(a_1, a_2, a_3, ..., a_n)$, починаючи від першої вершини, ми не розглядали, таку ж послідовність можна знайти, починаючи від другої вершини. Якщо граф не має циклів, то це свідчить, що ці дві вершини належать до двох симетричних підмножин вершин, тобто є n-інваріантами, що суперечить умові теореми. Якщо один зі шляхів від однієї з вибраних вершин має цикл, то обов'язково існує шлях від іншої вершини, який має цикл, бо лише в цьому разі цей шлях може бути нескінченним. Але дві початкові вершини не є n-інваріантами за умовою теореми, тобто послідовності номерів вершин на нескінченних шляхах по циклах мусять збігатися, але самі цикли повинні чимось відрізнятися, щоб не бути симетричними підмножинам вершин.

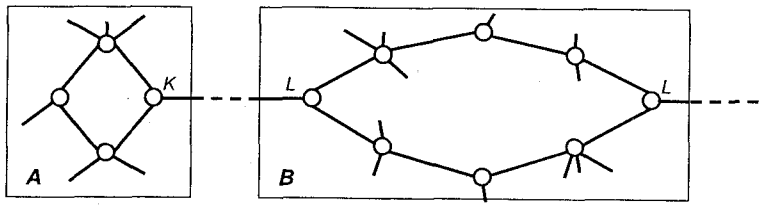


Рис. 12.3. Граф залежності робіт

Це означає, що один з циклів повторює повністю інший цикл кілька разів. Прикладом є граф, зображений на рис. 12.3. Але в цьому випадку вершини, подібні L, повинні бути ідентичними вершинам, подібним K, і чим більше таких повторень циклу A в циклі B, тим більше таких «вільних» вершин залишається.

Умова ідентичності послідовностей номерів вершин на шляхах від вершин з однаковими номерами є доказом того, що з вершинами, подібними до K, будуть врешті-решт з'єднані такі ж структури вершин, як і з вершинами, подібними до L. Серед циклів, які складаються з кількох ділянок, що мають однакові послідовності номерів вершин (вершини, що розміщені за визначеним порядком на одній такій ділянці і за таким же порядком на іншій, ми називатимемо *псевдосиметричними*), є хоча б один цикл, у якого всі вершини на цих ділянках складають симетричні підмножини, бо в протилежному разі описані вище «вільні» вершини мали б з'являтися в сукупностях вершин, що зв'язані з псевдосиметричними вершинами нескінченну кількість разів. А якщо граф має симетричні підмножини вершин, то він має і n-інваріанти, що суперечить умові теореми.

Теорему доведено.

Симетричні підмножини вершин графа G можуть мати свої внутрішні симетричні підмножини вершин (назвемо їх *крильцями*). Такі n-інваріанти, що не належать до жодного з крилець і є симетричними, назвемо *визначальними n-інваріантами* для даних симетричних підмножин вершин.

Легко довести наступну теорему.

Теорема 12.8. Симетричні підмножини вершин (СПВ), що мають вершини, які не належать крильцям, мають хоча б один визначальний n-інваріант.

Зациклювання способу Q свідчить про наявність n-інваріантів, проте не всі багатозначні вершини з мінімальної за розміром переповненої підмножини вершин є визначальними n-інваріантами. Тоді n-інваріанти, що належать до різного типу циклів (де одні цикли складаються з певної кількості будь-яких інших циклів), мають однакові номери після зациклення, але вони не є визначальними n-інваріантами СПВ. Щоб визначити, які з цих вершин є визначальними n-інваріантами СПВ, скористаємося способом Q_1.

Спосіб Q_1. Відокремимо підграф g, який складається з багатозначних вершин та їх сусідів. Цей підграф потребує змін для повного перетворення G в класичний граф. З мінімальної переповненої підмножини вершин (МППВ) вибираємо по черзі вершини і видаляємо їх з g, утворюючи, у такий спосіб, певну кількість різних нових графів. Застосовуємо до кожного з цих графів спосіб Q до зациклення і отримуємо певну нумерацію їх вершин. Впорядкувавши за зростанням утворені послідовності номерів вершин співвідносимо їх з вершинами, що були видалені. Змінюємо номери відповідних вершин графа G згідно з лексикографічним порядком послідовностей, зв'язаних з ними, за принципом, викладеним у кроці 1. Якщо у графі виявилися кілька рівних за розміром МППВ, то повторюємо Q_1 для кожної з них.

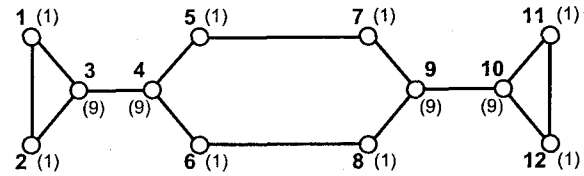


Рис. 12.4.

Теорема 12.9. Спосіб Q_1 залишає без змін нумерацію багатозначних вершин, що є визначальними n-інваріантами відповідних СПВ.

Доведення. Визначальні n-інваріанти є симетричними вершинами, тому видалення одного з них з підграфа g дасть такий же граф, який можна отримати після видалення будь-якого іншого визначального n-інваріанта, а отже, після дії Q_1 цим вершинам відповідатимуть однакові послідовності чисел, тому їх нумерація залишиться без змін.

Теорему доведено.

Спосіб Q_1 може змінити нумерацію багатозначних вершин, якщо вони не є визначальними n-інваріантами певних СПВ. Наприклад, граф, поданий на рис. 12.4, після застосування Q матиме позначену в дужках нумерацію вершин, а у разі видалення почергово вершин з номерами «9» зліва направо (3, 4, 9, 10) дасть такі відповідні їм кортежі впорядкованих номерів: {1, 1, 3, 3, 3, 3, 3, 3, 9, 9, 9}, {1, 1, 3, 3, 5, 5, 5, 5, 9, 9, 9}, {1, 1, 3, 3, 5, 5, 5, 5, 9, 9, 9}, {1, 1, 3, 3, 3, 3, 3, 3, 9, 9, 9}.

Наступна евристика ґрунтується на припущенні, що після виконання ітерацій $\{Q_1, Q$ (*Крок 2*)$\}$ доти, доки в нумерації вершин відбуваються зміни, у МППВ отримаємо визначальні n-інваріанти.

Спосіб Q_2. Будь-якій вершині з МППВ призначають новий номер згідно кроку 1. У такий спосіб, змінивши номер однієї з вершин СПВ і продовживши обробку графа за другим кроком Q, ми матимемо змінені номери вершин в усій СПВ. Такий крок є коректним, бо всі СПВ з однаковими номерами їх визначальних n-інваріантів рівні між собою.

У результаті отримаємо класичний граф для G.

Оскільки час, який займає кожен спосіб окремо, є поліноміальним, то якщо цикл $Q' = (Q($*Крок 1*$), \{\{Q($*Крок 2*$), Q_1\}, \forall V_i \in V_{(k)}(Q_2)\})$ теж здійснить

поліноміальну кількість ітерацій, тоді і весь алгоритм Q' матиме поліноміальну часову складність.

Теорема 12.10. Алгоритм Q' коректний і має поліноміальну часову складність.

Доведення. Коректність алгоритму випливає з наведеного вище аналізу. В реалізації алгоритму можна ввести додаткову процедуру Q_{add}, що перевіряє наприкінці роботи Q' рівність між собою СПВ, які будуть виділені після виконання кожної ітерації $\{(Q_2), \{Q(Крок\ 2), Q_1\}\}$. Тоді неточні виведення евристики можна буде якимсь чином відслідковувати (якщо такі існують).

Оскільки мета кожного циклу алгоритму Q' – зробити додаткове розбиття переповнених множин, то кількість ітерацій не може бути більшою, ніж $O(n)$, отже загалом Q' має поліноміальну часову складність.

Задачі до розділу 12

1. Напишіть програму, яка розв'язує задачу про мавпу і банани на основі базової ідеї «Загального вирішувача задач».
2. Проілюструйте на довільному прикладі зведення проблеми прийняття рішень до оптимізаційної задачі. Перелічіть відомі вам класи оптимізаційних задач.
3. Опишіть відому вам універсальну процедуру розв'язання задачі прийняття рішень. У чому полягають її позитивні та негативні риси?
4. Опишіть процедуру евристичного пошуку. Проілюструйте застосування цієї процедури до розв'язання задачі розфарбування графа.
5. Напишіть програму реалізації процедури евристичного пошуку, яка реалізовує пошук потрібного ходу для гри в шашки.
6. Охарактеризуйте поняття обмежувальних правил та евристик. В чому полягає їх значення при плануванні цілеспрямованих дій? Поясніть, чому жадібні алгоритми можна розглядати як застосування обмежувальних правил (розпишіть відповідні обмежувальні правила в явному вигляді).
7. Нагадаємо, що називають *реберним розфарбуванням* графа. У цьому разі мають справу з відображенням множини ребер на множину фарб, а вірним розфарбуванням називають розфарбування, за якого будь-які два ребра, що інцидентні одній вершині, пофарбовані різними фарбами. Розробіть евристичний алгоритм реберного розфарбування графа та побудуйте його часову оцінку.
8. Доведіть теорему Кенінга (теорема 12.1).
9. Доведіть теорему Секереша–Вільфа (теорема 12.2).
10. Напишіть програму та побудуйте часову оцінку точного алгоритму розфарбування з підрозділу 12.3.3.
11. Напишіть програму реалізації алгоритму розфарбування Єршова–Кожухіна та побудуйте його часову оцінку.
12. Розробіть евристичний алгоритм розфарбування графа другого типу на основі евристик вибору послідовності вершин розфарбування та оцініть час його виконання:
    а) вершини впорядковують за спаданням значень їх степенів (ПН-впорядкування);
    б) вершини впорядковують за зростанням значень їх степенів (НО-впорядкування);
13. Розробіть евристичний алгоритм розфарбування графа окільного типу та оцініть час його виконання.
14. Розробіть алгоритм складання розкладу виконання робіт на основі динамічного програмування та побудуйте його часову оцінку.
15. Розробіть алгоритм складання розкладу виконання робіт на основі жадібного алгоритму та побудуйте його часову оцінку.
16. Розробіть евристичний списковий алгоритм складання розкладу виконання робіт та оцініть час його виконання, що ґрунтується на наступних евристиках:
    а) найбільша зв'язність;
    б) найбільший або найменший час виконання;
    в) найменший критичний за кількістю операторів або за часом шлях від початкового оператора;
    г) найбільший критичний за кількістю операторів або за часом шлях до останнього оператора.

Література

1. *Адельсон-Вельский Г. М., Арлазаров В. Л., Донской М. В.* Программирование игр.– М.: Наука, 1978.
2. Алгоритмические исследования в комбинаторике / Под ред. И. А. Фараджева.– М.: Наука, 1978.– С. 172–183.
3. Алгоритмы и программы решения на графах и сетях / Под ред. А. Нечепуренко.– М., 1990.
4. *Анисимов А. В.* Рекурсивные преобразователи информации.– К.: Вища шк., 1987.– 230 с.
5. *Анисимов А. В.* Технология рекурсивно-параллельного программирования // Программирование.– 1991.– № 6.– С. 91–102.
6. *Анисимов А. В., Глибовец Н. Н., Сафронюк С. В.* Об одном способе раскраски графа. Модели и системы обработки информации.– К.: Вища шк., 1985.– С. 9–16.
7. *Анісімов А. В.* Алгоритмічна теорія великих чисел. Модулярна арифметика великих чисел.– К.: ВД «Академперіодика», 2001.– 153 с.
8. *Аоки М.* Введение в методы оптимизации.– М.: Наука, 1978.
9. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов.– М.: Мир, 1979.
10. *Басакер Р., Саати Г.* Конечные графы и сети.– М., 1974.
11. *Беллман Р.* Динамическое программирование.– М.: Изд-во иностр. л-ры, 1960.
12. *Ботвинник М. М.* О решении неточных переборных задач.– М.: Сов. радио, 1979.– 152 с.
13. *Вагин В. Н.* Дедукция и обобщение в системах принятия решений.– М.: Наука, 1988.
14. *Вагнер Г.* Основы исследования операций. В 3-х томах.– М.: Мир, 1973.
15. *Васильев В. И.* Распознающие системы.– К.: Наук. думка, 1983.
16. *Васильев Ф. П.* Лекции по методам решения экстремальных задач.– М.: Изд-во МГУ, 1974.
17. *Вентцель Е. С.* Исследование операций.– М.: Сов. радио, 1972.
18. *Верхаген К., Дейн Р., Грун Ф.* и др. Распознавание образов: состояние и перспективы.– М.: Радио и связь, 1985.– 104 с.
19. *Винер Н.* Кибернетика.– М.: Сов. радио, 1968.
20. *Вирт Н.* Алгоритмы + структуры данных = программы.– М.: Мир, 1985.
21. *Вирт Н.* Систематическое программирование. Введение.– М.: Мир, 1977.– 183 с.
22. Вычислительные машины и мышление / Под ред. Э. Фейгенбаума и Б. Фельдмана.– М.: Мир, 1967.
23. *Гаазе-Рапопорт М. Г., Поспелов Д. А.* От амебы до робота: модели поведения.– М.: Наука, 1987.– 288 с.
24. *Гасс С.* Линейное программирование (методы и приложения).– М.: Физматгиз, 1961.
25. *Гирсанов И. В.* Лекции по математической теории экстремальных задач.– М.: Изд-во МГУ, 1970.
26. *Глибовец Н. Н., Иващенко С. А.* Эвристический алгоритм определения изоморфизма графов // Кибернетика и системный анализ.– 2001.– № 170–176.
27. *Глибовец Н. Н., Глибовец А. Н.* Эвристический алгоритм составления расписания работы многопроцессорного комплекса // Проблемы программирования.– 1999.– № 2.– С. 34–40.
28. *Глибовец Н. Н., Гулаева Н. М.* Локальный алгоритм решения задачи поиска пути с максимальной пропускной способностью и его параллельная реализация // Управляющие машины и системы.– 2000.– № 1.– С. 27–32.
29. *Глибовец Н. Н., Заднепровский К. Г.* Об одном подходе к решению задачи нахождения всех контуров в ориентированных графах. Модели и системы обработки информации.– К.: Вища шк., 1989.
30. *Глибовець М. М., Іващенко С. А.* Про один підхід до розв'язку задачі мінімізації булевих функцій // Наук. записки НаУКМА. Комп'ютерні науки.– 2000.– Т. 18.– С. 29–33.
31. *Глибовець М. М., Олецький О. В.* Штучний інтелект.– К.: ВД «КМ Академія», 2002.– С. 365.
32. *Глибовец Н. Н.* Об одном методе описания управляющего пространства в асинхронных параллельных вычислениях. Модели и системы обработки информации.– К.: Вища шк., 1984.
33. *Глибовець М. М.* Основи комп'ютерних алгоритмів. Ч. 1. Методичний посібник для студентів НаУКМА.– К.: ВД «КМ Academia», 1995.– 41 с.
34. *Глушков В. М., Капитонова Ю. В., Летичевский А. А., Горлач С. П.* Макроконвейерные вычисления функций над структурами данных // Кибернетика.– 1981.– № 4.– С. 13–21.
35. *Глушков В. М., Капитонова Ю. В., Летичевский А. А.* Эффективность параллельных вычислений при ограниченных ресурсах // Докл. АН СССР.– 1980.– 254, № 3.– С. 527–530.
36. *Головкин Б. А.* Вычислительные системы с большим числом процессоров.– М.: Радио и связь, 1995.
37. *Головкин Б. А.* Параллельные вычислительные системы.– М.: Наука, 1980.
38. *Головкин Б. А.* Расчет характеристик и планирование параллельных вычислительных процессов.– М.: Радио и связь, 1983.
39. *Гэри М., Джонсон Д.* Вычислительные машины и труднорешаемые задачи.– М.: Радио и связь, 1983.– 277 с.
40. *Даффин Р., Питерсон Э., Зенер К.* Геометрическое программирование.– М., 1972.– 312 с.
41. *Джордж Ф.* Основы кибернетики.– М.: Радио и связь, 1984.– 272 с.
42. *Диниц Е. А.* Алгоритм решения задачи о максимальном потоке в сети со степенной оценкой // Докл. АН СССР. Сер. Математика, физика.– 1970.– № 4.– С. 754–757.
43. *Донец Г. А., Шор Н. З.* Алгебраический подход к раскраске плоских графов.– М., 1982.
44. *Евстигнеев В. А.* Применение теории графов в программировании.– М.: Наука, 1985.
45. *Ершов А. П.* Введение в теоретическое программирование.– М.: Наука, 1977.
46. *Загоруйко Н. Г., Скоробогатов В. А., Хворостов П. В.* Вопросы анализа и распознавания молекулярного строения на основе общих фрагментов. // Алгоритм анализа структурной информации: Вычисл. системы.– Новосибирск: ИМ СО АН СССР, 1984.– Вып. 103.– С. 26–50.
47. *Зайченко Ю. П.* Исследование операций.– К.: Вища шк., 1975.
48. *Зайченко Ю. П., Шумилова С. А.* Исследование операций.– К., 1990.– 239 с.
49. *Зельковиц М., Шоу А., Гешон Дж.* Принципы разработки программного обеспечения.– М.: Мир, 1982.– 183 с.
50. *Земляченко В. Н., Корнеенко Н. М., Тышкевич Р. И.* Проблема изоморфизма графов // Зап. науч. семинаров ЛОМИ. – Т. 118.– Л.: Наука, 1972.
51. *Зуев Е. А.* Программирование на языке TURBO PASCAL 6.0, 7.0.– М.: Радио и связь, 1993.– 384 с.
52. Искусственный интеллект: Справочник в 3-х т.– М.: Радио и связь, 1990.
53. *Калихман И. Л.* Сборник задач по математическому программированию.– М., 1975.– 270 с.
54. *Капитонова Ю. В.* Фундаментальные идеи и эволюция вычислительных систем // Кибернетика и систем. анализ.– 1995.– № 2.– С. 75–83.
55. *Карп Р. М.* Сводимость комбинаторных проблем // Кибернетический сборник.– Вып. 12.– 1975.
56. *Карпов А. Е., Гик Е. Я.* Неисчерпаемые шахматы.– М.: Изд-во Моск. ун-та, 1991.– 399 с.
57. *Кермек Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ.– М.: МЦНМО, 2001.– С. 955.
58. *Клин М. Х.* Об аксиоматике клеточных колец // Исследования по алгебраической теории комбинаторных объектов.– М.: ВНИИСИ, 1985.– С. 6–32.
59. *Кнут Д.* Искусство программирования для ЭВМ. Том 1: Основные алгоритмы.– М.: Мир, 1976.
60. *Кнут Д.* Искусство программирования для ЭВМ. Том 2: Получисленные алгоритмы.– М.: Мир, 1977.
61. *Кнут Д.* Искусство программирования для ЭВМ. Том 3: Поиск и сортировка.– М.: Мир, 1978.

62. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ.– М.: МЦНМО, 2001.– 960 с.
63. *Корн Г., Корн Т.* Справочник по математике.– М.: Наука, 1973.– 832 с.
64. *Кристофидес Н.* Теория графов. Алгоритмический подход.– М., 1978.
65. *Лингер Р., Миллс Х., Уитт Б.* Теория и практика структурного программирования.– М.: Мир, 1982.– 404 с.
66. *Липский В.* Комбинаторика для программистов.– М.: Мир, 1988.
67. *Лорьер Ж.-Л.* Системы искусственного интеллекта.– М.: Мир, 1991.– 568 с.
68. *Ляшко И. И., Макаров В. Л., Скоробогатько А. А.* Методы вычислений.– К.: Вища шк., 1977.– 408 с.
69. *Майер Б., Бодуэн К.* Методы программирования: В 2-х т.– М.: Мир, 1982.– Т. 1.– 356 с.; Т. 2.– 368 с.
70. *Максименко Н. В.* Анализ алгоритмов диспетчеризации задач мультипроцессорной ЭВМ // Управляющие системы и машины.– 1978.– № 3.
71. *Мальцев А. И.* Алгоритмы и рекурсивные функции.– М.: Наука, 1965.– 384 с.
72. *Минский М., Пейперт С.* Персептроны.– М.: Мир, 1971.
73. *Михалевич В. С., Шор Н. З.* Метод последовательного анализа вариантов при решении вариационных задач управлення, планирования и проектирования // Докл. на IV Всесоюз. мат. съезде.– Л., 1961.– 91 с.
74. *Михалевич В. С., Шкурба В. В.* Последовательные схемы оптимизации в задачах упорядочения выполнения работ // Кибернетика.– 1966.– № 2.– С. 34–40.
75. *Моисеев Н. Н., Иванилов Ю. П., Столярова Е. М.* Методы оптимизации.– М.: Наука, 1978.– 352 с.
76. *Нейман Дж., Моргенштерн О.* Теория игр и экономическое поведение.– М.: Наука, 1970.
77. *Нечипуренко М. И., Попков В. К., Найнагашев С. М.* и др. Алгоритмы и программы решения задач на графах и сетях.– Новосибирск: Наука, 1990.
78. *Нильсон Н.* Искусственный интеллект. Методы поиска решений.– М.: Мир, 1973.
79. *Нильсон Н.* Принципы искусственного интеллекта.– М.: Радио и связь, 1985.– 373 с.
80. *Ньюэлл А., Саймон Г.* GPS-программа, моделирующая процесс человеческого мышления.– М.: Мир, 1978.– С. 283–301.
81. *Петренюк А. Я.* Применение инвариантов в комбинаторных исследованиях // Вопросы кибернетики.– М.: Изд-во АН СССР, 1973.– С. 129–135.
82. *Понтрягин Л. С., Болтянский В. Г., Гамрелидзе Р. В., Мищенко Е. Ф.* Математическая теория оптимальных процессов.– М.: Наука, 1983.– 392 с.
83. *Попов Ю. Д., Тюптя В. І., Шевченко В. П.* Методи оптимізації.– К.: Абрис, 1999.– 217 с.
84. *Поспелов Д.* Фантазия или наука: на пути к искусственному интеллекту.– М.: Наука, 1982.– 224 с.
85. *Поспелов Д. А.* Ситуационное управление: теория и практика.– М.: Наука, 1986.– 160 с.
86. *Проценко В. С., Чаленко П. И., Сорока Р. А.* Техника программирования.– К.: Наук. думка, 1990.– 183 с.
87. *Пшеничный Б. Н., Данилин Ю. М.* Численные методы в экстремальных задачах.– М.: Наука, 1975.
88. *Пападиметру Х., Стайглиц К.* Комбинаторная оптимизация: алгоритмы и сложность.– М.: Мир, 1985.
89. *Рейнгольд Э., Нивергельт Ю., Део Н.* Комбинаторные алгоритмы. Теория и практика.– М.: Мир, 1980.
90. *Скурыхин В. И., Шифрин В. Б., Дубровский В. В.* Математическое моделирование.– М.: Техника, 1983.– 270 с.
91. *Слейгл Дж.* Искусственный интеллект.– М.: Мир, 1973.– 320 с.
92. *Ставровський А. Б., Коваль Ю. В.* Елементи програмування.– К.: Фонд «Молоді надії України», 1997.– 210 с.
93. Теория расписаний и вычислительные машины / Под ред. О. Г. Коффмана.– М.: Наука, 1984.
94. *Уинстон П.* Искусственный интеллект.– М.: Мир, 1980.
95. *Фараджев И. А.* Конструктивное перечисление комбинаторных объектов // Алгоритмические исследования в комбинаторике.– М.: Наука, 1978.– С. 3–11.
96. *Фу К.* Структурные методы в распознавании образов.– М.: Мир, 1977.

97. *Хант Э.* Искусственный интеллект.– М.: Мир, 1978.
98. *Харари Ф.* Теория графов.– М., 1973.
99. *Цейтлин Г. Е.* Введение в алгоритмику.– К.: Сфера, 1998.– 310 с.
100. *Цикритзис Д., Лоховски Ф.* Модели данных.– М.: Финансы и статистика, 1985.– 344 с.
101. *Чикул В. Н.* Об эвристическом программировании интеллектуальных игр.– Ташкент, 1968.
102. *Шабхазян К. В., Тушкина Т. А., Сохранская В. С.* Статистические испытания различных методов диспетчеризации многопроцессорных систем // Программирование.– 1976.– № 4.– С. 91–100.
103. *Ширяев А. Н.* Вероятность.– М.: Наука, 1980.
104. *Элти Дж., Кумбс М.* Экспертные системы: концепции и примеры.– М.: Финансы и статистика, 1987.– 191 с.
105. *Эндрю А.* Искусственный интеллект.– М.: Мир, 1985.– 264 с.
106. *Юдин Д. Б., Гольштейн Е. Г.* Линейное программирование.– М.: Наука, 1969.
107. *Янов Ю. И.* О логических схемах алгоритмов / Проблемы кибернетики.– Вып. 1.– С. 75–127.
108. *Ястремський О. І.* Моделювання економічного ризику.– К.: Либідь, 1992. – 176 с.
109. *Ястремський О. І., Грищенко О. Г.* Основи мікроекономіки.– К.: Знання, 1998.– 714 с.
110. *Bell A.* Games Playing With Computers.– London: Allen & Unwin, 1972.
111. *Bellman R. E.* On a routing problem // Quart. Appl. Math.– 1958.– V. 16.– P. 87–90.
112. *Bentley J. L., Haken D., Saxe J. B.* A general method for solving divide-and-conquer recurrences // SIGACT News.– 12(3).– 1980.– P. 36–44.
113. *Bitner J. R., Reingold E. M.* Backtrack programming techniques // C. ACM.– V.18.– 1975.– P. 651–656.
114. *Borodin A. V.* Fast modular transforms via division // IEEE 13th Annual Symposium on Switching and Automata Theory.– 1972.– P. 90–96.
115. *Cook S. A.* On the minimum computation time of functions, doctoral Thesis, Harvard University, Cambridge, Mass., 1966.
116. *Dutton R. D., Brigham R. C.* A new graph coloring algorithm // The Comp. Journal.– V. 24.– No. 1.– 1981.– P. 85–86.
117. *Edmonds J.* Matroids and the greedy algorithms // Math. Programming.– 1971.– V. 1.– P. 127–136.
118. *Edmonds J., Karp R. M.* Theoretical improvements in algorithmic efficiency for network flow problems // J. ACM.– 1972.– V. 19.– P. 248–264.
119. *Floyd R. W.* Algorithm 97: Shortest path // Comm. ACM.– 1962.– P. 701.
120. *Ford L. R.* Network flow theory // The Rand Corp.– August, 1956.– P. 923.
121. *Ford L. R., Fulkerson D. R.* Flows in networks.– NJ: Prinston Univ. Press, Princeton, 1962.
122. *Garey M. N., Johnson D. S.* The complexity of near-optimal graph coloring // J. ACM.– 1976.– V. 23.– No. 1.– P. 43–49.
123. *Garfinkel R., Nemhauser G.* Integer Programming.– John Wiley, 1973.
124. *Glibovets N.* The class of Neighboring Heuristic Algorithms for Graph Coloring.– Tampere University of Technology Software Systems Laboratory.– Report 12.– August, 1991.– P. 10.
125. *Glibovets N. N., Kurki-Suonio R., Zadneprovsky K. G.* An algorithm for the Isomophism Problem of Connected Graphs without Loops and Folded Edges.– Tampere University of Technology Software Systems Laboratory.– Report 13.– August, 1991.– P. 11.
126. *Golomb S., Baumen L.* Backtrack programming // J. ACM.– V. 12.– 1965.– P. 516–524.
127. *Hall M., Knuth D. E.* Combinatorial analysis and computers // American Mathematical Monthly.– V. 72.– Part II.– Feb., 1965.– P. 21–28.
128. *Horowitz E., Sahni S.* Fundamentals of Computer Algorithms.– Computer Science Press, 1978. – 625 p.
129. *Huffman D. A.* A metod for the construction of minimum-redundancy codes // Proceedings of the IRE.– V. 40(9).– 1952.– P. 1098–1101.
130. *Johnson D.* Efficient algorithms for shortest paths in sparse networks // J. ACM.– V. 24.– 1977.– P. 1–13.
131. *Johnson S. M.* Generation of permutations by adjacent transpositions // Math. Comp.– V. 17.– 1963.– P. 282–285.
132. *Knuth D. E.* Estimating the efficiency of backtrack programs // Math. Comp.– V. 29.– 1975.– P. 121–136.

133. *Kruskal J. B.* On the shortest spanning subgraph of a graph and the traveling salesman problem // Proc. Amer. Math. Soc.– V. 7.– 1956.– P. 48–49.
134. *Lawler E. L.* Combinatorial Optimization: Networks and Matroids.– NY: Holt, Rinehart and Winston, 1976.
135. *Lipson J.* Chinese remainder and interpolation algorithms // Proc. 2nd Symposium in Symbolic and Algebraic Manipulation.– 1971.– P. 372–391.
136. *Liu C. L.* Introduction to combinatorial mathematics.– McGraw-Hill, 1968.
137. *Malhotra V. M., Kumar P. M., Maheshwari S. N.* An $O(|V|^3)$ algorithm for finding maximum flows in networks.– Information Processing Lett.– V. 7.– 1978.– P. 277–278.
138. *Matula D., Marble G., Isaacson J.* Graph coloring algorithms // Graph Theory and Computing.– NY: Academic Press, 1972.– P. 109.
139. *McCulloch W. S., Pitts W.* A logical calculus of the imminent in nervous activity // Bull. Math. Biophysics.– V. 5.– 1943.– P. 115–137.
140. *Minsky M.* Steps toward artificial intelligence.– P. 406–450.
141. *Minsky M., Selfridge O. G.* Learning in Random nets. In: Information Theory.– London: Butterworth, 1961.– P. 335–347.
142. *Moenck R., Borodin A. V.* Fast modular transforms via division // IEEE 13th Annual Symposium on Switching and Automata Theory.– 1972.– P. 90–96.
143. *Neumann J.* Collected Works.– NY: Macmillan, 1963.
144. *Nijenhuis A., Wilf H. S.* Combinatoral Algorithms.– NY: Academic Press, 1975.
145. *Prim R. C.* Shortest connectionnetworks and some generalizations // Bell System Technical Journal.– V. 36.– 1957.– P. 1389–1401.
146. *Purdom P. W. Jr., Brown C. A.* The analysis of algorithms. Holt, Rinehart, and Winston, 1985.
147. *Rado R.* Note on independence function // Proc. London Math. Soc.– V. 7.– 1957.– P. 337–343.
148. *Tarjan R. E.* Depth first search and linear graph algorithms // SIAM J. Comput.– V. 1.– 1972.– P. 146–160.
149. *Trotter H. F.* Perm (Algorithm 115) // Comp. ACM.– V. 5.– 1962.– P. 434–435.
150. *Walker R. J.* An enumerative technique for a class of combinatorial problems // Proceedings of Symposia in Applied Mathematics.– V. X.– American Mathematical Society, Providence, R. I., 1960.
151. *Warshall S.* A theorem on Boolean matrices // J. ACM.– V. 9.– 1962.– P. 11–12.
152. *Welsh D. J* A Matroid Theory.– London: Academic Press, 1976.
153. *Whitney H.* On the abstract properties of linear independence // Amer. J. Math.– V. 57.– 1935.– P. 509–533.

Предметний покажчик

Г

Д

Е

З

І

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Іменний покажчик

Зміст

Наукове видання

Микола Миколайович
Глибовець

ОСНОВИ КОМП'ЮТЕРНИХ АЛГОРИТМІВ

Редактор *Л. В. Сивай*

Художнє оформлення *Н. В. Михайличенко*

Технічний редактор *Т. М. Новікова*

Комп'ютерна верстка *Г. Г. Пузиренка*

Коректор *І. Г. Ярошенко*

Підписано до друку 29.01.2003. Формат 60×90/16.
Гарнітура Times New Roman. Папір офсетний № 1.
Друк офсетний. Умов. друк. арк. 28,25.
Обл.-вид. арк. 28,0. Наклад 1000 прим.
Зам. 3-01.

Видавничий дім «КМ Академія».
Свідоцтво про реєстрацію № 770 від 15.01.2002 р.

Друкарня НаУКМА.

Адреса видавництва та друкарні:
04070, Київ, вул. Сковороди, 2.
Тел./факс: (044) 416-60-92, 238-28-26.
E-mail: phouse@ukma.kiev.ua

Глибовець М. М.

Г54 Основи комп'ютерних алгоритмів.– К.: Вид. дім «КМ Академія», 2003.– 452 с.: іл.– Бібліогр.: с. 428–432.

ISBN 966-518-193-9

Монографію присвячено розгляду питань конструктивної алгоритміки в інформатиці. У книзі розглядаються основні методи побудови алгоритмів: пошук на графах, «розділяй і пануй», жадібний підхід, динамічне програмування, бектрекінг, гілок і границь, символьні обчислення, використання евристик. Особливістю книги є цілісність викладення матеріалу. Воно починається з теоретичного огляду, опису алгоритму розв'язку задачі, характеристики та аналізу особливостей алгоритмів розв'язку задачі і завершується створенням програм реалізації відповідних алгоритмів на Паскалі.

Книгу адресовано студентам і аспірантам, спеціалістам і науковцям спеціалізацій «прикладна математика» та «комп'ютерні науки».

ББК 32.973.202-018