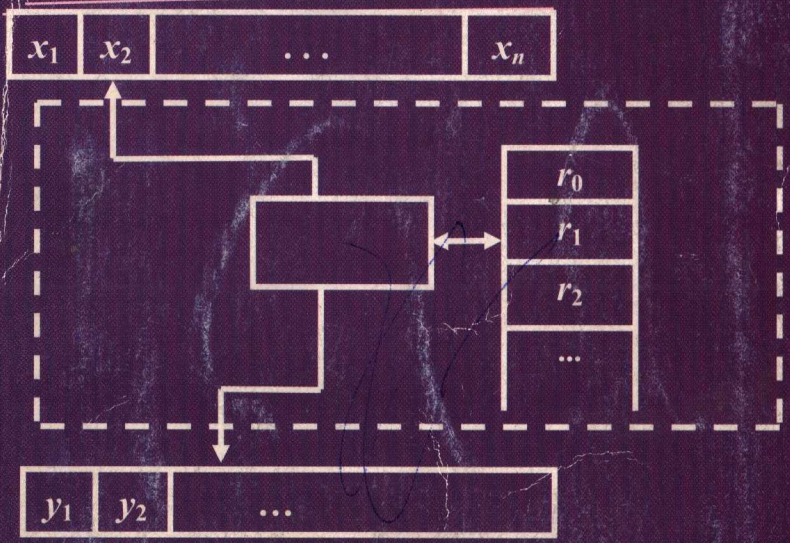
Л. Клакович,
С. Левицька,
О. Костів

ТЕОРІЯ АЛГОРИТМІВ



Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

Л.М. Клакович, С.М. Левицька

О.В. Костів

Теорія алгоритмів

Навчальний посібник

Рекомендовано

Міністерством освіти і науки України

Львів
Видавничий центр
ЛНУ імені Івана Франка
2008

УДК 510.5
ББК 13127
К 47

Рецензенти:

канд. фіз.-мат. наук, доц. *Ю.М. Щербина*
(Львівський національний університет імені Івана Франка)

д-р техн. наук, проф. *В.В. Пасічник*
(Національний університет "Львівська політехніка")

д-р фіз.-мат. наук, проф. *М.О. Недашковський*
(Тернопільський національний економічний університет)

Рекомендовано Міністерством освіти і науки України
як навчальний посібник для студентів вищих навчальних закладів
Лист №1.4/18–Г–849 від 11 квітня 2008 р.

Клакович Л. М. та ін.

К 47 Теорія алгоритмів: Навч. посібник / Л. М. Клакович, С.М. Левицька, О.В. Костів. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2008. – 140 с.

ISBN 978-966-613-637-7

Розглянуто основні поняття та проблеми теорії алгоритмів; описано класичні алгоритмічні системи: нормальні алгоритми Маркова, рекурсивні функції, машини Тьюрінга, Поста, РАМ-машини; досліджено клас важкорозв'язних задач. Наведено деякі методи розробки ефективних алгоритмів. До кожної теми складено низку завдань для самостійної роботи.

Для студентів та аспірантів факультету прикладної математики та інформатики, а також усіх, хто цікавиться розробкою обчислювальних

УДК 510.5
ББК 13127
ISBN 978-966-613-637-7

Зміст

ПЕРЕДМОВА

Матеріал, зібраний у навчальному посібнику, є результатом багаторічної роботи авторів над курсом "Теорія алгоритмів", який викладають у Львівському національному університеті імені Івана Франка студентам спеціальності "інформатика".

Запропонований навчальний матеріал розділено на чотири розділи. У першому розділі розглянуто базові поняття алгоритмів та їхні складності. Другий розділ присвячений класичним алгоритмічним системам. Тут наведено матеріал про нормальні алгоритми Маркова, рекурсивні функції, абстрактні машини Тьюрінга і Поста, операторні алгоритмічні системи. Особливу увагу приділено питанню існування універсального алгоритму та еквівалентності цих алгоритмічних систем.

Третій розділ присвячений важкорозв'язним задачам. Алгоритми, що їх розв'язують, мають експоненціальну складність і часто є неприйнятними у практичному використанні. Для введення поняття недетермінованого обчислення використано недетерміновані машини Тьюрінга. Досліджено важливі класи мов $\mathcal{P}$ та $\mathcal{NP}$, співвідношення між ними та задачі, які до них належать.

У четвертому розділі розглянуто методи розробки ефективних алгоритмів. Кожен запропонований алгоритм, крім теоретичного викладення на загальному рівні, супроводжується прикладом його реалізації для однієї з класичних задач.

Навчальний матеріал містить велику кількість прикладів, ілюстрацій, до кожного параграфа є завдання для самостійної роботи.

ВСТУП
У ТЕОРІЮ АЛГОРИТМІВ

Розвинута в XX ст. теорія алгоритмів стала базою для теорії обчислювальних машин, теорії та практики програмування, математичної логіки, інформатики. Предметом вивчення цієї науки є алгоритми.

Алгоритм – це формально описана обчислювальна процедура, що отримує вхідні дані, які називають також входом алгоритму, або його аргументом, і видає результат обчислень на вихід.

Слово "алгоритм" походить від імені узбецького вченого аль-Хорезмі, який у IX ст. розробив правила виконання чотирьох арифметичних дій над числами в десятковій системі числення. Сукупність цих правил у Європі почали називати "алгоризм". Пізніше цей термін перетворився в "алгоритм" (іноді "алгорифм") і ним називали правила розв'язування різних задач.

Першим алгоритмом, який дійшов до нас, у його інтуїтивному розумінні як скінченної послідовності елементарних дій, які розв'язують поставлену задачу, вважають запропонований Евклідом у III ст. до нашої ери алгоритм знаходження найбільшого спільного дільника двох чисел.

Аж до 30-х років XX ст. поняття алгоритму мало швидше методологічне, ніж математичне значення. Під алгоритмом розуміли скінченну сукупність точно сформульованих правил, які дають змогу розв'язувати задачі певного класу. Таке визначення алгоритму не є строгим, а скоріше інтуїтивним, оскільки спирається на інтуїтивні поняття (правило, клас задач). У 20-х роках XX ст.

задача строгого визначення алгоритму стала однією з центральних математичних проблем.

Початковою точкою відліку сучасної теорії алгоритмів можна вважати теорему про неповноту символічних логік, доведену німецьким математиком К. Геделем 1931 р. У цій роботі з'ясовано, що деякі математичні проблеми не можуть бути розв'язані алгоритмами з певного класу. Загальність результатів К. Геделя пов'язана з питанням про те, чи тотожний використовуваний ним клас алгоритмів з класом усіх алгоритмів в інтуїтивному понятті цього терміна. Зазначена праця дала поштовх до пошуку й аналізу різних формалізацій поняття "алгоритм".

Перші фундаментальні праці з теорії алгоритмів опубліковані в середині 30-х років XX ст. Аланом Тьюрінгом, Алоізом Черчем і Емілем Постом. Запропоновані ними машина Тьюрінга, машина Поста і клас рекурсивних функцій Черча стали першими формальними описами алгоритму, які використовували строго визначені моделі обчислень. Сформульовані гіпотези Поста і Черча–Тьюрінга постулювали еквівалентність запропонованих ними моделей обчислень як формальних систем та інтуїтивного поняття алгоритму.

Важливим розвитком цих праць стало формулювання і доведення існування алгоритмічно нерозв'язних проблем.

У 1950-х роках суттєвий внесок у розвиток теорії алгоритмів зробили праці А. Колмогорова і алгоритмічний формалізм Маркова, що ґрунтується на теорії формальних граматик. Формальні моделі алгоритмів Поста, Тьюрінга і Черча, як і моделі Колмогорова і Маркова, виявились еквівалентними в тому розумінні, що будь-який клас проблем, розв'язний в одній моделі, буде розв'язним і в іншій.

Поява доступних комп'ютерів і суттєве розширення кола задач, які можна розв'язати за їхньою допомогою, привели в 1960–1970-х роках до практично значимих досліджень алгоритмів і обчислювальних задач. На цій підставі тоді оформились такі розділи в теорії алгоритмів:

- класична теорія алгоритмів (формулювання задач у термінах формальних мов, поняття розв'язності задачі, опис класів задач за складністю, формулювання 1965 р. Ж. Едмондсом проблеми $\mathcal{P} = \mathcal{NP}$, відкриття класу $\mathcal{NP}$ -повних задач і його дослідження; введення нових моделей обчилень — алгебричного дерева обчислень (Бен-Ор), машин з довільним доступом до пам'яті, схем алгоритмів Янова, стандартних схем програм Котова та ін. [8]);

- теорія асимптотичного аналізу алгоритмів (поняття складності алгоритму, критерії оцінки алгоритмів, методи отримання асимптотичних оцінок, зокрема для рекурсивних алгоритмів, асимптотичний аналіз трудомісткості або часу виконання, отримання теоретичних нижніх оцінок складності задач), у розвиток якої суттєвий внесок зробили Д. Кнут, А. Ахо, Дж. Хопкрофт, Дж. Ульман, Р. Карп, Л. Кудрявцев та ін. [1, 2, 5, 6];

- теорія практичного аналізу обчислювальних алгоритмів (отримання явних функцій трудомісткості, інтервальний аналіз функцій, практично значимі критерії якості алгоритмів, методики вибору раціональних алгоритмів); основоположною в цьому напрямі вважають фундаментальну працю Д. Кнута "Мистецтво програмування" [5, 6].

З теорією алгоритмів тісно пов'язані дослідження, що стосуються розробки методів створення ефективних алгоритмів (динамічне програмування, метод гілок і меж, метод декомпозиції, "жадібні" алгоритми, спеціальні структури даних тощо) [2, 5, 8, 9, 13].

Отже, на сучасному рівні теорія алгоритмів є дисципліною теоретичної математики, яка надає їй апарат для дослідження розв'язності проблем, і дисципліною прикладної математики, яка безпосередньо вивчає певні явища реального світу.

Поняття алгоритму є концептуальною основою різноманітних процесів опрацювання інформації, бо власне наявність відповідних алгоритмів і забезпечує можливість автоматизації таких процесів. Разом з математичною логікою теорія алгоритмів утворює теоретичний фундамент сучасних обчислювальних наук.

Теорія алгоритмів тісно пов'язана не лише з інформатикою та програмуванням, а й з лінгвістикою, економікою, психологією та іншими науками. У рамках теорії алгоритмів сформувалось багато нових розділів, які мають яскраво виражений прикладний напрям – теорія алгоритмічних мов, складність алгоритмів і обчислень, аналіз і верифікація алгоритмів та програм тощо. Ці розділи, поряд з

іншими теоретичними напрямами інформатики, утворюють теоретичну базу опрацювання даних з використанням комп'ютерів.

У загальній теорії алгоритмів виділяють дві сторони:

- дескриптивну, яка вивчає питання наявності чи відсутності алгоритмів і числень, які приводять до заданої мети (але без оцінки затрат на досягнення цієї мети), і способи задання цих алгоритмів та числень;

- метричну, яка займається оцінюванням складності процесів обчислення. Ця сторона теорії алгоритмів сьогодні ще не склалась у єдину чітку систему.

Розділ 1

БАЗОВІ ПОНЯТТЯ АЛГОРИТМІВ ТА ЇХНІ СКЛАДНОСТІ

1.1. Оцінювання алгоритмів

Поняття алгоритму є головним у математиці. Детальне вивчення і глибоке розуміння його потрібне насамперед для розробки конкретних алгоритмів. Для того ж, щоб орієнтуватись у великій кількості алгоритмів, треба вміти порівнювати різні алгоритми розв'язування одних і тих самих задач, причому не тільки за якістю розв'язку, а й за характеристиками самих алгоритмів (кількістю операцій, потрібним обсягом пам'яті тощо). Таке порівняння неможливе без уведення точної мови для пояснення цих понять.

Для оцінки алгоритмів є багато критеріїв. Найчастіше користувача цікавить порядок зростання необхідних для розв'язування задачі часу й обсягу пам'яті зі збільшенням вхідних даних. З кожною конкретною задачею пов'язане деяке число, яке називають її ***розміром***. Це число виражає обсяг вхідних даних, потрібних для опису задачі. Наприклад, розміром задачі про сортування може бути кількість елементів масиву, який треба посортувати.

Час, затрачений алгоритмом, як функцію розміру задачі, називають ***часовою складністю*** цього алгоритму. Поведінку цієї складності в границі зі збільшенням розміру задачі називають ***асимптотичною часовою складністю***. Часова складність алгоритму відображає потрібні для його роботи витрати часу.

Аналогічно визначають ємнісну складність як затрачуваний обсяг пам'яті, необхідний для реалізації алгоритму, а також ***асимптотичну ємнісну складність***. Асимптотична складність алгоритму визначає розмір задач, які можна розв'язати цим алгоритмом.

Нехай A – алгоритм для розв'язування деякого класу задач, а n – розмір окремої задачі з цього класу. Визначимо $f_A(n)$ як робочу функцію, яка дає оцінку алгоритму для задачі розміру n. Будемо користуватись таким критерієм для оцінки якості алгоритму A.

Алгоритм A називають ***поліноміальним***, якщо $f_A(n)$ зростає не швидше, ніж поліном від n, в іншому випадку алгоритм A називають ***експоненціальним***. Тобто до експоненціальних належать і алгоритми, складність яких можна оцінити, наприклад, функцією $n^{\log n}$, яка не є експонентою.

Цей критерій ґрунтується на часі роботи в гіршому випадку, проте аналогічний критерій можна визначити і для середнього часу роботи.

Різниця між поліноміальними та експоненціальними алгоритмами дуже помітна, якщо проаналізувати вплив збільшення швидкодії комп'ютера на час роботи алгоритмів.

Приклад 1.1. Нехай задано п'ять алгоритмів A_1, ...,A_5 розв'язку однієї задачі з часовими складностями, наведеними в другому стовпці табл. 1.1.

Таблиця 1.1

Зміна розміру задач залежно від часової складності алгоритмів

Алгоритм	Часова складність	1 с	1 хв	1 год
A_1	n	1 000	$6\cdot10^4$	$3{,}6\cdot10^6$
A_2	$n\log_2 n$	140	4 893	$2\cdot10^5$
A_3	n^2	31	244	1 897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Тут часова складність – це кількість одиниць часу, потрібна для опрацювання входу розміру n. Нехай одиницею часу буде 1 мілісекунда. Тоді за 1 с алгоритм A_1 може опрацювати вхід розміром 1 000, а алгоритм A_5 – вхід розміру не більше 9. В табл. 1.1 наведено розміри задач, які можна розв'язати кожним з цих алгоритмів за 1 с, 1 хв, 1 год.

Припустимо, що швидкодія комп'ютера збільшилась на порядок, тобто у 10 разів.

Таблиця 1.2

Ефект десятиразового прискорення

Алгоритм	Часова складність	Максимальний розмір задачі	
		початковий	після прискорення
A_1	n	S_1	$10S_1$
A_2	$n\log_2 n$	S_2	Для великих S_2 приблизно $10S_2$
A_3	n^2	S_3	$3{,}16S_3$
A_4	n^3	S_4	$2{,}15S_4$
A_5	2^n	S_5	$S_5+3{,}3$

У табл. 1.2 наведено інформацію про те, як зростуть розміри задач, які можна розв'язати завдяки такому збільшенню швидкості тими ж алгоритмами. Зазначимо, що десятиразове збільшення швидкості для експоненціального алгоритму A_5 збільшує розмір задачі, яку можна розв'язати, лише на 3, тоді як для поліноміального алгоритму A_3 розмір задачі більш ніж потроюється.

Замість ефекту збільшення швидкості розглянемо ефект застосування дієвішого алгоритму. Розглянемо табл. 1.1. Якщо за основу для порівняння взяти 1 хв, то, замінюючи алгоритм A_4 алгоритмом A_3, можна розв'язати у 6 разів більшу задачу, а замінюючи A_4 на A_2, – у 125 разів більшу. Якщо для порівняння взяти 1 год, то результати будуть ще разючіші.

Звідси випливає висновок, що асимптотична складність алгоритму є важливим мірилом якості алгоритму, причому таким, який обіцяє стати ще важливішим у разі збільшення швидкості обчислень.

Крім порядку зростання значень, треба також враховувати мультиплікативну сталу. Алгоритм з більшим порядком зростання може мати меншу сталу, ніж алгоритм з меншим порядком. У такому випадку алгоритм зі складністю, яка швидко зростає, доцільніше використовувати для задач з малим розміром, можливо навіть для всіх задач, які нас цікавлять.

Наприклад, припустимо, що часові складності алгоритмів A_1, ..., A_5 дорівнюють, відповідно, $1\,000n$, $100n\log_2 n$, $10n^2$, n^3, 2^n. Тоді алгоритм A_5 буде найліпшим для задач розміру $2 \le n \le 9$, алгоритм A_3 – для задач розміру $10 \le n \le 58$, алгоритм A_2 – при $59 \le n \le 1024$, а алгоритм A_1 – при $n > 1024$.

Якщо алгоритм опрацьовує входи розміру n за час cn^2, де c – деяка стала, то кажуть, що складність цього алгоритму $O(n^2)$ (порядку n^2). Загалом, функцію $f(n)$ визначають як $O(g(n))$ і кажуть, що вона порядку $g(n)$ для великих n, якщо

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{const} \neq 0.$$

Функція $f(n) \in o(g(n))$ (o мале від $g(n)$) для великих n, якщо

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Інтуїтивно, якщо $f(n) \in O(g(n))$, то ці дві функції зростають з однаковою швидкістю при $n \to \infty$. Якщо $f(n) \in o(g(n))$, то $g(n)$ зростає набагато швидше, ніж $f(n)$.

Отже, алгоритм є поліноміальним, якщо $f_A(n) = O(P_k(n))$ або $f_A(n) = o(P_k(n))$, де $P_k(n)$ – деякий поліном від змінної n довільного фіксованого степеня k. В іншому випадку алгоритм є експоненціальним. Точні коефіцієнти $f_A(n)$ залежать від реалізації алгоритму, оскільки характеристика $O(g(n))$ властива саме алгоритму.

1.2. Необхідність формалізації поняття алгоритму

Інтуїтивне поняття алгоритму, наведене у параграфі 1.1, є хоча не строгим, проте настільки ясним, що практично завжди очевидно, чи буде алгоритмом той чи інший процес. У загальному вигляді проблему знаходження алгоритму можна сформулювати так: "задано певний клас початкових даних і задача, для якої ці початкові дані допустимі. Треба знайти алгоритм, який розв'язує цю задачу".

Задачу, для якої знайдено алгоритм розв'язування, називають ***розв'язною*** (такою, яку можна розв'язати). Щоб довести розв'язність задачі, достатньо побудувати відповідний алгоритм, а тому достатньо інтуїтивного поняття алгоритму. Для того ж, щоб довести, що задача є ***нерозв'язною*** (не існує алгоритму для її розв'язування), такого формулювання не достатньо. Для цього необхідно точно знати, що таке алгоритм, визначити це поняття чітко математично.

Є велика кількість задач, розв'язок яких не знайдено. Наприклад, **проблема Гольдбаха**: *знайти алгоритм, який дав би змогу для довільного цілого числа n ($n>6$) знайти хоча б один розклад на три прості доданки.* Для непарних n ця проблема розв'язана 1937 р. І. Виноградовим, однак для парних n вона нерозв'язна й дотепер.

Деякі задачі на знаходження алгоритму, які довго не піддавались розв'язанню, виявились такими, що ***не можуть бути розв'язані,*** і це строго доведено. До таких задач, наприклад, належать три стародавні геометричні проблеми:

- **задача про квадратуру круга** (за допомогою циркуля і лінійки знайти метод побудови квадрата, рівновеликого заданому кругу);
- **задача про трисекцію кута** (за допомогою циркуля і лінійки знайти метод ділення довільного кута на три однакові частини);
- **задача про подвоєння куба** (знайти метод, який дає змогу на стороні довільного куба за допомогою циркуля і лінійки

побудувати сторону куба, об'єм якого вдвічі більший від об'єму заданого).

Десята проблема Гільберта: побудувати алгоритм, який дає змогу для довільного діофантового рівняння $F(x,y,...)=0$ визначити, чи має воно цілочисловий розв'язок. Ця проблема сформульована 1901 р., а її нерозв'язність строго доведена 1970 р. Ю. Матіясевичем.

Тут $F(x,y,...)=0$ – поліном з цілими показниками степенів і цілими коефіцієнтами. Наприклад, рівняння $x^2+y^2-z^2=0$ має цілочисловий розв'язок $x=3, y=4, z=5$, а рівняння $6x^{18}-x+3=0$ не має цілочислового розв'язку.

Насправді алгоритмічна нерозв'язність певного класу задач означає, що алгоритм їхнього розв'язування жодним способом ніколи і ніким не буде побудовано. А це потребує обґрунтування у вигляді математичного доведення. Таке доведення можна побудувати лише тоді, коли поняття алгоритму є об'єктом математичної теорії.

Проблема формалізації поняння алгоритму стала однією із центральних математичних проблем у середині 30-х років XX ст. Її вирішенням займались Д. Гільберт, К. Гедель, А. Черч, С.-К. Кліні, А. Тьюрінг, Е. Пост, пізніше А. Марков, А. Колмогоров та ін. Результати їхніх досліджень започаткували нову математичну теорію – теорію алгоритмів, яка вивчає різні алгоритмічні системи.

Деякі класичні алгоритмічні системи розглянемо в розділі 2.

1.3. Алфавітні оператори й алгоритми

1.3.1. Абстрактні алфавіти й алфавітні оператори

Абстрактним алфавітом *називають довільну скінченну сукупність елементів.*

Природа елементів, які називають ***буквами*** алфавіту, може бути довільною, проте вони повинні бути попарно різними, а їхня кількість – скінченною.

Наприклад, алфавітами є $\mathcal{A}_1=\{x,y\}$ та $\mathcal{A}_2=\{0,1\}$.

Слово *у заданому алфавіті – це довільна скінченна впорядкована послідовність букв цього алфавіту.* ***Довжина слова*** *– кількість букв у слові.*

Наприклад, слова в алфавіті $\mathcal{A}_1$: *x, y, xy, xxy*. Будь-яке двійкове число є словом в алфавіті $\mathcal{A}_2$.

Множина слів, яку можна побудувати у скінченному алфавіті, є зліченною. Поряд зі словами додатної довжини розглядають порожнє слово (позначимо його Λ), яке не містить жодної букви і має довжину 0.

Алфавіти можна розширювати, додаючи до них нові букви. Якщо до алфавіту української мови додати пропуск і знаки пунктуації, то цілу сторінку можна розглядати як одне слово.

Слова можна впорядковувати в лексикографічному порядку, що визначений порядком букв в алфавіті.

На множині слів визначено дві операції: *конкатенація* (приєднання) та *підстановка* (заміна).

Конкатенацією *двох слів A та B називають слово AB, отримане приписуванням слова B до слова A.*

Ця операція не комутативна, проте асоціативна.

Приклад 1.2. Конкатенацією слів $A=abc$ та $B=xy$ буде слово $AB=abcxy$.

Слово A називають ***підсловом*** *слова B, якщо B можна записати в такому вигляді:*

$$B=CAD, \tag{1.1}$$

де C та D – деякі слова, можливо, порожні. Слово A є входженням у слово B.

Для деяких пар слів A та B можна навести кілька розкладів вигляду (1.1). Наприклад, нехай $B=xyxyxy$, $A=xy$, тоді існує три розклади вигляду (1.1):

$$B=Axyxy,\ B=xyAxy,\ B=xyxyA.$$

Для усунення такої неоднозначності вводять поняття i-го входження підслова у задане слово.

Якщо в розкладі (1.1) *підслово C має мінімальну довжину, то таке входження слова A в слово B називають* ***першим входженням***.

Нехай задано слова A, B, F і A є i-м входженням у слово B:

$$B=CAD.$$

Тоді *слово G отримують* ***операцією заміни*** (*підстановки*) *i-го входження слова A словом F, якщо*

$$G=CFD.$$

Якщо в операції заміни $i=1$, *то підстановку називають* ***стандартною*** (*або канонічною*).

Алфавітним оператором, *або алфавітним відображенням,* φ *називають відповідність між словами в одному або різних алфавітах.*

Функцію *називають* ***словниковою***, *якщо вона перетворює слово одного алфавіту в слово іншого алфавіту.*

Нехай $\{P\}_X$ – деяка множина вхідних слів в алфавіті X, $\{Q\}_Y$ – множина вихідних слів в алфавіті Y. Тоді алфавітним оператором φ є словникова функція такого вигляду

$$\varphi : \{P\}_X \to \{Q\}_Y .$$

Приклад 1.3. Якщо φ_1 – дзеркальне відображення вхідних даних, то $abcd \xrightarrow{\varphi_1} dcba$. Якщо φ_2 – оператор диференціювання, то $\sin x + \cos x \xrightarrow{\varphi_2} \cos x - \sin x$.

Нехай P – вхідне слово. *Якщо слову P оператор* φ *не ставить у відповідність жодного вихідного слова, то кажуть, що на слові P оператор* φ ***не визначений.***

Сукупність усіх слів, на яких оператор φ *визначений, називають* ***областю визначення*** (*областю застосування*) *оператора* φ.

Сукупність усіх слів, на яких оператор φ *невизначений, називають* ***областю заборони*** φ.

Завжди можна вважати, що вхідний алфавіт X і вихідний алфавіт Y оператора φ є одним і тим же алфавітом, оскільки їх можна об'єднати в один спільний алфавіт $Z = X \bigcup Y$ і розглядати оператор

$$\varphi : \{P\}_Z \to \{Q\}_Z .$$

Проте розширення вхідного алфавіту не розширює області визначення оператора φ.

Розрізняють однозначні та багатозначні оператори.

Однозначний оператор кожному вхідному слову ставить у відповідність не більше одного вихідного слова, а ***багатозначний*** оператор – декілька різних вихідних слів:

$$P \to \begin{cases} Q_1 \\ \vdots \\ Q_n \end{cases}$$

У цьому разі вибір вихідного слова відбувається або випадково, або відповідно до імовірностей, приписаних словам $Q_1, Q_2, \ldots, Q_n$ (*імовірнісний оператор*). Зазначимо, що розподіл імовірностей між $Q_1, Q_2, \ldots, Q_n$ повинен змінюватися в процесі опрацювання інформації. В іншому випадку багатозначний оператор вироджується в однозначний, оскільки за вихідне слово завжди вибирають одне і те ж слово Q_i, яке має найбільшу ймовірність.

Довільні процеси перетворення інформації можуть бути зведені до поняття алфавітного оператора. Наприклад, процеси перекладу з однієї мови іншою, написання рефератів, статей та інші перетворення лексичної інформації можна розглядати як реалізацію багатозначних алфавітних операторів.

1.3.2. Кодувальні алфавітні оператори

Серед алфавітних операторів особливу роль відіграють кодувальні оператори. Кодування застосовують, головно, для того, щоб слова в довільних алфавітах можна було задавати словами в деякому фіксованому, стандартному алфавіті.

Нехай $\mathcal{A}$ – деякий алфавіт, який називають ***стандартним,*** $\mathcal{B}$ – довільний алфавіт. Нехай $\{L\}_A$, $\{M\}_B$ – множини слів у відповідних алфавітах. Кажуть, що *множина* $\{M\}_B$ ***закодована*** *в алфавіті* $\mathcal{A}$, *якщо задано такий однозначний оператор:*

$$\varphi : \{M\}_B \to \{L\}_A .$$

Оператор φ *називають* ***кодувальним***, а *слова з* $\{L\}_A$ – ***кодами об'єктів з*** $\{M\}_B$.

Легко довести таке твердження.

Теорема *Кодувальний оператор є взаємно однозначним тоді й тільки тоді, коли:*

1) *коди різних букв алфавіту* $\mathcal{B}$ *різні;*

2) *код довільної букви алфавіту* $\mathcal{B}$ *не може бути першим входженням у коди інших букв цього алфавіту.*

Умова 2 виконується автоматично, якщо коди мають однакову довжину.

Кодування об'єктів алфавіту $\mathcal{B}$ *словами однакової довжини називають* ***нормальним кодуванням***.

Отже, у разі виконання умови 1 нормальне кодування забезпечує взаємну однозначність оператора кодування φ.

Розглянемо декілька кодувань, які трапляються найчастіше.

Кодування слів у багатобуквеному алфавіті. Нехай $\mathcal{B}$ – довільний алфавіт з n букв, $\mathcal{A}$ – стандартний алфавіт з m букв (m>1). Для довільної пари (n, m) завжди можна задати таке число l, що

$$m^l \geq n . \qquad (1.2)$$

Проте відомо, що величина m^l визначає кількість різних слів довжини l в m-буквеному алфавіті. Отже, нерівність (1.2) засвідчує, що всі букви алфавіту $\mathcal{B}$ завжди можна закодувати словами довжини l в алфавіті $\mathcal{A}$ так, щоб коди різних букв були різними. Кожне таке кодування є нормальним і забезпечує взаємну однозначність оператора кодування φ.

Приклад 1.4. Якщо n = 16, m = 2 (наприклад, $\mathcal{A} = \{0,1\}$), то $l = 4$

$$2^4 \geq 16$$

і кожну з 16 букв можна закодувати словами довжини 4:

0000, 0001,…,1111.

Власне ці коди використовують для кодування цифр шістнадцяткової системи в комп'ютері.

Кодування безконечних алфавітів. Згідно з визначенням алфавіту, він повинен бути скінченною множиною. Проте на практиці зручно розглядати безконечні алфавіти, які складаються зі скінченної кількості букв з індексами, які набувають натуральних значень 0, 1, 2,… Такий алфавіт формально безконечний.

Наприклад, якщо алфавіт $\mathcal{A}$ складається з символів

$$a, b, c, g, \ldots, r, f_j, d_j^i \quad (i, j = 0,1,2,\ldots),$$

то від $\mathcal{A}$ можна перейти до скінченного алфавіту

$$\mathcal{B} = \{a, b, \ldots, r, f, d\} \cup \{\alpha, \beta\} \quad (\alpha, \beta) \notin \mathcal{A}$$

і кодувати букви $\mathcal{A}$ в алфавіті $\mathcal{B}$ так:

1) символи $a, b, \ldots, r$ кодують ними самими;

2) символи f_j, d_j^i кодують словами

$$f_j = f\underbrace{\alpha\ldots\alpha}_{j}, \quad d_j^i = d\underbrace{\alpha\ldots\alpha}_{j}\underbrace{\beta\ldots\beta}_{i}.$$

Тоді скінченний алфавіт $\mathcal{B}$ завдяки такому кодуванню дає змогу виписати коди як скінченні слова для довільних $f_j, d_j^i \in \mathcal{A}$ з конкретними значеннями i, j, які б великі вони не були.

Отже, довільні слова у безконечному алфавіті, де безконечність виникає внаслідок індексації букв, можна закодувати у скінченному алфавіті.

1.3.3. Способи задання алфавітних операторів. Поняття алгоритму

Важливим моментом для алфавітних операторів є спосіб їхнього задання. Розрізняють два способи задання операторів.

1. ***Табличний спосіб задання операторів.*** Такий спосіб застосовують тоді, коли область визначення оператора скінченна. Оператор φ задають таблицею відповідності, у якій для кожного вхідного слова P з області визначення φ виписано відповідне вихідне слово Q:

$$P_1 \to Q_1,$$
$$\cdots$$
$$P_n \to Q_n.$$

У деяких випадках табличний спосіб можна використовувати і для задання операторів з безконечною областю визначення. Наприклад,

$$\underbrace{xx\ldots x}_{n \text{ разів}} \to \underbrace{yy\ldots y}_{n+1 \text{ разів}} \quad (n=1,2,\ldots).$$

Проте в загальному випадку для задання операторів з безконечною областю визначення табличний спосіб принципово не підходить.

2. ***Задання оператора скінченною системою правил***, яка дає змогу за скінченну кількість кроків знайти значення φ на довільному вхідному слові, наприклад, система правил для додавання натуральних чисел у системі числення з основою p або система правил знаходження найбільшого спільного дільника двох додатних чисел та ін.

Алфавітний оператор, заданий скінченною системою правил, називають ***алгоритмом.***

Очевидно, що оператор, заданий табличним способом, теж є алгоритмом.

Відмінність у поняттях алфавітного оператора й алгоритму полягає в тому, що в понятті оператора суттєвою є лише відповідність, яка усталюється між вхідними і вихідними словами, і не задано правил, які реалізують цю відповідність. У понятті алгоритму головним є спосіб задання відповідності. Тобто оператор визначає, що треба зробити, а алгоритм відображає не лише що, але і як це треба зробити.

Отже, ***алгоритм*** – це алфавітний оператор разом із системою правил, яка визначає його дію:

$$A = <\varphi, \mathcal{P} \mid \mathcal{P} - \text{система правил}>.$$

Два алфавітні оператори називають ***рівними*** *якщо вони мають одну і ту ж область визначення й однаковим вхідним словам з цієї області ставлять у відповідність однакові вихідні слова.*

Два алгоритми $A_1 = <\varphi_1, \mathcal{P}_1>$, $A_2 = <\varphi_2, \mathcal{P}_2>$ *називають* ***рівними****, якщо відповідні їм алфавітні оператори рівні та системи правил збігаються:*

$$A_1 = A_2 \Leftrightarrow \varphi_1 = \varphi_2,\ \mathcal{P}_1 = \mathcal{P}_2.$$

Два алгоритми A_1, A_2 *називають* ***еквівалентними****, якщо відповідні їм оператори рівні, а системи правил різні:*

$$A_1 \cong A_2 \Leftrightarrow \varphi_1 = \varphi_2,\ \mathcal{P}_1 \neq \mathcal{P}_2.$$

Еквівалентні алгоритми задають розв'язок однієї й тієї ж задачі, проте різними способами.

1.4. Властивості алгоритмів. Способи композиції алгоритмів

1.4.1. Основні властивості алгоритмів

В алгоритмі $A=<\varphi,\mathcal{P}>$ система $\mathcal{P}$ – це не просто довільний перелік скінченної кількості правил, які визначають послідовність виконання операцій під час розв'язування певної задачі. Система $\mathcal{P}$ повинна мати властивості, притаманні кожному алгоритму: дискретність, ефективність, скінченність, результативність, масовість.

Дискретність алгоритму. Алгоритм описує процес послідовної побудови величин, який відбувається в дискретному часі. У початковий момент t_0 задають скінченну систему σ_0 вхідних величин. Далі в кожен інтервал часу (t_i,t_{i+1}) із системи σ_i величин, які були в момент t_i, за певним правилом отримують систему величин σ_{i+1}. Перехід від σ_i до σ_{i+1} вважають елементарним кроком алгоритму. Інтервали (t_i,t_{i+1}) є різними для різних елементарних кроків.

Ефективність алгоритму. Елементарні кроки, які необхідно зробити в алгоритмі, повинні бути ефективними, тобто виконуваними точно і за короткий відрізок часу. Цю властивість ще називають зрозумілістю алгоритму. Це означає, що кроки алгоритму повинні містити лише операції з набору операцій виконавця. Тобто різні виконавці, згідно з алгоритмом, повинні діяти однаково і прийти до одного й того ж результату.

Скінченність. Алгоритм завжди повинен закінчуватися після скінченної (можливо, дуже великої) кількості кроків.

Результативність. Алгоритм завжди забезпечує отримання певного результату розв'язування задачі, що є наслідком скінченної кількості елементарних кроків та їхньої ефективності. Якщо деякий елементарний крок не дає результату, то має бути зазначено, що саме треба вважати результатом алгоритму.

Масовість. Алгоритм повинен бути застосовним до цілого класу задач, а не до одної задачі. Це означає, що початкову систему величин σ_0 можна вибирати з деякої потенційно безконечної множини.

Якщо система $\mathcal{P}$ з пари $<\varphi,\mathcal{P}>$ задовольняє всі перелічені вище властивості, крім властивості скінченності, то таку пару називають ***обчислювальним методом***.

Обчислювальним методом, наприклад, є процес ділення двох натуральних чисел.

Алгоритми, у яких кількість кроків скінченна, але дуже велика, доцільно вважати обчислювальними методами.

1.4.2. Різновиди алгоритмів

Наявність чи відсутність у системі правил $\mathcal{P}$ тієї чи іншої властивості розбиває множину всіх алгоритмів на два класи щодо конкретної властивості.

Розглянемо деякі з цих властивостей – детермінованість, самозмінність, самозастосовність, універсальність.

1. Нехай система $\mathcal{P}$ є такою, що система величин σ_{i+1} однозначно визначена системою σ_i. *Алгоритм $A=<\varphi,\mathcal{P}>$ називають* ***детермінованим*** *(визначеним), якщо $\mathcal{P}$ задовольняє цю властивість,* і ***недетермінованим*** *в іншому випадку.*

У детермінованому алгоритмі $<\varphi,\mathcal{P}>$ алфавітний оператор φ є *однозначним*, у недетермінованому – *багатозначним*.

2. *Алгоритм $A=<\varphi,\mathcal{P}>$ називають* ***самозмінним***, *якщо в процесі переробки алгоритмом вхідних слів система $\mathcal{P}$ змінюється,* і ***несамозмінним*** в іншому випадку.

Результат дії самозмінного алгоритму залежить не лише від заданого вхідного слова P_k, а й від того, які саме слова $P_1,...,P_{k-1}$ були опрацьовані алгоритмом до подання на вхід слова P_k. Кожна послідовність $P_1,...,P_{k-1}$ генерує різні зміни в системі правил $\mathcal{P}$.

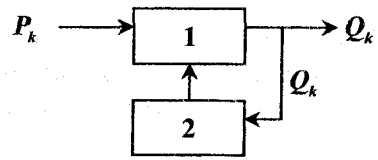


Рис. 1.1.

Самозмінний алгоритм можна розглядати як систему двох алгоритмів, перший з яких (робочий) виконує переробку вхідних слів, а другий (керівний) вносить зміни в робочий алгоритм (рис. 1.1).

Частковим випадком самозмінних алгоритмів є алгоритми, які ***самонавчаються***. В таких алгоритмах закладені лише найзагальніші принципи опрацювання інформації, а детальний алгоритм формується в процесі його роботи залежно від характеру інформації, яка надходить.

3. Нехай система правил $\mathcal{P}$ алгоритму $A=<\varphi,\mathcal{P}>$ закодована певним способом у вхідному алфавіті алгоритму A. Позначимо цей код $\mathcal{P}^{cod}$.

Алгоритм $A=<\varphi,\mathcal{P}>$ *називають* ***самозастосовним****, якщо слово* $\mathcal{P}^{cod}$ *входить в область визначення* A, і ***несамозастосовним*** в іншому випадку.

Приклад 1.5. Самозастосовним алгоритмом є тотожний алгоритм у довільному алфавіті $\mathcal{A}$, що містить не менше двох букв, який переробляє довільне слово в себе.

Несамозастосовним алгоритмом є нульовий алгоритм, який порожнє слово перетворює в букву з заданого алфавіту:

$$\Lambda \to a, \quad a \in \mathcal{A}, \quad \Lambda \text{ – порожнє слово.}$$

Цей алгоритм не застосовний до жодного слова, і тому й до P^{cod}.

4. *Алгоритм називають* ***універсальним****, якщо він еквівалентний довільному наперед заданому алгоритму* $A=<\varphi,\mathcal{P}>$.

Отже, універсальний алгоритм може виконувати роботу довільного алгоритму A. Іншими словами, за його допомогою можна реалізувати будь-які алгоритмічні процеси, якими б складними вони не були.

1.4.3. Композиція алгоритмів

Нові алгоритми можуть бути побудовані з уже відомих шляхом застосування різних способів композиції алгоритмів. Є чотири способи композиції алгоритмів: суперпозиція, об'єднання, розгалуження та ітерація.

1. Суперпозиція алгоритмів.

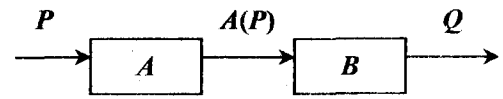


Рис. 1.2.

У випадку ***суперпозиції*** двох алгоритмів A та B вихідне слово одного з них розглядають як вхідне слово іншого (рис. 1.2).

Нехай $D(A)$ – область визначення алгоритму A, а $D(B)$ – область визначення алгоритму B. Результат суперпозиції A та B можна зобразити у такому вигляді:

$$C(P)=B(A(P)), \quad P\in D(A), \quad A(P)\in D(B).$$

У цьому випадку область визначення алгоритму B повинна включати в себе множину вихідних слів алгоритму A.

Суперпозиція може виконуватись для довільної скінченної кількості алгоритмів.

Приклад 1.6. Суперпозицією алгоритмів $A(P)=xP$ та $B(P)=Py$ є алгоритм $C(P)=xPy$.

2. Об'єднання алгоритмів.

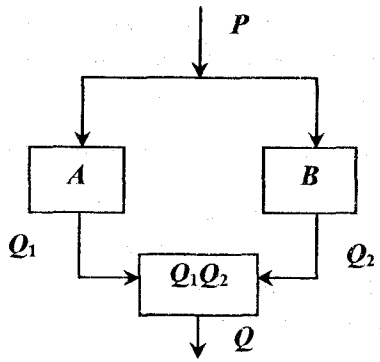


Рис. 1.3.

Розглянемо два алгоритми A та B в одному й тому ж алфавіті V. Нехай $D(A)$ та $D(B)$ – області визначення цих алгоритмів. ***Об'єднанням*** алгоритмів A та B є такий алгоритм C в цьому ж алфавіті, який перетворює довільне вхідне слово $P \in D(A) \bigcap D(B)$ в конкатенацію слів $A(P)$ і $B(P)$ (рис. 1.3):

$$C(P) = A(P)B(P).$$

На всіх інших словах алгоритм C вважають невизначеним.

Приклад 1.7. Об'єднанням двох алгоритмів $A(P) = xP$ та $B(P) = Py$ буде алгоритм $C(P) = xPPy$.

3. Розгалуження алгоритмів. Нехай задано три алгоритми A, B, C з областями визначення $D(A), D(B), D(C)$, відповідно.

Розгалуженням алгоритмів називають композицію трьох алгоритмів A, B, C, задану співвідношенням

$$F(P) = \begin{cases} A(P), & \text{якщо } C(P) = R; \\ B(P), & \text{якщо } C(P) \neq R, \end{cases}$$

де R – деяке фіксоване слово, переважно порожнє (рис. 1.4). Областю визначення отриманого алгоритму буде переріз областей визначення всіх трьох алгоритмів: $D(F) = D(A) \bigcap D(B) \bigcap D(C)$.

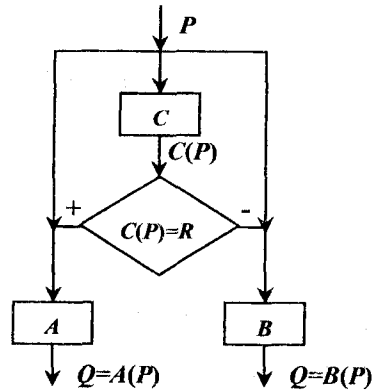


Рис. 1.4.

Приклад 1.8. Нехай алгоритми A, B, C задані такими правилами:

A: $ab \to ba$; $ba \to ab$,

B: $ab \to aab$; $ba \to aba$,

C: $ab \to a$; $ba \to \Lambda$,

$R = \Lambda$ (порожнє слово).

Тоді $D(ab) = aab$ і $D(ba) = ab$.

4. Ітерація алгоритмів.

Ітерацією (повторенням) двох алгоритмів A та B називають алгоритм C, який визначають так: для довільного вхідного слова P вихідне слово $C(P)$ отримують як результат послідовного багаторазового застосування алгоритму A доти, доки не буде отримано слово, перетворюване алгоритмом B в деяке фіксоване слово R (рис. 1.5).

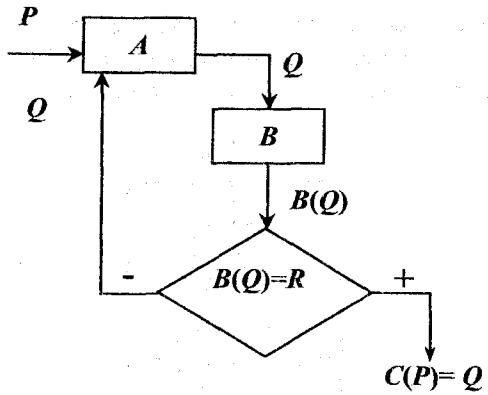


Рис. 1.5.

Отже, у процесі ітерації алгоритму A формується така послідовність слів $Q = Q_0, Q_1, ..., Q_n = C(P)$, що $Q_i = A(Q_{i-1})$ $(i = \overline{1,n})$, причому для $i = \overline{1,n-1}$: $B(Q_i) \neq R$, а $B(Q_n) = R$. Області визначення алгоритмів A та B повинні охоплювати множину вихідних слів алгоритму A.

Приклад 1.9. Нехай алгоритм A заданий правилом

$$P_1 abP_2 \to P_1 baP_2,$$

де P_1, P_2 – довільні (можливо, порожні) слова в алфавіті $V = \{a, b\}$.

Алгоритм B заданий правилами

$$bbbaa \to \Lambda, \quad P \to P,$$

де P – довільне слово в V, $P \neq bbbaa$.

Тоді при $R = \Lambda$

$$C(ababb) = bbbaa.$$

1.5. Завдання для самостійної роботи

1. Визначити час, необхідний для сортування мільйона чисел за алгоритмами вставки, який має оцінку $2n^2$ операцій, та злиття з оцінкою $50n\log_2 n$ (n – кількість чисел). Порівняти ефективність

сортування першим алгоритмом на потужному комп'ютері зі швидкодією 100 млн оп/с та другим алгоритмом на домашньому комп'ютері зі швидкодією 1 млн оп/с?

2. Скласти алгоритм, який перевіряє, чи задане натуральне число є простим. Оцінити часову та ємнісну складність цього алгоритму. Модифікувати його так, щоб оцінка кількості дій не перевищувала $C\sqrt{n}$, де n – число, яке перевіряють, C – деяка константа.

3. Скласти й оцінити алгоритм, який визначає кількість різних елементів у n-елементному лінійному масиві. Як зміниться оцінка цього алгоритму у випадку впорядкованого масиву?

4. Елементи двох лінійних масивів $A[1..m]$ та $B[1..n]$ розташовані у порядку неспадання. Скласти алгоритм, результатом дії якого є неспадний масив $C[1..m+n]$, у якому кожен елемент трапляється стільки разів, скільки разів він трапляється в масивах A та B разом. Оцінка алгоритму не повинна перевищувати $C(m+n)$.

5. У прямокутному масиві $A[1..m, 1..n]$ елементи в кожному рядку і в кожному стовпці не спадають. Скласти алгоритм, який визначає, чи є в масиві число x. Кількість дій не повинна перевищувати $C \cdot (m+n)$.

6. Нехай задано два рядки A та B. Скласти й оцінити алгоритм, який визначає, чи з букв рядка A можна отримати рядок B. Букви можна переставляти, але використовувати стільки разів, скільки разів вони трапляються у першому рядку.

7. Скласти й оцінити алгоритми сортування лінійного масиву: простим вибором, методом включення, методом "бульбашки", методом швидкого сортування.

8. Скласти й оцінити алгоритми спеціального сортування: сортування деревом, сортування підрахунком, цифрове сортування, сортування вичерпуванням (bucket sort).

9. Нехай в алфавіті $\{x,y\}$ задано два алгоритми: $A(P) = xxP$ та $B(P) = Pyy$, де P – довільне вхідне слово. Отримати алгоритми, що є суперпозицією та об'єднанням цих алгоритмів.

10. Нехай в алфавіті $\{a,b\}$ алгоритми A, B, C задані такими правилами:

$$A: aab \to bba;\ bba \to aab;$$
$$B: aab \to bab;\ bba \to aba;$$
$$C: aab \to ab;\ bba \to R,$$

де $R=\lambda$ (порожне слово). Отримати алгоритм, що є розгалуженням алгоритмів A, B, C.

11. Нехай в алфавіті $\{a,b\}$ алгоритми A, B задані такими правилами:

$$A: P_1aabP_2 \to P_1baP_2;\ B: baba \to \lambda;\ P \to \hat{P},$$

де P – довільне слово, відмінне від *baba*.

Отримати алгоритм C, що є ітерацією алгоритмів A, B. Яким буде вихідне слово, якщо на вхід алгоритму C подано слово *aaaabab*?

Розділ 2

КЛАСИЧНІ АЛГОРИТМІЧНІ СИСТЕМИ

2.1. Поняття про алгоритмічні системи

Нехай $A = <\varphi, \mathcal{P}>$ – деякий алгоритм із системою правил $\mathcal{P}$.

Алгоритм $A_F = <\varphi, \mathcal{P}_F>$, *де* $\mathcal{P}_F$ *– формальний опис системи* $\mathcal{P}$, *будемо називати* ***формальним еквівалентом*** *алгоритму* A.

Алгоритмічною системою *називають спосіб задання алгоритмів у деякому фіксованому формалізмі* F, *який дає змогу для довільного наперед заданого алгоритму* A *задати його формальний еквівалент* A_F.

Алгоритмічні системи можна вважати формалізацією поняття алгоритму. Виділяють три типи таких систем, які відрізняються початковими евристичними міркуваннями стосовно того, що таке алгоритм:

- у першому типі поняття алгоритму пов'язане з найтрадиційнішими поняттями математики – обчисленнями та числовими функціями (числовою називають функцію, значення якої та значення її аргументів – невід'ємні числа). Найпопулярніша система цього типу – *рекурсивні функції* – історично перша формалізація поняття алгоритму;
- другий тип ґрунтується на уявленні про алгоритм як детермінований пристрій, здатний виконувати в кожний окремий момент лише дуже примітивні операції. Основна теоретична система цього типу – *машина Тьюрінга*;

- третій тип алгоритмічних систем – це перетворення слів у довільних алфавітах. Приклади систем цього типу – *канонічна система Поста* й *нормальні алгоритми Маркова.*

Алгоритмічні системи мають специфічні для кожної з них формальні засоби, які дають змогу задавати строго математично довільні алгоритми, в тому числі й алгоритми особливого вигляду – універсальні. Побудова універсального алгоритму є важливим завданням для кожної алгоритмічної системи.

Один з фундаментальних результатів теорії алгоритмів такий: *усі алгоритмічні системи є рівносильними в тому сенсі, що в кожній з них описують (з точністю до еквівалентності) один і той самий клас алгоритмів.*

Тобто для будь-якого алгоритму в одній алгоритмічній системі є еквівалентний йому алгоритм у довільній іншій алгоритмічній системі. Згідно з положеннями теорії алгоритмів, цей клас охоплює всі алгоритми $A=<\varphi,\mathcal{P}>$, визначені на інтуїтивному рівні.

Головні формалізми прикладної теорії алгоритмів можна розділити на два напрями, які умовно називають алгебричним і геометричним.

Алгебричну теорію будують у деякій конкретній символіці, за якої алгоритми розглядають у вигляді лінійних текстів.

У геометричній теорії алгоритми будуються у вигляді множин, між якими вводять зв'язки, які є відображеннями або бінарними відношеннями. У цьому разі значне місце посідає геометрична інтерпретація об'єктів у вигляді графів, вершини яких задають елементи множини, а ребра – відношення між ними. За такої інтерпретації відображення задають у вигляді розмітки вершин або ребер графа.

До алгебричного напряму належать рекурсивні функції, машини Тьюрінга, операторні схеми Ван-Хао, А. Ляпунова, логічні схеми алгоритмів Ю. Янова та ін., до геометричного – зображення нормальних алгоритмів А. Маркова у вигляді граф-схем, запропонованих Л. Калужніним, блок-схемний метод зображення алгоритмів та ін.

Кожна алгоритмічна система охоплює об'єкти двох категорій – елементарні оператори та елементарні розпізнавачі.

Елементарні оператори – це достатньо прості алфавітні оператори. За допомогою послідовного виконання таких операторів відбувається реалізація довільних алгоритмів у заданій алгоритмічній системі.

Елементарні розпізнавачі використовують для розпізнавання певних властивостей інформації, перетворюваної алгоритмом, і для зміни (залежно від результатів розпізнавання) послідовності виконання елементарних операторів.

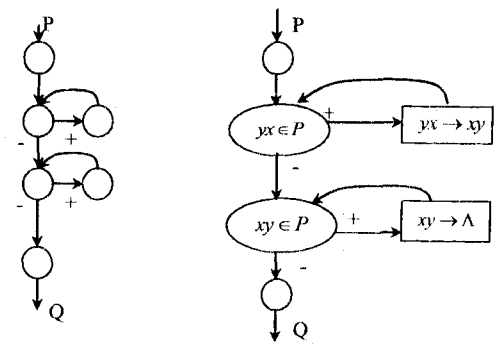
Для задання допустимих у системі елементарних операторів і розпізнавачів кожна система використовує свої засоби.

Розглянемо засоби формального опису деяких алгоритмічних систем – нормальних алгоритмів Маркова, рекурсивних функцій, абстрактних машин Тьюрінга і Поста, операторних алгоритмів.

2.2. Граф-схеми алгоритмів

Поняття граф-схеми алгоритму введене 1959 р. Л. Калужніним.

Скінченні орієнтовані графи спеціального вигляду, які задають набір елементарних операторів і порядок їхньої наступності один за одним, у разі реалізації конкретного алгоритму називають ***граф-схемами*** алгоритмів.



а *б*

Рис. 2.1.

Кожному вузлу граф-схеми, крім двох особливих – вхідного і вихідного, ставиться у відповідність елементарний оператор або елементарний розпізнавач. З кожного розпізнавача виходить дві дуги, з кожного оператора та вхідного вузла – одна дуга. Кількість дуг, що входять у вузол, може бути довільною (рис. 2.1, *а, б*).

Розглянемо, як працює алгоритм, визначений граф-схемою. Вхідне слово потрапляє на вхідний вузол і рухається у напрямі, вказаному стрілками. Під час проходження через операторні вузли слово перетворюється. У разі входження слова в розпізнавальний вузол відбувається перевірка відповідної умови. Якщо слово задовольняє задану умову, то воно виходить з вузла за стрілкою, поміченою "+", а якщо не задовольняє, – за іншою стрілкою, поміченою знаком "-". У розпізнавальних вузлах слово не змінюється.

Якщо вхідне слово P, подане на вхідний вузол граф-схеми, проходячи через вузли схеми та перетворюючись, потрапляє через скінченну кількість кроків у вихідний вузол, то вважають, що P входить в область визначення алгоритму. Слово Q, що в цьому разі з'являється на вихідному вузлі, називають ***вихідним*** словом. Якщо ж перетворення слова P і його рух по граф-схемі триває безмежно довго, не приводячи у вихідний вузол, то вважають, що P не входить в область визначення алгоритму.

За допомогою граф-схеми визначають лише порядок виконання допустимих операцій. У цьому випадку жодних обмежень на складність схеми або характер допустимих операцій не накладають. Конкретні алгоритми отримують з граф-схеми тією чи іншою інтерпретацією компонент схеми. Наприклад, на рис. 2.1, *а* зображена граф-схема, а на рис. 2.1, *б* – одна з її інтерпретацій.

2.3. Система нормальних алгоритмів Маркова

2.3.1. Нормальні алгоритми

Для формалізації поняття алгоритму А. Марков 1954 р. розробив систему нормальних алгоритмів. Алгоритми Маркова – це формальна математична система. Вони були основою для першої мови обробки рядів COMIT. Крім того, є подібність між моделлю Маркова і мовою СНОБОЛ, яка з'явилась після COMIT.

В алгоритмічній системі Маркова існує лише один тип елементарних операторів – оператор підстановки, та один тип елементарних розпізнавачів – розпізнавач входження.

Загальна стратегія роботи алгоритму Маркова полягає в тому, щоб, застосувавши декілька разів оператор підстановки до вхідного рядка R, перетворити його у вихідний рядок Q. Цей процес перетворення є звичайним у таких галузях застосування комп'ютерів, як редагування тексту або компіляція програми.

Простою продукцією (*формулою підстановки*) *називають запис вигляду*

$$u \to w,$$

де u, w – *рядки в алфавіті* V. У цьому разі V не містить символів '$\to$' та '.'. Величину u називають антицедентом, а w – консеквентом.

Уважають, що формула $u \to w$ може бути застосована до рядка $Z \in V$, якщо є хоча б одне входження u в Z. В іншому випадку ця формула не застосовна до рядка Z. Якщо формула може бути застосована, то канонічне (перше ліворуч) входження u в Z замінюється на w.

Наприклад, якщо формулу 'ba'$\to$'c' застосувати до вхідного рядка '$ababab$', то в підсумку отримаємо рядок '$acbab$'. Проте формула 'baa'$\to$'c' до рядка '$ababab$' не може бути застосована.

Заключною продукцією (*заключною підстановкою*) *називають запис вигляду* $u \to \cdot w$, *де* u, w – *рядки в* V.

Нормальним алгоритмом, чи алгоритмом Маркова, *називають упорядковану множину продукцій* $P_1, P_2, ..., P_n$.

Кожна з продукцій містить розпізнавання входження підрядка u в рядок Z та підстановку w замість u у разі успішного розпізнавання. Послідовність виконання продукцій залежить від того, чи може бути застосована до рядка чергова формула підстановки.

Виконання алгоритму починається з перевірки першої продукції. Якщо вона може бути застосована до рядка, то рядок перетворюється. Якщо ж формула не може бути застосована (тобто в рядку не знайдено підрядка, який можна було б замінити), то відбувається перехід до перевірки наступної продукції.

У випадку успішного розпізнавання входження та заміни виконання алгоритму припиняється, якщо підстановка була заключною, або продовжується з першої продукції. Тобто після заміни, визначеної звичайною підстановкою, завжди відбувається повернення до початку нормального алгоритму і знову пошук входження першої підстановки у змінене слово.

Алгоритм Маркова завершується в одному з двох випадків:

- до рядка не може бути застосована жодна з наявних формул підстановок;
- до рядка застосована заключна (термінальна) підстановка.

У кожному з цих випадків уважають, що нормальний алгоритм застосовний до заданого вхідного слова.

Якщо ж у процесі виконання нормального алгоритму безмежну кількість разів застосовувані не завершувальні формули, то алгоритм незастосовний до заданого вхідного слова.

Приклад 2.1. Нехай над словами з алфавіту $V = \{a,b,c\}$ задано алгоритм з такими формулами підстановки

$P_1 : 'ab' \to 'b'$,
$P_2 : 'ac' \to 'c'$,
$P_3 : 'aa' \to 'a'$.

Цей алгоритм вилучає всі входження символа $'a'$ з рядка, за винятком випадку, коли $'a'$ є в кінці рядка.

Простежимо роботу алгоритму, якщо вхідний рядок має вигляд *'bacaabaa'*. Нехай символ => означає результат перетворення, а підрядок, який підлягає заміні, будемо підкреслювати:

$$'baca\underline{ab}aa' \overset{P_1}{\Rightarrow} 'bac\underline{ab}aa' \overset{P_1}{\Rightarrow} 'b\underline{ac}baa' \overset{P_2}{\Rightarrow} 'bc\underline{ba}a' \overset{P_3}{\Rightarrow} 'bcba'.$$

Оскільки далі жодна з формул не може бути застосована, то на цьому робота алгоритму завершується.

Нормальні алгоритми іноді зображають за допомогою граф-схеми.

Алгоритми, які задають граф-схемами, складеними винятково з розпізнавачів входження і операторів підстановки, називають ***нормальними****, якщо їхні граф-схеми задовольняють такі умови:*

1) кожний розпізнавальний вузол Q і відповідний йому операторний вузол $Q \to R$ об'єднані в один узагальнений вузол, який називають операторно-розпізнавальним; усі узагальнені вузли схеми впорядковані за допомогою нумерації від 1 до n;

2) негативний вихід i-го вузла приєднаний до $(i+1)$-го вузла $(\overline{i = 1,n-1})$, а негативний вихід n-го вузла — до вихідного вузла граф-схеми;

3) позитивні виходи всіх узагальнених вузлів приєднані або до першого, або до вихідного вузла граф-схеми; у першому випадку підстановку оператора відповідного вузла називають звичайною, а в другому — заключною;

4) вхідний вузол приєднаний до першого узагальненого вузла.

Приклад 2.2. Алгоритм, зображений на граф-схемі (рис. 2.2), є нормальним і виконує перетворення довільного вхідного слова P в алфавіті $A = \{x, y\}$, що містить m значень x та n значень y, у вихідне слово довжини $|m-n|$, що складається тільки з x (якщо $m > n$), або тільки з y (якщо $n > m$). Нагадаємо, що Λ позначено порожнє слово.

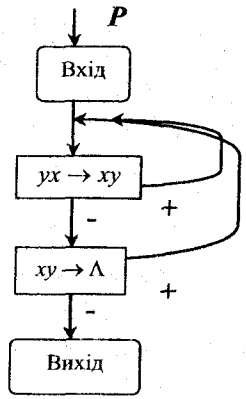


Рис. 2.2.

Нормальні алгоритми прийнято задавати не граф-схемами, а впорядкованими наборами підстановок, у яких кожна підстановка відповідає операторно-розпізнавальному вузлу. У цьому разі звичайні підстановки записують у вигляді $Q \to R$, а заключні – $Q \to \cdot R$.

Упорядкований набір підстановок визначеного типу називають ***схемою*** заданого алгоритму.

Порядок виконання підстановок однозначно визначений умовами 1 – 4 означення нормального алгоритму. Справді, з огляду на ці умови довільна i-та підстановка схеми алгоритму повинна виконуватися тоді й тільки тоді, коли вона є першою із застосовних підстановок, тобто всі підстановки від першої до (i-1)-ї не можуть бути застосовані.

Виконання алгоритму починається з першої формули. Послідовність виконання алгоритму залежить від того, чи може бути застосована до рядка чергова формула підстановки. Процес виконання підстановок закінчується лише тоді, коли жодна з підстановок схеми не може бути застосована до отриманого слова, або коли виконана (перший раз) деяка заключна підстановка.

Приклад 2.3. Нехай нормальний алгоритм заданий такими формулами підстановок (граф-схема цього алгоритму зображена на рис. 2.2):

1) $xy \to yx$;

2) $yx \to \Lambda$.

Тоді на вхідному слові $P = xxyyxyx$ одержимо

$$xxyyxyx \overset{1}{\Rightarrow} xyxyxyx \overset{1}{\Rightarrow} yxxyxyx \overset{1}{\Rightarrow} yxxyxxyx \overset{1}{\Rightarrow} yyxxxyx \overset{1}{\Rightarrow}$$

$$yyxxyxx \overset{1}{\Rightarrow} yyxyxxx \overset{2}{\Rightarrow} yyyxxxx \overset{2}{\Rightarrow} yyxxx \overset{2}{\Rightarrow} yxx \Rightarrow x.$$

Тобто $Q = x$ і цей алгоритм реалізує віднімання від кількості x кількість y.

2.3.2. *Принцип нормалізації*

Кожна алгоритмічна система повинна задовольняти дві вимоги: бути математично строгою та універсальною.

Математичної строгості досягають використанням певного математичного апарату (у цьому випадку розпізнавання входження і підстановка). Тому побудована А. Марковим теорія нормальних алгоритмів повністю задовольняє першу вимогу.

Універсальність теорії нормальних алгоритмів формулюють у вигляді принципу нормалізації, який є однією з головних гіпотез (тез) теорії алгоритмів.

А. Марков сформулював і довів необхідну умову універсальності алгоритмічної системи Маркова.

Теорема. *Для того, щоб реалізувати в схемах нормальних алгоритмів довільний алгоритм $A = \langle \varphi, \mathcal{P} \rangle$, необхідно, щоб у системі нормальних алгоритмів були як звичайні, так і заключні підстановки.*

Доведення (від супротивного). Припустимо, що заключних підстановок у системі немає. Тоді довільний алгоритм A може закінчити роботу лише у випадку, коли жодна з підстановок не є

застосовною. Звідси випливає, що повторна дія алгоритму A на слово $A(P)$, отримане внаслідок його застосування до довільного вхідного слова P, вже не може змінити цього слова. Отже, для A справджується тотожне співвідношення

$$A(A(P))=A(P). \qquad (2.1)$$

Проте є цілий клас алгоритмів $<\varphi,\mathcal{P}>$, які не задовольняють (2.1). Наприклад, алгоритм B, який приписує перед словом деяку фіксовану букву x:

$$P \to xP.$$

Справді,

$$B(B(P))=B(xP)=xxP.$$

Тобто алгоритм B не може бути реалізований нормальним алгоритмом без заключних підстановок. Проте B реалізують схемою з однієї заключної підстановки $\Lambda \to \cdot x$, бо порожнє слово входить першим зліва у кожне слово P.

Отже, щоб описати в системі Маркова довільний алгоритм, у ній повинна бути заключна підстановка.

Доведемо необхідність звичайних підстановок. Припустимо, що в системі Маркова є лише заключні підстановки. Тоді схема довільного алгоритму матиме вигляд

$$\begin{array}{l} P_1 \to \cdot Q_1 \\ \ldots \\ P_n \to \cdot Q_n \end{array} \qquad (2.2)$$

У цьому випадку під час обробки вхідного слова P виконається лише одна підстановка, і алгоритм закінчить роботу з підсумковим словом $A(P)$.

Позначимо довжини слів з i-ї підстановки схеми (2.2) через $|P_i|$ та $|Q_i|$ і знайдемо для кожної підстановки модуль різниці довжин слів

$$l_i = ||P_i|-|Q_i|| \qquad i=\overline{1,n}.$$

Нехай $L=\max\limits_{1\le i\le n} l_i$. З урахуванням скінченної кількості підстановок у схемі таке L існує для довільної схеми (2.2). Тому для схеми (2.2) справджується співвідношення

$$\big||P|-|A(P)|\big| \le L, \qquad (2.3)$$

де L не залежить від довжини вхідного слова P.

Проте є клас алгоритмів, для яких співвідношення (2.3) не виконується.

Наприклад, алгоритм B подвоєння слів

$$B(P)=PP,$$

для якого $||P|-|B(P)||$ визначене довжиною слова P.

Отже, для того, щоб у системі нормальних алгоритмів описати довільний алгоритм, необхідна наявність звичайних підстановок. Теорему доведено.

А. Марков запропонував тезу, згідно з якою наявність у системі нормальних алгоритмів заданих двох видів підстановок є не тільки необхідною, а й достатньою умовою універсальності системи, що приводить до такої теореми.

Принцип нормалізації. *Для будь-якого алгоритму $A=\langle\varphi,\mathcal{P}\rangle$ в довільному алфавіті X можна побудувати еквівалентний йому нормальний алгоритм над алфавітом X.*

Поняття нормального алгоритму над алфавітом X означає таке. Інколи не вдається побудувати нормального алгоритму, еквівалентного заданому в алфавіті X, використовуючи в підстановках лише букви алфавіту X. Однак потрібний нормальний алгоритм можна побудувати, розширюючи алфавіт X додаванням певної кількості нових букв, але залишаючи область визначення алгоритму попередньою.

Перехід від інших способів опису алгоритмів до еквівалентних нормальних алгоритмів називають ***зображенням у нормальній формі***, або ***нормалізацією***.

*Алгоритм $A=\langle\varphi,\mathcal{P}\rangle$ в алфавіті X називають **нормалізованим**, якщо можна побудувати еквівалентний йому нормальний*

алгоритм над алфавітом X. В іншому випадку алгоритм називають ***ненормалізованим***.

На підставі цього означення принцип нормалізації можна сформулювати так: **усі алгоритми є нормалізовані**.

Принцип нормалізації уточнює поняття алгоритму у формі нормальних алгоритмів. Він визначає відповідність між інтуїтивним поняттям алгоритму і точним математичним поняттям нормального алгоритму.

Водночас суть принципу нормалізації не вичерпується лише уточненням поняття алгоритму. Цей принцип стверджує також правильність такого уточнення, тобто можливість реалізувати кожен алгоритмічний процес у вигляді схеми нормального алгоритму. Згідно з принципом нормалізації, кожного разу, коли виникає питання про можливість алгоритмізації того чи іншого процесу, його можна замінити питанням про можливості реалізації цього процесу у вигляді нормального алгоритму.

Принцип нормалізації не може бути строго математично доведений або заперечений, оскільки поняття довільного алгоритму, що є в його формулюванні, не формалізоване строго математично. Тому принцип нормалізації не є математичною теоремою, а це гіпотеза, теза, яку підтверджують експериментально.

Наведемо головні факти, що підтверджують принцип нормалізації.

1. Правильність цього принципу ґрунтується на тому, що всі відомі сьогодні алгоритми є нормалізовані.
2. Усі відомі способи композиції алгоритмів, які дають змогу будувати нові алгоритми з уже відомих, не виводять за межі класу нормальних алгоритмів. Іншими словами, якщо вихідні алгоритми вже нормалізовані, то нормалізованою буде і композиція цих алгоритмів.
3. Можливим аргументом на користь принципу нормалізації є еквівалентність системи нормальних алгоритмів усім іншим алгоритмічним системам.

4. Усі спеціальні спроби побудови алгоритмів найбільш загального вигляду не вивели за межі класу нормалізованих алгоритмів.

Ці обґрунтування не повністю виключають можливість спростувати принцип нормалізації в майбутньому (побудовою прикладу ненормалізованого алгоритму). Принаймні нормалізовані алгоритми охоплюють значну частину алгоритмів (якщо не всі), тому систему нормальних алгоритмів прийнято вважати практично універсальною алгоритмічною системою.

2.3.3. Універсальний нормальний алгоритм

Важливе значення для нормальних алгоритмів, як і для кожної іншої алгоритмічної системи, має задача побудови універсального алгоритму.

Універсальним нормальним алгоритмом (УНА) називають алгоритм, здатний виконувати роботу довільного нормального алгоритму A.

В основі універсального нормального алгоритму є такий самий метод, як і в сучасних комп'ютерах. А саме: УНА отримує інформацію про схему конкретного алгоритму A, а також про вхідне слово P і виконує над P підстановки згідно з заданою схемою.

Задачу побудови УНА можна розв'язати різними способами. Розглянемо один з них.

Кожен конкретний нормальний алгоритм A має свій алфавіт. УНА, як нормальний алгоритм, теж має один фіксований алфавіт, назвемо його стандартним. Для того, щоб УНА міг сприймати схеми довільних нормальних алгоритмів, ці схеми повинні бути закодовані в стандартному алфавіті УНА.

Зафіксуємо деякий стандартний алфавіт S, наприклад двійковий $S=\{0,1\}$. Букви будь-якого нестандартного алфавіту $\mathcal{A}$ будемо кодувати так: усі букви нумерують натуральними числами, після чого i-й букві ставлять у відповідність двійковий код з i одиниць, обмежених нулями, наприклад $\underbrace{0111...10}_{i\ разів}$. Якщо алфавіт

A має n букв, то вводять ще додатково чотири букви для позначення знаків, використовуваних під час написання схем нормальних алгоритмів (стрілочка, крапка, знак розділення між підстановками), а також спеціального кінцевого знака, який ставлять на початку і в кінці схеми алгоритму.

Після записування схеми алгоритму одним словом і кодування букв цього слова заданим способом отримаємо слово в стандартному алфавіті, яке називають зображенням заданого алгоритму – A^{cod}.

Приклад 2.4. Нехай схема алгоритму A має такий вигляд:

$$xy \to yx;$$
$$y \to \cdot$$

Запишемо її у вигляді слова

$$|xy \to yx; y \to \cdot|$$

в алфавіті $\{x, y, \to, ;, \cdot, |\}$. Нумеруючи букви цього алфавіту зліва направо і виписуючи в слові їхні двійкові коди, отримаємо

$$A^{cod} = 060010020030020010040020030050060,$$

де для скорочення замість виписування підряд n одиниць виписано саме число n.

Поряд із зображенням алгоритму A засобами аналогічного кодування в стандартному алфавіті S можна отримати зображення P^{cod} довільного вхідного слова P алгоритму A.

Наприклад, нехай

$$P=xyxxy,$$

тоді

$$P^{cod} = 010020010010020.$$

А. Марков довів таку теорему про універсальний нормаль-ний алгоритм.

Теорема. *Існує такий нормальний алгоритм U, який називають універсальним, що для будь-якого нормального алгоритму A і будь-якого вхідного слова P з області визначення A перетворює слово $A^{cod}P^{cod}$, отримане конкатенацією зображень A і P, у слово Q^{cod}, де $Q = A(P)$:*

$$A^{cod}P^{cod} \xrightarrow{U} A(P)^{cod}.$$

Якщо ж P не входить в область визначення A, то універсальний алгоритм U також буде незастосовний до слова $A^{cod}P^{cod}$.

Ця теорема має фундаментальне значення для кібернетики, оскільки з неї випливає принципова можливість побудови машини, яка виконуватиме роботу будь-якого нормального алгоритму, а тому, з огляду на принцип нормалізації, – роботу довільного алгоритму $<\varphi, \mathcal{P}>$. Для цього в машину достатньо вкласти програму, тобто зображення того нормального алгоритму, роботу якого машина повинна виконати.

Водночас фактичне виконання нормалізації навіть для порівняно простих алгоритмів (наприклад, для алгоритму множення двох чисел) є досить непростою задачею, тому програмування для машини, що моделює універсальний нормальний алгоритм, було б дуже громіздким і неефективним. Тому на практиці універсальні машини будують на підставі інших алгоритмічних систем.

2.3.4. Асоціативне числення слів

Істотна різниця між поняттям нормального алгоритму та асоціативним численням слів полягає у тому, що в численні вводять допустимі операції підстановок без будь-яких обмежень на порядок їхнього виконання.

Асоціативним численням *називають сукупність усіх слів у заданому абстрактному алфавіті з деякою скінченною системою допустимих підстановок.*

Асоціативне числення задають алфавітом і системою допустимих підстановок. Це так звані неорієнтовані підстановки, які зображають так:

$$X \leftrightarrow Y,$$

де X, Y – слова в заданому алфавіті, причому знак $\leftrightarrow$ не входить у цей алфавіт.

Застосування підстановки $X \leftrightarrow Y$ до заданого слова P полягає в тому, що будь-яке входження слова X можна замінити на слово Y або будь-яке входження слова Y можна замінити словом X.

Приклад 2.5. Нехай асоціативне числення задане алфавітом $\{a, b, c\}$ і підстановкою

$$ab \leftrightarrow bcb.$$

Тоді для слова *abcbcbab* цю підстановку можна застосувати чотирма способами:

$\underline{ab}cbcbab \leftrightarrow \underline{bcb}cbcbab$,

$a\underline{bcb}cbab \leftrightarrow a\underline{ab}cbab$,

$abc\underline{bcb}ab \leftrightarrow abc\underline{ab}ab$,

$abcbcb\underline{ab} \leftrightarrow abcbcb\underline{bcb}$.

Якщо слово X є результатом одного застосування допустимої підстановки до слова Y, то очевидно, що слово Y теж є результатом застосування цієї ж підстановки до слова X; такі два слова називають суміжними словами.

Послідовність попарно суміжних слів $X_1, X_2, ..., X_n$ називають ***дедуктивним ланцюжком****, що зв'язує слова X_1 і X_n.*

Два слова X і Y називають ***еквівалентними*** *($X \sim Y$), якщо існує дедуктивний ланцюжок, який зв'язує ці слова.*

Для кожного асоціативного числення виникає **проблема еквівалентності слів:** *для довільних двох слів заданого асоціативного числення визначити, еквівалентні вони чи ні.*

Ця проблема є алгоритмічно нерозв'язною.

Проблема еквівалентності слів в асоціативному численні має важливе теоретичне і практичне значення. Наприклад, проблема розв'язності в булевій алгебрі є проблемою еквівалентності слів у деякому асоціативному численні, а саме: алфавіт цього асоціативного числення складається з букв, якими позначають змінні висловлювання, і знаків операцій булевої алгебри. Словами є формули булевої алгебри. Система тотожностей, що визначає

булеву алгебру, є системою допустимих підстановок. Проблему еквівалентності слів у цьому численні розв'язують алгоритмом зведення формул булевої алгебри до нормальної форми.

2.3.5. *Завдання для самостійної роботи*

1. Побудувати граф-схему нормального алгоритму, заданого підстановками

$$yyx \rightarrow y;$$

$$xx \rightarrow y;$$

$$yyy \rightarrow \cdot x,$$

в алфавіті $\mathcal{A} = \{x,y\}$.

Розглянути приклади дедуктивних ланцюжків, задаючи початкове слово довжиною не менше трьох символів.

2. Побудувати нормальні алгоритми Маркова, які реалізують віднімання A-B та додавання A+B, де значення A і B є натуральними числами, поданими як ланцюжки символів '1'. Перевірити роботу алгоритмів на декількох прикладах.

3. Побудувати нормальний алгоритм, який видаляє всі входження, крім першого, букви '*a*' з рядка символів, побудованого в алфавіті $\mathcal{A} = \{a, b, c\}$.

4. Скласти нормальний алгоритм, який дає змогу видаляти повторні пропуски у словах з алфавіту $\mathcal{A} = \{a, b, c, ' '\}$.

5. Скласти нормальний алгоритм, який дає змогу обчислити значення довжини слова в алфавіті $\mathcal{A}=\{a, b, c\}$ у десятковій системі числення.

6. Скласти нормальні алгоритми, які дають змогу закодувати та розкодувати у двійковому алфавіті слова, побудовані в алфавіті $\mathcal{A} = \{a, b, c\}$ (наприклад, так: $a \rightarrow 101$; $b \rightarrow 1001$).

2.4. Рекурсивні функції

Історично першою алгоритмічною системою була система, що ґрунтується на використанні конструктивно визначених арифметичних (цілочислових) функцій, які назвали **рекурсивними функціями.**

2.4.1. Зведення довільних алгоритмів до числових функцій. Обчислювані функції

Нехай задано деякий алгоритм A, який можна застосувати до цілого класу задач, тобто до ряду допустимих вхідних даних – умов задач. Пронумеруємо ці умови цілими числами $n_1, n_2, ..., n_k, ...$. Результати роботи алгоритму також пронумеруємо цілими додатними числами $m_1, m_2, ..., m_k, ...$ Тоді

$$m_i = A(n_j),$$

та оскільки m_i та n_j – числа, то алгоритм A визначає деяку числову функцію φ

$$m_i = \varphi(n_j), (i=1,2,\ldots, j=1,2,\ldots)$$

де $\varphi: N \to N$.

Отже, виконання довільного алгоритму A є еквівалентним обчисленню значень деякої числової функції φ.

Числові функції, значення яких можна обчислити за допомогою деякого (єдиного для заданої функції) алгоритму, називають ***обчислюваними функціями***.

Оскільки поняття алгоритму використане тут в інтуїтивному сенсі, то і поняття обчислюваної функції є інтуїтивним.

Формальним еквівалентом цього поняття є поняття рекурсивної функції, введене в працях К. Геделя, А. Черча, С.-К. Кліні у 30-х роках XX ст.

2.4.2. Найпростіші функції

Позначимо $N \equiv Z^+$ – множину всіх натуральних чисел, а $N^{(n)} = \{< x_1,...,x_n > | x_i \in N\}$ – множину усіх можливих n натуральних чисел.

Числову функцію $\varphi: N \to N$ *називають* ***функцією наступності****, якщо* $\varphi(x) = x+1$.

Числову функцію $\varphi: N^{(n)} \to N$ *називають* ***нуль-функцією****, якщо* $\varphi(x_1,...,x_n) = 0$.

Числову функцію $\varphi_i: N^{(n)} \to N$ *називають* ***функцією вибору аргументів****, якщо вона повторює значення свого і-го аргумента:* $\varphi_i(x_1,...,x_n) = x_i$, $(1 \le i \le n)$.

Позначимо функцію наступності через $S^1(x)$, нуль-функцію — $O^n(x_1,...,x_n)$ і функцію вибору аргументів — через $I_i^n(x_1,...,x_n)$.

Функції $S^1(x)$, $O^n(x_1,...,x_n)$, $I_i^n(x_1,...,x_n)$ *називають* ***найпростішими***.

Найпростіші функції є всюди визначені.

Приклад 2.6. Найпростішими є такі функції:

$S^1(5) = 6$,

$O^4(3,6,2,1) = 0$,

$I_2^3(4,3,8) = 3$.

2.4.3. Головні оператори

Операції над функціями називають операторами. Розглянемо три головні оператори, які використовують під час побудови частково-рекурсивних функцій.

1. Оператор суперпозиції. Нехай A,B,C – довільні множини, на яких визначено n часткових функцій (тобто не

обов'язково визначено для всіх значень аргументів) $f_i^m : A \to B, \left(i = \overline{1,n}\right)$ від однієї й тієї ж кількості змінних m:

$$f_1^m(x_1,...,x_m),$$
$$f_2^m(x_1,...,x_m),$$
$$...$$
$$f_n^m(x_1,...,x_m).$$

Нехай також задана часткова n-місна функція $f^n : B \to C$.

Розглянемо часткову m-місну функцію $g^m : A \to C$, таку що

$$g^m(x_1,...,x_m) = f^n(f_1^m(x_1,...,x_m),...,f_n^m(x_1,...,x_m)) \qquad (2.4)$$

для довільних $x_1,...,x_m$ з A.

Оператор, за допомогою якого з функцій f^n, $f_1^m, f_2^m,..., f_n^m$ утворюється функція g^m, що задовольняє рівність (2.4), називають ***оператором суперпозиції***.

Позначимо оператор суперпозиції $\mathbb{S}^{n+1}$, де $n+1$ — кількість функцій.

Приклад 2.7. А. Нехай $h(x) = 0$, $f(x) = x+1$, тоді

$$\mathbb{S}^2(f^1,h^1) = f(h(x)) = 1;$$
$$\mathbb{S}^2(f^1,f^1) = f(f(x)) = (x+1)+1 = x+2.$$

Б. Суперпозицією трьох функцій вибору аргументів I_2^2, I_1^3, I_2^3 є така функція:

$$\mathbb{S}^3(I_2^2, I_1^3, I_2^3) = I_2^2(x_1,x_2) = x_2.$$

Оператор $\mathbb{S}^{n+1}$ визначений тоді й тільки тоді, коли функції $f_1,..., f_n$ мають однакову кількість аргументів, а f є n-місною.

У разі суперпозиції функцій з різною кількістю аргументів за допомогою функції вибору аргументів уводять фіктивні змінні.

Наприклад, функцію двох змінних $\varphi(x_1,x_2)$ можна зобразити у вигляді функції трьох змінних, з яких одна фіктивна:

$$\varphi(x_1,x_2) = \varphi(I_1^3(x_1,x_2,x_3), I_2^3(x_1,x_2,x_3)) = \psi(x_1,x_2,x_3).$$

Нехай F^k – множина всіх часткових k-місних функцій $f^k : N^{(k)} \to N$. Тоді оператор $\mathbb{S}^{n+1}$ є всюди визначеною $n+1$-місною функцією

$$\mathbb{S}^{n+1} : F^n \times \underbrace{F^m \times ... \times F^m}_{n \text{ разів}} \to F^m.$$

Область визначення цієї функції є декартовий добуток:

$$F^n \times F^m \times ... \times F^m = \{< f^n, f_1^m,..., f_n^m > \mid f^n \in F^n,\ f_i^m \in F^m;\ i = \overline{1,n}\}.$$

Якщо множину всіх часткових числових функцій від довільної кількості аргументів позначити через F, то оператор $\mathbb{S}^{n+1}$ можна розглядати як часткову функцію

$$\mathbb{S}^{n+1} :\ F \to F.$$

Тобто оператор суперпозиції не погіршує області допустимих значень, а в разі застосування до часткових функцій ($F_{\text{ч.ф}}$) результатом є часткова функція:

$$\mathbb{S}^{n+1} : F_{\text{ч.ф}} \to F_{\text{ч.ф}}$$

Часткові функції, які можна отримати за допомогою оператора суперпозиції з функцій $f_1^{n_1},..., f_k^{n_k}$ і найпростіших функцій I_m^n $(m,n = 1,2,...)$, називають ***елементарними відносно функцій*** $f_1^{n_1},..., f_k^{n_k}$.

Приклад 2.8. Функція $x_1 \times x_2 + x_3$ є елементарною відносно функцій $+$ та $\times$, оскільки

$$x_1 \times x_2 + x_3 = \mathbb{S}^3(+, \mathbb{S}^3(\times, I_1^3, I_2^3), I_3^3).$$

2. Оператор примітивної рекурсії. Нехай задані часткові числові функції

$$g^n : N^{(n)} \to N;$$

$$h^{n+2} : N^{(n+2)} \to N.$$

Розглянемо часткову функцію $f^{n+1} : N^{(n+1)} \to N$, визначену так:

$$\begin{cases} f(x_1,...,x_n,0) = g(x_1,...,x_n), \\ f(x_1,...,x_n,y+1) = h(x_1,...,x_n,y,f(x_1,...,x_n,y)), \end{cases} \quad (2.5)$$

для довільних натуральних значень $x_1,...,x_n,y$.

У випадку одномісної функції $f^1 : N \to N$:

$$\begin{cases} f(0) = a, \\ f(y+1) = h(y,f(y)), \end{cases} \quad (2.6)$$

де $a \in N$.

Оператор, який за формулами (2.5) або (2.6) з функцій g і h дає змогу побудувати функцію f, називають ***оператором примітивної рекурсії*** *і позначають* $\mathbb{R}$:

$$f = \mathbb{R}(g,h).$$

Усі три функції g, h, f є числовими (з однією й тією ж областю визначення і значення), тому їхня суперпозиція завжди можлива, що зумовлює існування f^{n+1} для будь-якої пари g^n, h^{n+2}.

Так само, як і у випадку оператора суперпозиції, оператор примітивної рекурсії задає часткову функцію $\mathbb{R} : F_{\text{ч.р}} \to F_{\text{ч.р}}$.

3. Оператор мінімізації. Нехай задана часткова функція $f^n : N^{(n)} \to N$. Припустимо, що існує деяка механічна процедура для обчислення функції f^n, причому значення f^n невизначене лише в тому випадку, коли ця процедура працює безконечно довго і не видає жодного результату.

Зафіксуємо перші n-1 аргументів цієї функції і розглянемо рівняння

$$f(x_1,...,x_{n-1},y) = x_n. \quad (2.7)$$

Щоб знайти розв'язок y рівняння (2.7), будемо послідовно обчислювати значення $f(x_1,...,x_{n-1},y)$ при $y = 0, 1, 2,...$ і порівнювати результати з x_n.

Найменше значення $y = a$, для якого виконується рівність

$$f(x_1,...,x_{n-1},a) = x_n, \quad (2.8)$$

позначимо так:

$$\mu_y(f(x_1,...,x_{n-1},y) = x_n). \quad (2.9)$$

Процес знаходження значення виразу (2.9) буде безмежним у таких трьох випадках:

1) значення $f(x_1,...,x_{n-1},0)$ невизначене;

2) значення $f(x_1,...,x_{n-1},y)$ для $y = 0,1,2,...,k$ визначені, але не дорівнюють x_n, а для $y = k+1$ функція f не визначена;

3) значення $f(x_1,...,x_{n-1},y)$ визначені для всіх $y = 0,1,2,...$, але відмінні від x_n (тобто рівняння (2.8) натуральних розв'язків не має).

У цих трьох випадках значення виразу (2.9) вважають невизначеним. У всіх інших випадках процес буде скінченним і дає найменший розв'язок $y = a$ рівняння (2.8), який є значенням виразу (2.9).

Приклад 2.9. Розглянемо оператор мінімізації:

$$\mu_z(y+z = x) = x - y,$$
$$\mu_x(sg\ x = 1) = 1.$$

Значення виразу

$$\mu_y(y(y-(x+1)) = 0) \quad (2.10)$$

є невизначеним через невизначеність на множині N виразу $0(0-(x+1))$. Водночас рівняння

$$y(y-(x+1))) = 0$$

має розв'язок $y = x+1$, проте він не збігається зі значенням виразу (2.10).

Цей приклад засвідчує таке: якщо функція $f(x_1,...,x_{n-1},y)$ є частковою, то вираз $\mu_y(f(x_1,...,x_{n-1},y)=x_n)$, строго кажучи, не є найменшим розв'язком рівняння (2.8). Якщо ж функція $f(x_1,...,x_{n-1},y)$ всюди визначена і рівняння (2.8) має розв'язок, то $\mu_y(f(x_1,...,x_{n-1},y)=x_n)$ є найменшим розв'язком цього рівняння.

Значення виразу (2.9) за заданої функції f залежить від вибору значень для параметрів $x_1,...x_{n-1},x_n$. Тому значення виразу $\mu_y(f(x_1,...,x_{n-1},y)=x_n)$ є частковою функцією від аргументів $x_1,...,x_n$.

Оператор, за допомогою якого з функції $f(x_1,...,x_n)$ утворюється функція $\mu_y(f(x_1,...,x_{n-1},y)=x_n)$, називають ***оператором мінімізації (або найменшого кореня).***

Оператор мінімізації позначають $\mathbb{M}$, тоді

$$\mu_y(f(x_1,...,x_{n-1},y)=x_n)=\mathbb{M}f.$$

У загальному випадку функція $\mathbb{M}f$ є частковою.

2.4.4. Примітивно-рекурсивні функції. Частково-рекурсивні функції. Теза Черча

Нехай задана сукупність часткових функцій $\sigma=\{f_1^{n_1},...,f_k^{n_k}\}$.

Функції, які отримують з функцій системи σ і найпростіших функцій $S^1(x), O^1(x), I_m^n$ із застосуванням скінченної кількості операторів суперпозиції та примітивної рекурсії, називають ***примітивно-рекурсивними відносно системи*** σ*:*

$$\{S^1,O^1,I_m^n,\sigma\}\xrightarrow{S^{n+1},\mathbb{R}} f.$$

Функцію f називають ***примітивно-рекурсивною****, якщо її можна отримати із застосуванням скінченної кількості операторів суперпозиції і примітивної рекурсії на підставі лише найпростіших функцій S^1,O^1,I_m^n:*

$$\{S^1,O^1,I_m^n\}\xrightarrow{S^{n+1},\mathbb{R}} f.$$

У цих двох означеннях передбачена можливість виконання всіх допустимих операцій $(S^1,O^1,I_m^n,S^{n+1},\mathbb{R})$ у будь-якій послідовності та довільну скінченну кількість разів.

Усі примітивно-рекурсивні функції є всюди визначеними. Клас усіх примітивно-рекурсивних функцій позначимо $K_{\text{п.р}}$.

Часткову функцію f називають ***частково-рекурсивною відносно*** σ*, якщо її можна отримати з функцій системи σ і найпростіших функцій із застосуванням скінченної кількості операторів суперпозиції, примітивної рекурсії та мінімізації:*

$$\{S^1,O^1,I_m^n,\sigma\}\xrightarrow{S^{n+1},\mathbb{R},\mathbb{M}} f.$$

Часткову функцію f називають ***частково-рекурсивною****, якщо її можна отримати з найпростіших функцій із застосуванням скінченної кількості операторів суперпозиції, примітивної рекурсії та мінімізації:*

$$\{S^1,O^1,I_m^n\}\xrightarrow{S^{n+1},\mathbb{R},\mathbb{M}} f.$$

Клас частково-рекурсивних функції $(K_{\text{ч.р}})$ – це найзагальніший клас конструктивно визначених арифметичних функцій.

Аналогічно, як і для примітивно-рекурсивних функцій, в останніх двох означеннях допустимі операції $(S^1,O^1,I_m^n,S^{n+1},\mathbb{R},\mathbb{M})$ можна застосовувати в довільній послідовності та довільну скінченну кількість разів.

Поняття частково-рекурсивної функції – одне з головних понять теорії алгоритмів. Значення його полягає в такому.

1. Кожну задану частково-рекурсивну функцію можна обчислити за допомогою певної процедури механічного характеру (алгоритму).

2. Які б класи точно визначених алгоритмів не будували, у всіх випадках неодмінно виявляли, що числові функції, обчислювані за алгоритмами з цих класів, є частково-рекурсивними.

Ці факти відображені в гіпотезі Черча.

Теза Черча. *Клас алгоритмічно (або машинно) обчислюваних часткових числових функцій збігається з класом усіх частково-рекурсивних функцій:* $K_A \cong K_{ч.р}$.

Як уже зазначено, ця теза, як і теза Маркова, у принципі не може бути доведена, оскільки в її формулюванні використане інтуїтивне поняття алгоритму.

Водночас у теорії алгоритмів строго математично доведено таку **теорему**.

Алгоритм тоді і тільки тоді може бути нормалізований, коли він може бути реалізований за допомогою частково-рекурсивних функцій:

$$K_A \cong K_{ч.р}, \qquad K_Н = K_{ч.р}.$$

Тут $K_Н$ – клас нормальних алгоритмів.

2.4.5. *Загальнорекурсивні функції. Теза Тьюрінга*

Розглянемо оператор слабкої мінімізації, який ставить у відповідність довільній заданій функції f часткову функцію:

$$M' f = \begin{cases} Mf, & \text{якщо } Mf \text{ всюди визначена;} \\ \text{не визначена,} & \text{якщо } Mf \text{ визначена не всюди.} \end{cases}$$

Функції, які можна отримати з найпростіших функцій S^1, O^1, I_m^n *за допомогою оператора примітивної рекурсії, суперпозиції та слабкої мінімізації, називають* ***загальнорекурсивними***. Клас усіх загальнорекурсивних функцій позначимо $K_{з.р}$.

Якщо оператори S, R, M' застосовують до всюди визначених функцій, то вони в підсумку або нічого не дають, або знову дають функції, всюди визначені. Тому всі загальнорекурсивні функції є всюди визначені.

З іншого боку, якщо результат оператора M' визначений, то він збігається з результатом звичайного оператора M. Тому всі загальнорекурсивні функції є всюди визначеними частково-рекурсивними функціями.

У теорії алгоритмів доведено й обернене твердження: кожна всюди визначена частково-рекурсивна функція є загальнорекурсивною. Отже, клас всюди визначених частково-рекурсивних функцій збігається з класом загальнорекурсивних функцій. Клас примітивно-рекурсивних функцій вужчий, ніж клас загальнорекурсивних.

Практично поняттям частково-рекурсивних функцій користуються для доведення існування чи неіснування алгоритму розв'язку задачі. Використання ж частково-рекурсивних функцій для зображення того чи іншого конкретного алгоритму практично недоцільне через складність такого процесу алгоритмізації.

Теза Черча дає алгоритмічне пояснення поняттю частково-рекурсивної функції. Для поняття часткової рекурсивності функції відносно системи σ алгоритмічне пояснення вперше дав А. Тьюрінг.

Функцію f *називають* ***алгоритмічно обчислюваною відносно деякої системи функцій*** $\sigma = \{f_1, ..., f_k\}$, *якщо існує алгоритм, який дає змогу обчислити значення функції* f *за умови, що якимось чином можна знайти ті значення функцій* $f_1, ..., f_k$, *які потрібні для алгоритму.*

Тут уважають, що не існує алгоритмів обчислення функцій з σ і обчислити будь-яке значення кожної з них є математичною проблемою. Якщо ж значення всіх функцій з σ можуть бути обчислені за допомогою алгоритмів, то функція f з алгоритмічно обчислюваної відносно σ стає обчислюваною.

Поняття відносного алгоритму, як і поняття звичайного алгоритму, є інтуїтивним і його можна уточнювати різними

способами. Проте за всіх фактично випробуваних уточнень виявилось, що відносно обчислювані функції є відносно частково-рекурсивними.

Узагальненням тези Черча є така теза.

Теза Тьюрінга. *Клас функцій, алгоритмічно обчислюваних відносно деякого класу функцій σ, збігається з класом частково-рекурсивних функцій відносно σ.*

2.4.6. Універсальні рекурсивні функції

Як підсумок поняття класів примітивно-рекурсивних, частково-рекурсивних та загальнорекурсивних функцій можна записати таке співвідношення:

$$K_{\text{п.р}} \subset K_{\text{з.р}} \subset K_{\text{ч.р}}.$$

У кожній універсальній алгоритмічній системі повинен існувати універсальний алгоритм, еквівалентний довільному, наперед заданому алгоритму. Як доведено раніше, для системи нормальних алгоритмів Маркова такий універсальний алгоритм існує. Розглянемо питання про існування універсальних функцій для класів $K_{\text{п.р}}$, $K_{\text{з.р}}$, $K_{\text{ч.р}}$.

Часткову $(n+1)$-місну функцію $U(x_0,...,x_n)$ називають ***універсальною*** *для сім'ї σ всіх n-місних часткових функцій, якщо виконуються такі умови:*

1) *для кожного фіксованого числа i n-місна функція $U(i,x_1,...,x_n)$ належить σ;*

2) *для кожної функції $f(x_1,...,x_n)$ з σ існує таке число i, що для всіх $x_1,...,x_n$*

$$U(i,x_1,...,x_n) = f(x_1,...,x_n).$$

Іншими словами, функція U є універсальною для сім'ї σ, якщо всі функції з σ можна розташувати в такій послідовності:

$$U(0,x_1,...,x_n), U(1,x_1,...,x_n), ..., U(i,x_1,...,x_n), ...$$

Число i називають *номером* функції U.

У теорії рекурсивних функцій доведено такі три важливі теореми.

Теорема 1. *Система всіх n-місних примітивно-рекурсивних функцій не містить примітивно-рекурсивної універсальної функції* (n=1,2,...).

Теорема 2. *Система всіх n-місних загальнорекурсивних функцій не містить загальнорекурсивної універсальної функції* (n=1,2,...).

Теорема 3. *Для кожного n=1,2,... клас усіх n-місних примітивно-рекурсивних функцій містить $(n+1)$-місну загальнорекурсивну універсальну функцію.*

Наведена нижче теорема є головним результатом теорії частково-рекурсивних функцій і аналогом теореми Маркова про універсальний нормальний алгоритм.

Теорема 4. *Для кожного n=1,2,... існує частково-рекурсивна функція $U^{n+1}(x_0,x_1,...,x_n)$, універсальна для класу всіх n-місних частково-рекурсивних функцій.*

2.4.7. Рекурсивні та рекурсивно-перелічувані множини

Рекурсивні та рекурсивно-перелічувані множини відіграють важливу роль не тільки в теорії рекурсивних функцій, а й у низці інших дисциплін, що становить теоретичний фундамент автоматичного опрацювання даних.

Нехай M – деяка підмножина множини натуральних чисел.

Характеристичною функцією *множини M називають одномісну функцію таку, що*

$$\chi_M(x) = \begin{cases} 0, & \text{якщо } x \in M, \\ 1, & \text{якщо } x \notin M. \end{cases}$$

Частковою характеристичною функцією *множини M називають таку одномісну функцію:*

$$\tilde{\chi}_M(x)=\begin{cases}0, & \text{якщо } x\in M,\\ \text{не визначена}, & \text{якщо } x\notin M.\end{cases}$$

Наприклад, для порожньої множини M $\chi_M(x)\equiv 1$, а $\tilde{\chi}_M(x)$ є ніде не визначеною функцією.

Функції $\chi_M(x)$ та $\tilde{\chi}_M(x)$ збігаються лише для множини N усіх натуральних чисел, а для будь-якої іншої множини вони різні.

Множину M називають ***рекурсивною****, якщо її характеристична функція $\chi_M(x)$ є частково-рекурсивною.*

Згідно з тезою Черча, часткова рекурсивність функції означає існування алгоритму для її обчислення. Отже, рекурсивність множини M означає існування алгоритму, який дає змогу для довільного об'єкта x визначити, чи належить він до множини M, чи ні.

Проблемою входження для числової множини M *називають задачу відшукання алгоритму, який за стандартним записом довільного натурального числа m дає змогу з'ясувати, чи входить m в M, чи ні* (тобто дає змогу обчислити значення характеристичної функції $\chi_M(m)$ множини M).

Отже, рекурсивні множини – це множини з алгоритмічно розв'язною проблемою входження.

Прикладами рекурсивних множин є множини всіх парних чисел, усіх простих чисел і багато інших сукупностей чисел.

Доповнення рекурсивної множини, а також переріз і об'єднання довільної скінченної кількості рекурсивних множин є рекурсивною множиною.

Множину M називають ***рекурсивно-перелічуваною****, якщо вона або порожня, або збігається з сукупністю всіх значень деякої загальнорекурсивної функції $\varphi(t)$, яка набуває різних значень для різних аргументів t.*

Іншими словами, між будь-якою рекурсивно-перелічуваною множиною M і натуральним рядом чисел N існує взаємно однозначна відповідність, задана деякою загальнорекурсивною функцією $\varphi(t)$:

$$\varphi:\ N\to M.$$

Функцію $\varphi(t)$ можна розглядати як генератор елементів множини M.

Серед важливих результатів, визначених для рекурсивно-перелічуваних множин, які використовують під час доведення нерозв'язності багатьох проблем, назвемо такі.

1. *Об'єднання і переріз довільної скінченної системи рекурсивно-перелічуваних множин є рекурсивно-перелічуваними множинами.*

2. *Множина є рекурсивною тоді й тільки тоді, коли вона та її доповнення є рекурсивно-перелічуваними.*

3. *Для того, щоб часткова функція f була частково-рекурсивною, необхідно і досить, щоб її графік (тобто множина пар вигляду $\{x, f(x)\}$) був рекурсивно-перелічуваний.*

2.4.8. Завдання для самостійної роботи

1. Побудувати суперпозицію трьох функцій вибору аргументів I_2^2, I_1^3, I_2^3.

2. Довести елементарність функцій $g=x_1\times x_2+x_3\times x_3$ та $f=x_1\times(x_2+x_3)+x_2\times x_4$ відносно функцій двомісного множення $f_\times(x_1,x_2)=x_1\times x_2$ та двомісного додавання $f_+(x_1,x_2)=x_1+x_2$.

3. Довести примітивну рекурсивність таких функцій:

а) двомісного додавання $f_+(x_1,x_2)=x_1+x_2$,

б) двомісного множення $f_\times(x_1,x_2)=x_1\times x_2$,

в) укороченого віднімання $f_-(x_1,x_2)=x_1-x_2$ (коли зменшуване не менше, ніж від'ємник).

4. Довести примітивну рекурсивність функції n-місного додавання $f_+(x_1,\ldots,x_n)=x_1+\ldots+x_n$.

5. Розглянути рекурсивність функції n-місного множення $f_\times(x_1,\ldots,x_n)=x_1\times\ldots\times x_n$ та функції піднесення до степеня $f(x,y)=x^y$.

2.5. Алгоритмічна система Тьюрінга

Точний опис класу частково-рекурсивних функцій разом з тезою Черча дає один з можливих розв'язків задач про уточнення поняття алгоритму. Однак цей розв'язок не зовсім прямий, оскільки поняття обчислюваної функції є вторинним щодо інтуїтивного поняття алгоритму. Постає запитання: чи можна уточнити саме поняття алгоритму, а потім за його допомогою визначити клас обчислюваних функцій?

Відповідь на це питання дали 1936–1937 рр. Е. Пост і А. Тьюрінг та майже одночасно й незалежно А. Черч і С.-К. Кліні.

Головна думка праць Е. Поста і А. Тьюрінга полягає в тому, що алгоритмічні процеси — це процеси, які може виконувати відповідно побудована "машина". Відповідно до цього за допомогою точних математичних термінів вони описали досить вузькі класи машин, на яких можливо реалізувати або імітувати всі алгоритмічні процеси, які фактично будь-коли були описані математиками. Алгоритми, які можна реалізувати на машинах Тюрінга–Поста, запропоновано розглядати як математичний еквівалент алгоритмів у інтуїтивному понятті.

Отже, машина Тьюрінга – це математична модель пристрою, який породжує обчислювальні процеси. Її використовують для теоретичного уточнення поняття алгоритму та його дослідження.

2.5.1. Машина Тьюрінга

На змістовному рівні машина Тьюрінга (МТ) є деякою гіпотетичною (умовною) машиною, яка складається з трьох головних компонент: *інформаційної стрічки, головки для зчитування і запису та пристрою керування* (рис. 2.3).

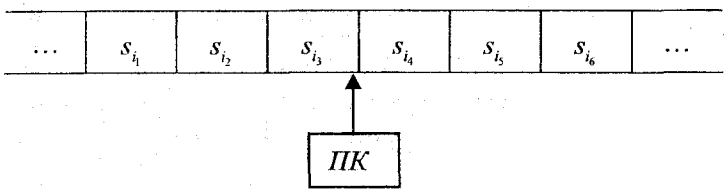


Рис. 2.3

Інформаційна стрічка призначена для записування вхідної, вихідної і проміжної інформації, яка виникає внаслідок обчислень. Стрічка потенційно безмежна в обидва боки і розбита на окремі не пронумеровані комірки (позиції, клітинки), в кожну з яких можна помістити один символ алфавіту $S = \{s_1, s_2, ..., s_k\}$, фіксованого для кожної МТ. Цей алфавіт називають ***зовнішнім алфавітом машини.***

Головка зчитування і запису – це спеціальний чуттєвий елемент, здатний читати і змінювати вміст комірок стрічки, зміщуючись уздовж неї вправо і вліво.

Рухається головка так, що в кожен момент часу t вона стоїть навпроти однієї певної комірки, і вважають, що в цей момент машина "сприймає" символ, записаний у цій позиції.

Пристрій керування (*функціональний механізм, логічний блок*) керує всією роботою машини відповідно до заданої програми обчислень. У кожний момент часу t функціональний механізм перебуває в одному зі станів, множина яких $Q = \{q_1, q_2, ..., q_n\}$ зафіксована для кожної МТ, її називають ***внутрішнім алфавітом машини***. В множині Q виділяють два особливі стани: *початковий* (q_0) і *заключний* (q_F), що відповідають за пуск і зупинку машини.

Функціонування МТ відбувається дискретними кроками. Кожен крок охоплює три елементарні операції:

1) символ s_i, на який вказує головка в цей момент, заміняється іншим символом алфавіту S:

$$s_i \to s_j \quad (i, j = \overline{1,k});$$

2) головка зсувається на одну позицію вліво (L) чи вправо (R) або ж залишається на місці (нейтральний зсув H);

3) функціональний механізм змінює свій стан q_l на новий стан з алфавіту Q:

$$q_l \to q_r \quad (l, r = \overline{1,n}).$$

Машину Тьюрінга можна використовувати для ***розпізнавання мов***. Символи на стрічці такої машини містять алфавіт мови, яку розглядають (її букви відіграють роль вхідних символів), порожній

символ і, можливо, інші символи. Спочатку на стрічці записано слово з вхідних символів (по одному символу в комірці), починаючи з першої ліворуч комірки. Всі комірки праворуч від комірок, що містять вхідне слово, порожні.

Слово з вхідних символів ***допустиме*** тоді й тільки тоді, коли машина Тьюрінга, почавши роботу у виділеному початковому стані, зробить послідовність кроків, які приведуть її в допускаючий (заключний) стан.

Мовою, яку ***розпізнає*** ця машина Тьюрінга, називають множину всіх слів із вхідних символів, які допустимі цією МТ.

Роботу машини Тьюрінга формально можна описати за допомогою конфігурацій. Під ***конфігурацією*** МТ в момент часу t (позначимо K_t) будемо розуміти таку сукупність умов:

- конкретне заповнення комірок стрічки в момент t буквами алфавіту S;
- конкретне положення головки на стрічці в момент часу t;
- стан функціонального механізму в цей момент.

Тоді процес роботи МТ полягатиме в послідовній зміні конфігурацій, починаючи від початкової конфігурації K_{t_0}, яка відповідає стану q_0 в початковий момент часу t_0. Якщо внаслідок роботи МТ породжує процес

$$MT(K_{t_0}) = K_{t_0}, K_{t_1}, K_{t_2}, \ldots, K_{t_m}, K_{t_{m+1}}, \ldots, K_{t_F},$$

який завершується заключною конфігурацією K_{t_F} (яка відповідає заключному стану q_F), то така МТ *застосовна* до початкової конфігурації K_{t_0}. Результатом роботи машини в такому разі вважають слово, яке буде записане на стрічці в заключній конфігурації з внутрішнім станом q_F.

Якщо ж заключна конфігурації K_{t_F} не досяжна, то МТ працює безмежно довго без зупинки (зациклюється). Вважають, що така МТ *незастосовна* до початкової конфігурації K_{t_0}.

Перехід МТ від конфігурації K_{t_m} до наступної конфігурації $K_{t_{m+1}}$ виконується за один крок, ініціалізований відповідною командою. ***Командою*** МТ називають таку п'ятірку:

$$< s_i, q_l > \rightarrow < s_j, q_r, Z >, \qquad (2.11)$$

де Z – одна з дій (зсув уліво, вправо чи на місці – {L, R, H}). Команда означає заміну s_i на s_j, q_l на q_r і зсув головки згідно з Z. Жодні дві команди не можуть мати однакових лівих частин. Стан q_F не може траплятися в лівій частині команди.

Множину команд вигляду (2.11), що визначає процес обчислень на МТ, називають ***програмою МТ***. Оскільки для кожної пари $< s_i, q_l >$ $(i = \overline{1,k};\ l = \overline{1,n})$ програма може містити відповідну п'ятірку, то розмір програми не перевищуватиме $k*n$ п'ятірок.

Програму можна задавати у вигляді таблиці, яку називають *функціональною схемою*, рядки і стовпці цієї таблиці позначені, відповідно, символами внутрішнього і зовнішнього алфавіту, а на перетині q_l рядка та s_i стовпця є трійка s_j, q_r, Z. Якщо один або декілька елементів залишаються незмінними, то їх не зазначають.

Зазначимо, що, крім інтерпретації МТ як пристрою для розпізнавання мов, її можна розглядати і як пристрій, який обчислює деяку функцію f. Аргументи цієї функції кодовані на вхідній стрічці у вигляді слова x зі спеціальним маркером '*', який відділяє їх один від одного. Якщо МТ зупиняється, а на стрічці є ціле число y (значення функції), то приймають $f(x)=y$. Отже, процес обчислення мало відрізняється від процесу розпізнавання мови.

Приклад 2.11. Побудуємо МТ для додавання двох натуральних чисел, записаних в унарному вигляді. На стрічці МТ задають два числа як послідовності відповідної кількості символів '1', розділених символом '*'. Програму для машини Тьюрінга задамо у вигляді такої таблиці:

T	1	A	*
q_0	Λq_2R	R	$\Lambda\, q_F$
q_1	L	q_0R	L
q_2	R	1q_1L	R

Множина станів МТ $Q = \{q_0, q_1, q_2, q_F\}$, зовнішній алфавіт $S = \{1, \Lambda, *\}$, де Λ – порожній символ.

Ця програма почергово заміняє символи '1' першого доданка, починаючи з лівого краю, на порожні символи і дописує відповідну кількість '1' після другого доданка. Такі дії відбуваються аж до роздільника *, який також заміняється Λ.

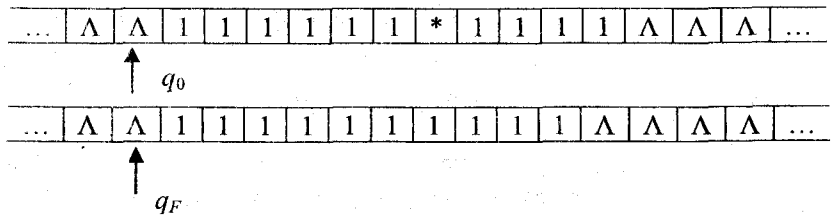


Рис. 2.4

На рис. 2.4 показано процес переробки початкової конфігурації в заключну у разі реалізації 6+4.

Можна побудувати й іншу програму МТ для розв'язування цієї задачі:

T	1	Λ	*
q_0	$\Lambda q_1 R$	R	-
q_1	R	-	$1q_2$
q_2	L	q_F	-

Ця програма перший символ '1' першого доданка заміняє на порожній символ, а роздільник * – на '1'.

Нагадаємо, що головними параметрами, які оцінюють складність алгоритму, є часова та ємнісна складність.

Часова складність T(n) машини Тьюрінга дорівнює найбільшій кількості кроків, зроблених нею під час опрацювання входу довжиною n (для всіх входів довжиною n). Якщо на деякому вході довжиною n МТ не зупиняється, то часова складність на цьому вході не визначена.

Ємнісна складність S(n) машини Тьюрінга дорівнює максимальній відстані, яку потрібно пройти головці під час опрацювання входу довжини n. Якщо головка для деякого входу довжини n невизначено довго рухається, то $S(n)$ на цьому вході не визначена.

2.5.2. *Формальне визначення МТ. Теза Тьюрінга*

З математичного погляду МТ – це певний алгоритм для переробки вхідних слів. Оскільки спосіб переробки машинних слів відомий, якщо відома програма машини, то МТ вважають заданою, якщо задані її зовнішній і внутрішній алфавіти, програма та визначено, які з символів відповідають порожній комірці й заключному стану.

Зазначимо, що кожна МТ реалізує таку функцію:

$$\delta : S \times \{Q \setminus q_F\} \to S \times Q \times Z ,$$

або інакше

$$\delta = \{< s_i, q_l > \to < s_j, q_r, Z >\}, \quad q_l \neq q_F .$$

Тоді ***формально машиною Тьюрінга*** *називають сімку*

$$(S, Q, q_0, q_F, I, \Lambda, \delta), \tag{2.12}$$

де $I \subseteq S$ *– множина вхідних символів;* Λ *– порожній символ.*

Якщо визначити строго формально МТ як математичний об'єкт, то можна на формальному рівні визначити поняття конфігурації, процесу обчислення і всі інші поняття, які використовують у доведеннях різних тверджень про процеси переробки інформації на МТ. Проте в теорії алгоритмів застосовують змістовний опис МТ і доведення виконують на змістовному рівні, вважаючи, що за ними легко відновити формальний еквівалент МТ.

Говорять, що слово P задане *в стандартному положенні*, якщо головка фіксує перший зліва символ вхідного слова. Якщо за початкову конфігурацію K_{t_0} взяти стандартне положення слова P,

то процес $MT(K_{t_0})$ є *процесом переробки вхідного слова P*, і його позначають $MT(P)$.

Машина Тьюрінга ***обчислює*** *деяку словникову функцію* $\varphi(P)$, *яка відображає слова з деякої множини слів у значущі (не порожні) слова, якщо для довільного слова P процес* $MT(P)$ *досягає заключної стандартної конфігурації, асоційованої зі словом* $\varphi(P)$ *– результатом цієї функції.*

Якщо для часткової словникової функції $\varphi(P)$ існує МТ, яка її обчислює, то функцію $\varphi(P)$ називають ***обчислюваною за Тьюрінгом.***

Якщо функція f є m-місною, то вважають, що обчислюється функція від одного аргументу, який є кодом слів $P_1,P_2,...,P_m$. Наприклад,

$$f(P_1,P_2,...,P_m)=f(P_1*P_2*...*P_m),$$

де символ $*\notin S$.

Теза Тьюрінга. *Для довільного алгоритму* $A=<\varphi,\mathcal{P}>$ *у довільному скінченному алфавіті X існує функція* $\varphi(P)\colon \{P\}_x\to\{Q\}_x$, *обчислювана за Тьюрінгом.*

Теза Тюрінга визначає відповідність між інтуїтивним поняттям алгоритму і точним математичним поняттям функції, обчислюваної на МТ: *будь-який алгоритм, заданий у довільній формі, можна замінити еквівалентною йому МТ.*

Згідно з цією тезою, питання про можливість алгоритмізації того чи іншого процесу рівносильне питанню про можливість реалізації цього процесу на МТ.

Як і принцип нормалізації, теза Тьюрінга стверджує неможливість побудови в майбутньому алгоритму, який не можна було б реалізувати на МТ.

Тезу Тьюрінга, як і тези Маркова і Черча, довести неможливо, оскільки в її формулюванні використане інтуїтивне поняття алгоритму.

Тюрінг довів такі важливі теореми.

Теорема 1. *Всі часткові словникові функції, обчислювані за Тьюрінгом, є частково-рекурсивними.*

Теорема 2. *Для кожної частково-рекурсивної словникової функції, визначеної в деякому алфавіті* $A=\{a_1,...,a_m\}$, *існує МТ з відповідним внутрішнім алфавітом, зовнішній алфавіт якої збігається з A* ($S=A$), *і яка обчислює задану функцію.*

Теореми Тьюрінга і теореми параграфів 2.3, 2.4 засвідчують еквівалентність трьох алгоритмічних систем – нормальних алгоритмів, частково-рекурсивних функцій і машин Тьюрінга.

2.5.3. *Універсальна машина Тьюрінга*

Дотепер ми розглядали різні алгоритми, виконувані на різних МТ, що відрізняються програмами, внутрішніми і зовнішніми алфавітами. За аналогією з універсальним нормальним алгоритмом можна побудувати універсальну машину Тьюрінга (УМТ), яка здатна виконувати роботу будь-якої конкретної МТ $T^{(k)}$ за умови, що УМТ отримує інформацію про програму П машини $T^{(k)}$ і вхідне слово P, подане на обробку в $T^{(k)}$.

В УМТ, як і в будь-якій МТ, інформація задана на стрічці. У цьому разі УМТ має фіксований скінченний зовнішній алфавіт S_u, який називають *стандартним*. Проте УМТ повинна бути пристосована до сприйняття програм і конфігурацій довільних $T^{(k)}$ і довільних вхідних слів, у яких можуть траплятися букви з різноманітних алфавітів і з як завгодно великою кількістю різних букв.

Це досягають шляхом кодування в стандартному алфавіті S_u $T^{(k)}=<S^{(k)},Q^{(k)},q_0,q_F,\Pi^{(k)}>$. Кодування можна виконати, як і у випадку УНА, при $S_u=\{0,1\}$: всі букви нумерують натуральними числами, після чого i-й букві ставлять у відповідність двійковий код $0\underbrace{11...1}_{i\ разів}0$.

Для кожної машини $T^{(k)}$ повинні бути закодовані всі символи на $S^{(k)}\cup Q^{(k)}\cup\{L,R,H\}\cup\{\Delta,*\}$, де символ Δ використовують для відділення один від одного команд програми $<s_i,q_l>\to<s_j,q_r,Z>$, а

символ * – для відділення лівої частини команди від правої. Якщо $S^{(k)}$ має n символів, а $Q^{(k)}$ – m символів, то всього повинно бути закодовано $n \times m + 5$ символів.

Якщо записати Π^{T_k} одним словом і кодувати його букви, то отримаємо слово в алфавіті S_U, яке називають *кодом* Π^{T_k} (позначають $\mathrm{cod}\,\Pi^{T_k}$).

А. Тьюрінг довів теорему про існування універсальної МТ.

Теорема 3. *Існує така МТ, її називають універсальною, яка для довільної машини Тьюрінга $T^{(k)}$ і довільного вхідного слова P перетворює вхідне слово $\mathrm{cod}\Pi^{T_k}\mathrm{cod}P$, отримане конкатенацією кодів Π^{T_k} і слова P, у код слова $T^{(k)}(P)$:*

$$\mathrm{cod}\Pi^{T_k}\mathrm{cod}P \xrightarrow{\text{УМТ}} \mathrm{cod}T^{(k)}(P).$$

Якщо $T^{(k)}$ до слова P не застосовна, то УМТ теж є незастосовною до слова $\mathrm{cod}\Pi^{T_k}\mathrm{cod}P$.

З огляду на існування УМТ будь-яку програму Π^{T_k} можна розглядати двояко:

- як програму деякої конкретної МТ,
- як вхідну інформацію для УМТ, що виконує роботу T_k.

Із теореми Тьюрінга випливає принципова можливість побудови машини, на якій можна реалізувати довільний алгоритм $<\varphi,\mathcal{P}>$. Водночас програмування для машини, яка моделює УМТ, є нереальним через його громіздкість. Тому на практиці універсальні машини будують на підставі інших алгоритмічних систем.

2.5.4. Різновиди машин Тьюрінга

Ми розглянули однострічкому одноголовкову МТ з одним функціональним пристроєм, яку називають *класичною машиною Тьюрінга*. Далі опишемо декілька варіантів МТ, які відрізняються один від одного кількістю зчитувальних головок, видом і кількістю стрічок, а також розглянемо операції композиції, застосовні до них.

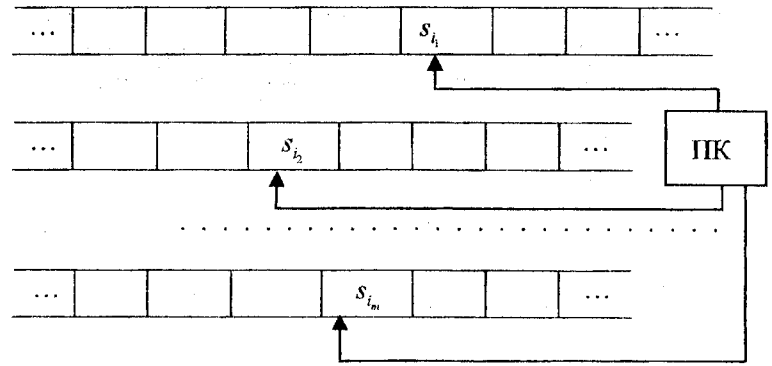


Рис. 2.5

Багатострічкова машина – це МТ зі скінченною кількістю стрічок (рис. 2.5). Її визначення можна розглядати на змістовному і формальному рівнях.

На змістовному рівні багатострічкова МТ складається з декількох, наприклад m, стрічок, кожна з яких має свою головку, що працює незалежно від інших. У кожен момент часу t машина спостерігає m комірок, по одній на кожній стрічці. Крок машини охоплює три елементарні операції:

а) заміна деяких або і всіх m символів комірок новими символами,

б) зсув кожної з m головок, або деяких з них, вліво, вправо або нейтральний зсув незалежно одна від одної,

в) перехід функціонального механізму в новий стан.

Формально m-стрічкова машина Тьюрінга задана сімкою:

$$(Q,S,I,\delta,\Lambda,q_0,q_F),$$

де Q – множина станів; S – множина символів на стрічках; I – множина вхідних символів ($I \subseteq S$); Λ – порожній символ ($\Lambda \in S - I$); q_0 – початковий стан; q_F – заключний стан. Функція переходів δ відображає деяку підмножину множини $Q \setminus \{q_F\} \times S^k$ в $Q \times (S \times \{\mathrm{L},\mathrm{R},\mathrm{S}\})^k$, тобто за довільним набором станів та k

символів на стрічках вона видає новий стан і k пар, кожна з яких складається з нового символа на стрічці і напряму зсуву головки.

Відповідно до цього команди складаються з $3m+2$ символів і мають такий вигляд:

$$< s_{i_1}, s_{i_2}, ..., s_{i_m}, q_l > \rightarrow < s_{j_1}, s_{j_2}, ..., s_{j_m}, q_r, Z_1, ..., Z_m > . \qquad (2.13)$$

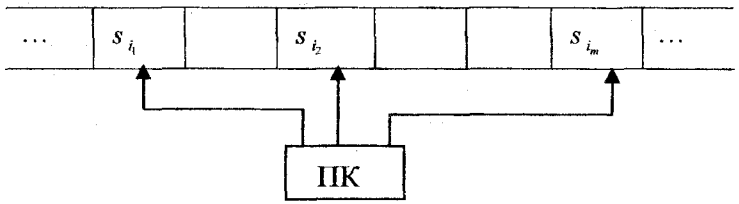


Рис. 2.6.

Багатоголовкова машина – це однострічкова МТ з m головками (рис. 2.6).

У кожен момент часу t машина спостерігає m комірок. Крок машини містить три елементарні операції, аналогічні операціям багатострічкової машини, а вигляд команди повністю збігається з (2.13). Тому за виглядом програми для багатострічкової та багатоголовкової МТ не відрізняються.

У цьому варіанті МТ можливий випадок, коли декілька головок спостерігають одну й ту ж комірку, проте команда передбачає різні записи в цю комірку.

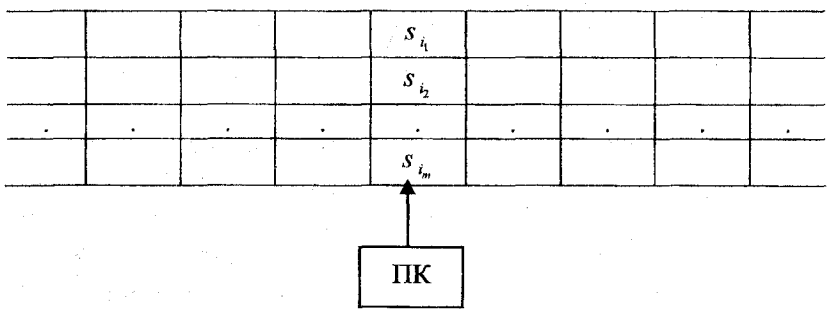


Рис. 2.7.

Наприклад, для двоголовкової машини це може бути така команда: $< s_i, s_i, q_l > \rightarrow < s_{j_1}, s_{j_2}, q_r, Z_1, Z_2 >$, де $s_{j_1} \neq s_{j_2}$. Для усунення такої неоднозначності вважають, що в комірці зберігається лише той запис, який заданий головкою з найменшим номером (s_{j_1}). Аналогічна домовленість прийнята і в загальному випадку – у разі визначення функціонування багатоголовкової машини з кількістю стрічок, меншою від кількості головок.

Машина з багатоповерховою стрічкою – це однострічкова, одноголовкова МТ (рис. 2.7). Стрічка має m поверхів і в кожний момент часу t машина оглядає відразу m символів, які розташовані на поверхах комірки, що є активною в цей момент.

Крок машини полягає в заміні m символів і стану, а також зсуві головки, а команда має такий вигляд:

$$< s_{i_1}, s_{i_2}, ..., s_{i_m}, q_l > \rightarrow < s_{j_1}, s_{j_2}, ..., s_{j_m}, q_r, Z > .$$

Одноголовкова машина з однією двовимірною стрічкою. Двовимірна стрічка – це площина, вистелена комірками. Для кожної комірки є сусідні праворуч, ліворуч, знизу і зверху, відповідно до чого і зсуви головок будуть чотирьох видів. Команда має вигляд

$$< s_i, q_l > \rightarrow < s_j, q_r, \pi > ,$$

де $\pi = \{L, R, S, U, D\}$, (U – уверх, D – униз).

Одностороння машина – це одноголовкова, однострічкова МТ, в якої стрічка є безмежною в один бік. Розглянемо випадок, коли стрічка безмежна вправо і має лівий кінець. Тоді команди з правим зсувом або без зсуву мають вигляд, як і в класичній МТ, та виконувані звичайним способом.

Вигляд команд з лівим зсувом і характер їхнього виконання інший. Команда

$$< s_i, q_l > \rightarrow \begin{cases} < s_{j_1} L q_{r_1} > \\ < s_{j_2} H q_{r_2} > \end{cases}$$

Означає таке: якщо видима комірка не є крайньою лівою, то виконується те, що задане у верхньому рядку, в протилежному випадку – у нижньому. Отже, машина здатна відчувати крайню комірку і робити відповідний вибір (тобто команда є умовною).

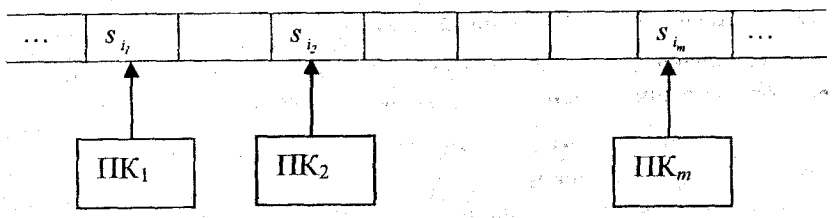


Рис. 2.8.

Машина з декількома функціональними механізмами – це однострічкова, багатоголовкова МТ, у якій головки пристроїв керування переміщаються вздовж стрічки та виконують свої команди незалежно одна від одної (рис. 2.8). Оскільки може трапитись, що кілька головок спостерігають одну й ту ж комірку, і відповідні пристрої керування передбачають різні записи в цю комірку, то вважають, що пристрої керування пронумеровані, а в комірці зберігається запис, змінений пристроєм з найменшим номером.

Є також інші варіанти МТ, які поєднують у собі ті чи інші компоненти перерахованих вище машин. Проте всі такі машини не виводять з класу функцій, обчислюваних за Тьюрінгом.

Операції композиції, виконувані над алгоритмами, дають змогу утворювати нові, складніші алгоритми з відомих простих.

Над усіма МТ можна виконувати такі операції композиції: *суперпозиція, об'єднання, розгалуження, ітерація*. Розглянемо деякі з них.

Нехай задано дві машини Тьюрінга T_1 та T_2, які мають спільний зовнішній $S=\{s_1,s_2,...,s_m\}$ і внутрішні $Q_1=\{q_0,q_1,q_2,...,q_n\}$ та $Q_2=\{q_0',q_1',q_2',...,q_t'\}$ алфавіти.

Суперпозицією, або ***добутком,*** машин Тьюрінга T_1 та T_2 називають машину T з тим же зовнішнім алфавітом S та внутрішнім $Q_1=\{q_0,q_1,q_2,...,q_n,q_{n+1},...,q_{n+t}\}$ і програмою, яку

отримують так. У всіх командах T_1, які містять заключний стан q_n, його заміняють на початковий стан машини T_2: $q_{n+1}=q_0'$. Всі інші символи в командах T_1 залишають без зміни. Кожен зі станів q_j' $(j=1,2,...,t)$ заміняють станом q_{n+j}.

Сукупність усіх команд T_1 та T_2, змінена таким способом, і утворює програму суперпозиції машин T_1 та T_2, яку позначають T_1*T_2. Таблиця програми T складається з двох частин: верхня описує T_1, а нижня – T_2.

Якщо одна з машин має більше одного заключного стану, то обов'язково повинно бути визначено, який із заключних станів попередньої машини ототожнюється із початковим станом наступної. Наприклад,

$$T=T_1*\begin{cases}(1)T_2\\(2),\end{cases}\quad\text{або}\quad T=T_1*\begin{cases}(1)\\(2)T_2.\end{cases}$$

Машина T у цьому випадку має також два заключні стани: один заключний стан T_1, інший – заключний стан T_2.

Операцію ітерації застосовують до однієї машини. Суть операції полягає в такому. Нехай машина T_1 має декілька заключних станів. Виберемо деякий r-й заключний стан і ототожнимо його в основній таблиці машини T_1 з її початковим станом. Отриману машину позначимо

$$T=\dot{T}_1\begin{cases}(1)\\...\\(r)\\...\\(s)\end{cases}$$

Вона є результатом ітерації машини $\dot{T}_1$. Тут символ '.' над T_1 означає ототожнення заключного стану з початковим станом машини T_1.

Якщо T_1 має лише один заключний стан, то після ітерації отримують машину, яка не має заключного стану взагалі.

2.5.5. Проблема розпізнавання самозастосованості алгоритмів

Алгоритмічну систему Тьюрінга використовують для доведення нерозв'язності задач. Розглянемо одну з алгоритмічно нерозв'язних проблем.

Нехай система правил $\mathcal{P}$ алгоритму $A=<\varphi,\mathcal{P}>$ закодована певним способом у вхідному алфавіті алгоритму A. Позначимо цей код системи $\mathcal{P}$ через $\mathcal{P}^{cod}$. Поняття самозастосовності алгоритму введено в 1.4.2.

Сформулюємо ***проблему*** розпізнавання самозастосовності алгоритмів так:

знайти алгоритм, здатний за кодом $\mathcal{P}^{cod}$ довільного алгоритму $A=<\varphi,\mathcal{P}>$ визначити, чи є A самозастосовним.

Теорема. *Проблема розпізнавання самозастосовності алгоритму є алгоритмічно нерозв'язною.*

Доведення. Цю теорему доводять на підставі алгоритмічної системи Тьюрінга.

Нехай на стрічці деякої МТ T_k зображено її власний код T_k^{cod} (тобто код програми і початкової конфігурації), записаний в алфавіті машини T_k. Якщо T_k застосовна до T_k^{cod}, то будемо називати її самозастосовною, в іншому випадку – несамозастосовною. Нагадаємо, що застосовність МТ до вхідного слова означає зупинку цієї машини через скінченну кількість кроків.

Припустимо, що існує МТ M, яка розпізнає самозастосовність довільної машини T_k. Тоді в M кожен самозастосовний код T_k^{cod} перетворюється в деякий символ ν (який позначає позитивну відповідь на поставлене питання про самозастосовність), а кожен несамозастосовний код – у символ τ (який позначатиме негативну відповідь).

Звідси випливає, що можна побудувати і таку машину M_1, яка перероблятиме несамозастосовні коди в τ, тоді як до самозастосовних кодів машина M_1 вже буде незастосовною. Цього можна досягти шляхом такої зміни програми M, щоб після появи символа ν машина, замість зупинки, нескінчено довго повторювала б цей символ.

Отже, M_1 буде застосовною до кожного несамозастосовного коду (у цьому разі видає символ τ) і незастосовна до самозастосовних кодів T_k^{cod} (нескінченно видає символ ν). Однак це приводить до суперечності. Справді:

1) нехай машина M_1 самозастосовна, тоді вона застосовна до свого коду M_1^{cod} і переробляє його в символ τ. Однак поява цього символа саме й повинна означати, що M_1 несамозастосовна;

2) нехай M_1 несамозастосовна, тоді вона застосовна до M_1^{cod}, а це означає, що M_1 є самозастосовною.

Отримана суперечність доводить теорему, оскільки припущення про існування машини M, яка розпізнає самозастосовність, не правильне.

Нерозв'язність цієї проблеми часто використовують для доведення нерозв'язності й інших масових проблем.

2.5.6. Завдання для самостійної роботи

1. Визначити, що реалізує задана таблицею машина Тьюрінга.

	1	Λ	*
q_0	Λ q_2 R	R	Λ q_F
q_1	L	q_0 R	L
q_2	R	q_1 1	R

2. Побудувати МТ, для обчислення найпростіших функцій.

3. Побудувати МТ для додавання двох чисел в унарній, двійковій та десятковій системах числення.

4. Побудувати МТ для віднімання, множення та ділення двох чисел в унарній системі числення.

5. Побудувати та оцінити МТ для переведення чисел:
   а) з унарної системи у двійкову,
   б) з унарної системи у десяткову,
   в) з двійкової системи в унарну.

6. Побудувати двострічкову МТ, яка розпізнає паліндроми в алфавіті {0, 1}. Перевірити її роботу на таких вхідних словах 0010 та 01110.

7. Реалізувати на тристрічковій МТ додавання двох двійкових чисел. Уважати, що два доданки задані на двох стрічках, а результат необхідно помістити на третю стрічку.

8. Побудувати МТ, яка за вхідним рядком з n символів '1' на першій стрічці друкує n^2 символів '1' на другій стрічці, тобто виконує перетворення 1^n у 1^{n^2}.

9. Побудувати й оцінити програму для МТ, яка допускає всі входи вигляду $0^n 1 0^{n^2}$.

10. Побудувати й оцінити програму для МТ, яка допускає всі входи вигляду $1^n 0 1^{2n}$.

2.6. Алгоритмічна система Поста

Система Поста дуже схожа на систему Тьюрінга, тому лише коротко опишемо машину Поста та сформулюємо головну тезу.

Машина Поста має нескінченну інформаційну стрічку, на яку записують інформацію в двійковому алфавіті {0,1}. Алгоритм задають як скінченну впорядковану послідовність перенумерованих правил — команд Поста.

Робота машини відбувається дискретними кроками, на кожному кроці виконується одна команда. Кожному кроку відповідає на інформаційній стрічці *активна комірка*. Для першої команди положення активної комірки фіксоване – це *початкова активна комірка*. Наступні положення активної комірки змінюються відповідно до команд Поста.

Є шість типів команд Поста:

1) відмітити активну комірку стрічки, тобто записати в неї 1 і перейти до виконання i-ї команди;

2) стерти відмітку активної комірки, тобто записати в неї 0 і перейти до виконання i-ї команди;

3) змістити активну комірку на одну позицію вправо і перейти до виконання i-ї команди;

4) змістити активну комірку на одну позицію вліво і перейти до виконання i-ї команди;

5) якщо активна комірка відмічена, то перейти до виконання i-ї команди, інакше – до виконання j-ї команди;

6) зупинка, закінчення роботи алгоритму.

Отже, в алгоритмічній системі Поста, на відміну від інших систем, у явному вигляді сформульовано принцип безумовного передавання (команди 1–4) і умовного передавання керування (команда 5).

Алгоритм, складений з довільної скінченної кількості команд Поста, називають ***алгоритмом Поста***.

Теза Поста: *Для кожного алгоритму* $A=<\varphi,\mathcal{P}>$ *існує алгоритм Поста, який реалізує словникову функцію* φ.

Доведено, що система Поста еквівалентна іншим системам, зокрема, системам Тюрінга, Маркова, Черча–Кліні.

2.7. Рівнодоступна адресна машина

2.7.1. Операторні алгоритмічні системи

Однією з особливостей розглянутих алгоритмічних систем, за винятком машини Поста, є незмінність набору допустимих засобів, використовуваних для запису алгоритмів і процесів їхнього виконання. В системі нормальних алгоритмів такими є один оператор (підстановки) і один розпізнавач (входження); у системі рекурсивних функцій фіксованим є набір базових функцій (S,O,I_m^n) і набір операторів $(\mathcal{S},\mathcal{R},M,M^l)$; під час обчислення на машині Тьюрінга елементарні такти обмежені фіксованим набором операцій (заміна символа в комірці, зміна стану машини і позиції головки).

Водночас під час вивчення конкретних алгоритмів бажано мати такі допустимі засоби їхнього опису, які найзручніші для цього класу алгоритмів. Наприклад, алгоритми лінійної алгебри найзручніше описувати за допомогою чотирьох арифметичних дій, а алгоритми обчислення функцій алгебри логіки – за допомогою тих логічних операцій, у термінах яких ці функції записані (або заперечення).

Операторні алгоритмічні системи першими почали орієнтувати на конкретні мови програмування, тобто на реальні обчислювальні машини. До визначення алгоритму в разі практичного використання ставлять такі вимоги:

- можливість вибору засобів опису алгоримів залежно від класу алгоритмів;
- дозвіл формування команд алгоритму в процесі його виконання;
- дозвіл використання широкого діапазону елементарних логічних операцій над об'єктами.

2.7.2. *Машина з довільним доступом до пам'яті*

Однією з найпоширеніших операторних алгоритмічних систем, які найбільше наближені до практичної реалізації, є машина з довільним доступом до пам'яті, або ***рівнодоступна адресна машина*** (***РАМ***). Машина з довільним доступом до пам'яті моделює обчислювальну машину з одним суматором, у якій команди програми не можуть самі себе змінювати.

РАМ складається з вхідної стрічки, з якої вона може лише читати, вихідної стрічки, на яку вона може лише записувати, і пам'яті (рис. 2.9).

Вхідна стрічка – це послідовність комірок, кожна з яких може містити ціле число (можливо, від'ємне). Кожного разу, коли символ зчитується з вхідної стрічки, її головка зчитування зміщується на одну комірку вправо. Вихідна стрічка теж розбита на комірки, які спочатку порожні. Під час виконання команди запису в тій комірці вихідної стрічки, яку визначає головка, друкується ціле число, і головка зміщується на одну комірку вправо. Як тільки символ записано, його вже не можна змінити.

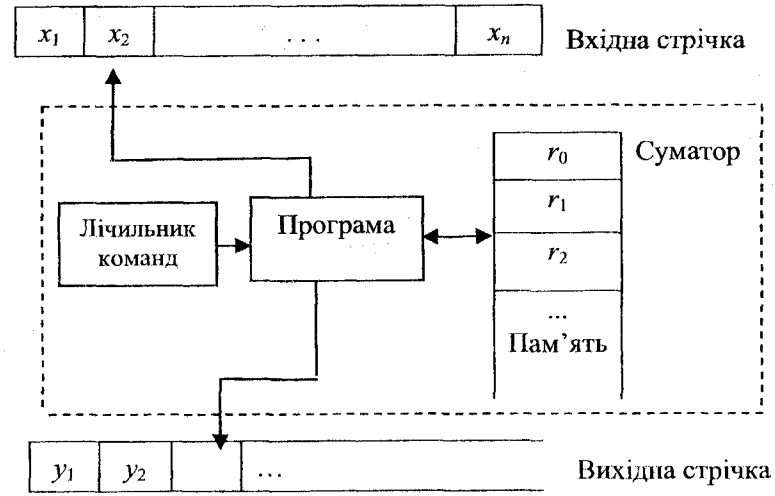


Рис. 2.9.

Пам'ять складається з послідовності регістрів r_0, r_1, ..., кожен з яких здатний зберігати довільне ціле число. Зазначимо, що верхня межа на кількість регістрів, які можна використовувати, не ввизначена. Така ідеалізація допустима у випадку, коли:

1) розмір задачі достатньо малий, щоб вона помістилась в основній пам'яті обчислювальної машини;

2) цілі числа, які беруть участь в обчисленнях, достатньо малі, щоб їх можна було помістити в одну комірку.

Програма для РАМ не записується в пам'ять і тому не може сама себе змінювати. Програма – це послідовність (можливо) помічених команд. Вони нагадують ті, які звичайно трапляються в реальних обчислювальних машинах. Передбачають, що є арифметичні команди, команди введення–виведення, непряма адресація (для індексації масивів) і команди розгалуження.

Всі обчислення відбуваються в першому регістрі r_0, який називають *суматором*; він, як і будь-який регістр пам'яті, може зберігати довільне ціле число.

Таблиця 2.3

Таблиця команд РАМ

Код операції	Адреса	Дія
LOAD	Операнд	Операнд заноситься в суматор
STORE	Операнд	Інформація з суматора заноситься в регістр (операнд)
ADD	Операнд	До вмісту суматора додається операнд
SUB	Операнд	Від вмісту суматора віднімається операнд
MULT	Операнд	Вміст суматора домножується на операнд
DIV	Операнд	Вміст суматора ділиться на операнд
READ	Операнд	Черговий вхідний символ заноситься в регістр (операнд)
WRITE	Операнд	Операнд або його вміст записують на вихідну стрічку
JUMP	Мітка	Безумовний перехід на команду із заданою міткою
JGTZ	Мітка	Якщо вміст суматора додатний, то відбувається перехід на команду із заданою міткою, інакше – на наступну команду
JZERO	Мітка	Перехід на команду із заданою міткою відбувається тоді, коли вміст суматора дорівнює нулю
HALT		Завершення програми

Приклад набору команд наведено в табл. 2.3. Кожна команда складається з двох частин – коду операції та адреси.

Тип операнда може бути одним з таких:

1) $=i$, означає ціле число, i називають літералом;

2) i, вміст регістра i (i повинно бути невід'ємним), будемо позначати $c(i)$;

3) $*i$, означає непряму адресацію, тобто значенням операнда є вміст регістру j – $c(j)$, де $j = c(i)$ – ціле число, що міститься в регістрі i; якщо $j<0$, то машина зупиняється.

Невизначені команди можна вважати еквівалентними HALT. У разі виконання перших восьми команд з табл. 2.1 лічильник команд збільшується на 1, тому такі команди виконуються послідовно; JUMP, JGTZ, JZERO – команди переходів.

Загалом РАМ-програма визначає відображення з множини вхідних стрічок у множину вихідних стрічок. Оскільки на деяких вхідних стрічках програма може не зупинятись, то це відображення є частковим (тобто для деяких входів воно може бути

невизначене). Це відображення можна інтерпретувати різними способами. Дві важливі інтерпретації – інтерпретація у вигляді функції та інтерпретація у вигляді мови.

Припустимо, що програма Π завжди зчитує з вхідної стрічки n цілих чисел і записує на вихідну стрічку не більше одного цілого числа. Нехай $x_1, x_2, \ldots, x_n$ – цілі числа, які містяться в n перших комірках вхідної стрічки, і нехай програма Π записує одне число y у першу комірку вихідної стрічки, а потім через якийсь час зупиняється. Тоді кажуть, що Π обчислює функцію $f(x_1, \ldots, x_n)=y$. Легко довести, що РАМ, як і будь-яка інша розумна модель обчислювальної машини, може обчислювати точно частково-рекурсивні функції. Іншими словами, для довільної частково-рекурсивної функції f можна написати РАМ-програму, яка обчислює f, і для довільної РАМ-програми можна задати еквівалентну їй частково-рекурсивну функцію ($K_{\text{ч.р.}}=K_{\text{РАМ}}$).

Інший спосіб інтерпретувати програму для РАМ – це розглянути її з погляду мови, яку вона розпізнає. Символи алфавіту можна подати цілими числами $1,2,\ldots,k$. РАМ розпізнає мову в такому сенсі. Нехай на вхідній стрічці міститься ланцюжок символів $s = a_1a_2\ldots a_n$, причому символ a_1 – у першій комірці, символ a_2 – у другій і так далі, а в $(n+1)$-й комірці розміщено 0-символ, який використовуватиметься як маркер кінця вхідного рядка (кінцевий маркер).

Вхідний ***ланцюжок s розпізнає РАМ-програма П,*** якщо Π прочитує всі його символи і кінцевий маркер, пише 1 у першій комірці вихідної стрічки і зупиняється.

Мовою, яку розпізнає програма П, називають множину всіх ланцюжків (слів), які вона допускає. Для вхідних ланцюжків, що не належать мові, яку розпізнає програма П, машина може надрукувати на вихідній стрічці символ, відмінний від 1, і зупинитись, а може не зупинитись узагалі.

Легко довести, що мову розпізнає деяка РАМ, тоді й лише тоді, коли вона рекурсивно-перелічувана. Мову розпізнає РАМ, яка зупиняється на всіх входах, тоді і лише тоді, коли вона рекурсивна.

Зазначимо, що ***рекурсивна мова*** – це мова з алгоритмічно розв'язною проблемою входження, тобто характеристична функція якої є частково-рекурсивною.

Приклад 2.12. Нехай на вхідну стрічку РАМ подано два цілі числа a_1, a_2. Побудувати РАМ-програму для знаходження значення виразу $S=a_1+a_2+1$.

Програма:

```
READ 1
READ 2
LOAD 1
ADD 2
ADD =1
STORE 1
WRITE 1
HALT
```

2.7.3. Обчислювальна складність РАМ-програм

Часова складність РАМ-програми – це функція $f(n)$, яка дорівнює найбільшому (на всіх входах розміру n) із підсумованих часів, затрачених на кожну команду, що працює.

Часова складність у середньому – це середнє, взяте по всіх входах розміру n, тих же сум.

Такі ж поняття визначають для ємнісної складності РАМ-програми, у цьому разі підсумовують ємність всіх регістрів, до яких було звертання.

Щоб точно визначити часову та ємнісну складності, треба задати час, необхідний для виконання кожної РАМ-команди, і обсяг пам'яті, використовуваної кожним регістром. Розглянемо два такі вагові критерії для РАМ-програм.

У випадку ***рівномірного вагового критерію*** кожна РАМ-команда затрачає на своє виконання одну одиницю часу і кожен регістр використовує одну одиницю пам'яті.

Друге визначення бере до уваги обмеженість розміру реальної комірки пам'яті, його називають ***логарифмічним ваговим крите-***

рієм. Нехай $l(i)$ – логарифмічна функція на цілих числах, задана рівностями

$$l(i)=\begin{cases}\lfloor log|i|\rfloor+1, & i\neq 0\\ 1, & i=0\end{cases},$$

де $\lfloor x\rfloor$ – нижнє ціле від x. Тут враховано, що для зображення цілого числа n в регістрі потрібно $\lfloor log|n|\rfloor+1$ бітів.

Логарифмічні ваги для трьох можливих видів операндів такі:

$=i$	$l(i)$
i	$l(i)+l(c(i))$
$*i$	$l(i)+l(c(i))+l(c(c(i)))$.

Таблиця 2.4

Логарифмічні ваги РАМ-команд

Команда	Вага
LOAD a	$l(a)$ – вага операнда a
STORE i	$l(c(0))+l(i)$
STORE $*i$	$l(c(0))+l(i)+l(c(i))$
ADD a	$l(c(0))+l(a)$
SUB a	$l(c(0))+l(a)$
MULT a	$l(c(0))+l(a)$
DIV a	$l(c(0))+l(a)$
READ $*i$	l(вхід)$+l(i)+l(c(i))$
READ i	l(вхід)$+l(i)$
WRITE a	$l(a)$
JUMP b	1
JGTZ b	$l(c(0))$
JZERO b	$l(c(0))$
HALT	1

Логарифмічний ваговий критерій ґрунтується на грубому припущенні, що ціна виконання команди (її вага) пропорційна до довжини її операндів (табл. 2.4).

Визначимо ***логарифмічну ємнісну складність*** РАМ-програми як суму за всіма регістрами, які працювали, включаючи суматор, величин $l(x_i)$, де x_i – найбільше за абсолютною величиною ціле число, що містилося в i-му регістрі за весь час обчислень.

Зрозуміло, що програма може мати різні часові складності залежно від того, який критерій використовують – рівномірний чи логарифмічний.

Якщо припустити, що для зберігання кожного числа, яке трапляється в задачі, достатньо одного машинного слова, то користуються рівномірною ваговою функцією. В іншому випадку для реалістичного аналізу складності більше може підходити логарифмічна вага.

Поряд з РАМ розглядають ще іншу модель обчислювальної машини – *машину з довільним доступом до пам'яті і програмою, що зберігається в пам'яті* (*РАСП*), оцінки якої не перевищують відповідні оцінки РАМ більше ніж у k разів (k = const).

2.7.4. Зв'язок машин Тьюрінга і РАМ

Головно машини Тьюрінга застосовують у визначенні нижніх оцінок, необхідних для розв'язання заданої задачі. У більшості випадків вдається знайти нижні оцінки лише з точністю до поліномної зв'язаності. Для точніших оцінок треба розглядати специфічніші деталі конкретних моделей.

Розглянемо зв'язок між РАМ та МТ. Очевидно, що РАМ може моделювати роботу k-стрічкової МТ, розміщуючи по одній клітинці МТ в регістр. Зокрема i-ну клітинку j-ї стрічки можна зберігати в регістрі $ki+j+c$, де c – деяка стала, яка дає змогу залишити певну кількість вільних регістрів для проміжних обчислень, у тому числі k регістрів для запам'ятовування положення k головок МТ. РАМ може читати з клітинок МТ, використовуючи непряму адресацію за допомогою регістрів, які містять положення головок на стрічках.

Припустимо, що задана МТ має часову складність $T(n) \geq n$. Тоді РАМ може прочитати її вхід, запам'ятати його в регістрах, які відображають першу стрічку, і змоделювати цю машину Тьюрінга за час $O(T(n))$ у разі рівномірного вагового критерію і

$O(T(n)\log_2 T(n))$ у випадку логарифмічного. У будь-якому випадку час на РАМ обмежений зверху поліномом від часу МТ, бо довільна функція $O(T(n)\log_2 n)$ є, зрозуміло, $O(T^2(n))$.

Обернене твердження правильне лише у разі логарифмічного вагового критерію для РАМ. За рівномірного критерію n-крокова програма для РАМ без входу може обчислювати числа порядку 2^{2^n}, що потребує порядку 2^n кліток МТ лише для запам'ятовування і зчитування. Тому у разі рівномірного критерію немає поліномального зв'язку між РАМ та МТ.

Якщо зростання чисел за порядком не перевищує розміру задачі, то варто використовувати РАМ з рівномірною вагою. Хоча під час аналізування алгоритмів зручніше застосовувати рівномірний ваговий критерій з огляду на його простоту, однак його треба відкинути, якщо визначають нижні межі на часову складність.

Для логарифмічної ваги виконується така теорема.

Теорема 1. *Нехай L – мова, яку розпізнає деяка РАМ за час $T(n)$ у випадку логарифмічної ваги. Якщо в РАМ-програмі немає множень і ділень, то на багатострічкових машинах Тьюрінга L має часову складність, не більшу від $O(T^2(n))$.*

Якщо в РАМ-програмі є команди множення і ділення, то можна написати підпрограми для МТ, які виконують ці операції за допомогою додавання та віднімання. Легко довести, що логарифмічні ваги цих підпрограм не більші, ніж квадрат логарифмічних ваг відповідних команд.

Теорема 2. *РАМ і РАСП з логарифмічною вагою і багатострічкові машини Тьюрінга поліномально зв'язані.*

Аналогічний результат отримують і для ємнісної складності.

2.7.5. Завдання для самостійної роботи

1. Написати програму для РАМ, яка з вхідної стрічки зчитує n додатних чисел, обмежених 0, і виводить їх у порядку неспадання на вихідну стрічку.

2. Написати програму для РАМ, яка за заданим числом n на вхідній стрічці обчислює n^n з часовою оцінкою $O(\log_2 n)$ у випадку рівномірного вагового критерію.

3. Написати та оцінити програму для РАМ, яка допускає всі ланцюжки, обмежені 0, у яких кількість 1 та 2 є однаковою.

4. Написати програму для РАМ, яка допускає всі входи вигляду $1^n 2^{n^2} 0$.

Розділ 3

ВАЖКОРОЗВ'ЯЗНІ ЗАДАЧІ

3.1. Поліноміальні алгоритми та важкорозв'язні задачі

Алгоритм можна розглядати, як "чорний ящик", який за вхідною послідовністю будує вихідну послідовність:

Зокрема, можна вважати, що вхідна і вихідна послідовності складаються з 0 і 1, які кодують вхід і вихід алгоритму. Тоді алгоритм розглядають як послідовність елементарних двійкових операцій, таких як *або, і, не, друк* і тощо, що працюють з пам'яттю двійкових символів, яка може бути досить великою.

Звичайно, можливі й інші зображення даних, однак на практиці природні способи зображення є еквівалентними, оскільки вони можуть бути поліноміально перетворені один в одний. У цьому разі, різниця між алгоритмами з поліноміальним і експоненціальним часом залишається незмінною: оскільки всі наші типи даних містять скінченну кількість двійкових символів, то переведення алгоритмів з мови високого рівня на рівень двійкових символів збільшує кількість операцій лише поліноміально (операції потребують поліноміального відносно довжини операнда часу).

Якщо алгоритм потребує не більше $T(n)$ операцій високого рівня на вході довжини n, то він потребує не більше $P(T(n))$ операцій над двійковими значеннями на такій же вхідній послідовності. Тут P – поліноміальна функція, яка визначає зростання кількості операцій у разі переходу від операцій високого рівня до двійкових операцій.

Отже, *поліноміальність чи експоненціальність часу роботи алгоритму* ***інваріантна***, оскільки $P(T(n))$ обмежене зверху деяким поліномом відносно n.

Якщо вхідні та вихідні послідовності кодовані "розумним" способом (код n не більший від $O(\log_2(n))$), то *поліноміальність чи експоненціальність довжини вхідної послідовності* також ***інваріантна***.

Нагадаємо, що ***алгоритм з поліноміальним часом*** – це алгоритм, час роботи якого (тобто кількість елементарних двійкових операцій, які він виконує на вхідному рядку довжини n) обмежений зверху деяким поліномом $P_k(n)$ (k – степінь полінома). Якщо стоїть питання про точні оцінки, то вони залежать від реалізації. Характеристика порядку, наприклад, $O(n^2)$ властива алгоритму і не залежить від реалізації.

Якщо оцінка алгоритму зростає швидше, ніж поліном, то такі алгоритми називають ***експоненціальними***.

Більшість задач, цікавих з практичного погляду, мають поліноміальні алгоритми розв'язування. Такі задачі вважають ***легкорозв'язними***. Задачу називають ***важкорозв'язною***, якщо не існує поліноміальних алгоритмів для її розв'язування.

Для експоненціальних алгоритмів важливо знати, наскільки "погано" вони працюють. Є доволі багато задач, для яких відомі лише експоненціальні алгоритми, однак їх необхідно розв'язати через практичну значимість. Тоді постає питання про вибір найефективнішого з цих експоненціальних алгоритмів. Справді, алгоритм, який розв'язує задачу розмірності n за $O(2^n)$ кроків, є "ліпшим", ніж алгоритм, який розв'язує її за $O(n!)$ або $O(n^n)$ кроків.

Поняття поліноміально розв'язної задачі прийнято вважати уточненням ідеї "практично розв'язної" задачі. Це пов'язано з тим, що:

- по-перше, використовувані на практиці поліноміальні алгоритми зазвичай справді працюють досить швидко;
- по-друге, обсяг цього класу практично не залежить від вибору конкретної моделі обчислень.

Наприклад, клас задач, які можна розв'язати за поліноміальний час на RAM-машині, збігається з класом задач, поліноміально розв'язних на машинах Тьюрінга. Клас буде таким самим і для моделі паралельних обчислень, якщо кількість процесорів обмежена поліномом від довжини входу.

По-третє, клас поліноміально розв'язних задач має природні властивості замкнутості. Наприклад, композиція двох алгоритмів теж працює за поліноміальний час. Це пояснюють тим, що сума, добуток і композиція поліномів є поліномом.

3.2. Недетерміновані машини Тьюрінга

Неформально недетермінованість можна пояснити, розглядаючи алгоритм, який виконує обчислення до певного місця, у якому повинен бути зроблений вибір з кількох альтернатив. Далі недетермінований алгоритм досліджує всі можливості одночасно, копіюючи себе для кожної альтернативи. Всі копії працюватимуть незалежно, не обмінюючись інформацією між собою. Якщо копія виявляє, що вона зробила неправильний або безрезультативний вибір, вона припиняє виконуватися, а якщо копія знайшла результат, вона оголошує про свій успіх і всі копії припиняють роботу.

На відміну від недетермінованого, детермінований алгоритм досліджуватиме кожну альтернативу по-черзі, кожного разу повертаючись для дослідження нової гілки.

Уведемо поняття ***стану*** як комбінації адреси виконуваної команди і значень усіх змінних. У детермінованому алгоритмі кожен стан визначає свого наступника, а в недетермінованому може бути більше одного допустимого стану. Недетерміновані алгоритми не є ймовірнісними, вони є алгоритмами, які можуть бути одночасно в багатьох станах.

Як приклад моделі недетермінованих алгоритмів, розглянемо **недетерміновану машину Тьюрінга (НМТ).**

НМТ має скінченну кількість можливих кроків, з яких у черговий момент вибирається якийсь один. Вхідний ланцюжок x *допускається* (*приймається*), якщо принаймні одна послідовність кроків для входу x приводить до допустимого заключного стану.

За заданого вхідного ланцюжка x можна вважати, що недетермінована машина Тьюрінга M паралельно виконує всі можливі послідовності кроків, доки не досягне допустимого заключного стану або доки не виявиться, що подальші кроки неможливі. Інакше кажучи, після i кроків можна вважати, що існує багато екземплярів M. Кожний екземпляр зображає конфігурацію, у якій M може виявитися після i кроків. На $(i+1)$- у кроці екземпляр C породжує j своїх екземплярів, якщо МТ, перебуваючи в конфігурації C, може вибирати наступний крок j способами.

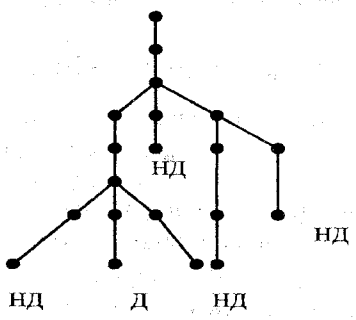


Рис. 3.1

Можливі послідовності кроків машини M на вході x організовують у вигляді дерева конфігурацій (рис. 3.1). Це означає, що можна побудувати ціле дерево, яке на заданому вході x буде передавати можливі конфігурації.

Шлях з кореня до листка – це шлях можливих кроків. Якщо σ – це найкоротша послідовність кроків, яка завершується допустимою конфігурацією, то коли машина зробила σ кроків, – вона зупиняється і допускає вхід x.

Час, затрачений на обробку входу x, дорівнює довжині послідовності σ ($|\sigma|$). Якщо для входу x жодна послідовність кроків не приводить до допустимої конфігурації, то НМТ відкидає x, а час, затрачений на обробку x, є невизначений.

Часто зручно уявити собі, що НМТ "вгадує" лише кроки з послідовності σ і перевіряє чи σ насправді закінчується допустимою конфігурацією. Та оскільки детермінована машина Тьюрінга (ДМТ) не може наперед "вгадати" допустиму послідовність кроків, то детерміноване моделювання машини потребувало б перевірки дерева всіх можливих послідовностей кроків. Цей перегляд продовжувався б до виявлення найкоротшої послідовності, яка закінчується допустимою конфігурацією. Якщо жодна послідовність кроків не приводить до допустимого входу, то детерміноване моделювання машини могло б тривати безконечно довго, якщо лише немає деякого апріорного обмеження на довжину найкоротшої допустимої послідовності. Тому природно очікувати, що НМТ здатні виконувати завдання, які жодні детерміновані машини не здатні виконати за той же час і з тією ж пам'яттю. Однак відкритим є питання про існування мов, які розпізнає недетермінована МТ з заданою часовою і ємнісною складністю, проте не розпізнає жодна детермінована МТ з такою ж складністю.

Формально НМТ визначимо так: ***k-стрічковою недетермінованою машиною Тьюрінга M*** називають сімку

$$(S, Q, q_0, q_F, I, \Lambda, \delta),$$

де значення всіх компонент ті ж, що і для детермінованої машини Тьюрінга (див. 2.5), за винятком того, що тут функція переходів δ є відображенням множини $\{Q \setminus q_F\} \times S^k$ в множину підмножин у $Q \times (S \times \{\mathrm{L}, \mathrm{R}, \mathrm{H}\})^k$.

Іншими словами, за заданим станом і списком з k символів на стрічках функція δ видає скінченну множину варіантів наступного кроку. Кожен варіант містить новий стан, k нових символів на стрічках і k зсувів головок. НМТ може вибирати будь-який з цих варіантів кроку, однак не може вибрати наступний стан з одного кроку, а символи на стрічках – з іншого.

Конфігурацію (*миттєвий опис*) для k-стрічкової НМТ визначають так само, як і для детермінованої машини Тьюрінга (ДМТ). Задана НМТ $M=(S,Q,q_0,q_F,I,\Lambda,\delta)$ робить крок, виходячи зі свого поточного стану, наприклад q, і символів, які оглядає кожна з k головок, нарпиклад $X_1,...,X_k$. З множини $\delta(q,X_1,...,X_k)$ вона вибирає деякий елемент $(r,(Y_1,D_1),...,(Y_k,D_k))$. Цей елемент означає, що новим станом повинен стати r, на i-й стрічці треба написати Y_i замість X_i і головку зсунути в напрямі $D_i, 1\le i\le k$. Якщо в разі деякого вибору наступного кроку конфігурація C переходить в конфігурацію D, то записують $C\vdash_M D$ (або просто $C\vdash D$).

Запис $C_1\vdash^* C'$ означає, що для деякого k виконується перехід $C_1\vdash C_2\vdash ...\vdash C_k=C'$, або $C_1=C'$.

***НМТ допускає ланцюжок x**, якщо існує перехід від початкового до заключного стану, тобто, якщо $(q_0x,q_0,...,q_0)\vdash^*(\alpha_1,\alpha_2,...,\alpha_k)$, де α_i містять заключний стан q_F.*

***Мовою L(M), яку розпізнає машина M,** називають множину всіх ланцюжків, які допускає M.*

НМТ має ***часову складність*** $T(n)$, якщо для кожного допустимого ланцюжка довжини n знайдеться послідовність, яка складається не більше ніж з $T(n)$ кроків і приводить до допустимого стану.

Машина має ***ємнісну складність*** $S(n)$, якщо для кожного допустимого ланцюжка довжини n існує послідовність кроків, яка приводить до допустимого стану і в якій кількість клітинок, переглянутих головкою на кожній стрічці, не перевищує $S(n)$.

У разі моделювання НМТ за допомогою детермінованої МТ складність змінює клас на "гірший". Довільну мову, яку розпізнає НМТ, розпізнає також і деяка детермінована МТ, однак у цьому випадку отримуємо збільшення складності. Найменша верхня межа для такого моделювання – це експоненціальна складність. Іншими словами, якщо $T(n)$ – "розумна" функція, що задає часову

складність ("конструйована за часом"), то для кожної НМТ M, час роботи якої обмежений функцією $T(n)$, можна знайти таку сталу c і детерміновану МТ M', що $L(M)=L(M')$ і час роботи машини M' становить $O(c^{T(n)})$.

Проте не відомо жодних нетривіальних нижніх оцінок, які стосуються складності моделювання НМТ за допомогою ДМТ. А саме невідомо жодної мови L, яку може допустити деяка НМТ з часовою складністю $T(n)$, але щодо якої можна довести, що її не розпізнає жодна ДМТ з часовою складністю $T(n)$.

Функцію $S(n)$ назвемо ***конструйованою за пам'яттю*** (ємністю), якщо деяка ДМТ M, почавши роботу над заданим входом довжини n, помістить спеціальний маркер на $S(n)$-у клітинку однієї зі своїх стрічок, переглянувши не більше $S(n)$ клітинок на кожній стрічці.

Наприклад, поліноми з цілими коефіцієнтами, 2^n, $n!$, $[n\log_2(n+1)]$ конструйовані за ємністю.

Аналогічно, як і для часової складності, якщо $S(n)$ – конструйована за ємністю функція, а M – НМТ з ємнісною складністю, що обмежена функцією $S(n)$, то знайдеться така ДМТ M' з ємнісною складністю $O(S^2(n))$, що $L(M)=L(M')$.

Якщо мову L розпізнає k-стрічкова НМТ M з часовою складністю $T(n)$, то її розпізнає й однострічкова НМТ з часовою складністю $O(T^2(n))$.

3.3. Класи $\mathcal{P}$ та $\mathcal{NP}$

Розглянемо два важливі класи мов.

Визначимо клас ***$\mathcal{P}$-TIME*** (або просто $\mathcal{P}$) *як множину всіх мов, які допускають ДМТ з поліноміальною часовою складністю*, тобто

$\mathcal{P}$–TIME={L| існують такі ДМТ M і поліном $P(n)$, що часова складність машини M дорівнює $P(n)$ і $L(M)=L$}.

*Клас **$\mathcal{NP}$-TIME*** (або просто $\mathcal{NP}$) *– це множина всіх мов, які допускають НМТ з поліноміальною часовою складністю.*

Зазначимо, що хоч класи $\mathcal{P}$ та $\mathcal{NP}$ визначені в термінах машин Тюрінга, проте можна було б скористатися будь-якою іншою моделлю обчислень (наприклад, РАМ).

Інтуїтивно клас $\mathcal{P}$ уявимо як клас задач, які можна швидко розв'язати, а $\mathcal{NP}$ – як клас задач, розв'язок яких можна швидко перевірити.

Кожна спроба прямого моделювання недетермінованого пристрою N за допомогою детермінованого пристрою D, який виконує всі можливі послідовності кроків, займає значно більше часу, ніж будь-яка одинична реалізація послідовності кроків пристрою N, оскільки D повинен простежувати роботу великої кількості копій N. На підставі результатів попереднього параграфа можемо стверджувати лише таке: якщо мова L належить до класу $\mathcal{NP}$, то її розпізнає деяка ДМТ з часовою складністю $k^{P(n)}$, де k – стала, $P(n)$ – поліном, залежний від L.

З іншого боку, не доведено, чи існують мови з $\mathcal{NP}$, які не належать $\mathcal{P}$. Отже, не відомо, чи є $\mathcal{P}$ власним підкласом класу $\mathcal{NP}$. Проте можна довести, що деякі мови не менш "важкі", ніж довільна мова з $\mathcal{NP}$, у тому сенсі, що якби у нас був детермінувантий алгоритм, який розпізнає одну з цих мов за поліноміальний час, то можна було б для кожної мови з $\mathcal{NP}$ знайти детермінований алгоритм, що розпізнає її за поліноміальний час. Такі мови називають $\mathcal{NP}$–повними.

Мову L_0 з $\mathcal{NP}$ називають ***повною для недетермінованого поліноміального часу*** (або ***$\mathcal{NP}$-повною***), *якщо за заданим детермінованим алгоритмом розпізнавання L_0 з часовою складністю $T(n) \geq n$ і довільною мовою L з $\mathcal{NP}$, можна ефективно знайти детермінований алгоритм, який розпізнає L за час $T(P_L(n))$, де P_L – деякий поліном, що залежить від L.* Говорять, що L (*поліноміально*) *зводиться* до L_0.

Термін $\mathcal{NP}$-повноти введений незалежно С. Куком (1971) та Л. Левіним (1973).

Отже, $\mathcal{NP}$-повноту мови L_0 можна довести, довівши, що L_0 належить $\mathcal{NP}$ і кожну мову $L \in \mathcal{NP}$ можна "поліноміально трансформувати" в L_0.

Мову L *називають* ***поліноміально трансформовною в L_0,*** *якщо деяка ДМТ M з поліноміально обмеженим часом роботи перетворює кожен ланцюжок w в алфавіті мови L в такий ланцюжок w_0 в алфавіті мови L_0, що $w \in L$ тоді і лише тоді, коли $w_0 \in L_0$.*

Якщо мова L трансформується в L_0, і мову L_0 розпізнає деякий детермінований алгоритм A з часовою складністю $T(n) \geq n$, то можна визначити, чи належить ланцюжок w мові L. Для цього w перетворюють у w_0 за допомогою машини M і потім застосовують A для визначення, чи належить w_0 до мови L_0. Якщо час роботи машини M обмежений поліномом $P(n)$, то $|w_0| \leq P(|w|)$. Отже, існує алгоритм, який визначає, чи належить w до мови L, за час $P(|w|) + T(P(|w|)) \leq 2T(P(|w|))$. Якби функція T була поліномом (тобто мова L_0 належала б до $\mathcal{P}$), то цей алгоритм, який розпізнає L, працював би поліноміально обмежений час, і мова L також належала б до $\mathcal{P}$.

Можна також визначити *мову L_0 як $\mathcal{NP}$-повну,* якщо вона належить до $\mathcal{NP}$ і кожна мова з $\mathcal{NP}$ поліноміально трансформується в L_0. Таке означення є вужчим, ніж попереднє, хоча не відомо, чи приводять ці два означення $\mathcal{NP}$-повних мов до різних класів.

Означення, яке ґрунтується на звідності, означає таке: якщо M_0 – детермінована машина Тьюрінга, яка розпізнає $\mathcal{NP}$-повну мову L_0 за час $T(n)$, то кожну мову з $\mathcal{NP}$ можна розпізнати за час $T(P(n))$, де P – деякий поліном, за допомогою детермінованої машини Тьюрінга, яка звертається до M_0 як до підпрограми нуль чи більше разів. Означення, яке ґрунтується на трансформовності, означає, що до M_0 можна звернутися лише один раз і потім використовувати результат її роботи наперед фіксованим способом.

Яке б з визначень ми не використовували, очевидно таке: якщо деякий детермінований алгоритм розпізнає L_0 за поліно-

міальний час, то всі мови з $\mathcal{NP}$ можна розпізнати за поліноміальний час. Отже, або всі $\mathcal{NP}$-повні мови належать до $\mathcal{P}$, або жодна з них не належить до $\mathcal{P}$. Перше справджується тоді й лише тоді, коли $\mathcal{P} = \mathcal{NP}$. На жаль, сьогодні можна висунути лише гіпотезу про те, що $\mathcal{P}$ – власний підклас класу $\mathcal{NP}$.

3.4. Мови і задачі

Ми визначили класи $\mathcal{P}$ і $\mathcal{NP}$ як класи мов. Причина цього двояка. По-перше, це спрощує систему позначень. По-друге, задачі з різних галузей, таких як теорія графів і теорія чисел, часто можна сформулювати як задачі розпізнавання мов.

Наприклад, розглянемо задачу, яка для кожного значення вхідних даних потребує відповіді "так" або "ні". Можна закодувати всі можливі значення вхідних даних такої задачі у вигляді ланцюжків і переформулювати її як задачу про розпізнавання мови, що складається з усіх ланцюжків вхідних даних нашої задачі, які приводять до відповіді "так".

Для того, щоб зв'язок *задача–мова* був зрозумілішим, задамо єдині "стандартні" зображення задач. Зокрема, приймемо такі узгодження:

1) цілі числа будемо зображати в десятковій системі;

2) вузли графа будемо зображати цілими числами 1, 2, …, n, закодованими у десятковій системі. Ребро будемо зображати ланцюжком (i_1, i_2), де i_1, i_2 – вузли, що визначають це ребро;

3) булеві вирази (формули) з n пропозиційними змінними будемо зображати ланцюжками, у яких * позначає "і", + позначає "або", $\overline{\ }$ – "не", а цілі числа 1, 2,…, n – зображають пропозиційні змінні.

Тепер можна сказати, що ***задача належить $\mathcal{P}$ або $\mathcal{NP}$, якщо її стандартне зображення належить до $\mathcal{P}$ або $\mathcal{NP}$, відповідно.***

Приклад 3.1. ***Задача про кліку для неорієнтованих графів.***

У графі G ***k-клікою*** називають k-вузловий повний підграф (кожна пара різних вузлів у цьому підграфі з'єднана ребром) графа G. Задачу про кліку формулюють так: *чи містить заданий граф G k-кліку, де k – задане ціле число?*

Мову L, яка описує задачу про кліку, утворюють такі ланцюжки вигляду

$$k(i_1, j_1)(i_2, j_2)...(i_m, j_m),$$

що граф з ребрами (i_r, j_r), $1 \le r \le m$, містить k-кліку. Це лінійне зображення заданої задачі ланцюжком.

Зазначимо, що коли йдеться про належність до класу $\mathcal{P}$ або $\mathcal{NP}$, то точний вибір мови для зображення задачі про кліку є неважливим, якщо застосовують "стандартне" кодування.

Задача про кліку належить до класу $\mathcal{NP}$. Недетермінована машина Тьюрінга спочатку може "здогадатися", які k вершин утворюють повний підграф, а потім перевірити, що довільна пара цих вершин з'єднана ребром. У цьому разі для перевірки достатньо $O(n^3)$ кроків, де n – довжина коду задачі про кліку. Це демонструє "силу" недетермінізму, бо перевірку всіх підмножин з k вершин виконують паралельно незалежні екземпляри розпізнавального пристрою.

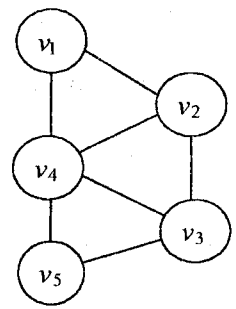


Рис. 3.2

Для прикладу розглянемо задачу про кліку для графа G, зображеного на рис. 3.2.

При k=3 цей граф можна закодувати ланцюжком

$$3\ (1,2)\ (1,4)\ (2,4)\ (2,3)\ (3,4)\ (3,5)\ (4,5). \qquad (3.1)$$

Перше число – це k, далі йдуть пари вершин, з'єднаних ребрами, причому вершина v_i зображена числом i.

Цей граф містить три 3-кліки: $\{v_1, v_2, v_4\}$, $\{v_2, v_3, v_4\}$ $\{v_3, v_4, v_5\}$.

Задача про кліку є $\mathcal{NP}$-повна. Зазначимо, що не відомі способи розв'язування задачі про кліку за детермінований поліноміальний час.

Приклад 3.2. ***Задача про виконанну булеву формулу.*** Цю задачу формулюють так: *чи виконанна задана булева формула?*

Булеву формулу $(p_1 + p_2) * p_3$ можна зобразити ланцюжком (1+2) 3, де число i зображає змінну p_i.

Булеву формулу називають ***виконанною***, якщо існує принаймні один набір значень аргументів, на якому ця формула дає істинне значення.

Розглянемо мову L, утворену всіма ланцюжками, які задають виконанні булеві формули.

Задача виконанності булевої формули є $\mathcal{NP}$-повною. Недетермінований алгоритм, який розпізнає, чи задана формула є виконанною, починає з того, що "вгадує" виконанний набір 0-1 істинних і хибних значень, для яких формула повертає істину (якщо такий набір існує). Потім відбувається підставляння "вгаданих" значень кожної змінної у вхідний ланцюжок і обчислення значень отриманого виразу, щоб перевірити його істинність.

Значення логічної функції можна обчислити за час, пропорційний до довжини виразу, багатьма алгоритмами синтаксичного аналізу. Однак навіть без них не важко обчислити значення логічної формули за $O(n^2)$ кроків, де n – довжина формули. Отже, деяка недетермінована машина Тьюрінга розпізнає виконанні булеві формули з поліноміальною часовою складністю, і тому ця задача належить до класу $\mathcal{NP}$. В 1971 р. С. Кук довів $\mathcal{NP}$-повноту цієї задачі.

Зазначимо, що недетермінізм використовували для того, щоб "паралелізувати" задачу, оскільки "вгадування" правильного

розв'язку насправді означає паралельну перевірку всіх можливих розв'язків.

Часто нас цікавлять задачі оптимізації, наприклад, знаходження найбільшої кліки в графі. Багато таких задач можна перетворити за поліноміальний час у задачі розпізнавання мови. Наприклад, найбільшу кліку в графі G можна знайти так. Нехай n – довжина зображення графа G. Для кожного k від 1 до n з'ясовуємо, чи містить граф кліку розміру k. Знайшовши таким способом розмір m найбільшої кліки, видаляємо по одній вершині доти, доки видалення чергової вершини v не зруйнує решту клік розміру m. Потім розглянемо підграф G', який складається з усіх вершин графа G, суміжних з v. Рекурсивно викликаючи цей процес, знаходимо кліку C розміру $m-1$ в G'. Найбільшою клікою для G буде $C \cup v$.

Можна перевірити, що час знаходження найбільшої кліки таким методом поліноміально залежить від довжини n зображення графа G і від часу визначення того, чи містить граф G кліку розміру k.

Приклад 3.3. ***Задача про комівояжера.*** В цій задачі задано множину міст і вартість подорожі між ними. Необхідно визначити порядок, у якому треба відвідати всі міста (по одному разу) і повернутися у початкове місто, щоб загальна вартість подорожі була мінімальною.

У термінах теорії графів список міст визначає гамільтонів цикл, що починається з базового міста і туди ж повертається після проходження через кожне місто по одному разу. Довжина гамільтонового циклу визначена сумою ваг усіх ребер, включених у нього (тобто сумою вартостей переїзду з міста в місто). Задача буде розв'язана, якщо знайдено гамільтонів цикл з найменшою довжиною.

Задача про комівояжера відома вже понад 200 років, вона має багато практичних застосувань. Наприклад, вибір найкоротшого шляху поширення інформації по всіх вузлах комп'ютерної мережі модельований нею.

Задача про комівояжера належить до класу $\mathcal{NP}$-повних задач, невідомо поліноміального алгоритму її розв'язання. В задачі з N містами необхідно розглянути $(N-1)!$ маршрутів, щоб вибрати маршрут мінімальної довжини. Тобто у випадку великих значень N неможливо за розумний час отримати результат.

Наприклад, вісім міст можна впорядкувати 40 320 можливими способами, а для десяти міст це число зростає вже до 3 628 800. Пошук найдешевшого маршруту потребує перебирання всіх цих можливостей. Припустимо, що в нас є алгоритм, здатний підрахувати вартість подорожі через 15 міст у заданому порядку. Якщо за секунду такий алгоритм здатен пропустити через себе 100 варіантів, то йому треба *більше чотирьох століть,* щоб дослідити всі можливості і знайти найдешевший маршрут. Навіть якщо в нашому розпорядженні є 400 комп'ютерів, все одно на це буде затрачено рік. Однак уже для 20 міст *мільярд* комп'ютерів повинен буде працювати паралельно протягом 9 місяців, щоб знайти найдешевший маршрут.

Чи можна знайти найдешевший маршрут, не проглядаючи їх всіх? До цього часу нікому не вдалось придумати алгоритм, який не займається, по суті, переглядом усіх шляхів. Коли кількість міст невелика, то задачу розв'язують досить швидко, однак нас цікавить розв'язок загальної задачі.

Розглянемо, як можна розв'язати задачу про комівояжера недетермінованим алгоритмом. На першому етапі "вгадування" розглядають деякий маршрут (що утворює гамільтонів цикл). Очевидно, що його можна побудувати за поліноміальний час: зберігаючи список міст, генеруємо випадковий номер і вибираємо зі списку місто з цим номером (одночасно видаляючи його зі списку, щоб воно не з'явилося вдруге). Така процедура виконується за $O(N)$ операцій, де N – кількість міст.

На другому кроці відбувається обчислення вартості маршруту. Для цього нам треба підсумувати вартості подорожей між послідовними парами міст у цьому маршруті, що теж потребує $O(N)$ операцій. Обидва кроки поліноміальні, тому задача про комівояжера належить до класу $\mathcal{NP}$.

3.5. Приклади $\mathcal{NP}$-повних задач

Можна довести, що задача, а точніше – її зображення у вигляді мови L_0, $\mathcal{NP}$-повна, довівши, що L_0 належить до $\mathcal{NP}$ і кожна мова з $\mathcal{NP}$ поліноміально трансформується в L_0. Завдяки "силі" недетермінізму належність цієї задачі до класу $\mathcal{NP}$ довести досить легко. Приклади 3.1–3.3 є типовими ілюстраціями цього кроку. Труднощі зумовлює доведення того, що кожну задачу з $\mathcal{NP}$ можна поліноміально трансформувати в задану задачу. Та оскільки $\mathcal{NP}$-повнота деякої задачі L_0 вже встановлена, то можна довести $\mathcal{NP}$-повноту нової задачі L, довівши, що L належить до $\mathcal{NP}$ і задачу L_0 можна поліноміально трансформувати в L.

Для визначення важливих $\mathcal{NP}$-повних задач стосовно неорієнтованих та орієнтованих графів розглянемо декілька означень.

Нехай $G=(V,E)$ – неорієнтований граф.

Вершинним покриттям графа G називають таку підмножину $S \subseteq V$, що кожне ребро графа G інцидентне деякій вершині з S; *гамільтоновим циклом* називають цикл у графі G, який міс-тить всі вершини з V; *k-розфарбуванням* графа G називають таке приписування натуральних чисел 1,2,...,k, (або кольорів) його вершинам, що жодні дві суміжні вершини не набувають однакового кольору. Найменшу можливу кількість кольорів у розфарбуванні називають *хроматичним числом.*

Нехай тепер $G=(V,E)$ – орієнтований граф.

Множиною вершин, які розрізають цикли, називають таку підмножину $S \subseteq V$, що кожен цикл у G містить вершину з S.

Множиною ребер, які розрізають цикли, називають таку підмножину $F \subseteq E$, що кожен цикл у G містить ребро з F.

Орієнтованим гамільтоновим циклом називають цикл у графі G, який містить усі вершини з V.

Теорема 3.1. Наведені нижче задачі належать до класу $\mathcal{NP}$:

1) ***Задача про кліку.***
2) ***Задача про виконанність булевих формул.***

3) ***Задача про вершинне покриття***: *чи існує в заданому неорієнтованому графі вершинне покриття потужністю k?*

4) ***Задача про гамільтонів цикл:*** *чи існує в заданому неорієнтованому графі гамільтонів цикл?*

5) ***Задача про розфарбовування:*** *чи існує розфарбування неорієнтованого графа в k кольорів?*

6) ***Задача про множину вершин, які розрізають цикли***: *чи існує в заданому орієнтованому графі k-елементна множина вершин, які розрізають цикли?*

7) ***Задача про множину ребер, які розрізають цикли***: *чи існує в заданому орієнтованому графі k-елементна множина ребер, які розрізають цикли?*

8) ***Задача про орієнтованний гамільтонів цикл***: *чи існує в заданому орієнтованому графі орієнтований гамільтонів цикл?*

9) ***Задача про покриття множинами***: *чи існує для заданої сім'ї множин* $S_1,S_2,...S_n$ *така підсім'я з k множин* $S_{i_1},S_{i_2},...,S_{i_k}$, *що їхнє об'єднання задовольняє такі рівності* $\bigcup_{j=1}^{k} S_{i_j} = \bigcup_{j=1}^{n} S_j$?

10) ***Задача про точне покриття***: *чи існує для заданої сім'ї множин* $S_1,S_2,...S_n$ *підсім'я множин, які попарно не перетинаються, що є покриттям?*

Належність задач 1 та 2 до класу $\mathcal{NP}$ показана в прикладах 3.1 та 3.2, відповідно. Аналогічно для кожної з решти задач можна побудувати недетермінований алгоритм з поліноміальною складністю, який "вгадує" розв'язок і перевіряє, що це справді розв'язок.

Доведено, що кожна мова з класу $\mathcal{NP}$ поліноміально трансформується в задачу виконанності; цим з'ясовано, що задача виконанності булевих формул є $\mathcal{NP}$-повна. Спираючись на це, доведено також $\mathcal{NP}$-повноту кожної задачі з теореми 3.1. Для цього визначають, що в неї прямо чи опосередковано перетворюється виконанність.

3.6. Застосування теорії $\mathcal{NP}$-повноти для аналізу задач

Однією з найскладніших проблем теорії обчислень є так звана ***проблема "$\mathcal{P}=\mathcal{NP}$"***, тобто ***чи тотожні класи $\mathcal{P}$ та $\mathcal{NP}$?*** Було зроблено багато спроб довести їхню еквівалентність чи нееквівалентність, однак сьогодні це питання відкрите.

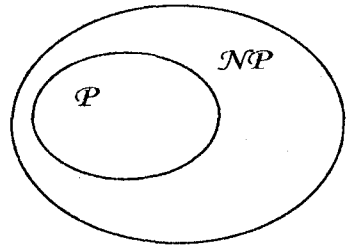


Рис. 3.2

Передбачуване співвідношення між розглянутими класами схематично можна зобразити так, як на рис. 3.2. Найсерйознішою причиною вважати, що ці класи не тотожні – це існування $\mathcal{NP}$-повних задач.

Як уже зазначено, у класі $\mathcal{NP}$ є $\mathcal{NP}$-повні (універсальні) задачі, тобто такі, що до них поліноміально зводиться довільна задача з класу $\mathcal{NP}$. Їх використовують як еталони складності.

Отже, $\mathcal{NP}$-трудною задачею є кожна задача, яка така ж важка, як і будь-яка задача з класу $\mathcal{NP}$ або важча від неї, а $\mathcal{NP}$-иовною задачею, можна вважати $\mathcal{NP}$-трудну задачу, яка не є важчою від кожної $\mathcal{NP}$ задачі. Іншими словами, задача є $\mathcal{NP}$-трудною, якщо будь-яка задача з класу $\mathcal{NP}$ до неї зводиться, а $\mathcal{NP}$-повною є задача, яка є $\mathcal{NP}$-трудною, але належить до класу $\mathcal{NP}$.

Важливою властивістю $\mathcal{NP}$-повних задач є те, що всі вони "еквівалентні" в такому сенсі: якщо хоча б для однієї з них буде доведено, що вона є легкорозв'язною, то такими будуть і решта задач з цього класу. Більшість з цих задач вивчали математики і

спеціалісти з обчислень десятиріччями і для жодної з них не вдалося знайти алгоритму поліноміальної складності. Тому природним є припущення про те, що таких поліноміальних алгоритмів не існує, а тому можна розглядати всі задачі з цього класу як важкорозв'язні.

Сформулюємо головну властивість $\mathcal{NP}$-повних задач у вигляді **теореми**.

Якщо деяка $\mathcal{NP}$-повна задача розв'язна за поліноміальний час, то $\mathcal{P}=\mathcal{NP}$. Іншими словами, якщо в класі $\mathcal{NP}$ існує задача, не розв'язна за поліноміальний час, то всі $\mathcal{NP}$-повні задачі такі ж.

Отже, гіпотеза $\mathcal{P}\neq\mathcal{NP}$ означає, що $\mathcal{NP}$-повні задачі не можуть бути розв'язані за поліноміальний час.

Доведення $\mathcal{NP}$-повноти деякої задачі є суттєвим аргументом на користь її практичної нерозв'язності. Для такої задачі доцільніше будувати достатньо точні наближені алгоритми, ніж затрачати час на пошук швидких алгоритмів, що розв'язують її точно.

"Трудність" задач можна порівнювати, зводячи одну задачу до іншої. Метод зведення є головним у доведенні $\mathcal{NP}$-повноти багатьох задач.

Неформально кажучи, ***задача Q зводиться до задачі Q',*** якщо задачу Q можна розв'язати для будь-якого входу, вважаючи відомим розв'язок задачі Q' для деякого іншого входу.

Сьогодні визначено $\mathcal{NP}$-повноту багатьох задач, еквівалентних між собою щодо поліноміальної звідності.

Великий практичний досвід розв'язуваня дискретних задач дає підстави вважати, що $\mathcal{NP}$-повні задачі й задачі з $\mathcal{P}$ сильно відрізняються за трудомісткістю розв'язування, проте в строгому сенсі ця різниця і, отже, різниця між класами $\mathcal{P}$ та $\mathcal{NP}$ не доведена. Це, зокрема, пояснюють тим, що класи $\mathcal{P}$ та $\mathcal{NP}$ визначають за допомогою поняття часу роботи обчислювального пристрою з потенційно необмеженою пам'яттю. Проте добре відомо, що час виконання алгоритму на машині погано піддається опису й аналізу загальноматематичними засобами.

Як зазначено раніше, знайти точний розв'язок задачі з класу $\mathcal{NP}$ важко, оскільки кількість можливих комбінацій вхідних значень, що потребують перевірки, надзвичайно велика. Для кожного набору вхідних значень I ми можемо створити множину можливих розв'язків PS_I. Тоді *оптимальним* уважають такий розв'язок $S_{optimal} \in PS_I$, що:

$$\text{Value}(S_{optimal}) < \text{Value}(S') \text{ для всіх } S' \in PS_I,$$

якщо ми маємо справу з задачою мінімізації, або

$$\text{Value}(S_{optimal}) > \text{Value}(S') \text{ для всіх } S' \in PS_I,$$

якщо розв'язуємо задачу максимізації.

Розв'язки, які пропонують наближені алгоритми для задач класу $\mathcal{NP}$, не будуть оптимальними, оскільки алгоритми розглядають тільки частину множини PS_I, часто досить маленьку. Якість наближеного алгоритму можна визначити, порівнюючи отриманий розв'язок з оптимальним. Деколи оптимальне значення – наприклад, мінімальний шлях комівояжера – можна знайти, і не знаючи оптимального розв'язку – самого шляху. Наприклад, якість наближеного розв'язку $A(I)$ визначають так:

$$Q_A(I) = \begin{cases} \dfrac{\text{Value}(A(I))}{\text{Value}(S_{optimal})} & \text{у задачах мінімізації;} \\ \dfrac{\text{Value}(S_{optimal})}{\text{Value}(A(I))} & \text{у задачах максимізації.} \end{cases}$$

Визначальним моментом в оцінюванні отриманого розв'язу може бути й те, чи розглядаємо ми випадки з фіксованою кількістю вхідних даних чи випадки із загальним оптимальним розв'язком. Тобто чи хочемо ми знати, наскільки добрим є наближений алгоритм на вхідних даних довжини 10 чи на вхідних списках різної довжини з оптимальним значенням 50?

Головне призначення теорії $\mathcal{NP}$-повних задач полягає в тому, щоб допомогти розробникам алгоритмів і спрямувати їхні зусилля на вибір таких підходів до розв'язування задач, які ймовірно приведуть до практично корисних алгоритмів.

Розділ 4

МЕТОДИ РОЗРОБКИ ЕФЕКТИВНИХ АЛГОРИТМІВ

Підсумовуючи матеріал попередніх розділів, зазначимо таке: якщо задача є розв'язною, то, як звичайно, вдається побудувати декілька різних за ефективністю алгоритмів її розв'язування. Для того, щоб розроблений алгоритм давав "найточніший" розв'язок задачі, необхідно мати знання про ефективні методи розробки алгоритмів, вміти застосовувати їх до розв'язування сформульованої задачі та оцінювати їхню ефективність.

Розглянемо деякі фундаментальні методи розробки алгоритмів та класичні алгоритми, побудовані з їхнім використанням.

Метод часткових цілей пов'язаний зі зведенням важкої (громіздкої) задачі до послідовності простіших задач. У цьому разі розв'язок початкової задачі отримують з розв'язків простіших задач.

Цей метод, особливо якщо його застосовувати рекурсивно, часто приводить до ефективного розв'язку задачі, підзадачі якої є її меншими версіями.

Як і більшість загальних методів розробки алгоритмів, метод часткових цілей не завжди легко перенести на конкретну задачу. Осмислений вибір простіших задач – швидше мистецтво чи справа інтуїції, ніж науки. Крім того, не існує загального набору правил для визначення класу задач, які можна розв'язати за допомогою такого підходу.

Метод підйому починається з прийняття початкового припущення або обчислення початкового розв'язку задачі. Потім відбувається якнайшвидший рух "вверх" від початкового розв'язку в напрямі до ліпших розв'язків. Коли алгоритм досягає такої точки, з якої більше неможливо рухатися вверх, алгоритм зупиняється.

Метод підйому є корисним, коли треба швидко отримати наближений розв'язок. У цьому разі, немає жодних гарантій, що кінцевий розв'язок буде оптимальним. Це й обмежує застосування методу підйому.

Наприклад, застосування цього методу для знаходження максимумів функцій декількох змінних може привести до розв'язку, який є локальним мінімумом функції.

Метод відпрацювання назад починають з цілі чи розв'язку і рухаються назад за напрямом до початкового формулювання задачі. Далі, якщо ці дії оборотні, рухаються знову від формулювання задачі до розв'язку. Цей метод широко застосовують під час розв'язування різноманітних головоломок.

Рекурсія. Процедуру, яка прямо чи опосередковано звертається до себе, називають рекурсивною. Застосування рекурсії часто дає змогу побудувати зрозуміліші та стислі алгоритми, ніж це було б зроблено без неї. Рекурсія сама по собі не приводить до ефективнішого алгоритму. Однак у поєднанні з іншими методами, наприклад "поділяй і володарюй", дає алгоритми, одночасно ефективні й елегантні.

4.1. Метод "Поділяй і володарюй"

Алгоритми вигляду "поділяй і володарюй" (ПВ) мають рекурсивну структуру: для розв'язування задачі вони рекурсивно викликають самі себе один чи декілька разів, щоб розв'язати допоміжну задачу, яка безпосередньо стосується сформульованої.

Такі алгоритми часто розробляють за допомогою методу часткових цілей: складну задачу розбивають на декілька простіших, які подібні до вихідної задачі, але мають менший обсяг;

далі ці допоміжні задачі розв'язують рекурсивним методом, після чого отримані розв'язки комбінують, щоб отримати розв'язок вихідної задачі.

У методі "поділяй і володарюй" на кожному рівні рекурсії виділяють три етапи [8].

1. Поділ задачі на декілька підзадач.

2. Рекурсивне розв'язування цих підзадач. Коли обсяг підзадачі достатньо малий, виділені підзадачі розв'язують безпосередньо.

3. Комбінування розв'язку вихідної задачі з розв'язків допоміжних задач.

Розглянемо загальний стандартний алгоритм вигляду ПВ.

Поділяй_І_Володарюй (*дані, N, розв'язок*)
якщо (*N* <= *граничнийРозмір*)
тоді **Прямий_Розв'язок** (*дані, N, розв'язок*)
в іншому випадку
Поділ_Даних (*дані, N, підмножини, розмПідмножин, M*)
для *i*=1 до *M*
повторювати **Поділяй_І_Володарюй** (*підмножини*[*i*], *розмПідмножин*[*i*], *підрозв'язки*[i])
Комбінація_Розв'язків(*підрозв'язки, M , розв'язок*)

Цей алгоритм має такі вхідні параметри: *дані* – набір вхідних даних, *N* – кількість значень у наборі. Результатом його роботи є *розв'язок*.

Алгоритм ПВ спочатку перевіряє, чи розмір задачі настільки малий (не більший, ніж *граничнийРозмір*), щоб розв'язок можна було знайти за допомогою простого алгоритму (**Прямий_Розв'язок**) і якщо це так, то відбувається його виклик.

Якщо задача занадто велика, то спочатку викликається процедура **Поділ_Даних**, яка розбиває вхідні дані на декілька менших наборів *підмножини* (їхня кількість *M*). Ці менші набори можуть бути всі одного розміру або їхні розміри можуть суттєво відрізнятися (*розмПідмножин*). Кожен з елементів початкового вхідного набору потрапляє принаймні в один з менших наборів,

однак один і той самий елемент може потрапити в декілька наборів.

Далі відбувається рекурсивний виклик алгоритму **Поділяй_І_Володарюй** на кожному з менших вхідних наборів, а процедура **Комбінація_Розв'язків** зводить отримані результати в один.

Алгоритми вигляду ПВ аналізувати досить просто, якщо кроки вашого алгоритму відповідають крокам запропонованого загального алгоритму цього виду. Якщо відомо, як ці кроки суміщаються один з одним і відома складність кожного з них, то для визначення складності алгоритму вигляду ПВ можна скористатися такою формулою:

$$T_{ПВ}(N) = \begin{cases} T_{ПР}(N), & \text{при } N < \textit{граничнийРозмір}, \\ T_{ПД}(N) + \sum_{i=1}^{M} T_{ПВ}(\textit{розмПідмножин}[i]) + T_{КР}(N), & \textit{в ін. в.} \end{cases} \quad (4.1)$$

де $T_{ПВ}$ – складність алгоритму **Поділяй_І_Володарюй**; $T_{ПР}$ – складність алгоритму **Прямий_Розв'язок**; $T_{ПД}$ – складність алгоритму **Поділ_Даних**; $T_{КР}$ – складність алгоритму **Комбінація_Розв'язків.**

Приклад 4.1. Розглянемо застосування методу ПВ для рекурсивного розв'язування задачі знаходження максимального і мінімального елементів у множині *S* з *n* елементів [1].

Найпростіший спосіб знаходження максимального та мінімального елементів полягає в тому, щоб шукати їх окремо. У цьому разі при $n \geq 2$ буде виконано $2n$-3 порівнянь елементів множини.

Застосуємо описану вище схему методу ПВ для ефективнішого розв'язування цієї задачі, шукатимемо максимальний і мінімальний елементи одночасно.

Вхідними даними алгоритму **Поділяй_І_Володарюй** будуть: множина *S*, кількість елементів *n*, а результатами – min (мінімальний) та max (максимальний) елементи. У цьому випадку:

1) розмір множини, для якої можна буде використати процедуру **Прямий_Розв'язок**, *граничнийРозмір*=2. Тобто якщо *S*

складається з двох елементів $S=\{a,b\}$, то процедура **Прямий_Розв'язок** знаходить максимальний і мінімальний елементи так:

$$\begin{cases}\min = a,\ \max = b, & \text{якщо } a \le b;\\ \min = b,\ \max = a, & \text{якщо } a > b.\end{cases}$$

2) процедура **Поділ_Даних** для заданої задачі розбиває вхідні дані на дві менші підмножини, тобто $M=2$. Для кожної з них застосовують рекурсивний виклик алгоритму **Поділяй_І_Володарюй** і отримують результати (min1,max1) та (min2,max2);

3) процедура **Комбінація_Розв'язків** об'єднує розв'язки підзадач:

$$\min = \begin{cases}\min 1, & \text{якщо } \min 1 \le \min 2;\\ \min 2, & \text{в іншому випадку;}\end{cases}$$

$$\max = \begin{cases}\max 1, & \text{якщо } \max 1 \ge \max 2;\\ \max 2, & \text{в іншому випадку.}\end{cases}$$

У випадку алгоритму знаходження найбільшого і найменшого елементів множини для оцінки кількості порівнянь з формули (4.1) отримаємо

$$T(n) = \begin{cases}1, & n = 2,\\ 2\cdot T(n/2)+2, & n > 2.\end{cases} \qquad (4.2)$$

Тут уважають, що розмір множини S *такий*: $n = 2^k$, $k \in N$.

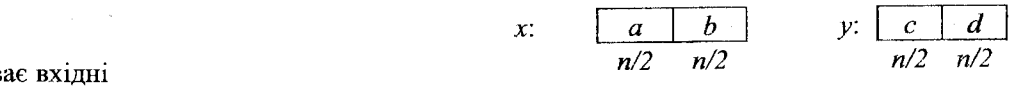
Розв'язком рекурентних співвідношень (4.2) є функція

$$T(n) = \frac{3}{2}n - 2.$$

Отже, у цьому випадку алгоритм ПВ дав змогу зменшити кількість порівнянь у фіксовану кількість разів. Цей підхід допомагає, інколи, зменшити навіть порядок зростання складності алгоритму.

Приклад 4.2. Розглянемо множення двох n розрядних двійкових чисел. Традиційний метод має оцінку $T(n)=O(n^2)$ бітових операцій. Алгоритм, який ґрунтується на принципі "поділяй і володарюй", дає $n^{\log_2 3} \approx n^{1,59}$ бітових операцій [1].

Нехай x і y – два n-розрядні двійкові числа. Вважатимемо, що n – степінь числа 2. Розіб'ємо x і y на дві однакові частини так:



Якщо розглядати кожну з цих частин як $(n/2)$-розрядне число, то добуток xy можна записати в такому вигляді:

$$x\cdot y = (a\cdot 2^{n/2}+b)(c\cdot 2^{n/2}+d) = \underset{*}{ac}\,\underset{<<}{2^n} + \underset{+}{(ad+bc)}\,\underset{<<}{2^{n/2}} + \underset{*}{bd} \qquad (4.3)$$

за допомогою чотирьох множень $(n/2)$-розрядних чисел і декількох додавань та зсувів (множень на степінь числа 2). Це дає змогу зменшити кількість операцій.

Розглянемо спочатку випадок, коли сума двох $(n/2)$-розрядних чисел є $(n/2)$-розрядне число. З урахуванням формули (4.3) добуток двох чисел можна записати за такою схемою:

$$\begin{aligned} &u = (a+b)*(c+d),\\ &v = a*c, \quad w = b*d, \end{aligned} \qquad (4.4)$$

$$x\cdot y = v*2^n + (u-v-w)*2^{n/2} + w.$$

Ця схема потребує лише три множення $(n/2)$-розрядних чисел і декілька додавань та зсувів. Для обчислення добутків u, v, w будемо застосовувати формули (4.4) рекурсивно. Додавання і зсуви займають $O(n)$ часу. Тому часова складність множення двох n-розрядних чисел обмежена зверху функцією

$$T(n) = \begin{cases}k, & n = 1,\\ 3T(\frac{n}{2}) + kn, & n > 1,\end{cases} \qquad (4.5)$$

де k – стала, що відображає додавання і зсуви в виразах, що входять в (4.4). Можна отримати розв'язок рекурентних рівнянь (4.5) у такому вигляді:

$$T(n) = 3kn^{\log_2 3} - 2kn.$$

Врахуємо випадок, коли $(a+b)$ чи $(c+d)$ мають $(n/2+1)$-розряди і тому добуток $(a+b)(c+d)$ неможливо обчислити безпосереднім рекурсивним застосуванням нашого алгоритму до задачі розміру $n/2$. В цьому випадку скористаємось таким прийомом:

$$a+b=a_1\cdot 2^{n/2}+b_1,\quad a_1=0 \text{ або } 1,$$

$$c+d=c_1\cdot 2^{n/2}+d_1,\quad c_1=0 \text{ або } 1.$$

Тоді

$$(a+b)(c+d)=a_1c_1 2^n+(a_1d_1+c_1b_1)2^{n/2}+\underbrace{b_1}_{n/2}\underbrace{d_1}_{n/2} \qquad (4.6)$$

Доданок b_1d_1 обчислюють за допомогою рекурсивного застосування (4.4) до задачі розміру $n/2$. Інші множення в (4.6) можна виконати за час $O(n)$, оскільки один з їхніх аргументів є або біт (a_1 чи c_1), або степінь числа 2.

Часова складність процедури визначена кількістю і розміром підзадач і менше – роботою, необхідною для розбиття цієї задачі на підзадачі. Оскільки рекурентні рівняння вигляду (4.2) і (4.5) часто трапляються під час аналізування рекурсивних алгоритмів типу ПВ, то наведемо розв'язок таких рівнянь у загальному вигляді.

Теорема 4.1. Нехай a, b, c – деякі невід'ємні сталі величини. Розв'язок рекурентних рівнянь

$$T(n)=\begin{cases} b, & n=1,\\ aT(n/c)+bn, & n>1,\end{cases}$$

де n – степінь числа c, має такий вигляд:

$$T(n)=\begin{cases} O(n), & a<c,\\ O(n\log_2 n), & a=c,\\ O(n^{\log_c a}), & a>c.\end{cases}$$

Із теореми 4.1 випливає, що асимптотично швидший алгоритм множення цілих чисел можна було б отримати, якби вдалося розбити вхідні цілі числа на чотири частини, щоб зуміти виразити початкове множення через 8 (або й менше) менших множень.

4.2. Евристичні алгоритми

Евристичні алгоритми визначають як алгоритми з такими властивостями [3]: алгоритм знаходить добрі, хоча не обов'язково оптимальні розв'язки; ці розв'язки можна швидше і простіше реалізувати, ніж відомий точний алгоритм, який гарантує оптимальний розв'язок.

Наприклад, якщо для розв'язку задачі всі відомі точні алгоритми потребують декількох років машинного часу, то можна прийняти довільний нетривіальний наближений розв'язок, який може бути отримано за розумний час. З іншого боку, маючи швидкий, близький до оптимального розв'язок, ми все ж можемо шукати точний розв'язок.

Не існує універсальної структури, якою можна описати евристичні алгоритми. Багато з них ґрунтується або на методі часткових цілей, або на методі підйому. Один із загальних підходів до побудови евристичних алгоритмів полягає в переліченні всіх вимог до розв'язку і поділу їх на два класи – ті, які легко задовольнити (тобто ті, які повинні бути задоволені), і ті, які важко задовольнити (і щодо яких можна піти на компроміс). Тоді будують алгоритм, який гарантує виконання умов першого класу, але не обов'язково умов другого класу.

Часто, добрі алгоритми потрібно розглядати як евристичні. Наприклад, нехай для деякої задачі ми побудували алгоритм, який працює на всіх тестах, проте не вдалося довести, що він правильний. Тоді про цей алгоритм не можна говорити як про точний у сенсі оптимальності, тобто він евристичний.

Приклад 4.3. ***Задача про комівояжера.*** Нехай задано N міст та матрицю вартостей C, елементи якої c_{ij} дорівнюють вартості подорожі з міста i в місто j. Нагадаємо (див. 3.4), що, починаючи з деякого міста u, необхідно відвідати всі міста (лише один раз) і повернутися в початкове місто. Отриманий таким способом шлях називають туром [3]. Задача полягає в знаходженні туру мінімальної вартості.

Як уже зазначено, у термінах теорії графів шуканий список міст визначає гамільтонів цикл і задача про комівояжера полягає в знаходженні гамільтонового циклу з найменшою довжиною.

Розглянемо евристичний алгоритм розв'язування задачі про комівояжера.

Алгоритм GTS [3]. Побудувати наближений розв'язок TOUR з вартістю COST для задачі комівояжера з N містами і таблицею вартостей C, починаючи з вершини u.

1. *Ініціалізація.* Помітимо вершину u як "використану", а всі інші вершини – "не використані". Вершина v визначає положення на графі в заданий момент: $v=u$.

2. *Відвідування всіх міст.* Для k від 1 до N-1 виконувати крок 3.

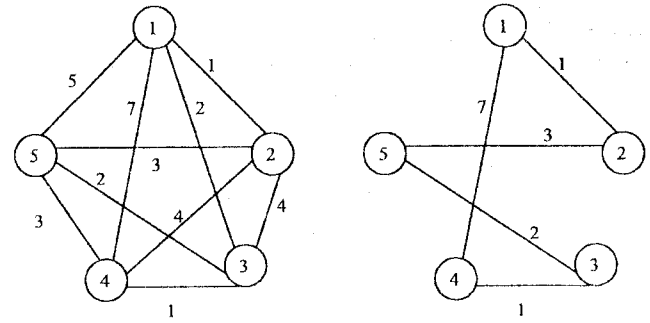
3. *Вибір наступної вершини.* Нехай ребро (v,w) – це ребро з найменшою вагою, яке веде з v в будь-яку невикористану вершину w. Враховуємо вартість подорожі з міста v в місто w: COST=COST+C_{vw}, та додаємо вершину w до розв'язку: TOUR=TOUR+(v,w). Помічає w як "використану" вершину: $v=w$.

4. *Завершення туру.* Додаємо початковий пункт u до розв'язку TOUR=TOUR+(v,u) і враховуємо вартість подорожі з v в u: COST=COST+$C(v,u)$.

Розглянемо роботу цього алгоритму на прикладі. Нехай задано п'ять міст і матрицю вартостей:

C	1	2	3	4	5
1	-	1	2	7	5
2	1	-	4	4	3
3	2	4	-	1	2
4	7	4	1	-	3
5	5	3	2	3	-

Граф, що моделює цю задачу, зображено на рис. 4.1, *а*. На рис. 4.1, *б* показано тур, отриманий алгоритмом GTS, що починається з вершини 1. Вартість цього туру 14. Проте ліпшим є тур вартості 13 (1-5-3-4-2-1). Тобто алгоритм GTS не завжди знаходить тур з мінімальною вартістю.



а *б*

Рис. 4.1

Алгоритм GTS ґрунтується на ідеї підйому. Його мета – знайти тур мінімальної вартості. Задача зведена до набору часткових цілей: знайти на кожному кроці "найдешевше" місто, щоб відвідати його наступним. Алгоритм не будує плану наперед; поточний вибір відбувається безвідносно до наступних виборів.

Оптимальний розв'язок задачі про комівояжера має дві головні властивості:

1) він складається з множини ребер, які разом утворюють гамільтонів цикл;

2) вартість жодного іншого туру не буде менша від вартості туру-розв'язку.

Наш евристичний алгоритм розглядає властивість 1 як обов'язкову, а властивість 2 як необов'язкову для виконання.

Приклад з п'ятьма містами засвідчує, що алгоритм GTS не гарантує властивості 2. Проте на кроці 3 відбувається деяка спроба зменшити вартість туру.

Оцінимо цей алгоритм. Для довільної задачі про комівояжера з n містами треба $O(n^2)$ операцій, щоб зчитати чи побудувати матрицю вартостей C. Тому нижня межа складності довільного алгоритму, спроможного дати нетривіальний можливий розв'язок цієї задачі, дорівнює $O(n^2)$. Легко побачити, що для довільної розумної реалізації кроків 2–4 потрібно не більше $O(n^2)$ операцій. Тому алгоритм GTS є настільки швидкий, наскільки це можливо.

Алгоритм GTS – це достатньо "добрий" евристичний алгоритм для великих n. Якість алгоритму можна значно поліпшити такою простою модифікацією. Найгіршою властивістю алгоритму є те, що вибір ребер дуже малої ваги в разі виконання кроку 3 на ранніх або середніх стадіях роботи алгоритму може привести до вибору дуже "важких" ребер на кінцевій стадії.

Один зі способів уникнути цього — застосовувати алгоритм для кожного з $i \le n$ різних випадково вибраних початкових міст. Інша модифікація може полягати у повторенні алгоритму для того ж початкового міста, але треба починати з другого за вагою ребра і потім, можливо, повернутись до проходження ребер з найменшою вагою для наступних вершин. Далі можна вибрати найдешевший з розглянутих турів.

Ще один варіант: зберігати лише найкоротший тур з тих, які знайдено, та відкидати частково побудовані тури, вартість яких уже перевищує вартість поточного найкоротшого туру. Очевидно, складність такого модифікованого алгоритму може досягти порядку $O(pn^2)$, де p – деяка константа.

4.3. Метод гілок і меж

Метод гілок і меж орієнтований головно на розв'язування задач оптимізації. Він досліджує деревоподібну модель простору розв'язків задачі та дає змогу серед елементів множини можливих розв'язків знайти найліпший (найоптимальніший). Головна властивість, яку повинна мати ця множина, – це можливість розділяти її на підмножини, що не перетинаються, для яких можна виконати деяку оцінку "оптимальності" можливого розв'язку. "Оптимальність" такого розв'язку означає, що не існує ліпшого розв'язку в межах вибраної підмножини (насправді такої оптимальності на цій підмножині може й не бути).

У разі практичного застосування цього методу всі можливі варіанти розв'язків задачі розбивають на класи (блоки), виконують оцінку знизу (у випадку мінімізації) для всіх розв'язків з одного класу, і якщо вона більша від раніше отриманої, то відкидають усі варіанти з цього класу.

Розглянемо роботу цього алгоритму на прикладі розв'язування задачі про комівояжера. Під час розв'язування цієї задачі алгоритмом гілок і меж визначають числову функцію вартості для кожної вершини, яка з'являється в дереві пошуку. Мета – знайти конфігурацію, на якій функція вартості досягає мінімального значення. У цьому разі множину можливих розв'язків становитимуть усі допустимі тури, а оцінками будуть їхні вартості. Найліпшим розв'язком уважатимемо той, що має найменшу вартість.

Два головні кроки в цьому алгоритмі – це *галуження* та *обчислення меж*.

Розглянемо ***галуження***. Корінь нашого пошукового дерева відповідатиме множині всіх можливих турів. У загальному випадку для довільної асиметричної задачі з N містами, корінь буде відображати повну множину всіх $(N\text{-}1)!$ можливих турів. Гілки, що виходять з кореня, визначені вибором одного ребра, наприклад, (i, j). Нашим завданням є розділити множину всіх турів на дві підмножини: одна, яка, досить імовірно, містить оптимальний тур, та інша, яка, імовірно, не містить його. Для цього вибираємо ребро (i, j) і розділяємо множину всіх турів на дві підмножини: $\{i, j\}$ – множина турів, які містять ребро (i, j), і $\overline{\{i, j\}}$ – множина турів, які не містять ребра (i, j).

Шлях від кореня до довільної вершини дерева виділяє певні ребра, які повинні чи не повинні бути включені в множину, відображену вершиною дерева.

Розглянемо ***обчислення меж***. З кожною вершиною дерева ми зв'язуємо нижню межу вартості довільного туру з множини, яку відображає ця вершина. Обчислення цих нижніх меж – головний чинник, який дає економію зусиль у будь-якому алгоритмі типу гілок і меж.

Обчислення нижніх меж відбувається у процесі зведення матриці вартостей.

Зведенням рядка (*чи стовпця*) матриці вартостей C називають віднімання одного і того ж числа h від всіх елементів цього рядка (чи стовпця). Зазначимо, що вартість кожного туру при новій матриці C' буде точно на h менша, ніж вартість того ж туру при матриці C. Це пояснюють тим, що кожен тур повинен містити

ребро з заданого рядка (виїзд з цього міста) чи заданого стовпця (в'їзд у це місто), тому і вартість усіх турів зменшиться на h.

Нехай t – оптимальний тур при матриці вартостей C. Тоді вартість туру t

$$z(t) = \sum_{(i,j)\in t} c_{ij} .$$

Якщо C' отримуємо з C зведенням рядка (чи стовпця), то t повинен бути оптимальним туром і при C', а

$$z(t) = h + z'(t) ,$$

де $z'(t)$ – вартість туру t при C'.

Зведення всієї *матриці* виконують послідовним зведенням усіх рядків, а потім усіх стовпців матриці С. У цьому разі від кожного елемента рядка (стовпця) віднімають найменший елемент h_i з цього рядка (стовпця).

Нехай

$$h = \sum_{\substack{\text{всі рядки}\\ \text{і стовпці}}} h_i .$$

Отриману в підсумку матрицю вартостей називають *зведеною* з C. Нижня межа вартості будь-якого туру з цієї матриці дорівнює h.

Якщо внаслідок зведення ми отримали матрицю, в якій є вибір по одному нулю в кожному стовпці та рядку, то тим самим ми знайшли шлях комівояжера з вартістю h.

Розглянемо приклад.

Приклад 4.4. Нехай задано шість міст і несиметричну матрицю вартостей C, елементи якої дорівнюють вартості переїзду між містами:

-	7	3	10	17	5
9	-	4	5	8	6
13	2	-	9	11	14
5	8	6	-	3	6
16	11	13	10	-	8
6	5	9	8	4	-

Після зведення матриці по рядках (h_1=3, h_2=4, h_3=2, h_4=3, h_5=8, h_6=4) та стовпцях (h_6=2, h_7=0, h_8=0, h_9=1, h_{10}=0, h_{11}=0) отримаємо матрицю C':

-	4	**0**	6	14	2
3	-	0	**0**	4	2
9	**0**	-	6	9	12
0	5	3	-	**0**	3
6	3	5	1	-	**0**
0	1	5	3	0	-

Сумарне зведення становить 27, тобто всі шляхи комівояжера зменшились на 27. Оскільки в підсумку ми отримали матрицю, у якій є вибір по одному нулю в кожному стовпці та рядку (вони виділені в матриці), а одержаний шлях утворює тур комівояжера вартості 27, то ми знайшли оптимальний тур. Це тур (1-3-2-4-5-6-1) з мінімальною вартістю, кожен інший тур матиме більшу вартість.

У цьому випадку вдалось досягнути мінімальної вартості 27 на конкретному турі. Очевидно, що не обов'язково після зведення отримують тур з ребрами нульової вартості.

Розглянемо інший приклад. Нехай задано матрицю вартостей C:

-	1	2	3	4	5
2	-	3	4	5	6
3	4	-	5	6	7
7	6	5	-	4	3
6	5	4	3	-	2
5	4	3	2	1	-

Після зведення матриці (сума констант зведення дорівнює 14) отримаємо таку матрицю C':

-	$\mathbf{0}^1$	0^0	1	3	4
0^0	-	**0**	1	3	4
0	1	-	1	3	4
4	3	1	-	1	0
4	3	1	**0**	-	0^0
4	3	1	0^0	**0**	-

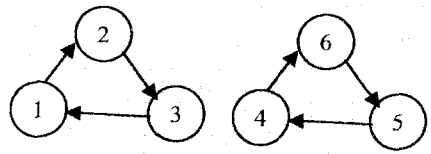


Рис. 4.2

Виділені нулі дають шлях (рис. 4.2), але це не тур комівояжера (його ребра не утворюють гамільтонів цикл).

Розглянемо нулі в зведеній матриці C', а саме – нуль у позиції (1, 2). Він означає, що вартість переїзду з першого в друге місто тепер дорівнює нулю. Однак якщо вилучити (заборонити) цей переїзд, то в'їжджати в друге місто все одно треба. Найдешевше з другого міста в'їжджати (шукаємо у другому стовпці) в третє місто. А з першого міста найдешевше їхати в третє – нульова вартість. Обчислюємо суму цих мінімумів: 1+0=1.

Суть цієї одиниці в такому: якщо не їхати з першого міста в друге, то доведеться додатково заплатити не менше 1. Ця оцінка нуля визначена в матриці верхнім індексом. Зробимо оцінки всіх нулів і виберемо елемент з максимальною оцінкою. Якщо таких нулів декілька, то вибираємо будь-який.

Далі виконуємо галуження. Множину всіх турів комівояжера розбиваємо на дві підмножини: 1) $\{1,2\}$, що включає всі тури, які містять ребро (1,2); і 2) $\overline{\{1,2\}}$, що включає всі тури, які не містять ребра (1,2). Для другої підмножини оцінка знизу збільшується на одиницю, тобто дорівнює 15. Тому далі досліджуємо підмножину $\{1,2\}$, з меншою оцінкою 14.

Для турів з підмножини $\{1,2\}$ продовжуємо працювати з матрицею C'. Вилучаємо (викреслюємо) перший рядок і другий стовпець. Крім того, на місце (2,1) ставимо прочерк, що означає заборону відповідного переїзду. Після цього матриця C'' набуде такого вигляду:

	1	3	4	5	6
2	-	0^2	1	3	4
3	0^5	-	1	3	4
4	4	1	-	1	0^1
5	4	1	0^0	-	0^0
6	4	1	0^0	0^1	-

Ця скорочена матриця є вже зведеною (мінімальні елементи у всіх рядках і стовпцях дорівнюють нулю), тому оцінка знизу для першої підмножини залишається 14.

Знову виконаємо крок галуження. Оцінимо нулі в скороченій матриці та виберемо нуль, що відповідає переїзду з третього міста в перше. Далі, аналогічно, розіб'ємо досліджувану підмножину $\{1,2\}$ на дві – $\{3,1\}$, що містить усі тури, які включають ребро (3,1), і $\overline{\{3,1\}}$, яка містить усі тури без ребра (3,1). Для множини $\overline{\{3,1\}}$ оцінка знизу буде 14+5=19.

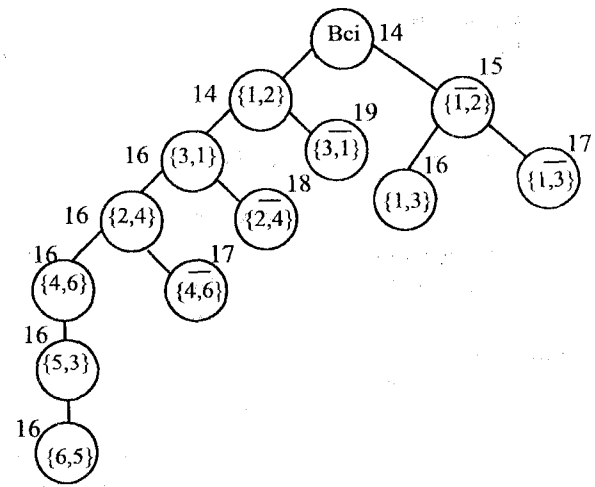


Рис. 4.3.

Продовжуємо алгоритм, для множини $\{3,1\}$ – вилучаємо другий стовпець і третій рядок, на місце (2, 3) в таблиці пишемо заборону на переміщення. Отримана матриця допускає крок зведення. Віднімаємо одиницю з елементів першого рядка і стовпця. Нижня оцінка маршрутів цього підкласу зросте на 2: 14+2=16.

Наступний крок галуження приводить до матриці ще меншого розміру. Весь процес триває доти, доки не отримаємо матрицю розміром 2×2. Тоді множина, що відповідає цій матриці, міститиме не більше ніж два тури.

Весь процес роботи за розглянутою схемою зображено на рис. 4.3. Вершини цього дерева відображають підмножини розв'язків. Числа біля вершин – це оцінки знизу для турів з заданої підмножини.

Унаслідок продовження алгоритму, отримаємо першу оцінку – 16 для туру 1-2-4-6-5-3-1. Усі підмножини розв'язків, у яких оцінка знизу є не менше 16, вилучено з розгляду. Залишилася тільки підмножина $\{\overline{1,2}\}$ без переїзду з першого міста в друге з оцінкою 15. Однак після першого ж галуження отримують підмножини з оцінками 16 та 17, які вилучаємо з розгляду. Отже, маршрут 1-2-4-6-5-3-1 з вартістю 16 і буде найліпшим туром комівояжера.

4.4. Динамічне програмування

Р. Беллман увів поняття динамічного програмування для характеристики алгоритмів, які діють залежно від зміни обставин. Методи оптимізації з елементами динамічного програмування були відомі здавна, однак саме Р. Беллман дав строге математичне обґрунтування цієї області.

Динамічне програмування, як звичайно, застосовують до задач оптимізації, у яких для того, щоб отримати оптимальний розв'язок, необхідно зробити певну множину виборів. Після того, як вибір зроблено, часто виникають допоміжні підзадачі, такі ж за виглядом, як і основа.

Динамічне програмування ефективне тоді, коли певна допоміжна підзадача виникає внаслідок декількох варіантів виборів. Головний метод розв'язування таких задач полягає у збереженні розв'язку кожної підзадачі, яка може виникнути повторно. Це гарантує, що один раз розв'язана підзадача вдруге не буде розв'язуватись. Завдяки цьому певні алгоритми, час роботи яких експоненціально залежить від вхідних даних, вдається замінити алгоритмом з поліноміальним часом роботи.

Процес розробки алгоритмів динамічного програмування можна розбити на чотири етапи [8].

1. Опис структури оптимального розв'язку.

2. Рекурсивне визначення значення, що відповідає оптимальному розв'язку.

3. Обчислення значення, що відповідає оптимальному розв'язку, за домогою методу висхідного аналізу.

4. Утворення оптимального розв'язку на підставі інформації, отриманої на попередніх етапах.

Етапи 1–3 є основою методу динамічного програмування. Четвертий етап можна опустити, якщо треба знайти лише значення, що відповідає оптимальному розв'язку. На цьому етапі деколи використовують додаткову інформацію, отриману на третьому етапі, що полегшує конструювання оптимального розв'язку.

Для того, щоб до задачі оптимізації можна було застосувати метод динамічного програмування, у ній повинні бути такі два інградієнти: *оптимальна підструктура* та *допоміжні програми, що перекриваються.*

Оптимальна підструктура виявляється в задачі в тому випадку, якщо в її оптимальному розв'язку містяться оптимальні розв'язки допоміжних підзадач. Друга складова частина полягає в тому, що простір допоміжних задач повинен бути "невеликим" у тому сенсі, що внаслідок виконання рекурсивного алгоритму одні й ті ж допоміжні задачі розв'язуються знову і знову, а нові підзадачі не виникають.

Приклад 4.5. ***Задача про перемноження матриць*** [2, 3]. У разі потреби перемножити послідовність матриць різних розмірів

порядок множення може суттєво впливати на ефективність усієї процедури. Розглянемо обчислення добутку n матриць:

$$A = A_1 * A_2 * \ldots * A_n, \qquad (4.7)$$

де A_i – матриця розміру $r_{i-1} \times r_i$. Розміщення матриць у (4.7) впливає на загальну кількість операцій, необхідних для обчислення добутку, незалежно від алгоритму, який застосовують для множення матриць.

Наприклад, для знаходження добутку чотирьох матриць A_1, A_2, A_3, A_4, відповідно, розміру 20×5, 5×35, 35×4 і 4×25, існує п'ять суттєво різних порядків множення, які потребують від 3 100 до 24 500 операцій множення (вважають, що множення (p×q)-матриці на (q× r)-матрицю потребує pqr операцій).

Справді, добуток $A = (A_1 * A_2) * (A_3 * A_4)$ потребує 24 500 операцій, а добуток $A = (A_1 * (A_2 * A_3)) * A_4$ – лише 3 100 операцій.

Задачу про перемноження послідовності матриць можна сформулювати так: для заданої послідовності матриць ($A_1, A_2, \ldots, A_n$), у якій матриця A_i, $i = 1,2,\ldots n$, має розмір $r_{i-1} \times r_i$, за допомогою дужок повністю визначити порядок множення в (4.7), за якого кількість скалярних множень зведеться до мінімуму.

Зазначимо, що саме перемноження матриць у задачу не входить. Наша мета – визначити оптимальний порядок перемноження. Зазвичай, час, затрачений на знаходження оптимального способу перемноження матриць, повністю окуповується, коли виконується саме множення.

Процес перебирання всіх можливих порядків обчислення добутку n матриць для мінімізації кількості операцій має експоненціальну складність, що за великих n є практично неприйнятним. Однак динамічне програмування приводить до алгоритму складності $O(n^3)$. Розглянемо цей алгоритм.

Нехай M – матриця, складена з елементів m_{ij} мінімальних складностей обчислення добутку $A_i * A_{i+1} * \ldots * A_j$, $1 \le i \le j \le n$. Очевидно, що

$$m_{ij} = \begin{cases} 0, & \text{якщо } i = j; \\ \min\limits_{i \le k < j} (m_{ik} + m_{k+1,j} + r_{i-1} r_k r_j), & \text{якщо } j > i. \end{cases} \qquad (4.8)$$

Тут m_{ik} – мінімальна складність обчислення $A' = A_i * A_{i+1} * \ldots * A_k$, $m_{k+1,j}$ – мінімальна складність обчислення $A'' = A_{k+1} * A_{k+2} * \ldots * A_j$. Третій доданок дорівнює складності множення A' на A''.

Якщо для обчислення величин m_{ij} побудувати рекурсивний алгоритм, що ґрунтується на рекурентному співвідношенні (4.8), то його складність буде експоненціальною.

Після аналізування формули (4.8) побачимо, що маємо досить мало підзадач: по одній для кожного вибору величин i та j, які задовольняють нерівність $1 \le i \le j \le n$. У рекурсивному алгоритмі кожна допоміжна задача може неодноразово траплятися в різних гілках рекурсивного дерева. Така властивість перекриття підзадач – одна з головних рис застосовності методу динамічного програмування.

Замість того, щоб рекурсивно розв'язувати (4.8), використаємо підхід динамічного програмування й обчислимо оптимальну вартість шляхом побудови таблиці у висхідному напрямі. Тобто величини m_{ij} обчислюються в порядку зростання різниць нижніх індексів. Починають з обчислення m_{ii} для всіх i, а далі $m_{i,i+1}$ для всіх i, потім $m_{i,i+2}$ і так далі. У цьому разі m_{ik} і $m_{k+1,j}$ у (4.8) будуть вже обчислені, коли ми прийдемо до обчислення m_{ij}. Це випливає з того, що при $i \le k < j$ різниця $j - i$ повинна бути більшою від k-i, а також від $j - (k+1)$.

У разі практичної реалізації цього алгоритму, крім значень матриці M, варто ще формувати допоміжну матрицю S [1..n,1..n], в яку заносять індекси k, за яких досягають оптимальних вартостей m_{ij}. Матриця S буде використана під час побудови оптимального розв'язку.

Проілюструємо описаний вище процес на прикладі послідовності з n=6 матриць, розміри $r_0, r_2, ..., r_6$ дорівнюють, відповідно, 30, 35, 15, 5, 10, 20, 25.

Таблиця 4.1

Матриця *M*

1	2	3	4	5	6	
0	15750	7875	9375	11875	15125	1
	0	2625	4375	7125	10500	2
		0	750	2500	5375	3
			0	1000	3500	4
				0	5000	5
					0	6

Оскільки величини m_{ij} визначені тільки при $i \le j$, то використовують лише частину матриці *M*, розміщену над її головною діагоналлю (табл. 4.1).

Таблиця 4.2

Матриця *S*

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

Те саме стосується і матриці *S* (табл. 4.2). Після обчислення елементів матриці *M* знаходимо, що мінімальна кількість скалярних множень шести матриць $m_{16} = 15125$.

Оптимальний розв'язок неважко побудувати, використовуючи матрицю *S*. В кожному елементі s_{ij} зберігається значення індексу k, де в разі оптимального розміщення дужок у послідовності $A_i A_{i+1} ... A_j$ відбувається розбиття. Отже, оптимальне обчислення

добутку матриць виводиться з такого рекурентного співвідношення:

$$(A_i * ... * A_{s[i,j]}) * (A_{s[i,j]+1} * ... * A_j). \qquad (4.9)$$

Перший раз співвідношення (4.9) записують для i=1, j=n, далі для оптимального розміщення дужок у добутках $A_i * ... * A_{s[i,j]}$ та $A_{s[i,j]+1} * ... * A_j$ знову застосовують формулу (4.9) і т. д.

Для нашого випадку оптимальне розміщення дужок отримуємо з матриці *S*: $((A_1 * (A_2 * A_3)) * ((A_4 * A_5) * A_6))$.

4.5. Жадібні алгоритми

Для розв'язування багатьох задач оптимізації застосовують так звані жадібні алгоритми. Ці алгоритми діють, використовуючи в кожен момент лише частину вхідних даних, і намагаються зробити найліпший вибір на підставі доступної інформації. Тобто відбувається локально оптимальний вибір у надії, що він приведе до оптимального розв'язку глобальної задачі.

Розглянемо простий приклад – задачу про видачу решти: щоб звести до мінімуму кількість монет, необхідних для видачі певної суми, застосовуючи жадібний алгоритм, достатньо кожного разу вибирати монету найбільшої вартості, яка не перевищує ту суму, яку залишилося видати.

Жадібні алгоритми не завжди приводять до оптимального розв'язку, однак є досить широкий клас задач, де вони дають ефективний результат.

У 1950 р. Е. Дейкстра і Р. Прім, працюючи незалежно, запропонували жадібний алгоритм побудови мінімального каркаса зв'язного зваженого графа, який отримав назву алгоритм Дейкстри–Пріма. Алгоритм Дейкстри знаходження найкоротшого шляху в графі та підхід Чватала до задачі про покриття множини також належать до жадібних алгоритмів.

Приклад 4.6. ***Алгоритм Дейкстри–Пріма знаходження мінімального каркаса.*** *Мінімальним каркасом* зв'язного зваженого

графа G називають його зв'язний підграф, який є деревом, що містить усі вершини графа G, причому сума ваг його ребер є мінімальною.

Задача знаходження мінімального каркаса – це оптимізаційна задача, для знаходження розв'язків якої застосовують жадібний алгоритм [8].

Алгоритм Дейкстри–Пріма полягає в тому, що на кожному кроці розглядають множину ребер (так зване оточення), які допускають приєднання до вже побудованої частини каркаса, і з неї вибирають ребро з найменшою вагою. Повторюючи цю процедуру, ми отримаємо каркас з найменшою вагою.

Розіб'ємо вершини графа на три класи: 1) вершини, які ввійшли у вже побудовану частину дерева; 2) вершини, які оточують побудовану частину (оточення); 3) вершини, які ще не розглядали.

Почнемо з довільної вершини графа і включимо її в каркас. Оскільки результатом побудови буде некореневе дерево, то вибір початкової вершини не впливає на кінцевий результат (за умови, що мінімальний каркас єдиний). Усі вершини, які з'єднані з заданою, позначаємо як оточення. Далі виконується цикл пошуку ребра з найменшою вагою, яке з'єднує вже побудовану частину каркаса з оточенням; це ребро разом з новою вершиною додають у дерево, і відбувається оновлення оточення. Після того, як у дерево потраплять всі вершини, робота буде завершена.

На рис. 4.4 проілюстровано виконання алгоритму на конкретному прикладі. Вихідний граф зображено на рис. 4.4, *а*. Ми починаємо побудову каркаса з вершини A. Всі вершини, безпосередньо зв'язані з A, утворюють вихідне оточення (див. рис. 4.4, *б*, штрихові лінії ведуть до вершин оточення).

На першому кроці знаходимо, що ребро найменшої ваги зв'язує вершини A і B, тому до каркаса додаємо вершину B і ребро AB. Унаслідок до оточення треба додати вершини E і G на наступному кроці. Треба також перевірити, чи є ребра, що ведуть з вершини A в C, D і F, найкоротшими серед ребер, які з'єднують ці вершини з деревом, чи є зручніші ребра, які виходять з B. Так

знаходимо, що ребро BD менше, ніж ребро AD, тому повинно його замінити (див. рис. 4.4, *в*, суцільні лінії зв'язують вершини каркаса).

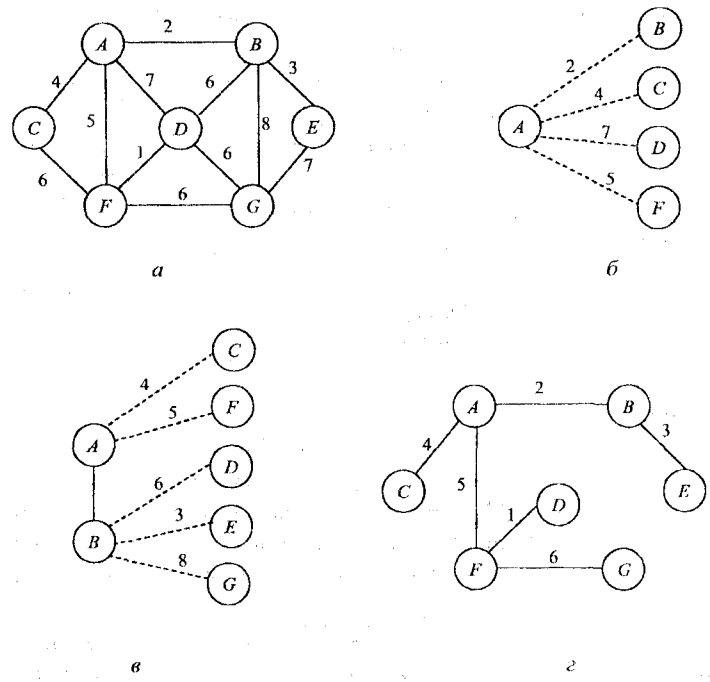


Рис. 4.4

Після продовження, отримаємо, що найменшу вагу з п'яти ребер, які ведуть в оточення, має ребро BE, тому до дерева додаємо вершину E. Вага ребра EG менша від ваги ребра BG, тому воно заміщає ребро BG. І так далі.

У підсумку ми отримаємо мінімальний каркас з коренем у вершині A (див. рис. 4.4, *г*).

Як визначити, чи здатен жадібний алгоритм розв'язати конкретну задачу оптимізації? Загального шляху не існує, однак

можна виділити дві головні складові задач, до яких застосовні жадібні алгоритми — властивість жадібного вибору і наявність оптимальної підструктури. Якщо вдається продемонструвати, що задача має дві ці властивості, то з великою імовірністю для неї можна розробити жадібний алгоритм.

Перша з названих – *властивість жадібного вибору*: глобальний оптимальний розв'язок можна отримати, використовуючи локальний оптимальний (жадібний) вибір.

У жадібному алгоритмі відбувається вибір, який виглядає в цей момент найліпшим, після чого розв'язують підзадачу, що виникає внаслідок цього вибору. Отже, на відміну від динамічного програмування, де підзадачі розв'язують у висхідному порядку, жадібна стратегія звичайно розгортається у низхідному порядку, коли жадібний вибір відбувається один за одним, унаслідок чого кожен екземпляр задачі зводиться до простішого.

Властивість жадібного вибору часто дає певну перевагу, яка дає змогу підвищити ефективність вибору в підзадачі. Наприклад, якщо в задачі про видачу решти попередньо відсортувати монети в порядку незростання вартостей монет, то кожну з них достатньо розглянути лише один раз.

Часто виявляється, що завдяки попередньому опрацюванню вхідних даних або застосуванню відповідної структури даних (як звичайно це черга з пріорітетами) можна прискорити процес жадібного вибору, що приведе до підвищення ефективності алгоритму.

Оптимальна підструктура виявляється в задачі, якщо в її оптимальному розв'язку міститься оптимальний розв'язок підзадач. Ця властивість є головною ознакою застосовності як динамічного програмування, так і жадібних алгоритмів.

Отже, якщо для деякої задачі вдається довести, що, використовуючи жадібний вибір, у вихідній задачі можна отримати підзадачу, і оптимальний розв'язок підзадачі в сукупності з раніше зробленим жадібним вибором приводить до оптимального розв'язку вихідної задачі, то цим буде доведено, що для цієї задачі може бути ефективно застосований жадібний алгоритм.

Приклад 4.7. ***Задача про ранець.*** Продемонструємо відмінності у застосуванні алгоритму динамічного програмування та жадібного алгоритму на прикладі двох різновидів класичної задачі оптимізації – дискретної та неперервної задачі про ранець [8].

Дискретну задачу про ранець формулюють так. Нехай задано n різних предметів і ранець з вантажопідіймальністю W (W – ціла величина). Предмет під номером i (i=1,...,n) має вартість v_i і вагу w_i, де v_i і w_i – цілі числа. Необхідно запакувати ранець так, щоб сумарна вартість запакованих предметів була якнайбільшою.

Формулювання *неперервної задачі про ранець* таке ж, як і дискретної, однак тепер той чи інший предмет ми можемо складати в ранець частинами (тобто кожен предмет є подільним).

В обох різновидах задачі про ранець виявляється властивість оптимальної підструктури. Справді, розглянемо в цілочисловій задачі найбільш цінне завантаження, вага якого не перевищує W. Якщо витягнути з ранця предмет під номером j, то решта предметів повинна бути найціннішою, вага яких не перевищує $W - w_j$ і які можна скласти з n-1 вихідних предметів, з множини яких вилучено предмет під номером j. Для аналогічної неперервної задачі можна застосувати такі ж міркування.

Щоб розв'язати неперервну задачу, обчислимо спочатку вартість одиниці ваги кожного предмета (питому вартість) V_i/w_i. Згідно з жадібною стратегією, спочатку необхідно завантажити якнайбільше предметів з максимальною питомою вартістю. Якщо запас цього предмета вичерпується, а вантажопідіймальність ранця – ні, то завантажуємо далі якнайбільше предмета, питома вартість якого буде другою за значенням. Так триває доти, доки вага завантажених предметів не досягне допустимого максимуму. Отже, разом із сортуванням предметів за питомою вартістю час роботи алгоритму буде O($n$lg$n$).

Легко довести, що неперервна задача про ранець має властивість жадібного вибору, а дискретна – ні.

Щоб переконатися, що подібна жадібна стратегія не працює в цілочисловій задачі про ранець, розглянемо приклад, проілюстрований на рис. 4.5, *а*. Нехай маємо три предмети і ранець,

здатний вмістити 50 кг. Маса першого предмета – 10 кг, він коштує 60 грн., другого предмета – 20 кг і 100 грн., третього предмета – 30 кг і 120 грн.

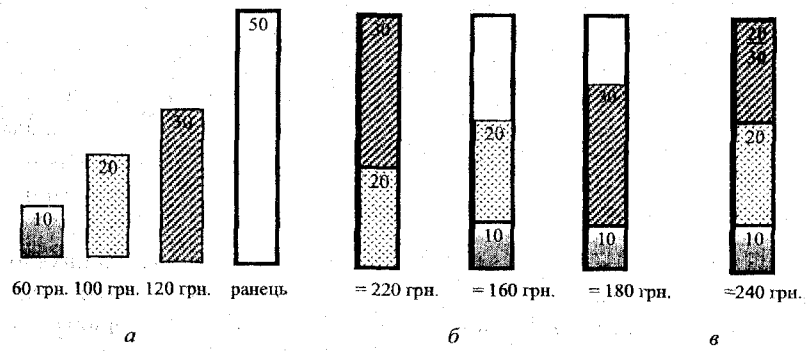


Рис. 4.5

Обчислимо їхні питомі вартості – це 6, 5 і 4 грн/кг, відповідно. Жадібна стратегія спонукає спочатку взяти перший предмет (з найбільшою питомою вартістю). Однак, як видно з рис. 4.5, *б*, оптимальне рішення – це взяти другий і третій предмети, а перший – залишити. Обидва можливі розв'язки, які включають перший предмет, не є оптимальними.

Легко побачити, що для аналогічної неперервної задачі жадібна стратегія, за якої спочатку завантажують перший предмет, дає змогу отримати оптимальний розв'язок (див. рис. 4.5, *в*).

Якщо ж спочатку завантажити перший предмет у дискретній задачі, то неможливо буде завантажити ранець повністю і залишене порожнє місце приведе до зниження ефективності вартості одиниці ваги. Приймаючи в дискретній задачі рішення про те, чи варто завантажувати той чи інший предмет у ранець, необхідно порівнювати розв'язок підзадачі, у яку входить цей предмет, з розв'зком підзадачі, у якій його нема, і тільки після цього можна робити вибір. У сформульованій таким способом задачі виникає множина підзадач, що перетинаються. А це є ознакою застосовності алгоритму динамічного програмування.

4.6. Завдання для самостійної роботи

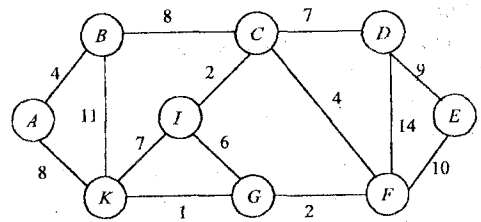
1. Алгоритм сортування злиттям ґрунтується на принципі "поділяй і володарюй". Побудувати алгоритм для сортування масиву з N чисел і, використовуючи формулу (4.1), отримати рекурентне співвідношення для оцінки часу його роботи.

2. Перевірити роботу асимптотично швидкого алгоритму множення цілих двійкових чисел (приклад 4.2), для випадку десяткових чисел.

3. За допомогою методу динамічного програмування визначити оптимальний спосіб розміщення дужок у добутку послідовності матриць, розміри яких: (5, 10, 3, 12, 5, 50, 6).

4. Побудувати алгоритм динамічного програмування для обчисленя чисел Фібоначчі.

5. Використовуючи алгоритм Дейкстри–Пріма, знайти мінімальний каркас такого зв'язного зваженого графа:



6. Побудувати жадібний алгоритм для розв'язування задачі про розкладання в ящики. Нехай ми маємо декілька ящиків одиничної ємності та набір предметів різних об'ємів $S_1, S_2, ..., S_N$ (об'єм кожного предмета не перевищує ємності ящика). Визначити найменшу кількість ящиків, необхідних для розкладання всіх предметів. Перевірити роботу алгоритму на наборі з десяти предметів таких розмірів: (0.5, 0.7, 0.3, 0.9, 0.6, 0.8, 0.1, 0.4, 0.2, 0.5).

Список літератури

1. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.
2. *Ахо А. В., Хопкрофт Д. Э., Ульман Д. Д.* Структуры данных и алгоритмы. М.: Вильямс, 2000.
3. *Гудман С., Хидетниеми С.* Введение в разработку и анализ алгоритмов. М.: Мир, 1981.
4. *Гэри М., Джонсон Д.* Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.
5. *Кнут В.* Искусство программирования. Т.1. Основные алгоритмы: 3-е изд. М.: Вильямс, 2000.
6. *Кнут В.* Искусство программирования. Т.3. Сортировка и поиск: 2-е изд. М.: Вильямс, 2000.
7. *Кожевникова Г.П.* Теория алгоритмов. Львов, ЛДУ, 1978.
8. *Кормен Т., Лейзер Ч., Ривест Р., Штайн К.* Алгоритмы: построение и анализ. М.: Вильямс, 2005.
9. *Макконелл Дж.* Основы современных алгоритмов: 2-е доп. изд. М.: Техносфера, 2004.
10. *Мальцев А.И.* Алгоритмы и рекурсивные функции. М.: Наука, 1986.
11. *Нікольський Ю. В., Пасічник В. В., Щербина Ю. М.* Дискретна математика. К.: Видавнича група BHV, 2007.
12. *Рейнгольд Э., Нивергельт Ю., Део Н.* Комбинаторные алгоритмы. Теория и практика. М.: Мир, 1980.
13. *Трахтенброт Б.А.* Алгоритмы и вычислительные автоматы. М.: Сов.радио, 1974.
14. *Успенский В. А., Семенов А. Л.* Теория алгоритмов: основные открытия и приложения. М.: Наука, 1987.
15. *David Harel.* Algorithmics. The Spirit of Computing. Addison-Wesley Publishing Company Inc., 1987.

Навчальне видання

Клакович Леся Миронівна
Левицька Софія Михайлівна
Костів Оксана Василівна

ТЕОРІЯ АЛГОРИТМІВ

Навчальний посібник

Рекомендовано
Міністерством освіти і науки України

Редактор *М. Мартиняк*
Технічний редактор *С. Сеник*
Комп'ютерне верстання *Н. Лобач*

Підп. до друку 6.11.2008. Формат 60x84/16.
Друк офсет. Гарнітура Times New Roman. Умовн. друк. арк. 8,1.
Обл.-вид. арк. 8,5. Тираж 300 прим. 352.
Видавничий центр Львівського національного університету
імені Івана Франка. 79000 Львів, вул. Дорошенка, 41

Свідоцтво
про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготівників і розповсюджувачів
видавничої продукції: Серія ДК № 3059 від 13.12.2007 р.

Клакович Л. М. та ін.

К 47 Теорія алгоритмів: Навч. посібник / Л. М. Клакович, С.М. Левицька, О.В. Костів. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2008. – 140 с.

ISBN 978-966-613-637-7

Розглянуто основні поняття та проблеми теорії алгоритмів; описано класичні алгоритмічні системи: нормальні алгоритми Маркова, рекурсивні функції, машини Тьюрінга, Поста, РАМ-машини; досліджено клас важкорозв'язних задач. Наведено деякі методи розробки ефективних алгоритмів. До кожної теми складено низку завдань для самостійної роботи.

Для студентів та аспірантів факультету прикладної математики та інформатики, а також усіх, хто цікавиться розробкою обчислювальних систем і алгоритмів.

УДК 510.5
ББК 13127