С.М. Прийма

ТЕОРІЯ АЛГОРИТМІВ

Навчальний посібник

Мелітополь
2018

УДК 519.683. (075.8)
П 15

Рекомендовано до друку рішенням Вченої ради
Таврійського державного агротехнологічного університету
(протокол № 5 від 26 грудня 2018 року)

Рецензенти:

професор кафедри комп'ютерних наук Таврійського державного агротехнологічного університету
д.т.н., професор **Малкіна В.М.**

професор кафедри інформатики і кібернетики
Мелітопольського державного педагогічного університету,
д.т.н., професор **Єремєєв В.С.**

П 15 Прийма С.М. Теорія алгоритмів: Навчальний посібник. – Мелітополь: ФОП Однорог Т.В., 2018. – 116 с.

ISBN 978-617-7566-66-2

У навчальному посібнику викладено основні розділи теорії алгоритмів. Теоретичний матеріал ілюстровано численними прикладами та доповнено практичним курсом, що дозволяє закріпити вивчений матеріал.

Навчальний посібник призначено для здобувачів вищої освіти спеціальності 122 Комп'ютерні науки.

УДК 519.683. (075.8)
ISBN 978-617-7566-66-2

ЗМІСТ

ВСТУП

Поняття «*алгоритм*» є концептуальною основою різноманітних процесів обробки інформації. З давніх часів у математиці склалося інтуїтивне уявлення про *алгоритм* як формальне розпорядження, що визначає сукупність операцій і порядок їх виконання для розв'язання задач деякого типу. Термін «алгоритм» зобов'язаний своїм походженням великому вченому середньовічного Сходу – Махамад ібн Муса ал-Хорезмі (Магомет, син Мойсея, із Хорезма 783 - 850 рр). У латинських перекладах з арабської арифметичного трактату ал-Хорезмі його ім'я транскрибувалося як *algorismi*.

З алгоритмами, тобто ефективними процедурами, які однозначно приводять до результату, математика мала справу завжди й у своєму розвитку накопичила множина різних алгоритмів. Отримавши відповідну інтерпретацію в конкретних додатках, вони становлять значну і найсуттєвішу частину математичного апарату, який використовується в техніці. Відомі зі шкільної програми методи множення "стовпчиком" та діленням "кутом", алгоритм Евкліда знаходження найбільшого спільного дільника двох додатних натуральних чисел, метод Гауса розв'язання системи лінійних рівнянь, правило диференціювання складної функції, добування квадратного кореня з раціонального числа із заданою мірою точності, обчислення рангу цілочислової матриці, спосіб побудови трикутника за трьома сторонами – все це алгоритми.

Доки математика мала справу в основному з числами й обчисленнями, а поняття алгоритму ототожнювалося з поняттям методу обчислення, необхідність у вивченні самого поняття алгоритму не виникало. Вважалося, якщо для розв'язання деякого класу задач пропонувався конкретний алгоритм, то вчені доходили згоди вважати його дійсно шуканим алгоритмом. І тільки доведення того, що не існує алгоритму розв'язання певного класу, яке є висловлюванням про всі можливі алгоритми, потребує уточнення цього важливого поняття. Саме уточнення поняття "алгоритм" і виконує така наукова дисципліна як *теорія алгоритмів*. Отже, *теорія алгоритмів* - наукова дисципліна, що

вивчає поняття про алгоритм на основі алгоритмічних моделей, їх функціонування та взаємозв'язку.

Теорія алгоритмів використовуються в теорії релейно-контактних схем, в теорії автоматів, у лінгвістиці, в економічних дослідженнях, у фізіології мозку і психології.

Теорія алгоритмів дуже важлива для професійної підготовки фахівців комп'ютерних наук, адже навчальна дисципліна сприяє розвитку алгоритмічного мислення.

ЗМІСТОВИЙ МОДУЛЬ 1. ОСНОВНІ ПОЛОЖЕННЯ ТА ОЗНАЧЕННЯ ТЕОРІЇ АЛГОРИТМІВ. АЛГОРИТМІЧНІ МОДЕЛІ

1.1. ТЕОРІЯ АЛГОРИТМІВ. ОСНОВНІ ПОЛОЖЕННЯ ТА ОЗНАЧЕННЯ ТЕОРІЇ АЛГОРИТМІВ

1.1.1. Поняття про алгоритм. Еволюція поняття алгоритм.

Література: *[8,108-113;8,116-118;12,360-362;18,7-12]*
Ключові поняття: *алгоритм, властивості алгоритму, вимоги до алгоритмів, підходи до визначення алгоритму, обчислювальна функція, детерміновані пристрої, фінітний комбінаторний процес Поста, абстрактна машина Тьюрінга, нормальні алгоритми Маркова.*

Поняття алгоритму належить до фундаментальних понять математики, кібернетики та інформатики.

Під алгоритмом розуміють чіткі інструкції про виконання в певній послідовності деякої системи операцій для рішення задач певного класу. Звичайно, запропоноване визначення швидше роз'яснює слово "алгоритм", а не дає точного математичного визначення поняття алгоритм. Незважаючи на це, в науці довгий час використовувалися саме інтуїтивні визначення цього поняття про що свідчать і наступні приклади.

Алгоритм - чітко визначений опис способу рішення задачі у вигляді кінцевої послідовності дій.

Алгоритм - правило, що сформульоване на деякій мові та визначає процес переробки вихідних даних в необхідні результати.

Алгоритм - сукупність правил, що визначає ефективну процедуру рішення будь-якої задачі деякого заданого класу задач.

Алгоритм - вказівки, що однозначно визначають процес перетворення вихідної інформації у вигляді послідовності елементарних дискретних кроків, які дозволяють за кінцеву їх кількість отримати необхідний результат.

1.1.1.1. Властивості алгоритмів

До основних властивостей алгоритму відносять наступні характеристики:

Детермінованість -визначення кроків алгоритму, тобто після кожного кроку або зазначається, який крок слід роботи далі, або дається команда на зупинку. Ця властивість означає, що застосування алгоритму до тих самих даних має призвести до одного і того самого результату.

Масовість-алгоритм може бути використаний для розв'язання цілого класу задач одного типу.

Результативність -виконання алгоритму має або закінчитися результатом, або інформацією про те, чому не може бути одержаний результат.

Зрозумілість-алгоритм має бути зрозумілим конкретному виконавцю, який повинен виконати кожну команду алгоритму у суворій послідовності з її призначенням.

Дискретність-можливість розбиття алгоритму на скінчену кількість етапів, причому результати попереднього етапу є вхідними для наступного.

1.1.1.2. Вимоги до алгоритмів

Серед вимог, що подані до алгоритму, необхідно вирізняти наступні:

- будь-який алгоритм застосовується для початкових даних і видає результат; маючи за мету в подальшому уточнити поняття алгоритму, необхідно уточнити також і поняття даних, тобто зазначити, які вимоги задовольняють об'єкти, щоб з ними могли працювати алгоритми;
- дані та алгоритм розміщуються в пам'яті, яка вважається однорідною та дискретною, іноді навіть нескінченною;
- у якості виконавця може виступати людина, різні технічні пристрої.

Слід зазначити, що навіть при виконанні всіх зазначених вимог не всяка послідовність дій є алгоритмом.

1.1.2. Підходи до визначення алгоритму

З метою формалізації поняття алгоритму, починаючи з 30-років XX ст., було здійснено цілий ряд досліджень, які мали за

мету виявити засоби, що фактично б залучалися для побудови алгоритмів. Проблема полягала в тому, щоб на цій основі дати визначення поняття алгоритм, яке було б досконалим не лише з точки зору формальної точності, але й з точки зору фактичного існування сутності даного поняття. При цьому дослідники виходили з точки зору різних технічних та логічних уявлень, внаслідок чого було запропоновано декілька алгоритмічних моделей, що на сьогодні використовуються для формалізації поняття алгоритм.

1.1.2.1. Алгоритм як обчислювальна функція.

Історично першою формалізацією поняття алгоритму були рекурсивні функції. Основана дана модель на тому положенні, що будь-який алгоритм чи алгоритмічну проблему можна звести до обчислення значення деякої цілочисельної функції цілочисельних аргументів.

В даній теорії алгоритмом прийнято називати систему обчислень, що для деякого класу задач Z із умови A дозволяє за допомогою однозначно визначеної послідовності операцій, що здійснюються "механічно", отримати результат B.

Для доведення, подамо всі початкові умови задачі Z у вигляді послідовності:

$\alpha_0, \alpha_1, \alpha_2, \alpha_3, ..., \alpha_n,$

а розв'язки як:

$\beta_0, \beta_1, \beta_2, \beta_3, ..., \beta_m.$

Якщо ж позначити через F множину номерів n тих умов α_n, які алгоритм може перетворити в рішення, то результат роботи алгоритму, що здійснює перетворення $\alpha_n \rightarrow \beta_m$ однозначно визначається заданою на F числовій функції m=f(n).

Таким чином, довільний алгоритм можна звести до обчислення значення деякої числової функції. І, навпаки, якщо для функції f існує алгоритм, який призводить до стандартного запису значення функції m=f(n), то функція f називається *алгоритмічно обчислювальною* або просто *обчислювальною*.

Як було зазначено раніше, у теорії алгоритмів, зокрема і в теорії обчислювальних функцій, прийнято конструктивний, фінітний підхід, основною рисою якого є те, що вся множина об'єктів (у нашому випадку – функцій) будується зі скінченого

числа початкових об'єктів за допомогою простих операцій та операторів, ефективна виконуваність яких досить очевидна.

1.1.2.2. Алгоритмічні моделі на основі детермінованих пристроїв

Другий підхід щодо уточнення поняття алгоритму базується на уявленні про алгоритм як деякий детермінований пристрій, що здатний виконувати в кожний окремий момент лише чітко фіксовані елементарні операції. Основними теоретичними моделями даного типу є фінітний процес Поста та машини Тьюрінга. Дані алгоритмічні моделі суттєво вплинули на розуміння логічної природи майбутніх ЕОМ. Іншою моделлю даного типу є машини з довільним доступом, що була запропонована в 70–х роках XX ст. і використовувалася для моделювання реальних ЕОМ та одержання оцінки складності обчислень.

1.1.2.3. Нормальні алгоритми Маркова

Алгоритмічними моделями третього типу є перетворення слів у довільному алфавіті за допомогою операцій підстановки, тобто заміни однієї частини слова на іншу. Основною теоретичною моделлю даного типу є нормальні алгоритми Маркова. Вони є строгими означеннями алгоритму, який уточнює інтуїтивне уявлення на основі алфавітно-словарного підходу.

Слід зазначити, що наведені алгоритмічні моделі, які були запропоновані для формалізації поняття алгоритму, математично еквівалентні. Але на практиці вони суттєво різняться з точки зору складності, що виникає при їх реалізації.

Незважаючи на це, всі алгоритмічні моделі, так сама як і вся теорія алгоритмів, мають великий вплив на розвиток ЕОМ та практику програмування. Адже саме на основі цих моделей були реалізовані різні напрямки в програмуванні. Так, мікропрограмування базується на ідеї машини Тьюрінга, структурне програмування запозичило свої конструкції з теорії рекурсивних функцій, а мови символьної обробки інформації (ПРОЛОГ, РЕФАЛ) беруть початок від нормальних алгоритмів Маркова.

1.2. АЛГОРИТМІЧНІ МОДЕЛІ

1.2.1. Обчислювальні функції

Література: *[8,116-117;8,118-121;12,365-369;15,30-49;16,23-29]*
Ключові поняття: *обчислювальна функція, нуль-функція, функція наступності, функція проекції, оператор суперпозиції, оператор примітивної рекурсії, оператор мінімізації, примітивно-рекурсивні функції, частково-рекурсивні функції, теза Черча.*

1.2.1.1. Поняття про обчислювальну функцію.

Алгоритмом прийнято називати систему обчислень, що для деякого класу задач Z із умови A дозволяє за допомогою однозначно визначеної послідовності операцій, що здійснюються "механічно", отримати результат B.

Для доведення, подамо всі початкові умови задачі Z у вигляді послідовності:

$\alpha_0, \alpha_1, \alpha_2, \alpha_3, ..., \alpha_n,$

а розв'язки як:

$\beta_0, \beta_1, \beta_2, \beta_3, ..., \beta_m.$

Якщо ж позначити через F множину номерів n тих умов α_n, які алгоритм може перетворити в рішення, то результат роботи алгоритму, що здійснює перетворення $\alpha_n \rightarrow \beta_m$ однозначно визначається заданою на F числовою функцією m=f(n).

Таким чином, довільний алгоритм можна звести до обчислення значення деякої числової функції. І, навпаки, якщо для функції f існує алгоритм, який призводить до стандартного запису значення функції m=f(n), то функція f називається *алгоритмічно обчислювальною* або просто *обчислювальною*.

Як було зазначено раніше, у теорії алгоритмів, зокрема і в теорії обчислювальних функцій, прийнято конструктивний, фінітний підхід, основною рисою якого є те, що вся множина об'єктів (у нашому випадку – функцій) будується зі скінченого числа початкових об'єктів за допомогою простих операцій та операторів, ефективна виконуваність яких досить очевидна.

При цьому слід зазначити, що у теорії обчислювальних функцій визначають множину натуральних чисел N=0,1,2,3,… і розглядають тільки числові функції, розуміючи під ними функції n змінних (n-місні функції), аргументи і значення яких належать N.

Виходячи з вищесказаного, надамо означення обчислювальної функції:

Обчислювальна функція – це числова функція $f(x_1, x_2, x_3,.., x_n)$, значення якої можна обчислювати за допомогою деякого алгоритму на підставі відомих значень аргументу.

Найпростіші функції

Розглянемо клас числових функцій, що використовуються в якості базису для побудови обчислювальних функцій.

$O(x)=0$ (нуль-функція)

$S(x)=x+1$ (функція наступності, але не додавання одиниці)

$I^n_m(x_1, x_2, x_3,.., x_m, ... x_n) = x_m$ (функція проекції або введення фіктивних змінних або вибору аргументу).

Оператори, що застосовуються до найпростіших функцій

В якості операторів, застосування яких до базисних функцій призводить до утворення нових функцій оберемо наступні три оператори:

- оператор суперпозиції
- оператор примітивної рекурсії
- оператор мінімізації або найменшого кореня.

Оператор суперпозиції (S^n_m)

Операція суперпозиції полягає у підстановці одних арифметичних функцій замість аргументів інших функцій.

Нехай задана m-місна функція $F(x_1, x_2, x_3,.., x_m)$ та m-на кількість n-місних функцій $f_1(x_1, x_2, x_3,.., x_n)$, $f_2(x_1, x_2, x_3,.., x_n)$, $f_3(x_1, x_2, x_3,.., x_n)$,...,$f_m(x_1, x_2, x_3,.., x_n)$. Тоді говорять, що n-місна функція $\varphi(x_1, x_2, x_3,.., x_n)$ утворилася в результаті підстановки у функцію F замість її аргументів m функцій $f_1, f_2, f_3,..., f_m$.

Така підстановка називається суперпозицією S^n_m.

$S^n_m(F, f_1, f_2, ..., f_m) = F(f_1(x_1, x_2, .., x_n) , f_2(x_1, x_2, .., x_n),...,f_m(x_1, x_2, .., x_n))=$

$=\varphi(x_1, x_2, .., x_n)$

Приклад: здійснюючи операцію суперпозиції функцій $f(x)=0$ та $g(x)=x+1$ отримуємо $h(x)=g(f(x))=0+1=1$.

Оператор примітивної рекурсії (Rn).

Оператор примітивної рекурсії (R^n) дозволяє будувати n+1-місну арифметичну функцію $f(x_1, x_2, x_3,.., x_n, y)$ з двох заданих функцій, одна з яких є n-місною $\varphi(x_1,x_2,..,x_n)$, а інша - n+2-місна функція $\psi(x_1,x_2,..,x_n,y,z)$ за наступною схемою:

$f(x_1, x_2, x_3,.., x_n,0)=\varphi(x_1,x_2,..,x_n)$

$f(x_1, x_2, x_3,.., x_n,y+1)= \psi(x_1,x_2,..,x_n,y, f(x_1, x_2, x_3,.., x_n, y))$

Таким чином, $f(x_1, x_2, x_3,.., x_n, y) = R^n(\varphi,\psi)$.

Для правильного розуміння операції примітивної рекурсії необхідно зазначити, що будь-яку функцію від меншої кількості аргументів можна розглядати як функцію від більшої кількості аргументів. Зокрема, функції-константи (n=0) є одномісними і відповідно:

$f(x)=a$

$f(x,y+1)=\psi(x,y, f(y))$, де a – константа.

Схеми примітивної рекурсії визначають функцію f рекурсивно не тільки через інші функції φ та ψ, а й через значення f у попередніх точках – значення f у точці (y+1) залежить від значення f у точці (y).

Наприклад, необхідно побудувати 2-місну функцію $f(x,y)=x+y$ з елементарних функцій за допомогою оператора примітивної рекурсії.

Функція $f(x,y) =x+y$ визначається функцією проекції $I^1_1(x) = x$ та функцією слідування $S(x,y,z)=z+1$.

Таким чином,

$f(x,0)= I^1_1(x) = x$

$f(x,1)= S(x,0,x)=x+1$

$f(x,2)= S(x,1,x+1)=x+2$

.....

$f(x,y-1)= S(x,y-2,x+y-2)=x+y-1$

$f(x,y)= S(x,y-1,x+y-1)=x+y$

або

$f(x,0)= I^1_1(x) = x$

$f(x,y+1)= f(x,y)+1=S(x+y)$

Таким чином, $f(x,y)= x+y=R^1(I^1_1(x), g(x,y,z))$, де $g(x,y,z)=S(z)=z+1$

Приклад: побудувати 2-місну функцію f(x,y) =x*y з елементарних функцій за допомогою оператора примітивної рекурсії.

$f_{xy}(3,2)=6$

$f_{xy}(3,2)=F_{xy}(3,1)+3=6$

$f_{xy}(3,1)=F_{xy}(3,0)+3=3$

$f_{xy}(3,0)=0=O(x)$

Приклад: побудувати 2-місну функцію f(x,y) $=x^y$ з елементарних функцій за допомогою оператора примітивної рекурсії.

$f_x^y(3,2)=9$

$f_x^y(3,2)=f_x^y(3,1)*3=9$

$f_x^y(3,1)=f_x^y(3,0)*3=3$

$f_x^y(3,0)=1=S(O(x))$

Оператор мінімізації (μу)

Розглянемо обчислювальну n-місну числову функцію $f(x_1, x_2, x_3,.., x_n)$. Зафіксуємо деякі значення $x_1, x_2, x_3,.., x_{n-1}$ її перших (n-1) аргументів та розглянемо рівняння $f(x_1, x_2, x_3,.., x_{n-1}, y) = x_n$

Значення виразу для заданої функції f залежить від значення $x_1, x_2, x_3,.., x_{n-1}, x_n$, тому вираз дає часткову функцію змінних параметрів $x_1, x_2, x_3,.., x_{n-1}, x_n$, що визначає μ-оператор.

Наприклад, отримаємо за допомогою μ-оператора функцію d(x,y):

$d(x,y)=\mu z[y+z=x]=\mu z[S((I^3_2(x,y,z), I^3_3(x,y,z),)]= I^3_1(x,y,z)$.

Обчислимо функція d(7,2). Для цього необхідно задати у значення 2 та встановити змінній Z послідовно значення 0,1,2.., кожного разу обчислюючи суму y+z. Як тільки вона дорівнюватиме 7, то відповідне значення прийняти за значення d(7,2).

Проведемо обчислення:

z=0 2+0=2 < > 7

z=1 2+1=3 < > 7

z=2 2+2=4 < > 7

z=3 2+3=5 < > 7

z=4 2+4=6 < > 7

z=5 2+5=7=7

Таким чином d(7,2)=5.

1.2.1.2. Примітивно-рекурсивні функції

Функція називається примітивно-рекурсивною, якщо вона бути утворена з найпростіших функцій за допомогою скінченого числа застосувань операторів суперпозиції S^n_m та примітивної рекурсії R^n

1.2.1.3. Частково-рекурсивні функції

Функція називається частково-рекурсивною, якщо вона бути утворена з найпростіших функцій за допомогою скінченого числа застосувань операторів суперпозиції S^n_m , примітивної рекурсії R^n та мінімізації μ_y.

1.2.1.4. Теза Черча

Кожна стандартно задана частково-рекурсивна функція є обчислювальною за певною процедурою, яка відповідає інтуїтивному уявленню алгоритму, а з іншого боку - які б досі не будувалися класи точно визначених алгоритмів, завжди з'ясовувалося, що числові функції, які обчислювалися за алгоритмами цих класів, були частково-рекурсивними. Тому загальноприйнятою є така наукова гіпотеза (теза Черча):

ТЕЗА ЧЕРЧА - *клас алгоритмічно обчислювальних часткових числових функцій збігається з класом усіх частково-рекурсивних функцій.*

У формулювання цієї тези входить інтуїтивне поняття обчислювальності, тому його не можна ні спростувати, ні довести. Це факт, на користь якого свідчить багаторічна математична практика.

1.2.2. Алгоритмічні моделі на основі детермінованих пристроїв

Література:*[8,116-118;8,124-129;8,118-124;8,137-140;12,402-408; 15,204-254;18,60-66;19,7-36;19,83-89]*

Ключові поняття: *фінітний комбінаторний процес Поста, нескінчена стрічка, пристрій керування, стан машини Поста, програма машини Поста, абстрактна машина Тьюрінга, функціональна схема машини Тьюрінга, конфігурація машини Тьюрінга, універсальна машина Тьюрінга, машини з довільним доступом, команди машин з довільним доступом .*

1.2.2.1. Фінітний комбінаторний процес Поста

Основні положення та означення фінітного комбінаторного процесу Поста

Фінітний комбінаторний процес Поста (тут та надалі - машина Поста) складається із стрічки та пристрою керування (каретки).

Стрічка нескінчена та складається з комірок однакового розміру (див. 2.1.).

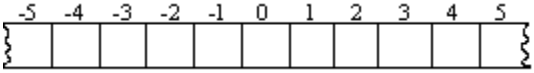


Рис. 2.1. Нескінчена стрічка

Послідовність, у якій розташовані комірки, відповідає послідовності, у якій розташовані натуральні числа.

В кожній комірці стрічки може бути записано або символ мітки ▼ або нічого (відповідна комірка називається або поміченою або пустою).

Інформація про заповнення стрічки називається її *станом*.

Каретка може переміщуватися вздовж стрічки праворуч та ліворуч. Коли ж вона нерухома, то стоїть проти однієї з комірок.

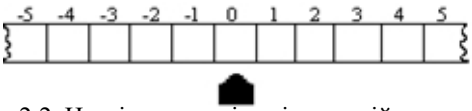


Рис. 2.2. Нескінчена стрічка і пристрій керування

Інформація про стан стрічки та місце розташування каретки називається *станом машини Поста*.

Робота машини Поста відбувається у дискретному часі. За одиницю часу каретка може виконати одну з операцій:

- зміститися праворуч
- зміститися ліворуч
- встановити мітку
- знищити мітку
- визначити наявність мітки.

Програма Поста

Кожна програма машини Поста складається з команд.

Кожна команда програми машини Поста складається з *номеру команди*, *операції* та *переходу*.

Наприклад,

- команда зміщення праворуч
  i → j
- команда зміщення ліворуч
  i ← j
- команда встановлення мітки
  i ⮟ j
- команда знищення мітки
  i ε j
- команда передачі керування
  i ?→ j1, j2
- команда зупинки
  i стоп або i ! (знак оклику)

Відповідно, *програма машини Поста* – це скінчений перелік команд, що має наступні властивості:

- на n-му місці записується команда з номером N;
- передача керування повинна відбуватися тільки до існуючого номеру команди.

Необхідні умови роботи машини Поста:

- визначеність стану машини Поста (місцерозташування каретки і міток);
- наявність програми машини Поста.

Зауваження щодо роботи машини Поста:

- виконання команди встановлення/знищення мітки не призводить до переміщення каретки і можливе тільки за умови пустої / відміченої комірки;
- виконання команди передачі керування з верхнім та нижнім індексом не змінює стан машини Поста. Верхній перехід відбувається у випадку, коли комірка, яку визначає каретка, пуста, і навпаки.

Наприклад,

Машина Поста	Мова програмування Pascal
i ?→ j1, j2	If “ ” then j1 else j2

Результат виконання програми машини Поста:

- в ході виконання програми машина Поста зустрічається із командою зупинки, що призводить до результативної зупинки.

- в ході виконання програми машина Поста зустрічається із не коректною командою, що призводить до без результативної зупинки.1
- в ході виконання програми машина Поста не зустрічається ні з однією з вищевказаних команд, що призводить до "зациклювання".

Зауваження: різні програми, що опрацьовують один і той же стан, можуть призводити до всіх трьох результатів, і навпаки, одна і та ж програма може давати різні результати для різних початкових станів.

1.2.2.2. Абстрактна обчислювальна машина Тьюрінга

28 травня 1936 року у журналі "Праці Лондонського математичного товариства" з'явилася стаття А.М.Тьюрінга «Про обчислювальні числа», в якій автор уточнював поняття алгоритму через абстрактний пристрій, що надалі отримав назву "*абстрактної машини Тьюрінга*".

Формальний опис машини Тьюрінга

Машина Тьюрінга (МТ) складається з:

- нескінченної стрічки, що поділена на комірки. В кожній комірці може бути записаний один із символів кінцевого алфавіту $A=\{a_0, a_1, a_2, \ldots, a_m\}$, що називається зовнішнім алфавітом (ЗА). Для кожної машини Тьюрінга можна задати власний ЗА.
- керуючого пристрою, що може знаходитися в одному із внутрішнього станів $Q=\{q_1, q_2, \ldots, q_n\}$. Кількість елементів Q визначає об'єм "внутрішньої пам'яті" машини Тьюрінга. У множині Q вирізняють два спеціальні стани: початковий q_1 та кінцевий q_z (або ! – знак оклику), де z – не числовий індекс, а мнемонічна ознака кінця. Таким чином, машина Тьюрінга починає роботу в стані q_1 та, потрапивши в q_z зупиняється.
- каретки, що переміщуючись вздовж стрічки, може:
  - *записувати в комірку символ зовнішнього алфавіту;*
  - *зміщуватися на комірку праворуч/ліворуч чи залишатися на місці*

Функціонування машини Тьюрінга можна описати так:

в залежності від внутрішнього стану машини Тьюрінга (q_i) та символу зовнішнього алфавіту на стрічці (a_j) відбувається запис

нового символу зовнішнього алфавіту (a'_j), зміщення каретки (d) та перехід до нового внутрішнього стану (q'_i).

Таким чином, робота машини Тьюрінга визначається системою команд вигляду:

$q_i\ a_j \rightarrow q'_i\ a'_j\ d$ (1)

Функціональна схема машини Тьюрінга

Враховуючи те, що для кожної пари $q_i\ a_j$ необхідна команда вигляду (1), програму машини Тьюрінга зручно записувати у вигляді прямокутної таблиці, стовпчики якої відповідають знакам внутрішніх станів машини Тьюрінга, а рядки – знакам зовнішнього алфавіту (див. *Таблиця 2.1.*). На перетині стовпчиків та рядків записана трійка знаків. Таку таблицю називають *функціональною схемою* машини Тьюрінга.

Таблиця 2.1.

Функціональна схема машини Тьюрінга

A \ Q	q_1	q_2	…	q_n
a_0	$q'_i a'_j d$			
a_1				
…				
a_m				

Конфігурація машини Тьюрінга

Повним станом машини Тьюрінга або конфігурацією машини Тьюрінга, за якою можна однозначно визначити поведінку машини, називається сукупність внутрішнього стану, стану стрічки та положення каретки.

Конфігурацію машини Тьюрінга надалі будемо записувати у вигляді

$a_1\ q_i\ a_2$ (2), де

q_i - поточний внутрішній стан,

a_1 - слово ліворуч від каретки,

a_2 - слово, що утворене символом, який визначається кареткою, та символами праворуч від нього.

Наприклад, конфігурація К з внутрішнім станом, словом abcde на стрічці та кареткою під символом С, записується як **abq_icde**.

Стандартною початковою конфігурацією назвемо конфігурацію вигляду q_1a, тобто конфігурація, що містить початковий стан, в якому каретка розглядає крайній лівий символ слова, записаного на стрічці.

Якщо машина Тьюрінга, почавши роботу з деяким словом, записаним на стрічці, прийде в заключний стан, то вона називається застосованою для цього слова. Результатом її роботи вважається слово, записане на стрічці в заключний момент. Якщо ж машина в жодний момент не прийде в заключний стан, то вона називається незастосовною до слова, і результат її роботи не визначений.

Приклад програми машини Тьюрінга

Написати програму для машини Тьюрінга для збільшення числа n на 1. Число записане у вигляді послідовності одиниць. Каретка знаходиться десь зліва від числа (див. рис. 2.3.).

λ	λ	λ	1	1	1	λ	λ
	q_1						

Рис. 2.3. Конфігурація машини Тьюрінга для задачі збільшення числа *n* на 1

Пояснення до рішення задачі: на першому такті визначається символ у комірці. Якщо це знак λ, то відбувається зміщення праворуч (символ в комірці та стан машини Тьюрінга залишаються незмінними). У випадку знаходження лівої позиції заданого числа (каретка зустріла в комірці перший символ 1), відбувається зміщення каретки ліворуч та перехід до стану $\mathbf{q_2}$ (символ в комірці залишається без змін). Зустрівшись зі символом λ у стані $\mathbf{q_2}$, машина Тьюрінга змінює його на символ 1 та припиняє роботу (переходить до стану $\mathbf{q_z}$).

Таблиця 2.2.

Функціональна схема машини Тьюрінга для задачі збільшення числа *n* на 1

A \ Q	q_1	q_2
1	$q_2$1L	
λ	q_1λR	q_z1E

Універсальна машина Тьюрінга

На сьогодні ми дотримувались точки зору, за якою різні алгоритми виконувалися в різних машинах Тьюрінга, що різнилися, звичайно, своїми функціональними схемами.

Але виникає природне запитання: *а чи можна побудувати універсальну машину Тьюрінга, яка б була здатна виконувати будь-який алгоритм, тобто була здатна виконувати роботу будь-якої машини Тьюрінга?*

Для відповіді на це запитання уявимо ситуацію, коли ми задали деякому виконавцю (людині) завдання, яке б полягало у демонстрації роботи машини Тьюрінга по опрацюванню деякої початкової інформації.

Звичайно, додатковою умовою буде повідомлення функціональної схеми даної машини Тьюрінга. Ми неодноразово, складаючи програми для машини Тьюрінга, самі виступали в якості виконавців, відтворюючи алгоритм машини.

Складемо словесний опис цього алгоритму:

Вказівка1: визначити на стрічці комірку, під якою підписана буква.

Вказівка2: знайти у функціональній схемі стовпчик, який позначений буквою, що написана під початковою коміркою.

Вказівка3: у стовпчику, що визначений у вказівці 2, знайти трійку символів, що розташована на перехресті із рядком, який позначений тим символом, що вписаний у комірку.

Вказівка4: змінити символ у комірці на перший символ трійки.

Вказівка5: якщо в трійці другим символом є символ !, то зупинити роботу. Якщо ж в трійці другим символом є символ Е, то зміни символ, що підписаний під визначеною коміркою, третім символом трійки.

Вказівка7: якщо в трійці другим символом є символ L/R, то зітри символ під коміркою та ліворуч/праворуч запиши третій символ із трійки.

Вказівка 8: перейти до вказівки 1.

Але на місці виконавця може бути і сама машина Тьюрінга. А це означає, що замість словесного алгоритму за допомогою 7 вказівок, алгоритм відтворення може бути заданий у вигляді функціональної схеми.

Вихідними даними до нашого алгоритму буде функціональна схема та початкова конфігурація деякої машини Тьюрінга.

Але при цьому є два зауваження:

- безпосередня подача функціональної схеми деякої машини Тьюрінга та відповідної конфігурації на стрічку універсальної машини Тьюрінга в якості вихідної інформації неможлива! Адже функціональна схема задається двомірною таблицею, конфігурація у вигляді одномірної таблиці з підписаним нижче символом стану;
- універсальна машина Тьюрінга може оперувати тільки фіксованим скінченим зовнішнім алфавітом. Але при цьому вона повинна бути пристосована до прийому в якості функціональної схеми та конфігурації з будь-якою кількістю будь-яких символів.

Таким чином постає **завдання**: знайти спосіб "одномірного" подання інформації та кінцевого алгоритму.

Вирішення проблеми

Замість подання функціональної схеми у вигляді двомірної таблиці запишемо на стрічці послідовність п'ятірок, де перший символ – позначення стовпчика, другий – рядку, наступні три – символи команди.

Таким чином, замість функціональної схеми програми для збільшення натурального числа N, поданого послідовністю одиниць на 1, виникає рядок (див. рис. 2.4.).

q_1	1	q_2	1	L	q_1	λ	q_1	λ	R	q_2	λ	q_z	1	E

Рис. 2.4. Лінійний запис функціональної схеми машини Тьюрінга

Для запису конфігурації машини Тьюрінга будемо дотримуватися такого правила: символ, що відповідає стану машини Тьюрінга, записуємо не під коміркою, а безпосередньо ліворуч від неї.

Наприклад, на рис. 2.5. зображена конфігурація машини Тьюрінга у звичному вигляді, на рис. 2.6. – її лінійне подання.

λ	λ	λ	1	1	1	λ	λ
	q_1						

Рис. 2.5 . Конфігурація машини Тьюрінга

Для однозначного визначення символів, що відповідають внутрішнім станам універсальної машини Тьюрінга (q_1, q_2 , q_3, …, q_m), зміщенню каретки (L,E,R) та зовнішньому алфавіту (a_1, a_2, a_3, a_4,…, a_k), змінемо ці символи на послідовності з 1 та 0, які надалі називатимемо кодовою групою.

λ	q_1	λ	λ	1	1	1	λ	λ

Рис. 2.6. Лінійний запис конфігурації машини Тьюрінга

За такого підходу важливо дотримуватися наступних правил:

- різні символи замінюються різними кодовими групами, але один і той самий символ завжди замінюється однією і тією ж кодовою групою.
- кодові групи повинні мати розподільні символи.

Ці правила можуть бути виконанні за умови спеціального кодування:

- в якості кодових груп беруться 3+k+m різновиди слів вигляду 10...01, де між символами 1 – послідовності 0.

Співвідношення кодових груп до вихідних букв здійснюється відповідно до таблиці 2.3.

Таблиця 2.3.

Співвідношення кодових групп до символів машини Тьюрінга

Тип групи	Символ	Кодова групи	Примітка	
символи зміщення каретки	L E R	101 1001 10001		
символи зовнішнього алфавіту	a_0 a_1 … a_k	100001 10000001 .... 10.....01	4 нулі 6 нулів 2(k+1) нулів	парна кількість нулів, більша 2
символи внутрішнього стану	q_1 q_2 … q_m	1000001 100000001 .... 10.....01	5 нулів 7 нулів 2(m+1)+1нулів	непарна кількість нулів, більша 5

За такої схеми кодування матимемо:

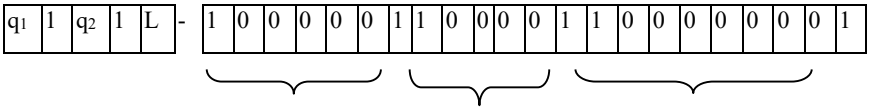


Рис. 2.7. Приклад кодування символів універсальної машини Тьюрінга

Такий рядок називатимемо *шифром* функціональної схеми машини Тьюрінга або шифром конфігурації машини Тьюрінга.

Звернемо увагу на те, як змінився алгоритм відтворення роботи універсальної машини Тьюрінга:

Вказівка1: визначити в шифрі конфігурації кодову групу, що розташована безпосередньо праворуч від кодової групи з непарною кількістю нулів.

Вказівка2-6: [див. 18, 87-88]

Вказівка7: якщо в трійці кодових груп з шифром схеми другою є група 1001, то шифрі конфігурації зміни кодову групу з непарною кількістю нулів третьою кодовою групою трійки.

Таким чином, алгоритм відтворення є описом деякої функціональної схеми машини Тьюрінга. Ця схема і є універсальною машиною Тьюрінга.

Практична значущість поняття універсальної машини Тьюрінга полягає у тому, що сучасні ЕОМ функціонують за таким же принципом як і вона, тобто: до запам'ятовуючого пристрою поряд із вихідними даними певної задачі вводиться також і програма її рішення.

Порівняйте сказане із фрагментом тексту підручника А.Ф. Верлань, Н.В. Апатової. Інформатика 10-11 кл. ст.13.

«…щоб розв'язати задачу, до пам'яті комп'ютера потрібно ввести програму, яка визначає процес розв'язування, а також необхідні дані…» «…Дж. фон Нейман (1947 р.) запропонував кілька принципів роботи комп'ютерів, названих принципами програмного керування. Головний з них – це принцип єдності лінійної пам'яті. "Єдиної" означає, що і програма, і дані зберігаються в одній пам'яті. "Лінійної" означає, що всі комірки пам'яті пронумеровані від 0 до деякого числа N. Одна і та сама комірка пам'яті для однієї задачі може зберігати команду, а для іншої дані».

2.2.2.3. Машини з довільним доступом

Машина з довільним доступом (МДД) як алгоритмічна модель була запропонована в 70-х роках XX ст. з метою моделювання реальних обчислювальних машин та аналізу складності обчислень.

Конфігурація та команди машин з довільним доступом.

Машина з довільним доступом складається з нескінченної кількості регістрів R_1, R_2, R_3, …, R_n, в кожному з яких може бути записане натуральне число N. Надалі позначатиме через r_n число, що записане в регістрі R_n.

Станом машини з довільним доступом або конфігурацією називається послідовність (r_1, r_2, r_3, …, r_n).

Функціонування машини полягає у зміні конфігурації шляхом виконання команд в порядку їх написання.

Машина з довільним доступом має наступні типи команд:

- *Команда обнулення (Z(n)).*

Дія команди Z(n) полягає у заміні змісту регістру R_n на 0. Зміст інших регістрів не змінюється.

Умовне позначення команди: $r_n:=0$

- *Команда додавання одиниці (S(n)).*

Дія команди S(n) полягає у збільшенні змісту регістру R_n на 1. Зміст інших регістрів не змінюється.

Умовне позначення команди: $r_n:= r_n+1$

- *Команда переадресації (T(m,n)).*

Дія команди T(m,n) полягає у заміні змісту регістру R_n числом r_m, що зберігається в регістрі R_m. Зміст інших регістрів (включно і R_m) не змінюється.

Умовне позначення команди: $r_n:= r_m$ або $r_m \rightarrow R_n$.

- *Команда умовного переходу (J(m,n,q)).*

Дія команди J(m,n,q) полягає в наступному:

  - порівнюється зміст регістрів R_n і R_m
  - якщо $r_m = r_n$, то МДД переходить до виконання команди з номером q в переліку команд
  - якщо $r_m <> r_n$, то МДД переходить до виконання наступної команди з переліку команд.

Кінцева впорядкована послідовність команд даних типів складає програму машини з довільним доступом.

Особливості функціонування машини з довільним доступом

Нехай зафіксована початкова конфігурація $K_0=(a_1, a_2, a_3, ..., a_n)$ чисел і програма $P=I_1, I_2, I_3, \ldots, I_s$. Тоді однозначно визначена послідовність конфігурацій $K_0, K_1, K_2,..., K_s$, де K_1 конфігурація, що отримана з K_0 застосуванням I_1. Нехай на деякому кроці виконана команда I_t та отримана конфігурація K_t. Тоді, якщо I_t не є командою умовного переходу, то наступною команда K_{t+1} є конфігурація, що отримана з K_t застосуванням I_{t+1}. Якщо I_t – команда умовного переходу, тобто $I_t = J(m,n,q)$, то K_{t+1} отримана з K_t застосуванням команди I_q, якщо $r_m = r_n$ в конфігурації K_{t+1} і команда I_{t+1}, якщо $r_m <> r_n$.

Робота машини з довільним доступом припиняється, якщо:

- виконана остання команда, тобто t=s і I_t не команда умовного переходу;
- якщо $I_t = J(m,n,q)$, $r_m=r_n$ в конфігурації K_t і q>s;
- якщо $I_t = J(m,n,q)$, $r_m <> r_n$ в конфігурації K_t і t=s.

Якщо обчислення припинилося, то послідовність $r_1, r_2, r_3,..., r_n$ змісту регістрів $R_1, R_2, R_3, \ldots, R_n$ називається заключною конфігурацією.

Програми машин з довільним доступом

Обчислення функції f(x,y)=x+y.

Ця функція може бути обчислення наступної програмою при початковій конфігурації (x,y,0,0).

$P=I_1 I_2 I_3 I_4$, де

I_1=J(3,2,5)

I_2=S(1)

I_3=S(3)

I_4=J(1,1,1).

Дана програма додає 1 до x до тих пір, доки r_3 не стане дорівнювати y.

Обчислення функції f(x,y)=x*y.

Нехай H – програма, що обчислює функцію x+y. Тоді f(x,y)=x*y обчислюється програмою:

J(2,3,p)

S(3)

H[1,4→5]
T(5,4)
J(1,1,1)
I_p T(5,1)

1.2.3.Теорія нормальних алгорифмів Маркова

Література: [8,137-140;12,362-365;18,23-37]
Ключові поняття: *нормальні алгорифми Маркова, підстановка нормальних алгорифмів Маркова, схема орієнтованих підстановок нормальних алгорифмів Маркова, означення нормальних алгорифмів Маркова.*

Основні положення та означення теорії нормальних алгорифмів

Теорія нормальних алгоритмів була розроблена радянським математиком **А.А. Марковим** (1903 р. – 1979 р.) наприкінці 40-х – на початку 50-х років XX ст. і є ще однією алгоритмічною моделлю для уточнення поняття алгоритму.

Алгорифмом (запропоноване Марковим поняття для позначення алгоритму) в теорії нормальних алгорифмів є правила по перетворенню слів в довільному алфавіті. Нормальні алгорифми оперують словами скінченої довжини, перетворюючи їх одне в одне за допомогою підстановок, тобто змін однієї частини слова на іншу.

Цей процес нагадує процедуру виконання завдання типу: “Шляхом поступової зміни літер слово “слон” перетворити на “муху””.

Поняття алфавіту нормального алгорифму

Алфавітом (як і випадку будь-якої формальної системи) називається будь-яка скінчена множина деяких символів.

Будь-яка скінчена послідовність N літер деякого алфавіту – це слова завдовжки N у цьому алфавіту. Наприклад, в алфавіту A з трьох літер {a,b,c} словами є послідовності a,b,c,ab,bac,aacbaccc.

Порожнє слово, що не містить жодного символу, позначається як λ.

Якщо слово α є частиною слова β, то кажуть, що слово α входить у слово β.

Наприклад, у слові d = abcbcbabaa є 4 підслова a, 2 підслова bcb, одне слово cba.

Поняття про підстановку

Підстановкою називається операція над словами, що задається за допомогою впорядкованої пари (α, β) та полягає у наступному: у довільному слові S знаходять перше входження слова α та, не змінюючи інших частин слова S, змінюємо в ньому це входження словом β.

Отримане слово називають **результатом підстановки** застосування підстановки (α, β) до слова S.

Якщо ж першого входження α в слово β немає (і відповідно немає ні одного входження α в S), то вважається, що підстановка (α, β) не застосована до слова S.

Для позначення підстановки (α, β) надалі будемо використовувати запис

$\alpha \to \beta$, який називається формулою підстановки (α, β).

Деякі підстановки будемо вважати заключними ($\alpha \to * \beta$ або $\alpha \to \beta$!).

$$\begin{cases} \alpha_1 \to \beta_1 // ! \\ \alpha_2 \to \beta_2 // ! \\ \ldots\ldots\ldots\ldots \\ \alpha_n \to \beta_n // ! \end{cases}$$

Схеми орієнтованих підстановок

Впорядкований скінчений перелік формул підстановок в алфавіті A називається *схемою орієнтованих підстановок* або *схемою нормальних алгорифмів.* Дана схема і визначає алгоритм перетворення слів, який називається нормальним алгорифмом Маркова.

Означення нормального алгоритму Маркова

Нехай задано алфавіт A і зафіксовано впорядковану (задану в певному порядку) систему орієнтованих підстановок P. Виходячи з довільного слова α в алфавіті A розглядаються підстановки в тому порядку, в якому їх задано. Перша підстановка, що зустрілася, з лівою частиною, яка є підсловом α, використовується для перетворення α, в яке замість першого входження лівої частини підстановки підставляється її права частина, внаслідок чого

утворюється слово α_1. Далі, виходячи з слова α_1, процес повторюється, поки він не зупиниться.

Ознаками зупинки процесу перетворення слова α є два випадки:

- коли утворюється таке слово α_n, що жодне з лівих частин допустимих підстановок не є його словами;
- коли при утворенні слова α_n використано останню підстановку (підстановку із знаком !).

Пара (A,P) визначає нормальний алгорифм Маркова.

Приклади нормальних алгорифмів Маркова

Приклад 1. Нехай задано алфавіт A={а,б,в,г,...я} і систему орієнтованих підстановок P

я→у	р→т	застосуємо алгорифм до слова “слон” та прослідкуємо перетворення:
л→у	т→р!	
с→м	о→х	слон→суон→муон→мухн→муха
в→б	н→а	застосуємо алгорифм до слова “ветер” та прослідкуємо перетворення:
		ветер→бетер→бетет→берет

Приклад 2. Нехай задано алфавіт A={1,+} і систему орієнтованих підстановок P={+→λ, 1→1}.

Слово 1+11+1111+1 алгорифм перетворює наступним чином:

1+11+1111+1
111+1111+1
1111111+1
11111111

Еквівалентним є алгорифм із P={1+→+1,+1→1, 1→1}.

Приклад 3. Нехай задано алфавіт A={1,0} і систему орієнтованих підстановок P={1→λ, 0→0}.

Слово 0101 алгорифм перетворює в слово 00.

Приклад 4. Нехай задано алфавіт A={0,1,2,3,4,5,6,7,8,9} і систему орієнтованих підстановок P

0b→1!	a0→0a	0a→0b
1b→2!	a1→1a	1a→1b
2b→3!	a2→2a	2a→2b
3b→4!	a3→3a	3a→3b
4b→5!	a4→4a	4a→4b
5b→6!	a5→5a	5a→5b
6b→7!	a6→6a	6a→6b
7b→8!	a7→7a	7a→7b
8b→9!	a8→8a	8a→8b
9b→b0	a9→9a	9a→9b
b→1!		λ→a

Застосуємо алгорифм до слова “499” та прослідкуємо перетворення:

499→a499→4a99→49a9→499a→499b→49b0→4b00→500

Як видно із прикладу, даний алгорифм виконує збільшення числа на 1.

1.2.4. Еквівалентність алгоритмічних моделей

Різноманітні підходи до уточнення (формалізації) поняття алгоритму привели до чітких алгоритмічних моделей.

В 1936 році Алонзо Черч сформулював свою тезу. Незабаром, після формулювання тези Черча, Алан Тьюрінг показав, що клас обчислювальних за Тьюрінгом функцій співпадає з класом частково-рекурсивних функцій та запропонував свій тезис, що формулюється так: “Класи обчислювальних та обчислювальних за Тьюрінгом функцій збігаються”. В свою чергу Марков сформулював свій тезис: “Будь-яка обчислювальна в інтуїтивному сенсі функція обчислювальна за Марковим”.

Загальні риси всіх алгоритмічних моделей.

- обчислювальна функція задається скінченим переліком елементарних інструкцій – програмою (програми машини Тьюрінга, схеми орієнтовних підстановок, схема примітивної рекурсії тощо);
- обчислення значень функції за вказаною програмою проводиться за чітко зафіксованому кінцевому переліку простих правил (опис підстановок нормальних алгорифмів чи кроків машини Тьюрінга);

- визначається єдиний спосіб подання вхідної та вихідної інформації (значення аргументів та функцій).

Всі вказані спільні риси обчислювальних функцій є доказом справедливості тези Черча в широкому сенсі: всі алгоритмічні моделі, що використовуються для точного визначення інтуїтивного поняття алгоритму – еквівалентні між собою.

Подамо більш чітке означення еквівалентності через визначення спільних ознак у різним алгоритмічних моделей.

Зв'язок частково-рекурсивних функцій з машиною Тьюрінга.

Для доведення еквівалентності частково-рекурсивних функцій з машиною Тьюрінга достатньо показати, що базисні функції та оператори суперпозиції S^n_m, примітивної рекурсії R^n та мінімізації μ_y можуть бути реалізовані на машині Тьюрінга .

Функція наступності.

Побудуємо програму для машини Тьюрінга із зовнішнім алфавітом A={λ,1} та внутрішнім станом $\{q_1\}$ і програмою, що записана у табл. 2.4.

Таблиця 2.4.

Функціональна схема функції наступності

A \ Q	q_1
1	$q_1 1R$
λ	$q_z 1E$

Нуль-функція.

Побудуємо програму для машини Тьюрінга із зовнішнім алфавітом A={λ,1} та внутрішнім станом $\{q_1\}$ і програмою, що записана у табл. 2.5.

Таблиця 2.5.

Функціональна схема нуль-функції

A \ Q	q_1
1	$q_1 \lambda R$
λ	$q_z \lambda E$

Функція проекції.

Побудуємо програму для машини Тьюрінга із зовнішнім алфавітом A={λ,1} та внутрішнім станом $\{q_1\}$ і програмою, що записана у табл. 2.6.

Таблиця 2.6.

Функціональна схема функції проекції

A \ Q	q_1
1	$q_1\lambda R$
λ	$q_z\lambda E$

Зв'язок частково-рекурсивних функцій з машинами з довільним доступом.

Оператор суперпозиції.

Нехай функція $g(y_1,y_2,..,y_n)$, f_1 $(x_1, x_2, x_3,.., x_m)$, f_2 $(x_1, x_2, x_3,.., x_m)$, ..., f_n $(x_1, x_2, x_3,.., x_m)$ обчислювальні на МДД.

Тоді обчислювальна і функція $h(x_1, x_2, x_3,.., x_m) = S(g,f,\ldots, f_n)$.

Доказ: нехай $G,F_1,\ldots,F_n$ – програми стандартного вигляду для обчислення функцій $g(y_1,y_2,..,y_n)$, f_1 $(x_1, x_2, x_3,.., x_m)$, ..., f_n $(x_1, x_2, x_3,.., x_m)$ відповідно. Необхідно написати програму H для обчислення h.

Покладемо $t=\max (n,m,r(G), r(F_1), \ldots, r(F_n))$.

Запам'ятаємо $x_1, x_2, x_3,.., x_m$ в регістрах $R_{t+1}, R_{t+2}, \ldots, R_{t+m}$, а F_1 $(x_1, x_2, x_3,.., x_m)$, .., F_n $(x_1, x_2, x_3,.., x_m)$ в регістрах $R_{t+m+1}, \ldots, R_{t+m+n}$.

Вказані регістри не використовуються для обчислення за програмами $G,F_1,\ldots,F_n$.

Програма H матиме вигляд:

$T(1,t+1)$

$T(2,t+2)$

…

$T(m,t+m)$

$F_1[t+1,\ldots t+m\rightarrow t+m+1]$

…

$F_n[t+1,\ldots t+m\rightarrow t+m+n]$

$G\ [t+m+1,\ldots t+m+n\rightarrow 1]$

Оператор примітивної рекурсії.

Нехай функції $g(x_1, x_2, x_3,.., x_n)$ та $h(x_1, x_2, x_3,.., x_n,y,z)$ – обчислювальні на МДД. Тоді функція $f=R(g,h)$ – обчислювальна.

Доказ: нехай G та H – програми стандартного вигляду для обчислення функцій g та h відповідно. Необхідно написати програму F для обчислення функції f=R(g,h).

За початковою конфігурацією (x_1, x_2, x_3,.., x_n,0) за програмою G обчислюється f (x_1, x_2, x_3,.., x_n,0) = g(x_1, x_2, x_3,.., x_n). Тепер, якщо y < >0, то застосовуємо (багаторазово) програму H для обчислення функцій f (x_1, x_2, x_3,.., x_n,1), f (x_1, x_2, x_3,.., x_n,2), …, f (x_1, x_2, x_3,.., x_n,y).

Нехай m =max (n+2,r(G), r(H)).

Запам'ятаємо x_1, x_2, x_3,.., x_n, y в регістрах R_{m+1}, R_{m+2}, …, R_{m+n}, R_{m+n+1}.

Номер циклу к, де к=*0,1,2,3...y* заносимо R_{m+n+2}.

Тимчасове значення f (x_1, x_2, x_3,.., x_n, к) розташуємо в регістрі R_{m+n+3}.

Програма F матиме вигляд:

T(1,m+1)
T(2,m+2)
…
T(n+1,m+n+1)
T(n+2,m+n+2)
…
G [1,2,3,.., n → m+n+3]
I_q: J(t+2,t+1,p)
H [m+1,m+2,.., m+n, t+2, t+3 → t+3]
S(t+2)
J(1,1,q)
I_p: T(t+3,1)

1.3. АЛГОРИТМІЧНО НЕРОЗВ'ЯЗУВАНІ ПРОБЛЕМИ

Література: *[14,208-209;14,210-217;16,62-70;18,89-94]*
Ключові поняття: *алгоритмічно нерозв'язувані проблеми, теорема Райса, проблема самозастосованості, проблема зупинки .*

Алгоритмічно нерозв'язувані проблеми – не невдача, а науковий факт. Знання можливої алгоритмічної нерозв'язуваності має бути таким самим елементом наукової культури, як для фізика знання про неможливість створення вічного двигуна.

Якщо ж важливо мати справу з розв'язуваною задачею, то потрібно чітко уявляти дві обставини:

- відсутність загального алгоритму, який вирішує певну проблему не означає, що в кожному конкретному випадку не можна досягти успіху;
- поява нерозв'язаності – це, зазвичай, результат надмірної загальності задачі.

Теорема М. Райса.

Теорема Райса є однією з найбільш загальних теорем теорії алгоритмів, що пояснює природу багатьох проблем в практиці програмування.

Для пояснення теореми дамо деякі визначення. Нехай існує деяка множина А натуральних чисел.

Характерною функцією множини А будемо називати функцію

$$\chi_A(x) = \begin{cases} 1, & якщо x \in A \\ 0, & якщо x \notin A \end{cases}$$

Множина А називається рекурсивною, якщо її характерна функція рекурсивна.

Наведемо змістовне формулювання теореми Райса.

Нехай Q – деяка властивість одномісних частково-рекурсивних функцій. Властивість Q називається нетривіальною, якщо є функції, яким характерна дана властивість Q, і яким вона не властива.

Враховуючи те, що частково-рекурсивні функції можна задати програмою їх обчислення, виникає питання: чи можливо за програмою визначити, чи має відповідна функція певну нетривіальну властивість?

У відповідності до тез Черча і Тьюрінга задача є алгоритмічно розв'язуваною тоді і тільки тоді, коли існує деяка машина Тьюрінга M^0, що вирішує дану задачу. Нехай необхідно визначити, чи характерна функції $\varphi_м(x)$, що реалізується машиною М, властивістю Q. На стрічці машини M^0 повинна бути записана інформація про програму машини М. Цю інформацію задамо у вигляді шифру Ш(М). Результатом роботи машини M^0 повинна бути відповідь "так" чи "ні", в залежності від того, характерна

функції $\varphi_м(x)$ властивість Q чи ні. Першому випадку домовимося співставляти символ 1, другому – 0. Таким чином, машина Тьюрінга M^0 розпізнає властивість Q одномісних частково-рекурсивних функцій, якщо конфігурацію $q_1Ш(M)$ вона перетворює в q_11, якщо функції $\varphi_м(x)$ характерна властивість Q, та в q_10 в іншому випадку.

Із сказаного робимо наступний висновок: якою б не було нетривіальна властивість Q одномісних частково-рекурсивних функцій, задача розпізнання цієї властивості алгоритмічно нерозв'язувана, тобто не існує машини M^0, яка вирішує дану задачу (теорема Райса). Для доказу даного висновку покажемо, що з існування машини M^0, яка розпізнає властивість Q, витікає рекурсивність множини номерів F_q функцій, яким властивість Q також характерна.

За наявності машини M^0 обчислення функції $\varphi_{N_{F_q}}(n)$ може бути здійснено наступним чином. Спочатку запис числа n переводиться в шифр $Ш(M_n)$ машини з номером n, потім застосовується машина M^0, яка видає 1, якщо функція $\varphi_{N_{F_q}}(n)=\varphi_м(x)$ має властивість Q, і 0 – в іншому випадку. В результаті значення $\varphi_{N_{F_q}}(n)$ обчислюється в алфавіті {0,1}. Потім 0 перетворюється в |, в 1 – в ||.

З обчислювальності функції $\varphi_{N_{F_q}}(n)$ витікає її рекурсивність. Але в силу нетривіальності властивості Q множина F_q не пуста і відмінна від сукупності всіх одномісних частково-рекурсивних функцій, ось чому у відповідності до теореми Райса функція $\varphi_{N_{F_q}}(n)$ не може бути рекурсивна. Отримане протиріччя доводить теорему.

Аналогічні результати можуть бути встановлені і для інших програм, за допомогою яких можна подати частково-рекурсивні функції, зокрема і для програм алгоритмічних мов програмування. Звідси витікає нерозв'язуванність багатьох програм, що пов'язані з програмуванням. Наприклад, якщо є деяка програма, то за її допомогою неможливо визначити функцію, яку дана програма реалізує. За двома програмами неможливо встановити, чи реалізують вони одну і ту ж функцію, а це призводить до

нерозв'язуванності багатьох задач, що пов'язані з еквівалентними перетвореннями і мінімізацією програм.

Приклади алгоритмічно нерозв'язуваних проблем

Проблема самозастосованості.

Проблема самозастосованості одна з найактуальніших алгоритмічно нерозв'язуваних проблем.

Зміст проблеми в наступному. Будемо розглядати машини Тьюрінга, у зовнішньому алфавіті яких присутні, поряд з іншими, символи 1 та 0.Нехай на стрічку машини М записаний її шифр Ш(М) і машина запущена в початковому стані q_1. Якщо після деякого скінченого числа кроків машина М прийде в заключний стан, то таку машину будемо називати самозастосованою, в іншому випадку – несамозастосовано. Проблема самозастосованості полягає в тому, щоб на основі машини М визначити, чи є вона самозастосованою. Машина Тьюрінга M^0 вирішує проблему самозастосованості, якщо для будь-якої машини М конфігурацію q_1Ш(М) вона перетворює в $q_1$1, якщо М самозастосована, і в $q_1$0 – якщо несамозастосована.

Звідси, проблема самозастосованості алгоритмічно нерозв'язувана, тобто не існує машини Тьюрінга, щоб вирішувала дану задачу.

Для доведення уявимо протилежне – машина М, що вирішує проблему самозастосованості існує. На її основі побудуємо нову машину M'. Для цього стан q_0 зробимо не заключними, та введемо новий заключний стан q_0' та додамо до програми M^0 дві нові команди:

$q_0 1 \to q_0 1E$

$q_0 0 \to q_0' 0E$

Машина M' застосована до шифрів несамозастосованих машин і не застосована до шифрів самозастосованих машин. Дійсно, якщо деяка машина М несамозастосована, то на початку M', працюючи так як M^0, перейде в конфігурацію $q_0 0$, а потім зупиниться у відповідності до команди $q_0 0 \to q_0' 0E$. Якщо ж М само застосована, то M' перейде в конфігурацію $q_0 1$ і ця конфігурація буде повторюватися нескінчено у відповідності до команди $q_0 1 \to q_0 1E$.

Сама машина M' є або само застосованою або несамозастосованою. У першому випадку вона застосована до

власного шифру, тобто до шифру само застосованих машин, що неможливо з побудови M'. В іншому випадку вона не застосована до власного шифру, тобто до шифру несамозастосованих машин, що також неможливо. Зазначене протиріччя виникло із припущення існування машини M^0, що вирішує проблему самозастосованості.

Проблема зупинки

Серед загальних вимог до алгоритмів відзначалася вимога результативності.

Найрадикальнішим формулюванням тут була б вимога, щоб за будь-яким алгоритмом А і даними α можна було б визначити, чи призведе робота А з початковими аргументами α до результату чи ні? Іншими словами, треба побудувати такий алгоритм В, щоб $B(A,\alpha)=I$, якщо $A(\alpha)$ дає результат, та $B(A,\alpha)=X$, в іншому випадку.

З урахуванням тези Тьюрінга цю задачу можна сформулювати як задачу про побудову машини Тьюрінга: побудувати машину T_0 таку, що для будь-якої машини Тьюрінга Т і будь-яких початкових даних для машини $T_0(\xi_T, \alpha)=I$, якщо машина $T(\alpha)$ зупиняється, та $T_0(\xi_т, \alpha)=X$, якщо вона не зупиняється.

Ця задача називається проблемою зупинки, а її формулювання нагадує задачу побудови універсальної машини Тьюрінга.

ЗМІСТОВИЙ МОДУЛЬ 2. СКЛАДНІСТЬ АЛГОРИТМІВ

2.1. СКЛАДНІСТЬ АЛГОРИТМІВ

2.1.1. Поняття про складність алгоритмів

Література:*[3,11-14;3,47-54;3,93-122;4,22;4,222-264;5,74-108;11,85-421; 12,374-385;13,11-23;13,148-166]*
Ключові поняття: *складність алгоритмів, часова і ємкісна складності алгоритмів, псевдокод, методика визначення часової складності алгоритмів, алгоритми впорядкування, впорядкування обміном, впорядкування вибором, впорядкування вставками, впорядкування підрахунком.*

Поняття алгоритмічно нерозв'язуваних проблем звичайно важливе і практично значуще. Розв'язування багатьох задач, як це не парадоксально, пов'язане саме з алгоритмічною нерозв'язуваністю. Але з розвитком обчислювальної техніки все більше уваги стали приділяти не просто наявності алгоритму рішення деякого класу задач, а й ефективності цих алгоритмів.

Використовуючи алгоритмічні моделі (фінітний комбінаторний процес машини Поста, абстрактну машину Тюрінга чи машини з довільним доступом) ми не замислювалися над обмеженням ресурсів (стрічка була нескінчена, а час необмежений). Але в реальних ЕОМ і пам'ять і час обмежені. Ось чому замало знати про існування того чи іншого алгоритму – необхідно мати уявлення про необхідні ресурси, а саме:

- чи може певна програма (алгоритм) розміститися в пам'яті ЕОМ;
- чи дасть вона результат за належний час.

Дослідженням цих питань і займається розділ теорії алгоритмів – аналіз складності алгоритмів.

Складність алгоритмів – кількісна характеристика, яка визначає час, що необхідний для виконання алгоритму (часова складність), і об'єм пам'яті, необхідний для його розміщення (ємкісна складність).

Складність розглядається, за звичай, для машинних алгоритмічних моделей (ЕОМ), оскільки в них час і пам'ять присутні у явному вигляді.

Часова характеристика (фізичний час виконання) складності алгоритму – це величина τ*t, де t – кількість дій алгоритму (елементарних команд), а τ - середній час виконання однієї операції (команди).

Кількість команд t визначається описом алгоритму в певній алгоритмічній моделі і не залежить від фізичної реалізації цієї моделі.

Середній час τ - величина фізична і залежить від швидкості обробки сигналів в елементах і вузлах ЕОМ. Ось чому об'єктивною математичною характеристикою складності алгоритму в певній моделі є кількість команд t.

Ємкісна характеристика складності алгоритмів визначається кількістю комірок пам'яті, що використовуються в процесі його обчислення. Ця величина не може перевищувати кількість дій t, що перемножена на певну константу (кількість комірок, які використовуються при виконанні однієї команди). В свою чергу, кількість дій t може перевищувати об'єм пам'яті (за рахунок використання повторень в одних і тих же комірках). До того ж проблема пам'яті технічно долається легше, аніж проблема швидкодії, яка має фізичне обмеження – швидкість розповсюдження фізичних сигналів (300 км/с). Ось чому часова складність вважається більш суттєвою характеристикою алгоритму, і в подальшому увагу ми зосередимо саме на ній.

Слід зазначити, що часова складність алгоритму не є постійною величиною і залежить від розмірності задачі (об'єм пам'яті для зберігання даних) – кількість комірок для різних даних. Отже, складність алгоритму – функція, значення якої залежить від розмірності *n* даних задачі.

Зазвичай говорять, що *час виконання алгоритму* або його *часова складність* має порядок *T(n)* від вихідних даних розмірністю *n*. Наприклад, деякий алгоритм має часову складність $T(n)=cn^2$, де *c* – деяка константа. Одиниця виміру *T(n)* точно не визначена, проте ми будемо розуміти її як кількість кроків, що виконані на ідеалізованому комп'ютері.

У більшості випадків, при визначенні часової складності алгоритму *T(n)*, розуміють максимальний час виконання алгоритму

по всім вхідним даним, так звану часову складність алгоритму $T_{max}(n)$ *у найгіршому випадку.* Також розглядають і $T_{avg}(n)$ як середній (в статистичному сенсі) час виконання алгоритму на всіх вихідних даних розмірністю n. Хоча $T_{avg}(n)$ є досить об'єктивною мірою визначення часової складності, проте часто неможливо стверджувати рівнозначність всіх вхідних даних. Таким чином, на практиці середній час виконання алгоритму знайти складніше, ніж час у найгіршому випадку. Ось чому, зазвичай, використовують час у найгіршому випадку як міру часової складності алгоритму $T(n)$. Обчислення часової складності алгоритму $T_{min}(n)$ у *найсприятливішому випадку* хоча і буде предметом нашої уваги, проте ми повинні розуміти, що значення цієї функції не повинно суттєво впливати на вибір того чи іншого алгоритму. Це пояснюється декількома причинами:

• по-перше, знаючи час роботи в *найгіршому випадку*, можна гарантувати, що виконання алгоритму закінчиться за якийсь час, навіть не знаючи, якого вигляду буде вихідна послідовність;

• по-друге, на практиці «погані» входи (для яких час роботи близький до максимуму) можуть часто потраплятися. Наприклад, для бази даних поганим запитом може бути пошук відсутнього елемента (досить поширена ситуація).

Виконуючи аналіз алгоритмів використовують різні способи їх подання: словесний опис, блок-схеми чи запис однією з мов програмування (машинно-орієнтованої чи високого рівня). Кожен із вказаних способів має свої переваги і оптимальнішим було б їх комплексне використання. Проте, ми скористаємося підходом, за яким спочатку пояснимо ідею того чи іншого методу, що необхідно реалізувати алгоритмом, а потім запишемо алгоритм *псевдокодом*.

Псевдокод – штучна неформальна мова, яка використовується для подання алгоритмів. Псевдокод - зручна і досить проста мова, що нагадує повсякденну мову і не є справжньою мовою програмування. Ретельно підготовлена на псевдокоді реалізація алгоритму може бути легко перетворена на програму на тій чи іншій мові програмування – для цього досить змінити оператори псевдокоду на їх еквіваленти мови програмування.

Слід зазначити, що ідея використання спрощеної моделі мови програмування високого рівня з'явилася в роботі [3, с.47-54]. Так, А. Ахо, Дж. Хопкрофт, Дж. Ульман для більш наочного подання

алгоритмів вводять так званий *Спрощений Алгол*. У більш пізніх роботах автори розуміють під псевдокодом композицію мови *Pascal* і менш формальних та узагальнених операторів „людською" мовою [4, с.22]. Псевдокод використовує традиційні конструкції математики і мов процедурного програмування, зокрема такі як вирази, умови, оператори і процедури. Специфічним для цієї мови є те, що вона дозволяє використовувати будь-який тип математичних записів. Псевдокод не має фіксованого набору типів даних. Змінними можуть бути цілі числа, слова і масиви.

Програма на *псевдокоді* – це один із наведених нижче операторів:

1. змінна ← вираз
2. ***if*** умова ***then*** оператор1 ***else*** оператор2
3. ***while*** умова ***do*** оператор
4. ***repeat*** оператор ***until*** умова
5. ***for*** змінна ← початкове значення ***step*** розмір кроку ***until*** кінцеве значення ***do*** оператор
6. ***begin***
   оператор
   оператор
   …
   оператор
   оператор
   end
7. підпрограми

procedure ім'я (перелік параметрів) :
оператор
function ім'я (перелік параметрів) : тип результату
оператор

8. ***comment*** коментар
   { коментар***}***

Дамо короткий опис основних операторів псевдокоду.

1. Оператор присвоювання.

Оператор присвоювання *змінна* ← *вираз* означає, що необхідно обчислити вираз праворуч від стрілки і його значення привласнити змінній, що розташована ліворуч від стрілки. Часова складність оператора присвоювання визначається часом, що затрачується на обчислення значення виразу і присвоювання цього значення змінній.

2. Оператор умови.

В операторі умови ***if*** *умова* ***then*** *оператор1* ***else*** *оператор2* умовою, що вказана за ***if***, може бути будь-який вираз, що приймає значення *true* або *false*. Якщо ця умова має значення *true*, то треба виконувати *оператор1*, що слідує за ***then***. В іншому випадку треба виконувати *оператор2*, що стоїть за ***else*** (якщо ***else*** присутнє). Часова складність оператора умови дорівнює сумі складностей, необхідних для обчислення значення і перевірки його, і складності *оператора1*, що знаходиться відразу за ***then***, або *оператора2*, що стоїть за ***else***, у залежності від того, який з них виконується.

3. Оператори повторення.

Оператори

- ***while*** умова ***do*** оператор
- ***repeat*** оператор ***until*** умова
- ***for*** змінна ← початкове значення ***step*** розмір кроку ***until*** кінцеве значення ***do*** оператор

призначені для організації повторень.

У *операторі повторення з передумовою*

while *умова* ***do*** *оператор*

обчислюється значення умови, що йде після ***while***. Якщо воно істинне (приймає значення *true*), то виконується *оператор*, що стоїть після ***do***. Цей процес повторюється доти, поки *умова* не стане хибною. Якщо спочатку ця умова була істинною, то виконання *оператора* повинно зрештою привести цю *умова* до значення *false*, щоб закінчилося виконання ***while***-оператора. Для обчислення часової складності ***while***-оператора додаються складності всіх перевірок *умови* і усіх виконаних *операторів*.

Оператор повторення з післяумовою

repeat *оператор* ***until*** *умова*

трактується аналогічно, але тільки тепер оператор, що знаходиться після ***repeat***, виконується перед перевіркою умови.

У *оператора повторення з параметром*

for *змінна* ← *початкове значення* ***step*** *розмір кроку* ***until*** *кінцеве значення* ***do*** *оператор*

„*початкове значення*”, „*розмір кроку*” і „*кінцеве значення*” є виразами. У випадку коли значення *кроку* позитивне, *змінна*

приймає значення, що відповідає значенню виразу для *початкового значення*. Якщо воно більше *кінцевого значення*, то виконання *оператора* закінчується. В іншому випадку виконується *оператор*, що стоїть після ***do***, значення *змінної* збільшується на *розмір кроку* і порівнюється із *кінцевим значенням*. Процес повторюється доти, поки значення *змінної* менше, ніж *кінцеве значення*. Випадок, коли *розмір кроку* від'ємний, трактується аналогічно з тією лише різницею, що закінчення відбувається, коли значення *змінної* стає меншим за *кінцеве значення*. Часова складність оператора повторення з параметром аналогічна часовій складності ***while***-оператора. Проте у такому випадку враховуються і часова складність обчислення значення *змінної* при кожній ітерації.

4. Підпрограми.

Фрагменти програмного коду, що часто використовуються, можна оформлювати в якості підпрограм. Підпрограми визначаються і надалі використовуються в алгоритмах.

Підпрограми використовуються одним з двох способів.

Перший спосіб полягає у використані підпрограми за допомогою оператора виклику підпрограми-процедури

procedure ім'я (перелік параметрів):

оператор

Цей оператор є, по суті, іменем процедури, за яким вказується перелік параметрів. Завершується виконання підпрограми-процедури завершенням виконання останнього оператора підпрограми.

Другий спосіб полягає у тому, що використовується оператор виклику підпрограми-функції

function ім'я (перелік параметрів) : тип результату

оператор,

який дозволяє обчислити деяке значення і присвоїти його значення функції. Наприклад, вираз c←MIN(a,b) означає, що значенню змінної *c* присвоюється значення мінімальної серед змінних *a* і *b*.

На увагу заслуговує аналіз складності виклику **підпрограм**. Для алгоритмів, що містять кілька процедур (серед яких немає рекурсивних), можна підрахувати загальний час виконання алгоритму шляхом послідовного знаходження часу виконання процедур, починаючи з тієї, яка не має викликів інших процедур. Потім

необхідно визначити час виконання процедур, що викликають цю процедуру, використовуючи вже обчислений час виконання цієї процедури. Продовжуючи цей процес можна знайти час виконання всіх процедур і, нарешті, час виконання всього алгоритму.

Якщо ж в алгоритмі присутні рекурсивні підпрограми, то не можна упорядкувати всі підпрограми таким чином, щоб кожна з них викликала тільки підпрограми, час виконання яких підраховано на попередньому кроці. У цьому випадку з кожною рекурсивною підпрограмою пов'язують часову складність *T(n)*, де *n* визначає кількість аргументів підпрограми. Потім одержують рекурентне співвідношення для *T(n)*, тобто рівняння (або нерівність) для *T(n)*, де беруть участь значення *T(k)* для різних значень *k*.

Техніка рішення рекурентних співвідношень залежить від виду цих співвідношень.

Пояснимо сказане на прикладі аналізу підпрограми обчислення факторіалу *n!*.

```
   function factorial (n): цілочисельна змінна
     begin
(1)  if n≤1 then
(2)      factorial ←1
     else
(3)    factorial ←n* factorial (n-1)
   end
```

Природною мірою кількості вхідних даних для функції *factorial* є значення *n*. Позначимо через *T(n)* часову складність алгоритму. Часова складність рядків (1) і (2) має значення 1, а рядку (3) – (1+1)+*T(n-1)*. Таким чином, для деяких констант *c* і *d* маємо:

$$T(n)=\begin{cases} c+T(n-1), & \text{якщо } n>1,\\ d, & \text{якщо } n\le 1. \end{cases}$$

Припустивши, що *n>2* і підставивши у співвідношення (1) *n-1* замість *n*, отримаємо *T(n)=2c+ T(n-2).* Аналогічно, якщо *n>3*, отримаємо *T(n)=3c+ T(n-3).* Продовжуючи цей процес, в загальному випадку для деякого *i, n>1,* маємо *T(n)=ic+T(n-i).* Поклавши в останньому виразу *i= n-1,* отримуємо

T(n)=c(n-1)+ T(1)= c(n-1)+d.

Таким чином, загальний метод рішення рекурентних співвідношень, полягає у послідовному розкритті виразу *T(k)* у

правій частині рівняння (шляхом підстановки у вихідне співвідношення k замість n) до тих пір, поки не вийде формула, в якій у правій частині відсутнє $T(n)$.

Оскільки для різних конкретних даних однакової розмірності n алгоритм може витрачати різну кількість команд, функція складності $T(n)$ визначається різними способами. Розглянемо деякі з них на прикладі *алгоритмів впорядкування*.

Під *впорядкуванням* розуміється процес перестановки об'єктів деякої множини у певному порядку [5, с. 74]. Методи впорядкування є блискучою ілюстрацією ідеї аналізу складності алгоритмів, тобто ідеї, що дозволяє оцінювати робочі характеристики алгоритмів, і відповідно, зважено підходити до вибору адекватного алгоритму рішення конкретної задачі.

Д. Кнут в третьому томі „*Сортировка и поиск*" своєї монографії „*Искусство программирования для ЭВМ*" наводить наступний факт: „Постачальники обчислювальних машин вважають, що в середньому більше 25% машинного часу систематично витрачається на впорядкування. В багатьох обчислювальних системах на неї (операцію впорядкування) припадає більше половини машинного часу. Виходячи з цієї статистики, можна стверджувати, що або (1) впорядкування має багато важливих галузей застосування, або (2) нею користуються без потреби, або (3) використовуються неефективні алгоритми впорядкування" [14, с.8]. Як зазначає Д. Кнут, в кожному із вказаних тверджень є доля істини. І якщо в першому і другому випадках вплинути на ситуацію складно (сфера застосування алгоритмів, що в якості проміжного етапу використовують принцип впорядкування, об'єктивно значна, а необхідність використання того чи іншого алгоритму суб'єктивна і залежить від „людського фактору"), то виправданим буде зосередження уваги на ефективності цих алгоритмів.

Існують декілька алгоритмів впорядкування (саме такий термін для позначення процедури ранжування використовує Д. Кнут, хоча і визнає, що термін „сортування" набув більшого розповсюдження серед програмістів), зокрема: впорядкування обміном („бульбашкове" впорядкування), впорядкування вибором, впорядкування вставками, швидке впорядкування, шейкер-впорядкування, впорядкування Шелла, пірамідальне і порозрядне впорядкування.

Введемо основну термінологію і позначення, що використовуються в алгоритмах впорядкування.

Дана послідовність A, що складається із n елементів:

$a[1], a[2], .. , a[n]$

Впорядкування передбачає розміщення цих елементів у порядку зростання (спадання) відповідно до лінійного відношення порядку, такому як „≤" („≥") для чисел.

Звичайно, впорядкування може застосовуватися не лише до елементів з „чисельною" природою. Так, в узагальненому випадку операцію впорядкування можна сформулювати наступним чином: „Впорядкувати послідовність записів таким чином, щоб значення ключових полей цих записів складали зростаючу (спадаючу) послідовність". Іншими словами, записи $a[1],a[2],..,a[n]$ зі значеннями ключів $k[1], k[2], .. , k [n]$ необхідно розташувати у такому порядку, щоб $k[1] \le k[2] \le .. \le k [n]$.

Існує декілька основних алгоритмів впорядкування (Д. Кнут наводить 25 основних алгоритмів [11,с.85]) та велика кількість їх модифікацій. Така кількість алгоритмів впорядкування пояснюється перевагами і недоліками кожного з алгоритмів для деяких конфігурацій даних і апаратури. Корисно розглянути характеристики хоча б основних з них, щоб для конкретного випадку можна було зробити розумний вибір. Проте, алгоритми впорядкування будуть використані нами не лише для розгляду методики оцінки часової складності алгоритмів, а й для ознайомлення з основними методами розробки ефективних алгоритмів.

Для пояснення методики оцінки часової складності алгоритмів $T(n)$ скористаємося так званими „простими" алгоритмами. До цієї групи відносять алгоритми впорядкування обміном, упорядкування вибором та впорядкування вставками. Для демонстрації методів удосконалення „простих" алгоритмів з метою зниження їх часової складності можна скористатися алгоритмами швидкого впорякування, шейкер-впорядкування та алгоритмом впорядкування Шелла.

Слід зазначити, що поділ алгоритмів впорядкування на алгоритми впорядкування обміном-вставками-вибором (тим більше на групи „простих" і „складних") досить умовний. Адже всі алгоритми використовують операцію обміну елементів місцями, яка передбачає як вибір елементів, так і їх вставку. Проте, саме

така класифікація є найбільш вживаною в літературі, що присвячена аналізу алгоритмів впорядкування, а отже саме її ми і будемо притримуватися.

Впорядкування обміном

Алгоритм впорядкування обміном базується на принципі порівняння пари сусідніх елементів до тих пір, доки не будуть впорядковані всі елементи. Щоб описати основну ідею цього методу, який іноді називають методом „бульбашки", уявимо, що елементи зберігаються в послідовності (масиві), розташованому вертикально. Елементи, що мають малі значення, є більш „легкішими" і „спливають" нагору подібно бульбашкам. При першому проході уздовж масиву (перегляд починається знизу), береться перший елемент послідовності і його значення по черзі порівнюється зі значеннями наступних елементів. Якщо зустрічається елемент з більш „важким" значенням, то ці елементи міняються місцями. При зустрічі з елементом, що має більш „легше" значення, цей елемент стає „еталоном" для порівняння, і всі наступні елементи порівнюються з цим новим, більш „легшим" елементом. У результаті елемент з найменшим значенням опиниться вгорі послідовності. Під час другого проходу уздовж масиву знаходиться елемент із другим по величині значенням, що міститься під елементом, який було знайдено при першому перегляді послідовності. Процес повторюється до тих пір, доки не будуть впорядковані всі елементи послідовності. Відзначимо, що під час другого і наступних переглядів послідовності немає необхідності переглядати елементи, знайдені за попередні перегляди, адже вони мають значення менші за значення елементів, що залишилися. Іншими словами, під час *i*-го перегляду не перевіряються елементи, що знаходяться на позиціях вище *i*.

Проілюструємо роботу алгоритму впорядкування обміном наступним прикладом, що записаний на псевдокоді.

(1) ***for*** i←n-1 ***step*** -1 ***until*** 1 ***do***
(2) ***for*** j←1 ***step*** 1 ***until*** I ***do***
(3) ***if*** a[j] > a[j+1] ***then***
(4) swap(a[j], a[j+1])

Звернемо увагу на те, що підпрограма-процедура *swap* рядку (4) використовується в багатьох алгоритмах впорядкування для

перестановки елементів місцями. Код цієї процедури наведено нижче:

```
procedure swap (x,y)
  begin
     temp ← x
     x ← y
     y ←temp
  end
```

Для визначення часової складності алгоритму виконаємо наступні дії. Біля кожного рядку алгоритму зазначимо його вартість (число операцій) і кількість разів, за яку виконується цей рядок. Зауважимо, що рядки усередині циклу виконуються на один раз менше, ніж перевірка, оскільки остання перевірка виводить з циклу. Для кожного j від 1 до i підрахуємо, скільки разів буде виконаний рядок (3), і позначимо це число через k_j. Аналогічну дію виконаємо і для рядку (4).

Таблиця 2.7.

Визначення часової складності алгоритму впорядкування обміном

№	Команда	Вартість	Кількість виконань
(1)	***for*** i←n-1 ***step*** -1 ***until*** 1 ***do***	c_1	n
(2)	***for*** j←1 ***step*** 1 ***until*** i ***do***	c_2	n-1
(3)	***if*** a[j] > a[j+1] ***then***	c_3	$\sum_{j=1}^{i} k_j$
(4)	swap (a[j], a[j+1])	c_4	$\sum_{j=1}^{i} t_j$

Рядок вартістю c, що повторений m разів, дає внесок cm в загальну кількість операцій. Склавши внески всіх рядків, одержимо вираз, що позначає часову складність алгоритму впорядкування обміном:

$$T(n) = c_1 n + c_2 (n-1) + c_3 \sum_{j=2}^{i} k_j + c_4 \sum_{j=2}^{i} t_j$$

Обчислимо суму кількості виконань для рядку (3).

$$\sum_{j=1}^{i} k_j = \frac{1+n-1}{2}(n-1) = \frac{n(n-1)}{2}$$

Таким чином, часова складність алгоритму *T(n)* дорівнює:

$$T(n) = c_1 n + c_2(n-1) + c_3(\frac{n(n-1)}{2}) + c_4 \sum_{j=2}^{i} t_j =$$

$$= c_1 n + c_2(n-1) + c_3(\frac{n^2-n)}{2}) + c_4 \sum_{j=2}^{i} t_j =$$

$$= c_1 n + c_2(n-1) + c_3(\frac{1}{2}(n^2-n)) + c_4 \sum_{j=2}^{i} t_j =$$

$$= c_1 n + c_2 n - c_2 + \frac{c_3 n^2}{2} - \frac{c_3 n}{2} + c_4 \sum_{j=2}^{i} t_j =$$

$$= (\frac{c_3}{2})n^2 + (c_1 + c_2 - \frac{c_3}{2})n - c_2 + c_4 \sum_{j=2}^{i} t_j$$

Як бачимо, функція *T(n)* – квадратична, тобто має вигляд $T(n)=an^2+bn+c$, де константи *a, b* і *c* визначаються значеннями $c_1, \ldots c_4$.

Слід зауважити, що час роботи алгоритму залежить не тільки від *n*, але і від того, який саме масив розмірністю *n* поданий їй на вхід. Для алгоритму впорядкування обміном найбільш сприятливий випадок, коли послідовність уже впорядкована.

Процедура обміну *swap* у такому разі не буде виконана жодного разу, а часова складність алгоритму $T_{min}(n)$ дорівнюватиме:

$$T_{\min}(n) = (\frac{c_3}{2})n^2 + (c_1 + c_2 - \frac{c_3}{2})n - c_2$$

Таким чином, у найбільш сприятливому випадку час $T_{min}(n)$, необхідний для обробки масиву розміру *n*, залишається бути квадратичною функцією від *n*.

Якщо ж масив розташований у зворотному (спадному) порядку, час роботи алгоритму буде максимальним: кожен елемент *a[j]* прийдеться міняти місцями з елементом *a[j+1]* . При цьому $t_j = j$.

Обчислимо суму кількості виконань для рядку (4).

$$T_{\max}(n) = (\frac{c_3}{2})n^2 + (c_1 + c_2 - \frac{c_3}{2})n - c_2 + c_4 \sum_{j=2}^{i} t_j =$$

$$= (\frac{c_3}{2})n^2 + (c_1 + c_2 - \frac{c_3}{2})n - c_2 + c_4(\frac{n^2 - n)}{2}) =$$

$$= (\frac{c_3}{2} + \frac{c_4}{2})n^2 + (c_1 + c_2 - \frac{c_3}{2} - \frac{c_4}{2})n - c_2$$

Як бачимо, у гіршому випадку час роботи алгоритму $T_{max}(n)$ також є квадратичною функцією від n.

Аналіз середнього значення часової складності алгоритму впорядкування обміном $T_{avg}(n)$ залежить від обраного розподілу ймовірностей, і на практиці реальний розподіл може відрізнятися від передбачуваного, який, зазвичай, вважають рівномірним. Іноді рівномірний розподіл моделюють, використовуючи генератори випадкових чисел. Проте, можна припустити, що в середньому обмін відбувається приблизно у $i/2$ випадках і його загальна кількість приблизно дорівнює значенню $(1+2+..+n)/2 \approx n^2/4$. У такому випадку середнє значення часової складності алгоритму впорядкування обміном $T_{avg}(n)$ можна показати формулою:

$$T_{avg}(n) = (\frac{c_3}{2})n^2 + (c_1 + c_2 - \frac{c_3}{2})n - c_2 + c_4(\frac{n^2 - n)}{4}) =$$

$$= (\frac{c_3}{2} + \frac{c_4}{4})n^2 + (c_1 + c_2 - \frac{c_3}{2} - \frac{c_4}{4})n - c_2$$

Звичайно, алгоритм впорядкування обміном можна легко модифікувати. Один із шляхів полягає у тому, що можна позбавитися великої кількості непотрібних порівнянь, запам'ятавши чи відбувався обмін на попередньому кроці. Якщо обміну не було, то послідовність вважається впорядкованою, і алгоритм закінчує роботу. Цей процес удосконалення алгоритму можна продовжити, якщо запам'ятовувати не тільки сам факт обміну, але і місце (індекс) останнього обміну. Адже зрозуміло, що всі пари елементів з індексом, меншим від j, вже упорядковані, і наступні перегляди можна закінчувати на цьому індексі.

Проте, всі запропоновані нами вдосконалення жодним чином не впливають на кількість обмінів – вони лише зменшують кількість надлишкових повторних перевірок. Нажаль, обмін двох елементів – більш ресурсоємна операція, ніж порівняння, тому всі наші вдосконалення дають лише незначний ефект.

Таким чином, аналіз алгоритму впорядкування обміном показав, що ніяких переваг, окрім легкозапам'ятовуючої назви, цей алгоритм не має. Проте, саме на прикладі цього алгоритму ми розглянули методику визначення часової складності алгоритмів, яку надалі застосуємо при аналізі інших алгоритмів.

Впорядкування вибором

Ідея методу впорядкування вибором полягає у тому, що на *i*-му кроці вибирається найменший елемент серед елементів послідовності *a*[i]..*a*[n], який міняється місцями з елементом *a*[i].

У залежності від результату пошуку елемент *a*[i] або залишається в *i*-ій позиції або вони міняються місцями і процес повторюється.

Таблиця 2.8.

Визначення часової складності алгоритму впорядкування вибором

№	Команда	Вартість	Кількість виконань
(1)	***for*** *i*←1 ***step*** 1 ***until*** *n-1* ***do***	c_1	n
	begin{for}		
(2)	*min*←*a[i]*	c_2	n-1
(3)	*index*←*i*	c_3	n-1
(4)	***for*** *j*←*i*+1 ***step*** 1 ***until*** *n* ***do***	c_4	$\sum_{j=i+1}^{n} t_j$
	begin {for}		
(5)	***if*** *a[j]* < *min* ***then***	c_5	$\sum_{j=i+1}^{n} (t_j - 1)$
	begin{if}		
(6)	*min*←*a[j]*	c_6	$\sum_{j=i+1}^{n} k_j$

№	Команда	Вартість	Кількість виконань
(7)	*index←j*	c_7	$\sum_{j=i+1}^{n} k_j$
	end{if}		
(8)	*swap(a[i], a[index])*	c_8	n-1
	end{for}		
	end{for}		

Склавши внески всіх рядків, одержимо вираз, що позначає часову складність алгоритму впорядкування вибором:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=i+1}^{n} t_j + c_5 \sum_{j=i+1}^{n} (t_j - 1) +$$

$$+ c_6 \sum_{j=i+1}^{n} k_j + c_7 \sum_{j=i+1}^{n} k_j + c_8(n-1)$$

Обчислимо суму кількості виконань оператора повторення рядка (4) і оператора умови, що знаходиться в рядку (5).

$$\sum_{j=i+1}^{n} t_j = \frac{2+n}{2}(n-1) = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=i+1}^{n} t_j - 1 = \frac{1+n-1}{2}(n-1) = \frac{n(n-1)}{2}$$

Таким чином, часова складність алгоритму *T(n)* дорівнює:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n^2+n}{2} - 1\right) +$$

$$+ c_5\left(\frac{n^2-n}{2}\right) + + c_6 \sum_{j=i+1}^{n} k_j + c_7 \sum_{j=i+1}^{n} k_j + c_8(n-1)$$

Слід зауважити, що, як і у випадку впорядкування обміном, час роботи алгоритму впорядкування вибором залежить не тільки від *n*, але і від того, яка саме вихідна послідовність розмірністю *n*. Обчислення часових складностей $T_{max}(n)$ і $T_{avg}(n)$ алгоритму впорядкування вибором відбувається аналогічним до алгоритму впорядкування обміном чином.

У випадку, коли послідовність уже впорядкована оператори в рядках (6)-(7) виконуватися не будуть. Таким чином, часова складність алгоритму $T_{min}(n)$ дорівнюватиме:

$$T_{\min}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n^2+n}{2} - 1) +$$

$$+ c_5(\frac{n^2-n}{2}) + c_8(n-1)$$

Як бачимо, навіть у найбільш сприятливішому випадку час $T_{min}(n)$, необхідний для обробки послідовності, залишається бути квадратичною функцією від n.

Слід зазначити, що, незважаючи на відсутність значної різниці в значенні функції $T(n)$ алгоритму впорядкування вибором у порівнянні з алгоритмом впорядкування обміном, впорядкування вибором є більш придатнішим. Пов'язано це, зокрема, з таким моментом. Кожен з алгоритмів використовує процедуру обміну *swap (x,y)*. В алгоритмі впорядкування обміном ця процедура виконується не менш, ніж $n(n-1)/2$ разів. Але, оскільки процедура *swap (x,y)* виконується після оператора умови, то можна очікувати, що кількість обмінів буде дещо меншим за $n(n-1)/2$. Однак, в алгоритмі впорядкування вибором процедура *swap (x,y)* знаходиться поза внутрішнім оператором повторення і тому виконується рівно $n-1$ раз для послідовності розмірністю n. А це значно менше за розглянуту в алгоритмі впорядкування обміном кількість.

Впорядкування вставками

Ідея методу полягає у тому, що розглядаються дії алгоритму на його i-му кроці. При цьому, послідовність розділена на дві частини: впорядковану a[1]...a[i] і неупорядковану a[i+1]...a[n]. На кожному наступному, ($i+1$)-му кроці алгоритму береться a[i+1] елемент і вставляється на потрібне місце в упорядковану частину послідовності.

Пошук потрібного місця для чергового елемента вхідної послідовності здійснюється шляхом послідовних порівнянь з елементом, що знаходиться перед ним. У залежності від результату порівняння елемент або залишається на поточному місці (вставка закінчена), або вони міняються місцями і процес повторюється.

Таким чином, у процесі вставки елемент *x* якби „просівається” до початку масиву. При цьому зупинка відбувається у випадку коли:

- знайдено елемент, менший від *x* або
- досягнуто початок послідовності.

Біля кожного рядку процедури зазначимо його вартість (число операцій) і кількість разів, за яку виконується цей рядок. Для кожного *i* від 2 до *n* підрахуємо, скільки разів буде виконаний рядок (4), і позначимо це число через t_i.

Визначення часової складності алгоритму впорядкування вставками

№	Команда	Вартість	Кількість виконань
(1)	***for*** *i*←2 ***step*** 1 ***until*** *n* ***do***	c_1	n
	begin{for}		
(2)	*key*←*A[i]*	c_2	n-1
(3)	*j*←*i-1*	c_3	n-1
(4)	***while*** (*j*>0) ***and*** (*a[j]*>*key*) ***do***	c_4	$\sum_{i=2}^{n} t_i$
	begin{while}		
(5)	*a[j+1]*←*a[j]*	c_5	$\sum_{j=2}^{n} (t_j - 1)$
(6)	*j*←*j-1*	c_6	$\sum_{i=2}^{n} (t_i - 1)$
	end{while}		
(7)	*a[j+1]*←*key*	c_7	n-1
	end{for}		

Склавши внески всіх рядків, одержимо вираз, що позначає часову складність алгоритму впорядкування вставками:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i +$$

$$+ c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

Як і у випадку аналізу алгоритму обміном, зауважити, що час роботи процедури залежить не тільки від *n*, але і від того, який саме масив розмірністю *n* поданий їй на вхід.

Якщо вихідна послідовність вже упорядкована у зворотному (спадному) порядку, час часова складність роботи алгоритму $T_{max}(n)$ буде максимальною: кожен елемент $a[j]$ прийдеться порівнювати з усіма елементами $a[j\text{-}1]$.. $a[1]$. При цьому $t_i = i$.

Обчислимо суми

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

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

Як бачимо, у найгіршому випадку часова складність роботи алгоритму $T_{max}(n)$ дорівнює

$$T_{\max}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) +$$

$$+ c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + + c_7(n-1) =$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 +$$

$$+ \left(c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7)$$

Функція $T_{max}(n)$ - квадратична, тобто має вигляд багаточлена (полінома) $T_{max}(n)=an^2+bn+c$, де константи *a, b* і *c* визначаються значеннями c_1, … c_7. Середнє значення часової складності роботи алгоритму $T_{avg}(n)$ буде доволі близьким до $T_{max}(n)$, адже в середньому біля половини елементів послідовності $a[1]..a[j\text{-}1]$ більше за $a[j]$. Це означає, що t_i в середньому можна вважати рівним $i/2$, отже і функція часової складності роботи алгоритму $T_{avg}(n)$ буде квадратичною:

$$T_{avg}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n(n+1)}{4} - 2) +$$

$$+ c_5(\frac{n(n-1)}{4}) + c_6(\frac{n(n-1)}{4}) +$$

$$+ c_7(n-1) = (\frac{c_4}{4} + \frac{c_5}{4} + \frac{c_6}{4})n^2 +$$

$$+ (c_1 + c_2 + c_3 + \frac{c_4}{4} + \frac{c_5}{4} + \frac{c_6}{4} + c_7)n - (c_2 + c_3 + 2c_4 + c_7)$$

Цікаво, що у найбільш сприятливішому випадку (послідовність впорядкована), рядок (4) завершується після першої ж перевірки, так що всі t_i рівні 1, і часова складність $T_{min}(n)$ дорівнює

$$T_{\min}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) +$$
$$+ c_7(n-1) =$$
$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

Таким чином, у найбільш сприятливішому випадку час $T_{min}(n)$, необхідний для впорядкування послідовності розмірністю n, є лінійною функцією від n, тобто має вигляд $T_{min}(n)=an+b$ для деяких констант a і b (ці константи знову визначаються обраними значеннями c_1, ...,c_7).

Наведений приклад показує, що час роботи у найгіршому та найбільш сприятливішому випадках може значно відрізнятися.

Підводячи підсумок, порівняємо ефективність розглянутих нами алгоритмів. Для цього скористаємося наведеними Н.Віртом аналітичними формулами [5, с.106], які дозволяють визначити часові складності $T_{min}(n)$, $T_{avg}(n)$ і $T_{max}(n)$ (див. *Таблиця 2.10.*). Зазначимо, що Н.Вірт зосереджує увагу лише на операціях порівняння (С) і обміну (М).

Аналітичні формулами для визначення часової складності алгоритмів

	C/M	$T_{min}(n)$	$T_{avg}(n)$	$T_{max}(n)$
Впорядкування обміном	C	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	M	0	$(n^2-n)*0.75$	$(n^2-n)*1.5$

	С/М	$T_{min}(n)$	$T_{avg}(n)$	$T_{max}(n)$
Впорядкування Вибором	С	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	М	$3(n-1)$	$n(\ln n+0.57)$	$n^2/4+3(n-1)$
Впорядкування Вставками	С	$n-1$	$(n^2+n-2)/4$	$(n^2-n)/2 -1$
	М	$2(n-1)$	$(n^2+-9n-10)/4$	$(n^2+3n-4)/2$

Як бачимо, для кожного із розглянутих нами алгоритмів функція визначення часової складності в середньому і найгіршому випадках є квадратичною. Існують більш ефективні алгоритму, які іноді називають „логарифмічними" за тієї причини, що функція визначення часової складності в середньому і найгіршому випадках є $T(n)=an\log n+b+c$. Різницю між „квадратичними" і „логарифмічними" алгоритмами наочно демонструє приклад, наведений у роботі [13,с.29-30]. Нехай потрібно впорядкувати послідовність розмірністю в один мільйон елементів. Виникає питання: що швидше – впорядковувати його алгоритмом вставок з часовою складністю, щодо відповідає квадратичній функції $T(n)=c_1n^2$ на комп'ютері, який виконує 100 мільйонів операцій за секунду, чи алгоритмів впорядкування злиттям з часовою складністю $T(n)=c_2n\log n$. При цьому алгоритм впорядкування вставками написаний надзвичайно економно і для сортування n чисел потрібно лише $2n^2$ операцій. У той час алгоритм впорядкування злиттям написаний без особливої турботи про ефективність і вимагає $50n\log n$ операцій.

Для впорядкування мільйона елементів послідовності одержуємо для потужного комп'ютера:

$$\frac{2(10^6)^2 \text{операцій}}{10^8 \text{операційза секунду}} = 20000 \text{сек} \approx 5{,}56 \text{год}$$

натомість для повільного лише

$$\frac{50(10^6)\log(10^6) \text{операцій}}{10^6 \text{операцій за секунду}} \approx 1000 \text{сек} \approx 17 \text{хв}.$$

Як бачимо, перевага більш ефективного алгоритму очевидна.

Алгоритми впорядкування, які були нами використанні для ознайомлення з методикою визначення складності алгоритмів, базувалися на операціях порівняння та обміну елементів. Проте, існують алгоритми впорядкування які не тільки не використовують

операцію порівняння елементів, а й мають часову складність у найгіршому випадку $T_{max}(n)=an+b$. Серед цих алгоритмів виділимо *алгоритм впорядкування підрахунком* і проаналізуємо його.

Нехай дана послідовність розмірністю n елементів, кожен з яких має цілочисельне додатнє значення, і відоме значення k найбільшого елементу з всієї послідовності. Ідея впорядкування підрахунком полягає у записі до допоміжної послідовності в позицію, що відповідає значенню елемента вихідної послідовності, кількості таких елементів.

Проілюструємо роботу алгоритму впорядкування підрахунком наступним прикладом, що записаний на псевдокоді.

(1) ***for*** i←1 ***step*** 1 ***until*** *k* ***do*** *c*[*i*] ←0
(2) ***for*** i←1 ***step*** 1 ***until*** *n* ***do*** *c*[*a*[*i*]] ← *c*[*a*[*i*]] +1
(3) ***for*** i←2 ***step*** 1 ***until*** *k* ***do*** *c*[*i*] ← *c*[*i*] + *c*[*i*-1]
(4) ***for*** i←*n* ***step*** -1 ***until*** *1* ***do***
begin
(5) *b*[*c*[*a*[*i*]]] ← a[i]
(6) *c*[*a*[*i*]] ← *c*[*a*[*i*]]-1
end

Проаналізуємо роботу алгоритму впорядкування підрахунком, в якому використовуються вихідна *a*[1..*n*], результуюча *b*[1..*n*] і допоміжна *c*[1..*k*] послідовності. Після ініціювання допоміжної послідовності *c*[1..*k*] оператором повторення в рядку (1), до *i* позиції цієї послідовності записується кількість елементів послідовності *a*[1..*n*], значення яких дорівнює *i* (рядок 2). В рядку (3) алгоритму відбувається обчислення часткових сум послідовності *c*[1], *c*[2],.., *c*[*k*], що відповідають кількості елементів, значення яких не перевищує *i.* В рядках (4)-(5) кожен елемент послідовності *a*[1..*n*] записується в необхідну позицію послідовності *b*[1..*n*]. Дійсно, якщо всі *n* вихідної послідовності *a*[1..*n*] відмінні один від одного, то в упорядкованій послідовності *b*[1..*n*] елемент *a*[*i*] повинен бути записаним в позицію *c*[*a*[*i*]]. У випадку, коли в вихідній послідовності *a*[1..*n*] зустрічаються елементи з однаковими значеннями, то після кожного запису елемента *a*[*i*] в послідовність *b*[1..*n*] (рядок 5) значення *c*[*a*[*i*]] елемента зменшується на 1 (рядок (6)). Це дозволить на наступному етапі записати елемент, значення якого

дорівнюватиме $a[i]$, в позицію зліва. Такий підхід демонструє одну із властивостей алгоритмів впорядкування, що називається *стійкістю*. Саме стійкість алгоритму впорядкування вказує на необхідність запису елементів з однаковими значеннями в тому порядку, в якому вони були записані у вихідній послідовності. Така властивість особливо актуальна за умови впорядкування послідовностей елементів, кожен з яких є записом, що разом із чисельними значеннями зберігає і додаткову інформацію.

Аналіз складності алгоритму впорядкування підрахунком показує, що час $T_{max}(n)$ у найгіршому випадку є лінійною функцією від n і k, тобто має вигляд $T_{max}(n)=an+bk$ для деяких констант a і b.

2.1.2. Асимптотична часова складність алгоритмів

Література:[4,29-35;4,265-275;5,105-108;13,26-30;13,811-837;17,352-416]

Ключові поняття: *асимптотична часова складність, асимптотично точна оцінка складності, швидкість росту часової складності, Θ-символіка, поліноміальна й експоненціальна складність, класи P і NP.*

Раніше було зазначено, що одиниця виміру $T(n)$ точно не визначена, проте розумілася нами як кількість кроків, що виконані на ідеалізованому комп'ютері.

Справді, застосувати такі стандартні одиниці виміру часу як секунди і мілісекунди ми не можемо, адже на час виконання програми, що реалізовує той чи інший алгоритм, впливають ще й такі фактори як кількість і якість скомпільованого коду, архітектура і набір внутрішніх інструкцій ЕОМ тощо.

Час виконання також може суттєво залежить від вибраної множини тестових вхідних даних. Звичайно, як зазначає Н.Вірт, для практичних цілей корисно мати експериментальні дані, що можуть „пролити світло" на коефіцієнти c_i, тим самим провести подальшу оцінку різних методів впорядкування. Наведені у роботі [5, с.105-108] експериментальні результати (в мілісекундах) (див. *Таблиця 2.11.*) базуються на обчисленнях, що були проведені на системою CDC 6400 (в якості мови програмування для реалізації алгоритмів була, звичайно ж, обрана мова Pascal).

Таблиця 2.11.

Експериментальні дані роботи алгоритмів (в мілісекундах)

	Упорядкована послідовність		Випадковий розподіл		Послідовність упорядкована у зворотному порядку	
	256	512	256	512	256	512
Впорядкування обміном	540	2165	1026	4054	1492	5931
Впорядкування вибором	489	1907	509	1956	695	2675
Впорядкування вставками	12	23	366	1444	704	2836

В якості критерію визначення ефективності алгоритмів можуть використовуватися різні показники. Зокрема, Д.Кнут наводить в якості одного з поширених тестів для порівняльної оцінки різних алгоритмів задачу впорядкування послідовності розмірності n 100-символьних записів. Так, автор зазначає, що терабайтова послідовність – 10^{10} записів по 100 символів – була впорядкована у вересні 1997 року за 2,5 години. При цьому використовувалася система Silicon Graphics Origin2000, що складалася з 32 процесорів, 8 Гбайт оперативної пам'яті, 559 дисків по 4 Гбайти, та програмне забезпечення *NSort*, в основі якого лежить неопублікований алгоритм впорядкування [11, с.420].

Наведені приклади, як і більш сучасні результати експериментального порівняння алгоритмів впорядкування, не спростовують жодної з оцінок складності, що були отримані теоретично. Тим більше, що ця залежність стає очевидною при реалізації одного і того ж алгоритму на різних комп'ютерах, компіляторах, при використанні різних вхідних даних.

Ось чому у більшості випадків роблять висновки на зразок: „часова складність того чи іншого алгоритму пропорційна n^2". Константи пропорційності у більшості випадків також не можна точно визначити. Пов'язано це з тим фактом, що у більшості випадків алгоритми реалізуються програмами, які записані операторами мов програмування високого рівня. Отже, виникає необхідність запису цих алгоритмів в термінах „кроків", кожен з

яких повинен виконуватися за скінчений фіксований час після трансляції їх у машинну мову будь-якої ЕОМ. Проте, точно визначити час виконання будь-якого кроку алгоритму дуже важко, адже цей час залежить не тільки від „природи" самого кроку, а й від процесу трансляції і машинних інструкцій певного комп'ютера. Ось чому замість точної оцінки ефективності алгоритму, що придатна (актуальна, валідна) для певної обчислювальної системи, прийнято використовувати менш точну, але більш загальну *асимптотичну часову складність* як основну міру ефективності виконання алгоритму.

Для опису *швидкості росту* часової складності алгоритмів використовується *Θ-символіка* („тета-символіка"). Наприклад, коли говорять що час виконання $T(n)$ деякого алгоритму має порядок $\Theta(n^2)$, тобто $T(n) = \Theta(n^2)$, то вважають, що існують такі константи $c_1, c_2 > 0$ і таке число n_0, що $c_1 n^2 \le T(n) \le c_2 n^2$ для всіх $n \ge n_0$. Оскільки, як було зазначено вище, складність алгоритму – це функція, значення якої залежить від розмірності n даних задачі, то запис $T(n) = \Theta(n^2)$ можна подати у вигляді $f(n) = \Theta(n^2)$. Враховуючи різні значення пропорційності для n різних алгоритмів, точнішим буде використання запису $f(n) = \Theta(g(n))$, де $g(n)$ – деяка функція від n. Зміст цього запису полягає у тому , що існують такі константи $c_1, c_2 > 0$ і таке число n_0, що $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ для всіх $n \ge n_0$. Якщо для деякого алгоритму $f(n) = \Theta(g(n))$, то функцію $g(n)$ вважають *асимптотично точною оцінкою* його складності.

Для пояснення скористаємося прикладом. Нехай часова складність деякого алгоритму $T(n) = (1/2)n^2 - 3n$. Перевіримо, що цей алгоритм має швидкість росту $\Theta(n^2)$, тобто $(1/2)n^2 - 3n = \Theta(n^2)$. Для цього зазначимо константи c_1, c_2 і число n_0, так щоб нерівність $c_1 n^2 \le (1/2)n^2 - 3n \le c_2 n^2$ виконувалася для всіх $n \ge n_0$. Розділимо нерівність на n^2 і отримаємо $c_1 \le 1/2 - 3/n \le c_2$. Таким чином, для виконання другої нерівності достатньо прийняти $c_2 = 1/2$. Перша ж є нерівність виконується, наприклад, при $n_0 = 7$ і $c_1 = 1/14$.

Зазначимо, що запис $f(n) = \Theta(g(n))$ включає в себе дві оцінки росту швидкості: верхню $O(g(n))$ („*о* від *же* від *ен*") і нижню $\Omega(g(n))$ („*омега* від *же* від *ен*").

Вважають, що:

- $f(n) = O(g(n))$, якщо знайдеться така константа $c > 0$ і таке число n_0, що $0 \le f(n) \le cg(n)$ для всіх $n \ge n_0$;

- $f(n) = \Omega(g(n))$, якщо знайдеться така константа $c>0$ і таке число n_0, що $0 \le cg(n) \le f(n)$ для всіх $n \ge n_0$.

Із наведеного випливає, що для функції $f(n)$ властивість $f(n) = \Theta(g(n))$ виконується тоді і тільки тоді, коли $f(n) = O(g(n))$ і $f(n) = \Omega(g(n))$.

Спрощення, яких ми припускалися при оцінці складності алгоритму $T(n)$, базуються на правилах виконання операцій додавання та множення в Θ-символіці.

Правило сум. Нехай $T_1(n)$ і $T_2(n)$ - час виконання двох програмних фрагментів P_1 і P_2. $T_1(n)$ має швидкість росту $O(f(n))$, $T2(n)$ - $O(g(n))$. Тоді $T_1(n) + T_2(n)$, тобто час послідовного виконання фрагментів P_1 і P_2, має швидкість росту $O(max(f(n),g(n)))$. Для доказу цього нагадаємо, що існують константи c_1, c_2, n_1 і n_2 *такі, що за умови* $n \ge n_1$ виконується нерівність $T_1(n) \le c_1 f(n)$, та аналогічно $T_2(n) \le c_2 g(n)$, якщо $n \ge n_2$. Нехай $n_0 = \max(n_1, n_2)$. Якщо $n \ge n_0$, то, очевидно, що і $T_1(n)+T_2(n) \le c_1 f(n)+c_2 g(n)$. Звідси випливає, що при $n \ge n_0$ справедлива нерівність $T_1(n)+T_2(n) \le (c_1 + c_2)\max(f(n),g(n))$. Остання нерівність і означає, що $T_1(n)+T_2(n)$ має порядок росту $O(max(f(n), g(n)))$. Наприклад, скористаємося правилом сум, для обчислення часу послідовного виконання програмних фрагментів з повторенням і розгалуженнями. Нехай є три фрагменти з часами виконання відповідно $O(n^2)$, $O(n^3)$ і $O(nlogn)$. Тоді час послідовного виконання перших двох фрагментів має порядок $O(max(n^2,n^3))$, тобто $O(n^3)$. Час виконання всіх трьох фрагментів має порядок $O(max(n^3,nlogn))$, тобто знову $O(n^3)$. У загальному випадку час виконання кінцевої послідовності програмних фрагментів, без врахування констант, має порядок фрагмента з найбільшим часом виконання.

З правила сум також випливає, що якщо $g(n)<f(n)$ для всіх n, що перевищують n_0, то вираз $O(f(n)+g(n))$ еквівалентний $O(f(n))$. Наприклад, вираз $O(n^2+n)$ еквівалентний виразу $O(n^2)$.

Правило добутку. Якщо $T_1(n)$ і $T_2(n)$ мають швидкість росту $O(f(n))$ і $O(g(n))$ відповідно, то добуток $T_1(n)*T_2(n)$ має швидкість росту $O(f(n)*g(n))$. З правила добутку випливає, що $O(n^2/2)$ еквівалентно $O(n^2)$.

Як було показано раніше, часову складність алгоритмів можна оцінити за допомогою функцій часу виконання, зневажаючи при цьому константами пропорційності. За такої умови, алгоритм з

часом виконання $\Theta(n^2)$, наприклад, ефективніше алгоритму з часом виконання $\Theta(n^3)$. Константи пропорційності залежать не тільки від компілятора і комп'ютера, але і від властивостей самого алгоритму. Нехай при заданій комбінації „компілятор-комп'ютер" один алгоритм виконується за $100n^2$ мілісекунд, а інший - за $5n^3$ мілісекунд. Чи може другий алгоритм бути ефективнішим, ніж перша?

Відповідь на це питання залежить від розміру вхідних даних n. При розмірі вхідних даних $n< 20$ алгоритм з часом виконання $5n^3$ завершиться швидше, ніж алгоритм з часом виконання $100n^2$. З цього робимо висновок про те, що, коли алгоритм в основному опрацьовує вхідні дані незначної розмірності, перевагу необхідно віддати алгоритму з часом виконання $\Theta(n^3)$. Однак при зростанні n відношення часу виконання $5n^3/100n^2= n/20$ також росте. Тому при значних n алгоритм з часом виконання $\Theta(n^2)$ стає переважнішим алгоритму з часом виконання $\Theta(n^3)$. Якщо навіть при порівняно невеликих n, коли час виконання обох програм приблизно однаково, вибір ефективнішого алгоритму є дещо складним, то для більшої надійності роблять вибір на користь алгоритму з меншою швидкістю росту складності. Ще однією причиною, що змушує віддавати перевагу алгоритмам з найменшої швидкістю росту складності, є той факт, що чим меншою є швидкість росту складності алгоритму, тим більший розмір задачі, яку можна обчислити на ЕОМ. Іншими словами, якщо збільшується швидкість обчислень комп'ютера, то росте також і розмір задач, розв'язуваних на комп'ютері. Однак незначне збільшення швидкості обчислень комп'ютера приводить тільки до невеликого збільшення розмірності задач, розв'язуваних протягом фіксованого проміжку часу.

Для пояснення скористаємося прикладом, наведеним в [3, с. 12-14]. Нехай є п'ять алгоритмів A_1... A_5 з часовими складностями n, $n\log_2 n$, n^2, n^3 і 2^n відповідно (див. *Таблиця 2.12.*). Припустимо, що одиниця інформації перероблюється за одну мілісекунду. Тоді алгоритм A_1 може обробити за одну секунду вхідну інформацію розмірністю 1000 одиниць, A_1 -140, в той час як алгоритм A_5 – лише 9 одиниць. Складемо залежність можливих розмірностей задачі від складності алгоритму.

Таблиця 2.12.

Залежність можливих розмірів задачі від складності алгоритму

Алгоритм	Часова складність	Розмірність задачі		
		1 с	1 хв	1 год
A_1	n	1000	$6*10^4$	$3{,}6*10^6$
A_2	$n\log_2 n$	140	4893	$2*10^5$
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Розглянемо, як вплине збільшення швидкодії обчислювальної техніки на можливості використання алгоритмів A_1... A_5.

Враховуючи, що алгоритм обчислюється на одній ЕОМ, вважаємо час виконання алгоритму дорівнює $T_a=f_a(n)$. Зафіксуємо максимально припустимий час T, на протязі якого працювати ЕОМ. Тоді, вирішуючи рівняння $f_a(n)=T$ відносно n, можна знайти найбільшу розмірність N, для якої ще можна застосувати алгоритм А, не перевищуючи час Т. Наприклад, $f_a(n)=n^2$, то $N=\sqrt{T}$.

Тепер, використовуємо для виконання цих алгоритмів ЕОМ із збільшеною в 10 разів швидкодією (див. *Таблиця 2.13.*). Здавалося, що такий підхід призведе до збільшення в 10 разів максимального часу роботи старої ЕОМ, тобто $f_a(n)=10T$. Дійсно, розмірність n (стара) і N(нова) пов'язані відношенням $f_a(N)=10\,f_a(n)$. Зокрема, якщо $f_a(n)=n^2$, то $(N^2)=10(n^2)$, звідки $N=n\sqrt{10}\approx 3.16n$. Проте, у випадку $f_a(n)=2^n$, то $2^N=10*2^n$, звідки $N=n+\log_2 10\approx n+3.3$.

Як видно з табл. 2.13., десятикратне збільшення швидкодії ЕОМ призвело до збільшення розмірності, що може бути обчислена алгоритмом A_3, у три рази, а алгоритмом A_5 - всього на три одиниці. Якщо як основу для порівняння взяти 1 хвилину, то, заміняючи алгоритм A_4 алгоритмом A_3, можна вирішити у 6 разів більшу задачу, а заміняючи A_4 на A_2 – більшу в 125 разів. Ці результати роблять набагато більше враження, ніж дворазове поліпшення, що досягається за рахунок десятикратного збільшення швидкості. Якщо за основу для порівняння взяти 1 годину, то розходження виявляється ще значніше.

З табл. 2.13. видно, що у випадку поліноміальних алгоритмів розмірність задачі із збільшенням швидкодії ЕОМ збільшується на мультиплікатну константу, тоді як для експоненціальних алгоритмів має місце збільшення на адитивну константу.

Таблиця 2.13.

Залежність можливих розмірів задачі від складності алгоритму

Алгоритм	Часової складності	Розмірність задачі		
		x1	x10	x100
A_1	n	a_1	$10*a_1$	$100*a_1$
A_2	$n\log_2 n$	a_2	$10*a_2$	$10*a_2$
A_3	n^2	a_3	$3{,}16*a_3$	$10*a_3$
A_4	n^3	a_4	$2{,}15*a_4$	$4{,}64*a_4$
A_5	2^n	a_5	$a_5+3{,}3$	$a_5+6{,}64$

Дослідження складності алгоритмів привели до усвідомлення важливого факту: подібно до того як існують алгоритмічно нерозв'язувані проблеми, існують і задачі об'єктивно складні (такі, складність яких суттєво не зменшується із покращенням швидкодії ЕОМ).

Підвищення фізичної швидкодії зменшує час фізичного часу в константне число раз, однак не змінює ступінь поліному у функції складності. Зміна архітектури ЕОМ іноді може дещо зменшити цей ступінь. Якщо ж задача має складність вище поліноміальної (2^n, 3^n тощо), то складність цієї задачі зберігається при будь-якому прогресі обчислювальної техніки.

Слід зазначити, що множина проблем (задач), для якої існують алгоритми з поліноміальною складністю, називається класом *P*. Звідси виходить, що проблеми з класу *P* є *поліноміально розв'язуваними*. Клас проблем *P* прийнято ототожнювати з класом „практично розв'язуваних" проблем. На відміну від класу *P*, існують проблеми експоненціальної складності (клас *NP*), які відповідають повному перебору підмножини *n* – елементів множини або навіть перевищують його за складністю. Скорочення *NP* походить від англійських слів „*Nondeterministic Polynomial* (*time*)", що перекладається як „недетермінований поліноміальний час". Назва класу пов'язана з тим фактом, що історично клас *NP* задач вперше був описаний у термінах так званих недетермінованих машин Тьюрінга.

Вивчення питання співвідношення класів *P* і *NP* призвело до постановки проблеми $P \overset{?}{=} NP$, яка на сьогодні лишається однією з найскладніших проблем теорії обчислень. Вважається, що *NP* задачі не можуть бути вирішені за поліноміальний час, а, отже,

проблему $P = ?\ NP$ можна сформулювати гіпотезою P≠NP. Найбільш вірогідне співвідношення між класами *P* і *NP* в такому випадку можна зобразити рис. 2.8.

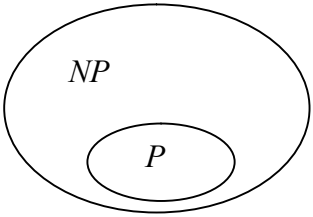


Рис. 2.8. Співвідношення між класами *P* і *NP*

Звичайно, хоча експоненціальна функція (така, як 2^n) росте швидше будь-якої поліноміальної функції від *n*, для невеликих значень *n* алгоритм, що вимагає $\Theta(2^n)$ часу, може виявитися ефективніше багатьох алгоритмів із поліноміально обмеженим часом роботи. Наприклад, сама функція 2^n не перевищує n^{10} до значення *n*, рівного 59. Однак швидкість росту експоненціальної функції настільки стрімка, що такі задачі називають „*важко розв'язними*", якщо у всіх алгоритмів, які її вирішують, складність щонайменше експоненціальна. Ще одна перевага поліноміальних алгоритмів полягає в тому, що, коли на практиці вдається вирішити задачу за поліноміальний час, то ступінь поліному в ході експериментальних досліджень часто зменшується. Це пов'язано з тим, що різні дослідники вдосконалюють алгоритм, іноді покращуючи його часову складність до $\Theta(n^3)$ чи ще нижче. Експоненціальні алгоритми, навпаки, на практиці вимагають стільки ж часу як і в теорії. Ось чому від таких алгоритмів відмовляються як тільки розроблено алгоритм з поліноміальною часовою складністю. На сьогодні відомо багато задач класу *NP*, пов'язаних із різними областями математики та інформатики: математичною логікою, теорією графів, комп'ютерними мережами, множинами, математичним програмуванням, алгеброю і теорією чисел, оптимізацією програм тощо. Наявність цих задач, практична значущість яких не дозволяє обмежитись простим віднесенням їх до задач класу *NP*, вказує на необхідність перегляду стратегій їх виконання. Серед підходів, що є менш безнадійними, аніж спроби виконання задач класу *NP*, слід виділити:

1. *Наближені* алгоритми. Такі алгоритми дозволяють одержувати не оптимальні рішення, а рішення, що відрізняються

від дійсного оптимуму не більше ніж на фіксовану частку цього оптимуму. На практиці таке, близьке до оптимального, рішення може бути достатнім.

2. *Ймовірнісні* алгоритми. Іноді виявляється можливим побудувати алгоритми, які дуже часто щодо деякого ймовірнісного розподілу на індивідуальних задачах працюють досить непогано – як в сенсі якості одержаних рішень, так і часу, витраченого на їх одержання.

3. *Евристичні* алгоритми. Алгоритми, математична основа яких часто дуже сумнівна, дозволяють на практиці отримувати рішення, що відрізняються на декілька відсотків від оптимального.

Звісно, запропоновані підходи не виключають використання експоненціальних алгоритмів, тим більше що такі алгоритми при незначних *n* можуть, зрештою, бути досить оптимальними. Крім того, використання методів розробки ефективних алгоритмів (наприклад, методу розгалужень і меж чи динамічного програмування) у поєднанні з евристичними ідеями (наприклад, „жадібна" стратегія чи генетичний алгоритм) дозволяють проектувати алгоритми, результати роботи яких на практиці є досить прийнятними.

2.2. МЕТОДИ РОЗРОБКИ АЛГОРИТМІВ

На сьогодні відомо декілька методів побудови ефективних алгоритмів. Серед них слід виділити такі як декомпозиції, метод розгалужень і меж, динамічне програмування, „жадібні" методи, пошук з поверненням і локальний пошук. Слід зазначити, що жоден із вказаних методів не є універсальним, адже вибір того чи іншого методу дуже часто залежить від природи даних, які необхідно опрацювати. Проте знання цих методів є необхідною умовою при розробці ефективних алгоритмів.

2.2.1. Декомпозиція

Література:[*3,75-81;4,276-280;13,15-23]*

Ключові поняття: *декомпозиція, лінійний перегляд, впорядкування злиттям, збалансоване і незбалансоване розбиття* .

Одним з найбільш широко застосованим методом проектування ефективних алгоритмів є *метод декомпозиції* (*метод*

розбивки або *метод "розділяй і пануй"*). Цей метод припускає таку декомпозицію (розбивку) задачі на більш дрібні задачі, що на основі рішень цих більш дрібних задач можна легко одержати рішення вихідної задачі. Продемонструємо принцип роботи методу декомпозиції на прикладі декількох алгоритмів.

Спочатку скористуємося задачею пошуку *k*-елемента у послідовності розмірністю *n*. Найбільш прості алгоритми базуються на послідовному перегляді кожного елемента на відповідність певній властивості, зокрема, чи дорівнює *k*-елемент шуканому. Такий метод називають *лінійним* переглядом (пошуком), а його часова складність відповідає $\Theta(n)$.

Задача пошуку істотно спрощується, якщо послідовність упорядкована. У такому випадку пошук полягає у поділі послідовності розмірністю *n* на дві рівні частини і перевірці приналежності шуканого *k*-елемента лівій (1..*m*) чи правій (*m*+1..*n*) частині. Далі процес рекурсивно продовжується вже над однією з половин розмірності до тих пір, доки не буде знайдено шуканий елемент. Описаний підхід, в основі якого лежить метод декомпозиції, називається „бінарним („логарифмічним") пошуком". Часова складність алгоритму „бінарного пошуку" відповідає $\Theta(\log_2 n)$.

Метод декомпозиції можна використати для впорядкування послідовності. Справді, застосування методу декомпозиції до алгоритму впорядкування вставками дозволяє зменшити його часову складність до $\Theta(n\log_2 n)$. Пояснюється це тим, що при впорядкуванні методом простих вставок *j*-ий елемент порівнюється в середньому близько з *j*/2 раніше впорядкованих елементів. Ось чому загальна кількість порівнянь алгоритму дорівнює приблизно $(1+2+..+n)/2 \approx n^2/4$, що досить багато, навіть при незначних *n*. Проте, алгоритм впорядкування вставками можна вдосконалити, скориставшись „бінарним пошуком": позиція вставки *j*-ого елемента буде визначена за $\log_2 n$ операцій порівняння з відповідним чином вибраними елементами впорядкованої послідовності. Наприклад, при необхідності визначення позиції елементу *a*[64], його значення спочатку порівнюється елементом *a*[32], потім, якщо це значення менше, воно порівнюється з *a*[16], якщо ж більше – то з *a*[48] тощо. Таким чином, позиція елемента *a*[64] буде знайдена за 6 порівнянь. Загальна кількість порівнянь

для послідовності розмірністю n приблизно дорівнює $n\log_2 n$. Наведений алгоритм називається *алгоритмом впорядкування бінарними вставками*, а його часова складність відповідає $T(n)=\Theta(n\log_2 n)$

Проілюструємо роботу алгоритму впорядкування бінарними вставками наступним прикладом, що записаний на псевдокоді.

```
for i←2 step 1 until n do
 begin{for}
  key←a[i]
   If b < a[i-1] then
       begin
        l ← 1;
        r ← i;
        repeat
         m ← (l + r) \ 2; { „\" - цілочисельне ділення}
         if b < a[m] then
                    r ← m
           else  l← m + 1;
        until (l = m);
        for l←i-1 step -1 until m do   a[l+1] ← a[l];
       a[m] ←b;
       end;
 end{for}
```

Алгоритм впорядкування бінарними вставками неєдиний алгоритм, що використовує метод декомпозиції. Розглянемо принцип роботи так званого *алгоритму впорядкування злиттям*.

У алгоритмі впорядкування злиттям послідовність спочатку поділяється на дві половини меншої розмірності. Потім відбувається впорядкування кожної з половин окремо. Після цього дві упорядковані послідовності половинного розміру зливаються в одну. Рекурсивне розбиття задачі на менші відбувається доти, поки розмірність послідовності не дорівнюватиме 1 (будь-яку послідовність розмірністю в 1 елемент можна вважати упорядкованим).

Підпрограму впорядкування злиттям можна записати на псевдокоді у наступному вигляді:

```
procedure Megre_Sort (i,j)
 if i<j then
 begin
     m←(i+j)\2
     Megre(Megre_Sort(i,m), Megre_Sort(m+1,j))
end
```

Виконаємо аналіз складності цієї підпрограми. Як було сказано раніше, метод декомпозиції базується на рекурсивному виклику деякої підпрограми для вирішення задачі меншої розмірності. Таким чином, ми повинні враховувати час, витрачений на рекурсивні виклики. Внаслідок цього одержуємо рекурентне співвідношення, яке і дозволяє оцінити часову складність алгоритму. Припустимо, що алгоритм розбиває задачу розмірністю n на a підзадач, кожна з яких має в b разів меншу розмірність. Будемо вважати, що розбиття вимагає часу $D(n)$, а з'єднання отриманих рішень — часу $C(n)$. Тоді одержуємо співвідношення для часу роботи $T(n)$ на задачах розмірністю n (у найгіршому випадку): $T(n)=aT(n/2)+D(n)+C(n)$. Це співвідношення виконане для досить великих n, коли задачу має сенс розбивати на підзадачі. Для малих n, коли така розбивка неможлива або не потрібно, застосовується якийсь прямий метод рішення задачі. Припустимо, що розмірність послідовності є ступінь двійки (хоча це обмеження не дуже істотне). Тоді на кожному кроці послідовність поділяється навпіл. Розбиття на частини вимагає часу $\Theta(1)$, а злиття – часу $\Theta(n)$. Одержуємо співвідношення:

$$T(n)=\begin{cases}\Theta(n), & \text{якщо } n=1,\\ 2T(n\setminus 2)+\Theta(n), & \text{якщо } n>1.\end{cases}$$

Це співвідношення приводить до значення $T(n)=\Theta(n\log_2 n)$.

Таким чином, часова складність алгоритму впорядкування злиттям є $T(n)=\Theta(n\log_2 n)$, що виявляється значно ефективнішою за інші алгоритми зі часовою складністю в $T(n)=\Theta(n^2)$.

Рекурентні рівняння часто зустрічаються при аналізі складності алгоритмів. Тому наведемо рішення таких рівнянь в узагальненому вигляді.

Нехай a,b і c – додатні константи. Рішення рекурентного рівняння:

$$T(n)=\begin{cases} b, & \text{якщо } n=1, \\ aT(n\backslash c)+bn, & \text{якщо } n>1 \end{cases}$$

де n – ступінь числа c, має вигляд

$$T(n)=\begin{cases} \Theta(n), & \text{якщо } a<c, \\ \Theta(n\log n), & \text{якщо } a=c, \\ \Theta(n^{\log_c a}), & \text{якщо } a>c. \end{cases}$$

В якості доведення покажемо, що коли n – ступінь числа c, то $T(n)=bn\sum_{i=0}^{\log_c n} r^i$, де $r=a\backslash c$. Якщо $a<c$, то $\sum_{i=0}^{\log_c n} r^i$ сходиться, а відповідно $T(n)=\Theta(n)$. Якщо $a=c$, то кожним членом ряду буде 1, а всього в ньому $\Theta(\log n)$ членів. Ось чому $T(n)=\Theta(n\log n)$. І якщо $a>c$, то $bn\sum_{i=0}^{\log_c n} r^i = bn\frac{r^{\log_c n}-1}{r-1}$, що складає $\Theta(n^{\log_c a})$.

Слід зазначити, що при використанні методу декомпозиції бажано, щоб підзадачі були однакової розмірності. Наприклад, алгоритм впорядкування вставками можна розглядати як поділ послідовності на дві половини – одна розмірністю в 1, а інша – в n-1 елементів (див.рис.2.9.).

Оскільки процедура поділу займає час $\Theta(n)$, для часу роботи $T(n)$ одержуємо співвідношення $T(n)=T(n-1)-\Theta(n)$. Оскільки $T(1)=\Theta(1)$, то

$$T(n)=T(n-1)+\Theta(n)=\sum_{k=1}^{n}\Theta(k)=\Theta\left(\sum_{k=1}^{n}(k)\right)=\Theta(n^2)$$

Як бачимо, за умови максимально незбалансованого розбиття час роботи алгоритму вставками складає $\Theta(n^2)$. Найкращим випадком вважається ситуація, коли послідовність розподіляється на дві рівні частини (див. рис. 2.10.).

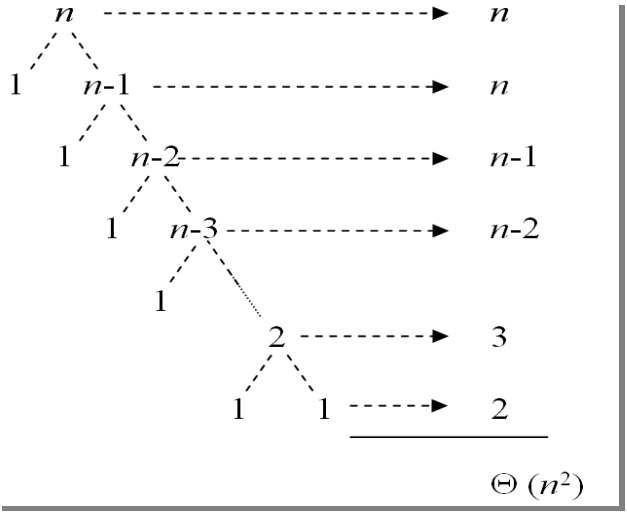


Рис. 2.9. Приклад незбалансованого розбиття

Рекурентне співвідношення в такому випадку має наступний вигляд $T(n) = 2T(n\backslash 2) + \Theta(n)$, що відповідає $T(n) = \Theta(n \log_2 n)$.

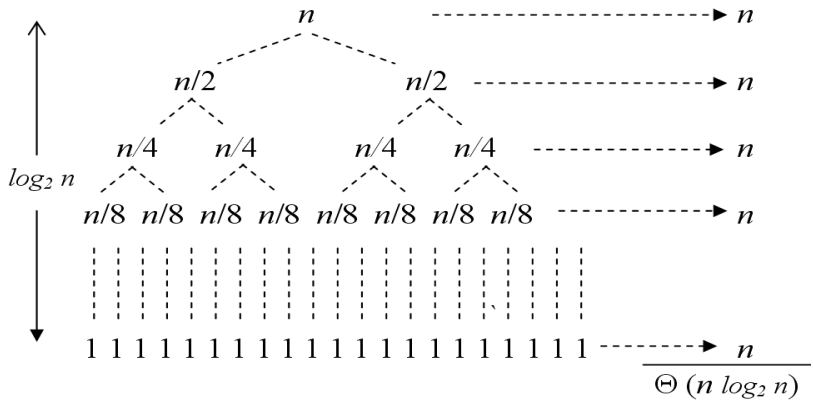


Рис. 2.10. Приклад збалансованого розбиття

2.2.2. Метод розгалужень і меж

Література:*[4,296-298;17,446-461]*
Ключові поняття: *метод розгалужень і меж, оптимізаційні задачі, задача комівояжера (traveling-salesman problem)*, *повний перебір, дерево рішень, нижня оцінка вартості, правила прийняття рішень.*

Для пояснення принципу роботи *методу розгалужень і меж* серед множини задач класу *NP* виділимо клас так званих *дискретних оптимізаційних задач*.

Всі задачі цього класу мають перелік характерних властивостей, зокрема:

1. В кожній задачі є лише кінцева кількість варіантів, з яких необхідно зробити вибір.
2. Кожному з варіантів зіставлена деяка чисельна характеристика.
3. Необхідно обрати варіант, чисельна характеристика якого досягає екстремуму.

Однією із перших таких задач, що привернули увагу математиків, є так звана *задача комівояжера (traveling-salesman problem)*. Зміст задачі полягає у наступному. Нехай є *n* міст $A_1, A_2, A_3,.., A_n$ із заданими відстанями між ними $d_{ij}(i, j=1,2,..n)$. Необхідно, вирушаючи з міста A_1, обрати такий маршрут переміщення $A_1, A_{i,1}, A_{i,2},.., A_{1,j}, A_{2,j},.., A_{1,j}$, за якого комівояжер, побувавши в кожному з міст, повернувся б в A_1, пройшовши мінімально можливий сумарний шлях.

У загальному випадку задача комівояжера формулюється в термінах *теорії графів* і полягає в пошуку в неорієнтованому *графі* з ваговими значеннями *ребер* такого маршруту (простого циклу, що включає всі вершини), у якому сума всіх ваг ребер буду мінімальною.

В якості основи для задачі комівояжера скористаємося графом з п'ятьма вершинами (див. рис.2.11.). Вага (вартість) кожного ребра, що зазначена цифрами на рис. 2.11., не обов'язково дорівнює довжині ребра. Це може бути, наприклад, час шляху чи плата за проїзд.

Найбільш очевидним методом рішення задачі комівояжера, як і інших дискретних оптимізаційних задач, є повний перебір всіх варіантів. Проте, алгоритм знаходження оптимального маршруту шляхом повного перебору всіх маршрутів з n вузлами має часову складність $\Theta(n!)$, оскільки необхідно переглянути $(n\text{-}1)!$ різних перестановок, а оцінка кожної з перестановок займає час $\Theta(n)$, тобто загальна кількість маршрутів $n!=1*2*3*4..(n\text{-}1)*n$.

Цей процес можна зобразити наступним фрагментом *дерева рішень* (див. рис. 2.12.)

Таким чином, виникає об'єктивна необхідність в пошуку методів побудови більш ефективних алгоритмів для рішення задачі комівояжера, що дозволять скоротити повний перегляд, відкидаючи заздалегідь непридатну множину варіантів.

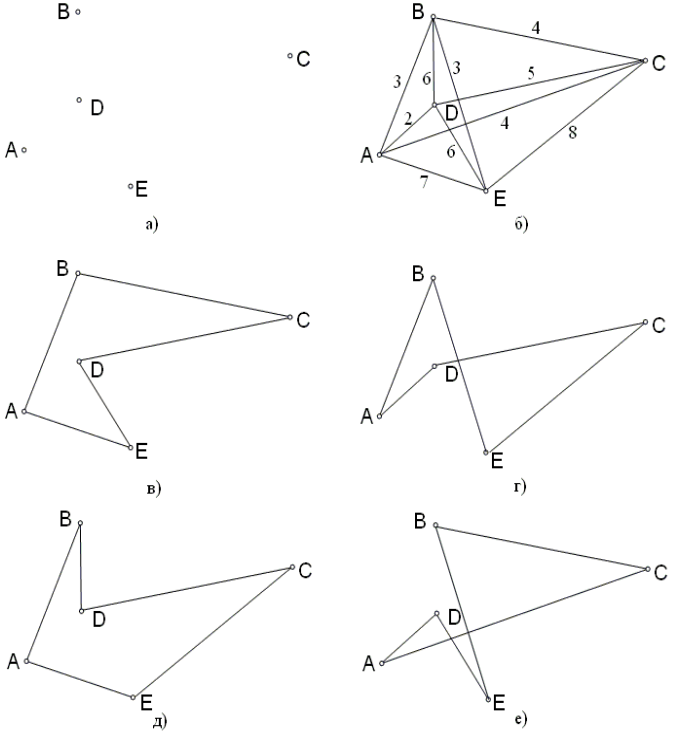


Рис. 2.11. Приклади побудови маршрутів для задачі комівояжера

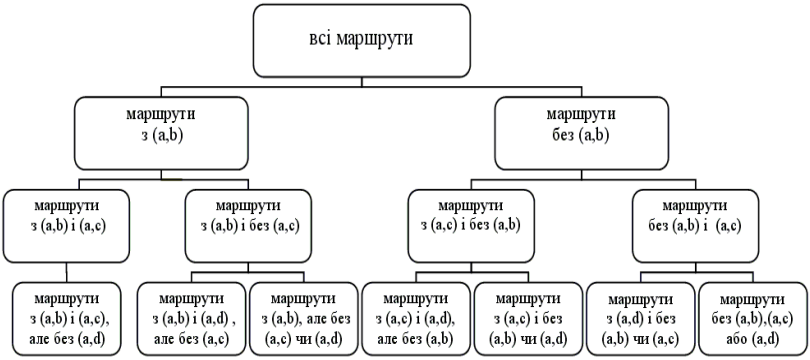


Рис. 2.12. Фрагмент дерева рішень повним переглядом для задачі комівояжера

Метод розгалужень і меж базується на ідеї „розумного" перегляду всіх припустимих варіантів комбінаторної задачі оптимізації. Застереження „розумного" має важливе значення, оскільки відбувається не просто перегляд всіх варіантів, як у випадку повного перегляду, а будується доказ оптимальності деякого рішення на основі послідовного розбиття простору рішень.

Слово „розгалужень" у назві методу відноситься саме до цього процесу розбиття. Термін „меж" відноситься до процесу визначення так званих *нижніх оцінок* (меж) вартості (наприклад, вартості того чи іншого маршруту), що використовуються при побудові доказу оптимальності без повного перегляду всіх варіантів.

Як і у випадку рішення задачі комівояжера повним перебором всіх маршрутів з n вузлами, при використанні методу розгалужень і меж також будується дерево рішень. При цьому кожна вершина (вузол) має два розгалуження (сина): те, що містить певне ребро і те, у якого такого ребра немає.

Ребра дерева рішень переглядаються в лексикографічному порядку, тобто: *(a,b), (a,c),..,(a,f),(b,c),..,(b,f),(c,d)* тощо.

Рішення про необхідність включення чи вилучення того чи іншого ребра приймається на основі наступних *правил*:

правило 1: вибір ребра не призводить до появи вершини зі ступенем три чи більше, тобто кількість ребер, що виходять з цієї вершини не повинна перевищувати два ребра;

правило 2: вибір ребра не призводить до утворення замкнутого контуру (циклу) з ребрами, що були прийняті раніше;

правило 3: якщо вилучення ребра (*x,y*) призводить до того, що у вершини *x* чи *y* немає інших ребер на даному маршруті, тоді ребро (*x,y*) необхідно включити;

правило 4: якщо включення ребра (*x,y*) призводить до того, що у вершини *x* чи *y* буде більше двох ребер на даному маршруті чи утворюється цикл з раніше включеними ребрами, тоді ребро (*x,y*) необхідно вилучити.

На наступному етапі обчислюється нижня межа вартості для всіх рішень задачі. При цьому приймається до уваги, що вартість будь-якого маршруту можна обчислити як половину суми по всім вузлам *n* вартості пар ребер цього маршруту. Таким чином, сума вартості двох ребер, що входять до маршруту, не може мати вартість меншу, ніж половина суми по всім вершинам *n* вартості двох ребер найменшої вартості. Отже, жоден з маршрутів не може мати вартість меншу, ніж половина суми по всім вершинам вартості двох ребер найменшої вартості, що виходять з цієї вершини.

У випадку задачі комівояжера, поданої мал. 2.13, нижня межа вартості маршруту становить 17.5 і обчислюється наступним чином. Спочатку обирають два ребра вершини *A* з найменшою вартість. Це рабра (*a,b*) і (*a,d*) з сумарною вартістю 5. Для вершини *B* такими ребрами є (*a,b*) і (*b,e*) із сумарною вартістю, що дорівнює 6. Аналогічно, сумарна вартість двох ребер з найменшою вартістю вершин *C, D* і *E* дорівнює відповідно 8, 7 і 9. Отже, нижня межа вартості маршруту складатиме (5+6+3+7+9)/2=17.5. Проте, прийняті правила включення і вилучення ребер, зокрема правило 3 і 4, вимагають включити в маршрут ребро (*a*,*e*) і виключити ребро (*b*,*c*). Таким чином, двома ребрами для вершини *A* будуть ребра (*a, d*) і (*a, e*), загальна вартість яких дорівнює 9. Для вершини *b* залишаються ребра (*a,b*) і (*b,e*) із сумарною вартістю, що дорівнює 6. Для вершини *C* вибір ребра (*b*, *c*) неможливий, отже обираються ребра (*a, c*) і (*c, d*), загальна вартість яких дорівнює 9.

Для вершини *D* обраними залишаються ребра (*a,d*) і (*c,d*), тоді як для вершини *E* необхідно вибрати ребра (*a,e*) і (*b,e*). Таким чином, нижня межа маршруту з урахуванням прийнятих обмежень дорівнює (9+6+9+7+10)/2=20.5.

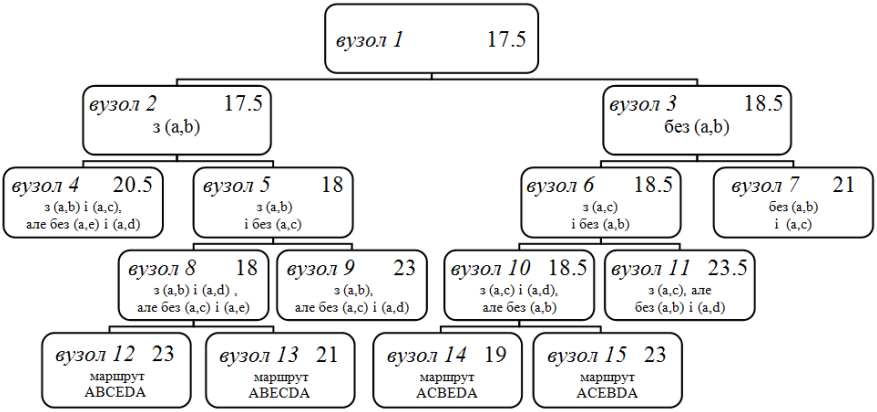


Рис. 2.13. Фрагмент дерева рішень методом розгалужень і меж для задачі комівояжера

При побудові дерева рішень необхідно враховувати той факт, що коли нижня межа для деякого розгалуження буде вищою за найменшу вартість знайденого на даний момент маршруту, то розгалуження вилучається і його нащадки не розглядаються. Якщо жодного з розгалужень вилучити неможливо, то розглядається спочатку розгалуження з меншою нижньою межею, з надією швидше досягти рішення, що буде мати нижчу вартість за вартість найкращого на даний момент рішення. Розглянувши одне розгалуження, можна ще раз перевірити, чи не можливо вилучити інше розгалуження, адже можливо знайдено нове „найкраще" рішення. Таким чином, одержане нове дерево рішень демонструє принцип роботи методу розгалужень і меж (див. рис. 2.13.).

Приймаємо за вихідну точку маршруту вершину A і позначаємо її як вузол 1. Першим ребром в лексикографічному порядку буде ребро (a,b). Далі розглядаються сини вершини A - вузли 2 і 3, до маршрутів яких включено і виключено ребро (a,b) відповідно. Зауважимо, що виключене ребро зазначається із знаком заперечення (інверсії), тобто $\overline{a,b}$. У відповідності до прийнятих правил включення ребра (a,b) в маршрут не підвищує його нижню межу, проте виключення цього ребра призводить до підвищення нижньої межі до 18.5, адже тепер вартість найменших двох ребер для вершин A і B становить 6 і 7 відповідно. Отже, спочатку аналізуються нащадки вузла 2.

Наступним ребром у лексикографічному порядку буде ребро (*a,c*). Таким чином, ми переходимо до вузлів 4 і 5, що відповідають маршрутам, в яких ребро (*a,c*) відповідно включається або виключається. Стосовно вузла 4 можна зробити висновок, що ні ребро (*a,d*), ні ребро (*a,e*) не можуть входити в маршрут (правило 1 прийняття ребер). Відповідно розглядаються спочатку вузли 5, а потім 4 і переходимо до ребра (*a,d*). Нижні межі для вузлів 8 і 9 дорівнюють відповідно 18 і 23. Для кожного з цих вузлів нам відомі три ребра з ребер, що виходять з вершини *A*, тому ми можемо зробити висновок стосовно ребра (a,e). Далі розглянемо синів вузла 8. Першим ребром, що залишилося, у лексикографічному порядку буде ребро (*b*,*c*). Якщо ми включимо в маршрут це ребро, то не зможемо включити ребро (*b,d*) чи (*b*,e), тому що уже виключено ребро (*a,b*). Оскільки ми уже виключили ребра (*a,e*) і (*b*,*c*), у нас повинні бути ребра (*c,e*) і (*d,e*). Ребро (*c,d*) не може входити в маршрут, інакше у вершин *C* і *D* три прилеглі до них ребра входили б у маршрут. Залишається один маршрут (*a,b,c,e,d,a*), вартість якого дорівнює 23. Аналогічно можна довести, що вузол 13 з виключеним ребром (*b*,*c*) є маршрутом (*a,b,e,c,d,a*), вартість якого дорівнює 21. Далі повертаємося до вузол 5 і розглядаємо його другого сина, вузол 9. Проте, нижня межа цього вузла дорівнює 23, що перевершує найкращу на даний момент вартість – 21. Отже, ми видаляємо вузол 9 і повертаємося до вузла 2, де виконуємо аналіз його другого сина, вузла 4. Нижня межа для вузла 4 дорівнює 20.5, проте її цілочисельне значення - 21- вказує на те, що жоден з маршрутів, що містить вузол 4, не може мати вартість меншу за 21. Але, оскільки у нас вже є маршрут з вартістю 21, нам не прийдеться розглядати нащадків вузла 4, а отже, вилучаємо вузол 4.

На наступному етапі повертаємося у вузол 1 і розглядаємо його другого сина – вузол 3. На рівні вузла 3 ми розглянули тільки одне ребро (*a,b*). Вузли 6 і 7 є синами вузла 3. Вузол 6 відповідає тим маршрутам, що містять ребро (*a,c*), але не містять ребро (*a,b*). Нижня межа вузла 6 становить 18,5. Вузол 7 відповідає тим маршрутам, що не містять ні ребро (*a,c*), ні ребро (*a,b*), звідси випливає висновок, що ці маршрути містять ребра (*a,d*) і (*a,e*). Нижня межа для вузла 7 дорівнює 21, тому можемо вилучити його, оскільки уже відомий маршрут з такою вартістю.

Продовжується аналіз над синами вузла 6. Це вузли 10 і 11. Вузол 11 вилучається, оскільки його нижня межа перевершує вартість найкращого зі знайдених на даний момент маршрутів. Синами вузла 10 є вузли 14 і 15, що відповідають маршрутам, що включають і вилучають ребро (*b*,*c*). Проте, у відповідності до правила 1 і 2 прийняття ребер робити висновок, що вузли 14 і 15 є окремими маршрутами. Один з них (*a,c,b,e,d,a*) має найменшу серед усіх маршрутів вартість – 19, а це означає, що він і є самим оптимальним маршрутом комівояжера.

Експериментальні дані дозволяють стверджувати, що метод розгалужень і меж для рішення задачі комівояжера є розумною альтернативою повному перебору, а його часова складність дорівнює $\Theta(1.26^n)$.

2.2.3. Динамічне програмування

Література:[3,83-85;4,280-288;13,299-326;17,461-465]
Ключові поняття: *динамічне програмування, принцип оптимальності Р.Белмана, табличний підхід.*

Рекурсивна техніка, яка була використана при розгляді метода декомпозиції, корисна, якщо задачу можна розбити на підзадачі за розумний час, а сумарна кількість підзадач буде незначною. Якщо, наприклад, сума розмірностей підзадач дорівнює *an* для деякої константи $a>1$, то рекурсивний алгоритм матиме поліноміальну часову складність. За умови, коли розподіл задачі розмірністю *n* зводить до появи *n* задач розмірністю *n*-1, то рекурсивний алгоритм, ймовірніше, матиме експонентціальну складність. У такому випадку часто можна одержати більш ефективні алгоритми за допомогою табличної техніки, яку часто називають *динамічним програмуванням*.

Слід зазначити, що *динамічне програмування* не обов'язково пов'язане з процесом написання програм. Сам термін „*динамічне програмування"* походить від теорії керування і означає процес покрокового рішення задач, коли на кожнім кроці вибирається одне рішення з множини допустимих (на цьому кроці) рішень, причому таке, що оптимізує задану цільову функцію або функцію критерію. В основі теорії динамічного програмування лежить *принцип оптимальності Р. Белмана*.

Подібно методу декомпозиції, динамічне програмування вирішує задачу, розбиваючи її на підзадачі і поєднуючи їх рішення. Проте, існує важлива відмінність. Метод декомпозиції поділяє задачу на незалежні підзадачі, в свою чергу підзадачі - на більш дрібні підзадачі і так далі, а потім збирає рішення основної задачі «знизу вгору». Динамічне програмування ж застосовують тоді, коли підзадачі мають спільні „*підпідзадачі*". За такої умови, рекурсивний алгоритм виконує зайві обчислення, вирішуючи ті самі підпідзадачі по декілька разів. Алгоритм, що заснований на динамічному програмуванні, розв'язує кожну з підзадач один раз і запам'ятовує відповіді в спеціальній таблиці. Такий підхід дозволяє не обчислювати заново відповідь до підзадач, які вже зустрічалися.

Застосуємо метод динамічного програмування до вирішення задачі комівояжера. Зазначимо, що алгоритм знаходження мінімального маршруту комівояжера був запропонований *Л. Фордом* (1956) і *Р. Белманом* (1958). Розробка власного алгоритму вирішення задачі комівояжера і його програмна реалізація методом динамічного програмування була виконана О.Г. Тюріним в процесі підготовки даного видання. Саме цим алгоритмом ми з вдячністю і скористаємося.

Таблиця 2.14.

Таблиця X. Значення ребер графу з рис. 2.11.

	A	B	C	D	E
A	0	3	4	2	7
B	3	0	4	6	3
C	4	4	0	5	8
D	2	6	5	0	6
E	7	3	8	6	0

Спочатку подамо у табличній формі – таблиця X - значення всіх фрагментів можливих маршрутів, тобто ребер графу з рис. 2.11.

Для визначення найкоротшого шляху комівояжера, що розпочинається з вершини *A* і цією вершиною закінчується, будемо будувати варіанти його переміщення послідовно, вершина за вершиною.

Для цього нам знадобиться ще дві таблиці. Перша таблиця - таблиця Y (див. *Таблиця 2.15.*) - буде містити значення мінімальних маршрутів з вершини *A* у вершину *n*, що складаються з мінімальних маршрутів з вершини *A* у вершину *n*-1 та маршрутів з вершинами *n*-1 у вершину *n.* Таким чином, обчислення мінімального маршруту *A..n* базується на раніше обчисленому значенні маршруту *A..n*-1, що зберігається в таблиці. Наприклад, комірка Y[*B*,3] (перехрестя рядку *B* і стовпчика 3) таблиці Y містить мінімальне значення маршруту з вершини *A* у вершину *B*, що проходить через 3 вершини, наприклад, *ACB, ADB* чи *AEB,* і обчислюється наступним чином:

Таблиця 2.15.

Таблиця Y. Значення мінімальних маршрутів

	1	2	3	4	5
A	0	0	0	0	0
B	0	3	8	15	∞
C	0	4	7	14	∞
D	0	2	9	12	17
E	0	7	6	11	18

$$Y[B,3]=\min\begin{cases}A-C+c-b=Y[C,2]+X[c,b]=4+4=8\\A-D+d-b=Y[D,2]+X[d,b]=2+6=8\\A-E+e-b=Y[E,2]+X[e,b]=7+3=10\end{cases}$$

Наведена формула означає, що у вершину *B* можна пройти маршрутами *ACB, ADB* чи *AEB,* мінімальна вартість яких становить 8, 8 і 10 відповідно. Зрозуміло, що маршрут з вершини *A* у вершину *B*, який складається з 3 вершин і проходить через вершину *B*, неможливий, відповідно й значення його не враховується. Це дає право стверджувати, що у вершину *B* можна пройти двома мінімальними маршрутами, що містять 3 вершини (два ребра), і це маршрути *ACB* і *ADB.* Зазначимо, що у відповідності до принципу оптимальності Р.Белмана доцільніше скористатися саме

маршрутом *ACB*, вибір якого базується на оптимальнішому значенні ребра (*c*-*b*) у порівнянні з ребром (*d*-*b*).

Зауважимо, що нулі в рядку *A* таблиці Y означаються неможливість прокладання маршруту з вершину *A* у вершину *A* через 1,2,3,..*n* вершин, адже у такому випадку вершина *A* мала б 3 ребра, що виходять з неї (з врахуванням необхідно повернення у вершину *A*). Нулі ж стовпчика 1 вказують на те, що пройти з вершину *A* у вершину *B,C,D,..,n* маршрутом, який би містив лише одну вершину, також неможливо. Знак „∞" в комірках Y[*B*,5] і Y[*C*,5] означає, що вершини *B* і *C*, якими повинні скінчитися маршрути з 5 вершин, вже були задіяні у маршрутах, а їх повторне використання призведе до наявності у вершин *B* і *C* третього ребра. Отже, такі маршрути також неможливі.

Таблиця 2.16.

Таблиця Z. Чисельне позначення маршрутів

	1	2	3	4	5
A	0	0	0	0	0
B	1	3	4	2	0
C	1	2	5	3	0
D	1	3	2	5	4
E	1	3	4	2	5

Третя таблиця - таблиця Z - містить безпосередньо маршрут, поданий послідовністю чисел, що відповідають номеру вершини. Наприклад, рядок *E* складається з елементів, що утворюють послідовність 1,3,4,2,5. Ця послідовність - чисельне позначення маршруту *ACDBE*. Зауважимо, що всі маршрути починаються з вершини *A*, отже стовпчик 1 таблиці Z міститиме одиниці. Значення, записані у другому стовпчику, відповідають значенню самої вершини, на яку повинен закінчитися маршрут. Дійсно, маршрут з вершини *A* у вершину *B* буде закінчений у вершині *B*, чисельне значення якої дорівнює 2. Слід зазначити, що всі маршрути, наведені у таблиці Z, не включають фрагмент, що вказує повернення у вершину *A*. Це несуттєво, адже перехід з передостанньої вершини *n*-1 у вершину *A* однозначно визначений ребром ((*n*-1)-*a*).

Прослідкуємо визначення послідовності з 3 вершин для маршруту, що повинен закінчитися у вершині *B*. Зрозуміло, що ми опрацьовуємо рядок *B*. Послідовність для маршруту з 3 вершин вказує на те, що ми повинні до послідовності, що визначає мінімальний маршрут для 3 вершин, дописати значення вершини *B*, тобто 2. Мінімальний маршрутом для маршруту, що включає 3 вершини і закінчується у вершині *B*, є сума з маршруту, який включає 2 вершини і закінчується у вершині *C*, та ребра (*c*-*b*). А це означає, що необхідно до послідовності 1-3, що визначає маршрут *A*-*C*, дописати значення вершини 2. Нову послідовність 1-3-2 маршруту *A*-*C*-*B* зберігаємо до тих пір, доки не визначенні всі значення мінімальних маршрутів з 3 вершин, і потім записуємо у рядок *B* таблиці Z.

Прослідкуємо за роботою алгоритму вирішення задачі комівояжера для графа, що поданий на рис.2.11.

```
(1) for i←1 step 1 until n do
(2)   for j←1 step 1 until n do
      begin
(3)    if j=2 then
(4)              Y[i,j] ← X[1,i]
(5)          else Y[i,j] ← 0
(6)    if j=1 then
(7)              Z[i,j] ← 1
(8)          else if j=2 then
(9)                          Z[i,j] ← i
(10)                         else Z [i,j] ← 0
(11)  TEMP[i,j] ← 0
      end
(12) for j←3 step 1 until n do
      begin
(13)  for i←2 step 1 until n do
        begin
(14)   Y[i,j] ← ∞
(15)   for k←2 step 1 until n do
       begin
(16)    y←1
(17)    for l←1 step 1 until j-1 do
```

```
(18)        if Z[k,l]=i then
(19)            y← 0
(20)        if y=1 then
                begin
(21)            if Y[k,j-1]+X[k,i]< Y [i,j] then
                  begin
(22)              Y[i,j] ← Y [k,j-1]+X[k,i]
(23)              for l←1 step 1 until j-1 do
(24)                 TEMP[i,l] ←Z[k,l]
(25)              TEMP[i,j] ← i
(26)              s← X[k,i]
                  end
(27)              else if (Y[k,j-1]+X[k,i]= Y[i,j]) and (X[k,i]<s) then
                   begin
(28)               Y[i,j] ← Y [k,j-1]+X[k,i]
(29)               for l←1 step 1 until j-1 do
(30)                    TEMP[i,l] ←Z [k,l]
(31)               TEMP[i,j] ←i
(32)               s←X[k,i]
                   end
                  end
          end
           end
(33)   for i←2 step 1 until n do
(34)        for l←1 step 1 until j do
(35)          Z[i,l] ← TEMP[i,l]
       end
(36) for i←1 step 1 until n do
           begin
(37)     for j←1 step 1 until n do
             begin
(38)        write(X[i,j])
            end
(39)     writeln
           end
(40) for i←1 step 1 until n do
           begin
(41)     for j←1 step 1 until n do
```

```
            begin
(42)        write(Y[i,j])
            end
(43)    writeln
        end
(44) for i←1 step 1 until  n do
        begin
(45      for j←1 step 1 until  n do

         begin
(46)     write(Z[i,j])
         end
(47)     writeln
    end
```

У рядках (1)-(11) відбувається встановлення початкових значень для елементів таблиць *Y* і *Z*. Враховуючи те, що мінімальний маршрут з 2 вершин відповідає ребру цих вершин, то до стовпчика 2 рядків *B-E* таблиці *Y* записуємо значення ребер (*a-b*)-(*a-e*) відповідно. До стовпчика 1 рядків *B-E* таблиці *Z* записуємо 1, а до стовпчика 2 – чисельні значення вершин, в яких повинні закінчитися відповідні маршрути. Таблиця *TEMP,* що зберігатиме тимчасові значення послідовностей мінімальних маршрутів, заповнюється 0.

Основна робота алгоритму відбувається в рядках (12)-(35). Оператори повторення, що записані у рядках (12) і (13), дозволяють організувати опрацювання всіх стовпчиків і рядків таблиць *X* і *Y* відповідно. Оператор повторення, що записаний у рядках (15) необхідний для використання попередніх значень з таблиць *X* і *Y*. Комірку *Y*[*i,j*], яка використовується для зберігання мінімального значення маршруту, що складається з *j* вершин і закінчується у вершині *i*, на початку роботи оператору повторення рядку (13) заповнюється максимально можливим значенням. Саме з цим значенням на першому етапі порівнюється маршрут, що складається з мінімального маршруту з *j*-1 вершинами (Y[*k,j-1*]) і ребром з вершини *k* у вершину *i* (*X*[*k,i*]). Зазначимо, що значення змінної *y* і відповідні операції над нею в рядках (16)-(19) дозволяють уникнути перевірки маршруту з вершини, наприклад, *B* у вершину *B*. Оператор умови, що записаний у рядку (21)

встановлює у поточну комірку *Y*[*i,j*] значення мінімального з маршрутів *Y*[*k,j-1*]+*X*[*k,i*], де *k* змінюється від 2 до *n*. Оператор повторення рядку (23) дозволяє запам'ятати у таблицю *TEMP* значення попереднього оптимального маршруту, що складається з *j-1* вершин і закінчується у вершині *i*. Рядок (25) доповнює цей оптимальний маршрут вершиною *i*, в якій і повинен цей маршрут закінчуватися. Зміна *s* у рядку (26) отримує значення ребра *X*[*k,i*] і використовується в якості додаткової умови у рядку (27). Саме вона дозволяє обрати оптимальний маршрут, виходячи із значення поточного ребра *X*[*k,i*], у випадку коли є декілька попередніх маршрутів з однаковими значеннями. Рядки (27)-(32) дозволяють зробити оптимальний вибір. Наприклад, визначення оптимального маршруту, що проходить через 3 вершини і закінчується у вершині *B*, відбувається на основі пошуку мінімального значення маршрутів, що проходять через 2 вершини і закінчується у вершинах *C,D* чи *E* з врахуванням ребер (*c-b*), (*d-b*) і (*e-b*) відповідно.

За умови двох однакових значень мінімальних маршрутів саме умова рядку (27) і значення змінної s дозволяють обрати оптимальний маршрут.

Оператори повторення рядків (33) і (34) після аналізу всіх попередніх маршрутів записують послідовності оптимальних маршрутів з таблиці TEMP у таблицю Y.

Оператори повторення рядків (36) і (47) дозволяють подати у табличній формі значення ребер між всіма вершинами (таблиця X), мінімальних маршрутів (таблиця Y) і самі послідовності, що відповідають цим маршрутам (таблиця Z).

В результаті роботи алгоритму одержуємо значення двох мінімальних маршрутів *A-C-B-E-D*(1-3-2-5-4) і *A-C-D-B-E* (1-3-4-2-4), значення яких 17 і 18 відповідно. Враховуючи те, що повний маршрут повинен закінчуватися вершиною *A,* до значення маршрутів *A-C-B-E-D* і *A-C-D-B-E* додаємо значення ребер (*d-a*) і (*e-a*). Одержуємо значення 19 (17+2=19) і 25 (18+7=25) відповідно. Отже, робимо висновок, що маршрут *A-C-B-E-D-A* із значенням 19 є оптимальним маршрутом комівояжера.

Враховуючи те, що основна робота алгоритму зосереджена в операторах повторення (рядки (12), (13) і (15)), часова складність алгоритму буде $\Theta(n^3)$. Незважаючи на показник ступеня, функція часової складність алгоритму, що побудований на основі методу

динамічного програмування, росте набагато повільніше, ніж функція часової складність алгоритму повного перебору всіх варіантів. І якщо при $n \leq 5$ часова складність алгоритму динамічного програмування вища за складність алгоритму повного перебору ($n!=5!=120<n^3=5^3=125$), то вже при $n=6$ ситуація змінюється ($n!=6!=720>n^3=6^3=216$), а за умови наявності 50 вершин ця перевага ще значніша ($n!=50!=3{,}04*10^{64} >> n^3=50^3=125000$).

2.2.4. Евристичні алгоритми

Література:[4,288-291;13,326-346]

Ключові поняття: *евристичний алгоритм, „жадібна" стратегія, найкращий вибір, критерії прийняття рішення.*

Для рішення багатьох оптимізаційних задач є більш прості і швидкі евристичні алгоритми, ніж динамічне програмування. Зокрема, це так звані „жадібні" алгоритми. „Жадібний" алгоритм на кожному кроці робить локально оптимальний вибір з надією, що остаточне рішення також виявиться оптимальним. Таким чином, принцип „жадібного" вибору застосовується для оптимізаційних задач у тому випадку, якщо послідовність локально оптимальних (жадібних) виборів дає глобально оптимальне рішення. Різницю між жадібними алгоритмами і динамічним програмуванням можна пояснити так: на кожному кроці жадібний алгоритм бере самий „жирний" шматок, а потім уже намагається зробити *найкращий вибір* серед тих, що залишилися; алгоритм динамічного програмування навпаки приймає рішення тільки після того, як прорахуються заздалегідь усі варіанти.

Пояснимо принцип „жадібного" метода побудови алгоритмів наступним прикладом. Нехай у нас є монети номіналом в 25, 10 і 1 копійки. Нам потрібно здійснити покупку вартістю 88 копійок, при цьому скориставшись мінімальною кількістю монет. Оптимальним вибором буде три монети номіналом 25 копійок, одна монета в 10 копійок і три монети номіналом в 1 копійку. Наш вибір можна пояснити наступним описом алгоритму. Спочатку ми спробували скористатися монетою найбільшого номіналу (25 копійок), яка є, звичайно, менша за необхідну суму. Потім ми обчислили різницю (88-25=63) і дійшли до висновку про можливість використання ще двох монет номіналом в 25 копійок, тобто (63-25=38) і (38-25=13). Після третього вибору ми розуміємо, що скоритися монетами в 25

копійок неможливо (13<25) і переходимо до використання монет меншим номіналом тощо. Описаний алгоритм побудований за „жадібним" методом. Слід зазначити, що „жадібні" алгоритми зазвичай працюють швидше, ніж алгоритми, що базуються на динамічному програмуванні. Проте „жадібні" методи не завжди дають оптимальне рішення. Зокрема, для вирішення аналогічної задачі з монетами для суми в 30 копійок „жадібним" методом потрібно шість монет (одна номіналом в 25 копійок і п'ять в 1 копійку). Проте оптимальним рішенням є використання трьох монет номіналом в 10 копійок. Незважаючи на це, „жадібні" методи дозволяють отримувати досить оптимальні рішення, що відрізняються від дійсно оптимальних лише на декілька відсотків.

Застосуємо „жадібну" стратегію для побудови алгоритму вирішення задачі комівояжера. Як і випадку розгляду попередніх методів використовуємо рис. 2.11. „Жадібний" алгоритм розглядає спочатку найкоротші ребра. *Критерієм прийняття* ребра є виконання двох умов:

а) вибір не призводить до появи вершини зі ступенем три чи більше, тобто кількість ребер, що виходять з цієї вершини не повинна перевищувати два ребра;

б) вибір не призводить до утворення замкнутого контуру (циклу) з ребрами, що були прийняті раніше.

Сукупність ребер, що були обрані у відповідності до вказаного критерію, утворює сукупність не з'єднаних шляхів. Таке положення зберігається до виконання останнього кроку, коли останнє ребро приєднується до першої вершини, утворюючи маршрут.

Прослідкуємо процес утворення маршруту за „жадібним" алгоритмом. На першому етапі обирається ребро (*a,d*), довжина якого дорівнює 2. Далі обираємо два ребра (*a,b*) і (*b,e*), довжина яких дорівнює 3 і також приймаємо їх. На наступному етапі розглядаємо ребра (*a,c*) і (*b,c*), довжина яких дорівнює 4. Проте, відмовляємося від них, так як їх прийняття призводить до підвищення ступенів вершин *a* і *b* до трьох, що суперечить критерію прийняття ребер. Ребро (*c,d*), довжиною 5 приймаємо на наступному кроці. Два ребра (*b,d*) і (*d,e*) довжиною 6 відхиляємо відповідно до першої умови критерію прийняття. Ребро (*a,c*), довжиною 7 відхиляємо у відповідності до другої умови критерію, адже його прийняття призведе до утворення замкнутого контуру

(циклу) з ребрами (*a,d*) і (*c,d*). Ребро (*c,e*), довжиною 8 приймаємо на останньому етапі. Таким чином, з використанням „жадібного" алгоритму був одержаний маршрут (a→b→e→c→d→a), вартість якого дорівнює 21. Як бачимо „жадібна" стратегія дозволила отримати маршрут, що відрізняється всього на 9 відсотків від оптимального.

Ми розглянули основні методи, що використовуються при проектуванні ефективних алгоритмів. Зрозуміло, ці методи не є універсальними і абсолютними. Проте, саме їх потужна техніка дозволяє проектувати елегантні і природні алгоритми. Адже вони спонукають до вирішення таких завдань як оптимальне подання даних і знаходження розумного порядку виконання операцій. Проте, як слушно зазначають А. Ахо, Дж. Хопкрофт і Дж. Ульман [3, с.86], найважливішим принципом майстерного розробника алгоритмів повинне бути почуття незадоволеності. Розробнику необхідно вивчати задачу з різних точок зору, поки він не переконається, що одержав алгоритм, який найбільш підходить для досягнення його мети.

ПРАКТИЧНИЙ КУРС

Алгоритмічні моделі на основі обчислювальних функцій

1. Використовуючи частково-рекурсивні функції виконати наступні задачі:
1.1. Довести, що функція f(x,y)=x+y задовольняє схему примітивної рекурсії.
1.2. Довести, що функція f(x,y)=x*y задовольняє схему примітивної рекурсії.
1.3. Довести, що функція $f(x,y)=x^y$ задовольняє схему примітивної рекурсії.
1.4. Довести, що наступні функції є примітивно рекурсивними:

- $\left[\frac{x}{y}\right]$ - залишок від ділення x на y, де [x/0]=x;
- rest (x,y) - залишок від ділення x на y, де rest(x,0)=x;
- k(x,y) – найменше спільне ділене чисел x та y, де k(x,0)=k(0,y)=0;
- d(x,y) – найбільший спільний дільник чисел x та y, де d(0,0)=0
- $\left[\sqrt{x}\right]$

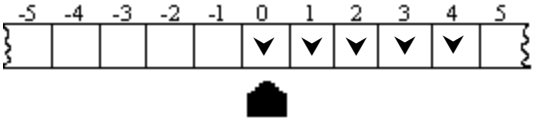
1.5. Довести, що функція, яка перелічує числа Фібоначчі

$$\begin{cases} f(0)=0, \qquad f(1)=1, \\ f(n+2)=f(n)+f(n+1) \end{cases}$$

є примітивно рекурсивною.

Алгоритмічні моделі на основі детермінованих пристроїв

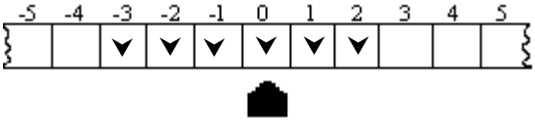
2. Написати програму машини Поста для рішення наступної задачі:
2.1. Яким би не було натуральне число N, якщо початковий стан машини Поста такий, що на стрічці є запис n та каретка знаходиться проти самої лівої секції запису, то виконання програми повинно привести до результативної зупинки, після чого на стрічці повинно бути записане число N+1. Каретка після виконання програми може знаходитися в будь-якому положенні.



Приклад. 3.1.

1 → 2	переходимо праворуч, не перевіряючи зміст секції, бо машина знаходиться проти заповненої самої лівої секції
2 ? 3,1	перевіряємо зміст секції. Якщо вона порожня переходимо до команди 3, якщо наповнена, повертаємось до команди 1.
3 ⮟ 4	ставимо мітку в порожній секції та переходимо до команди 4.
4 !	зупиняємось.

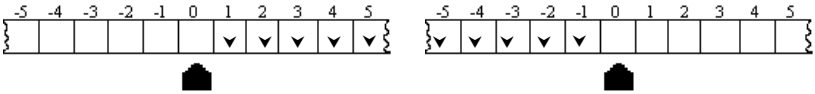
2.2. Яким би не було натуральне число N, якщо початковий стан машини Поста такий, що на стрічці є запис n та каретка знаходиться проти однієї секції запису, то виконання програми повинно привести до результативної зупинки, після чого на стрічці повинно бути записане число N+1. Каретка після виконання програми може знаходитися в будь-якому положенні.



- Знайдіть додаткові 2 рішення задачі 3.2, програма яких складається з 4 команд. Перевірте, щоб програма містила команди зміщення вліво, встановлення мітки, передачі керування та зупинки.

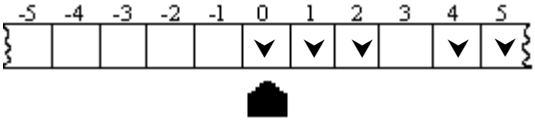
- Довести, що існує ще 11 рішень задачі 3.2., що складаються з 4 команд та містять команду зміщення вліво.

2.3. Яким би не було натуральне число N, якщо початковий стан машини Поста такий, що на стрічці є запис n та каретка знаходиться зліва / зправа від запису, то виконання програми повинно привести до результативної зупинки, після чого на стрічці повинно бути записане число N+1. Каретка після виконання програми може знаходитися в будь-якому положенні.

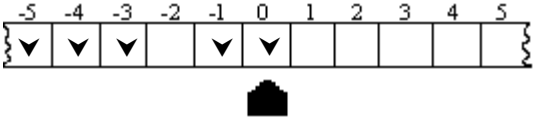


2.4. Яким би не було натуральне число N, якщо початковий стан машини Поста такий, що на стрічці є запис n та каретка знаходиться в довільному місці стрічки, то виконання програми повинно привести до результативної зупинки, після чого на стрічці повинно бути записане число N+1. Каретка після виконання програми може знаходитися в будь-якому положенні. Запропонуйте декілька варіантів рішення задачі.

2.5. Додати два числа, що записані на відстані в 1 комірку одне від одного. Каретка знаходиться проти:
самої лівої комірки лівого числа

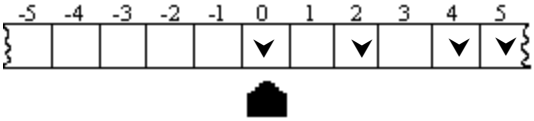


самої правої комірки правого числа.



2.6. Додати два числа, що записані на довільній відстані одне від одного. Каретка знаходиться проти самої лівої комірки лівого числа.

2.7. Додати довільну кількість чисел, що записані на відстані в 1 комірку одне від одного. Каретка знаходиться проти самої лівої комірки лівого числа.



2.8. Знайти найбільше з двох натуральних чисел N і M, що записані на відстані в одну пусту комірку одне від одного. Каретка знаходиться проти самої лівої комірки лівого числа.

2.9. Перемножити два числа, що записані на довільній відстані одне від одного. Каретка знаходиться проти самої лівої комірки лівого числа.

2.10. Виконати ділення числа, що записане на стрічці на задане число К.
Примітка: у випадку "нульового" ділення програма не повинна призводити до результативної зупинки.

3. Написати програму машини Тьюрінга для рішення наступної задачі:
3.1. Збільшити число *p*, записане у двійковій системі числення, на 1.

Приклад. 4.1. Вхідним алфавітом буде *A={0,1,λ }*. Отже,

A \ Q	q_1	q_2	q_3
0	$q_1 0R$	$q_3 1L$	$q_3 0L$
1	$q_1 1R$	q_2 0L	$q_3 1L$
λ	q_2 λL	q_3 1L	q_z λR

У стані q_1 каретка знаходиться проти лівої цифри числа *p* та зсувається праворуч за командами $q_1 0 q_1 0R$ та $q_1 1 q_1 1R$ до тих пір, доки не зчитає пустий символ λ. Команда q_2 λL переведе стан машини на молодший розряд числа *p*. У стані q_2 відбувається перехід ліворуч, якщо у черговому розряді числа міститься

одиниця. Це забезпечується командами $q_2 1 q_2 0 L$. Якщо ж міститься нуль, то команда $q_2 0 q_3 1 L$ додає одиницю та забезпечує перехід до стану q_3. Команда $q_2 \lambda q_3 1 L$ ставить на місце першої прогалини зліва від числа, якщо у результаті додавання одиниці виникає перенесення з старшого розряду вихідного числа *p*. У стані q_3 повертає машину на перший символ зліва результату – числа *p+1*.

3.2. Додати два числа, що записані у вигляді послідовності одиниць між якими стоїть знак "+". Каретка знаходиться проти самої лівої комірки лівого числа.

λ	1	1	+	1	λ

q_1

3.3. Знайти найбільше з двох натуральних чисел N і M, що записані у вигляді послідовностей одиниць між якими стоїть знак "=".

3.4. Яким би не було натуральне число N, подане в десятковій системі числення, на стрічці записати N+1. Каретка встановлена проти розряду одиниць (в самій правій позиції числа).

λ	5	8	λ	λ	λ

q_1

3.5. Скопіювати натуральне число N, що записані у вигляді послідовності одиниць зі знаком "*" наприкінці.

λ	1	1	1	*	λ

q_1

3.6. Перемножити два натуральні числа, що записані у вигляді послідовності одиниць та розподілені знаком "*".

λ	λ	1	1	1	*	1	1	λ	λ	λ	λ	λ	λ

q_1

3.7. Додати два натуральні числа, що подані в десятковій системі числення.

λ	5	8	+	3	λ

q_1

3.8. Відняти два натуральні числа, що подані в десятковій системі числення.

λ	5	8	-	3	λ
	q_1				

3.9. Нескінчено збільшувати на одиницю натуральне число N, що подане в десятковій системі числення.

3.10. Знайти найбільший спільний дільник двох натуральних чисел, що подані в десятковій системі числення.

3.11. Виконати ділення числа N, що записане у вигляді послідовності одиниць, на задане число К (послідовність *).

4. Скласти схеми орієнтовних підстановок для нормальних алгорифмів Маркова для рішення наступної задачі:

4.1. Перетворити будь-яке слово в алфавіті A=(a_0, a_1, a_2, ..., a_n) в порожнє слово.

Приклад. 4.1. За умовами задачі є алфавіт A = (a_0, a_1, a_2, ..., a_n) . Нехай у загальному вигляді слово P цього алфавіту має вигляд $a_0a_1a_2...a_n$, де a_0, a_1, a_2, ..., a_n – букви з алфавіту A. Якщо усі букви цього алфавіту замінити їхніми номерами отримаємо новий алфавіт B = (0, 1, , 2, …, n). Будь-яке слово, складене з елементів утвореного алфавіту B буде порожнім, бо не буде містити букв.

4.2. Перетворити будь-яке парне число в алфавіті A=(1) в порожнє слово, а непарне – в слово 1.

4.3. Перетворити довільне слово в алфавіті A=(1) в 1, якщо слово ділиться на 3, в іншому випадку – записати порожнє слово.

4.4. Збільшити довільне число в алфавіті A=(0, 1, 2, ..., 9) на 1.

4.5. Зменшити довільне число в алфавіті A=(0, 1, 2, ..., 9) на 1.

Складність алгоритмів

5.1. Розташуйте наступні функції в порядку зростання швидкості росту: а)n, б)$\sqrt{n}$, в)log n, г)log log n, д) $\log^2 n$, е)n^2, ж) $(1/3)^n$, з) $(3/2)^n$.

Приклад. 5.1. Порядку зростання швидкості росту функції залежить від розмірності n даних задачі. Отже, порядок буде наступним

ж) $(1/3)^n$ – степінь числа, меншого одиниці, буде найменшою
г) log log n- подвійний логарифм більше, ніж степінь малого числа
б) $\sqrt{n}$ - корінь з числа n менший за саме число
в) log n – логарифм числа більший, ніж його корінь
д) $\log^2 n$- квадрат логарифма більший за сам логарифм
а) n- саме число більше, ніж його корінь, логарифм
з) $(3/2)^n$ – ступінь числа, більшого за одиницю більше самого числа
е) n^2 – квадрат числа більший, ніж саме число, його кореня чи логарифма.

5.2. Побудуйте таблицю залежності максимально допустимих розмірностей задачі для розв'язання за одну добу для алгоритмів з часовими складностями n, log n , n^2, 2^n.

5.3. Запишіть вираз $n^3/1000-100\ n^2-10n+1$ використовуючи Θ-символіку.

5.4. При якому найменшому значенні n алгоритм, що виконує $10n^2$ операцій, ефективніше алгоритму, що виконує 2^n операцій?

5.5. Використовуючи Θ-символіку, визначте час (в найгіршому випадку) виконання наступної підпрограми як функції від n.

```
(1) procedure A
(2)       for i←1 step 1 until  n do
(3)       for j←1 step 1 until  n do
            begin
(4)         C[i,j]←0
(5)         for k←1 step 1 until  n do
(6)             C[i,j]← C[i,j]+A[i,k]*B[k,j]
            end
```

МОДУЛЬНО-ТЕСТОВІ ЗАВДАННЯ ДЛЯ САМОКОНТРОЛЮ

МОДУЛЬ №1. АЛГОРИТМІЧНІ МОДЕЛІ НА ОСНОВІ ДЕТЕРМІНОВАНИХ ПРИСТРОЇВ. ОБЧИСЛЮВАЛЬНІ ФУНКЦІЇ

І. Визначте варіанти правильних відповідей на наступні запитання

1. Частково-рекурсивні функції як алгоритмічна модель для уточнення поняття алгоритму були запропонована в:
   a) 20-х роках XX ст.
   b) 30-х роках XX ст.
   c) 40-х роках XX ст.
   d) 50-х роках XX ст.
   e) 60-х роках XX ст.
   f) 70-х роках XX ст.
   g) правильна відповідь відсутня

2. До класу найпростіших функцій, що використовуються в якості базису для побудови обчислювальних функцій, відносяться:
   a) нуль-функція
   b) функція суперпозиції
   c) функція примітивної рекурсії
   d) всі вищевказані варіанти
   e) правильна відповідь відсутня

3. В якості операторів, що застосовуються до базисних функцій для створення нових обчислювальних функцій, використовують:
   a) оператор наступності
   b) оператор проекції
   c) оператор введення фіктивних аргументів
   d) всі вищевказані варіанти
   e) правильна відповідь відсутня

4. Підстановка, в результаті якої n-місна функція утворюється з підстановки у функцію F заміст її аргументів m функцій f_1 , f_2 ,... f_m , називається:
   a) оператором примітивної рекурсії
   b) оператором мінімізації
   c) оператором суперпозиції
   d) оператором проекції
   e) правильна відповідь відсутня

5. Функція називається частково-рекурсивною, якщо вона утворена з базисних функцій та:
   a) оператора примітивної рекурсії
   b) оператора мінімізації
   c) оператора проекції
   d) оператора суперпозиції
   e) всі вищевказані варіанти
   f) правильна відповідь відсутня

6. Оператор примітивної рекурсії дозволяє будувати n+1-місну арифметичну функцію G *(x_1, x_2,y)* з двох заданих функцій:
   a) n+2-місної функції Q (x_1, x_2,y,z) та n+1-місної функції W (x_1, x_2,y)
   b) n+2-місної функції Q (x_1, x_2,y,z) та n+2-місної функції W (x_1, x_2,y,z)
   c) n+1-місної функції Q (x_1, x_2,y) та n+1-місної функції W (x_1, x_2,y)
   d) n+1-місної функції Q (x_1, x_2,y) та n+2-місної функції W (x_1, x_2,y,z)
   e) правильна відповідь відсутня

7. Функція називається примітивно-рекурсивною, якщо вона утворена з базисних функцій та:
   a) оператора примітивної рекурсії
   b) оператора мінімізації
   c) оператора проекції
   d) оператора суперпозиції
   e) всі вищевказані варіанти
   f) правильна відповідь відсутня

8. Оператор примітивної рекурсії позначається як:
   a) R^n
   b) I^n_m
   c) *S(x)*
   d) *M(z)*
   e) S^n_m
   f) *правильна відповідь відсутня*

9. Оператор суперпозиції позначається як:
   a) R^n
   b) I^n_m
   c) *S(x)*
   d) *M(z)*
   e) S^n_m
   f) *правильна відповідь відсутня*

10. Функція введення фіктивних аргументів позначається як:
   a) R^n
   b) I^n_m
   c) *S(x)*
   d) *M(z)*
   e) S^n_m
   f) *правильна відповідь відсутня*

11. Тезис, стосовно того, що клас алгоритмічно обчислювальних часткових числових функцій збігається з класом усіх частково-рекурсивних функцій, запропонував:
   a) *Пост*
   b) *Тьюрінг*
   c) *Марков*
   d) *Гільбер*
   e) *правильна відповідь відсутня*

12. В результаті виконання функції проекції $I^n_m (x_1, x_2,.. x_m,.. x_n)$ отримується значення:
   a) x_1
   b) x_2
   c) x_m
   d) x_n

e) правильна відповідь відсутня

13. Двомісна функція f_+ (x,y) = x+y задовольняє наступну схему примітивної рекурсії:
    a) $x+0=S(x)$ та $x+(y+1)=(x+y)+1=S(x+y)$
    b) $x+0=S(x)$ та $x+(y+1)=(x+y)+1=S(y+1)$
    c) $x+0=I^1_1(x)$ та $x+(y+1)=(x+y)+1=S(x+y)$
    d) $x+0= I^1_1(x)$ та $x+(y+1)=(x+y)+1=S(y+1)$
    e) правильна відповідь відсутня

14. Двомісна функція f_* (x,y) = x*y задовольняє наступну схему примітивної рекурсії:
    a) $x+0=O(x)$ та $x*(y+1)=x*y+x=f_*(x,y)+x= f_+(x, f_*(x,y))$
    b) $x+0=S(x)$ та $x*(y+1)=x*y+x=f_*(x,y)+x= f_+(x, f_*(x,y))$
    c) $x+0=S(0)$ та $x*(y+1)=x*y+x=f_*(x,y)+x= f_+(y, f_*(x,y))$
    d) $x+0=S(x)$ та $x*(y+1)=x*y+x=f_*(x,y)+x= f_+(y, f_*(x,y))$
    e) правильна відповідь відсутня

15. Функція наступності позначається як:
    a) R^n
    b) I^n_m
    c) $S(x)$
    d) $M(z)$
    e) S^n_m
    f) правильна відповідь відсутня

МОДУЛЬ №2. АЛГОРИТМІЧНІ МОДЕЛІ НА ОСНОВІ ДЕТЕРМІНОВАНИХ ПРИСТРОЇВ

Визначте варіанти правильних відповідей на наступні запитання

1. Машина з довільним доступом як алгоритмічна модель для уточнення поняття алгоритму була запропонована в:
   a) 20-х роках XX ст.
   b) 30-х роках XX ст.
   c) 40-х роках XX ст.
   d) 50-х роках XX ст.
   e) 60-х роках XX ст.
   f) правильна відповідь відсутня

2. Фінітний комбінаторний процес Поста був:
   a) результатом паралельного, у відношенні до машин Тьюрінга, дослідженням
   b) теоретичною основою абстрактної машини Тьюрінга
   c) всі вищевказані варіанти
   d) правильна відповідь відсутня

3. До алгоритмічних моделей на основі детермінованих пристроїв відносять:
   a) машини з довільним доступом
   b) нормальні алгоритми Маркова
   c) фінітний комбінаторний процес Поста
   d) частково-рекурсивні функції
   e) абстрактну машину Тьюрінга
   f) правильна відповідь відсутня

4. Команда програми машини Поста складається з:
   a) операції машини Поста
   b) символу внутрішнього стану
   c) переходу
   d) символу зовнішнього алфавіту
   e) *номеру команди*

5. Верхній перехід команди передачі керування в машині Поста відбувається у випадку, коли комірка, яку визначає каретка,:
   a) відмічена міткою

b) пуста
c) правильна відповідь відсутня

6. В ході виконання програми машина Поста не зустрічається ні з командою зупинки, ні з некоректною командою, що призводить до:
   a) результативної зупинки
   b) безрезультативної зупинки
   c) "за циклювання" програми
   d) правильна відповідь відсутня

7. Абстрактна машина Тьюрінга складається з:
   a) каретки
   b) керуючого пристрою
   c) команди машини Тьюрінга
   d) нескінченної кількості регістрів
   e) нескінченної стрічки
   f) всі вищевказані варіанти
   g) правильна відповідь відсутня

8. Прямокутна таблиці, що використовується для запису програм машини Тьюрінга, називається:
   a) функціональною схемою машини Тьюрінга
   b) конфігурацією машини Тьюрінга
   c) програмою машини Тьюрінга
9. Конфігурацією машини Тьюрінга називається сукупність:
   a) функціональної схеми, внутрішнього стану та положення каретки машини Тьюрінга
   b) внутрішнього стану, програми та стану стрічки машини Тьюрінга
   c) стану стрічки, функціональної схеми та положення каретки
   d) *правильна відповідь відсутня*

10. Конфігурація К з внутрішнім станом $\mathbf{q_1}$, словом ***алгоритм*** на стрічці та кареткою під символом ***р*** записується як:
    a) аq$_1$лгоритм
    b) алгоq$_1$ритм
    c) алгор q$_1$итм

d) алгори q_1тм
e) алг q_1оритм
f) алгоритмq_1м
g) правильна відповідь відсутня

11. Машини з довільним доступом мають наступні типи команд:
a) команду обнуління
b) команду безумовного переходу
c) команду умовного переходу
d) команду віднімання одиниці
e) команду додавання одиниці
f) команду встановлення мітки
g) команду зміщення вправо/вліво
h) всі вищевказані варіанти

12. Символам, що використовують для позначення символів зміщення каретки машини Тьюрінга, відповідають наступні кодові групи універсальної машини Тьюрінга:
a) 11
b) 101
c) 1001
d) 10001
e) 100001
f) 1000001
g) 10000001
h) правильна відповідь відсутня

13. Вкажіть варіанти правильних записів програм машини Поста:

a) 1 E3	*b) 1E2*	*c) 1 E2*	*d) 1 V2*
3 V 2	*2 V4*	*2 V3*	*2 стоп*
2 стоп	*3 стоп*	*3 стоп*	*3 E1*

14. Робота машини з довільним доступом з конфігурацією $K=(a_1,a_2,...a_s)$ припиняється після виконання команди I_t, якщо:
a) $I_t=J(m,n,q)$, $r_n=r_m$ і ... *a) $t=s$*
b) $I_t\neq J(m,n,q)$ і ... *b) $q>s$*
c) $I_t=J(m,n,q)$, $r_n\neq r_m$ і ... *c) $t=s$*

15. Інформація про стан стрічки та місцерозташування каретки машини Поста називається:
    a) станом стрічки машини Поста
    b) станом каретки машини Поста
    c) функціональною схемою машини Поста
    d) командою машини Поста
    e) правильна відповідь відсутня

МОДУЛЬ №3. СКЛАДНІСТЬ АЛГОРИТМІВ

1. При якому найменшому значенні n алгоритм, що виконує $100n^2$ операцій, ефективніше алгоритму, що виконує 2^n операцій?

(2 балів)

2. Алгоритми сортування вставками і злиттям мають складність $8n^2$ і $64n\log n$ відповідно. Для яких значень n алгоритм сортування вставками є більш ефективнішим?

(3 балів)

3. Визначте складність алгоритму пошуку заданого елемента у впорядкованому масиві розмірністю n.

(4 балів)

4. Алгоритм впорядкування вирішує задачу розмірністю n за $f(n)$ мікросекунд. Визначте максимальну розмірність задачі впорядкування для функцій і часу, наведених у таблиці.

	1 сек.	1 хв.	1 год.	1 доба
$\log n$				
n				
$n\log n$				
n^2				
n^3				
2^n				
$n!$				

(6 балів)

5. Визначте складність алгоритму (в найкращому і найгіршому випадках), сутність роботи якого полягає у наступному:

І етап: вихідний масив рекурсивно поділяється на дві частини меншої розмірності до тих пір, доки розмірність частини не становитиме 1 (будь-який масив розмірністю в один елемент можна вважати впорядкованим);

II етап: окремо виконується впорядкування кожної із частин;

III етап:здійснюється поєднання двох впорядкованих частин масиву в одну і процес продовжується етапом I.

(10 балів)

Правильні відповіді до модульно-тестових завдань для самоконтролю

№	Модуль 1.	Модуль 2.
1	b	f
2	a	a
3	e	a,c,e
4	c	a,b,c
5	a,b,d	b
6	e	c
7	a,d	a,b,e
8	a	a
9	e	d
10	b	b
11	e	a,c,e
12	e	b,c,d
13	c	c,d
14	a	a-b,b-a,c-c
15	c	e

ЛІТЕРАТУРА

Основна:

1. Emil L. Post Finite combinatory processes – formulation 1 // The Journal of Symbolic Logic. - 1936. –№4, P. 122-134.
2. Turing A.M. On Computable numbers with an application to the Entscheidungsproblem //Proc. London Math. Soc., Ser.2. – 1936. – V.42. - №3-4, P.230-265.
3. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. - М.: Мир, 1979. – 536 с.
4. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы.: Пер. с англ.:Уч. пос. – М.: Издательский дом «Вильямс», 2000.-384 с.:ил.
5. Вирт. Н Алгоритмы + структуры данных = программы.- М.: Мир, 1985.-406с.
6. Грин Д., Кнут Д. Математические методы анализа алгоритмов.- М.: Мир, 1987.- 120 с.
7. Гуц А.К. Математическая логика и теория алгоритмов: Учебное пособие. – Омск: Диалог – Сибирь, 2003. – 108 с.
8. Дискретна математика: Підручник / Ю.М. Бардачов, Н.А. Соколова, В.Є Ходаков; За ред. В.Є. Ходакова. – К.: Вища шк., 2002. –287 с.: іл.
9. Игошин В.И. Задачник-практикум по математической логике. – М.: Просвещение, 1986. – 159 с.
10. Игошин В.И. Математическая логика и теория алгоритмов. - Саратов: Изд-во Саратовского ун-та, 1991. – 256 с.
11. Кнут Д. Искусство программирования для ЭВМ. – В 3–х т. Т.3 Сортировка и поиск. – М.:Мир, 1977.
12. Комп'ютерна дискретна математика: Підручник / М.Ф. Бондаренко, Н.В. Білоус, А.Г. Руткас. – Харків: Компанія СМІТ, 2004.-480 с.
13. Кормен Т и др. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р.Ривест. – М.:МЦНМО, 2001. - 960 с.

14. Лісовик Л.П., Шкільняк С.С. Теорія алгоритмів: Навч. посібник.- К.: Видавничий поліграфічний центр "Київський університет", 2003.-163 с.
15. Мальцев А. И. Алгоритмы и рекурсивные функции.- М.:Наука, 1986/-368 с.
16. Носов В.А. Основы теории алгоритмов и анализа их сложности. Курс лекций. М.: 1992. – 112 с.
17. Пападимитриу Х., Стайглиц К. Комбинаторная оптимизация. Алгоритмы и сложность. – М.: Мир, 1984. – 510 с.
18. Трахтенброт Б.А. Алгоритмы и машинное решение задач. М.: Наука, 1957.- 98 с.
19. Успенский В.А. Машина Поста. 2-е изд., испр. М.: Наука, 1988.- 96 с.
20. Эдельман С.Л. Математическая логика. Уч. пос. для ин-тов. – М.: Высшая школа, 1975. – 176 с.

Додаткова:

1. Алферова З.В. Теория алгоритмов.- М.: Статистика, 1973.- 164 с.
2. Асанов М.О., Баранкский В.А., Расин В.В. Дискретная математика: графы, матроиды, алгоритмы. – Ижевск, НИЦ «Регулярная и хаотическая динамика», 2001.-288с.
3. Гильберт Д., Бернайс П. Основания математики. Т.1, Т.2. – М.: Наука, 1982.
4. Глушков В.М. Введение в кибернетику. Киев: Изд-во АН Укр ССР. 1964. 324 с.
5. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. - М. Мир. 1982.
6. Евстигнеев В.А. Применение теории графов в программировании. - М. Наука. 1986.
7. Зубенко В.В., Шкільняк С.С. Теорія алгоритмів у прикладах та задачах. – К.: Інтелектуальні системи, 1993. – 84 с.
8. Иванов Б.Н. Дискретная математика. Алгоритмы и программы. Уч. пос. - М.: Лаборатория Базовых Знаний, 2001.
9. Игошин В.И. Задачник-практикум по математической логике. – М.: Просвещение, 1986. – 159 с.

10. Катленд Н. Вычислимость. Введение в теорию рекурсивных функций: Пер. с англ. М.: Мир, 1983.- 256 с.
11. Клини С. Введение в метаматематику. – М.: Иностранная литература, 1957. – 526 с.
12. Клини С. Математическая логика. – М.: Мир, 1973. – 480 с.
13. Колмогоров А.Н., Драгалин Ф.Г. Математическая логика. Дополнительные главы. – М.: МГУ, 1984. – 120 с.
14. Кристофидес Н. Теория графов. Алгоритмический подход. - М. Мир. 1980.
15. Кузьмин И.В., Березюк И.Т., Фурманов К.К., Шаронов В.Б. Синтез вычислительных алгоритмов управления. - Л.: Техника, 1975. - 248 с.
16. Лавров И.А., Максимова Л.Л. Задачи по теории множеств, математической логике и теории алгоритмов. М.: Наука, 1984. - 224 с.
17. Лісовик Л.П., Редько В.Н. Алгоритмы и формальные системы. – К.: КГУ, 1981. – 112 с.
18. Лісовик Л.П., Шкільняк С.С. Основи теорії алгоритмів. – К.: Інтелектуальні системи, 1993. – 94 с.
19. Лісовик Л.П., Шкільняк С.С. Теорія алгоритмів. – Київ, 2001. – Деп. в ДНТБ України 23.07.2001, № 132-Ук2001. – 148 с.
20. Мальцев А.И. Алгоритмы и рекурсивные функции. – М.: Наука, 1965. – 392 с.
21. Манин Ю.И. Вычислимое и невычислимое. – М.: Советское радио, 1980. – 128 с.
22. Марков А. А. Теория алгорифмов. Труды математического института им. В. А. Стеклова, – Т.42. – М. – Л.: Издательство Академии наук СССР, 1954. – 375 с.
23. Марков А.А., Нагорный Н.М. Теория алгорифмов. – М.: Наука, 1984. – 432 с.
24. Математические методы построения и анализа алгоритмов. – Л.: Наука, 1990. – 241 с.
25. Машины Тьюринга и рекурсивные функции / Г. Д. Эббинхауз, К. Якобе, Ф. К. Ман, Г. Хермес.— М.: Мир, 1972.—264 с.
26. Мендельсон Э. Введение в математическую логику. – М.: Наука, 1976. – 320 с.
27. Новиков В.А. Дискретная математика для программистов. Санкт-Петербург.: Питер, 2001.

28. Прийма С.М. Теорія алгоритмів: Навч. посібник. - Мелітополь: МДПУ, 2004. - 48 с.:іл.
29. Роджерс Х. Теория рекурсивных функций и эффективная вычислимость. – М.: Мир, 1972. – 624 с.
30. Роджерс Х. Теория рекурсивных функций і эффективная вычислимость. М., 1972.
31. Справочная книга по математической логике (под ред. Дж.Барвайса). Ч.1–Ч.4. – М.: Наука, 1982–1983.
32. Тарьян Р.Э. Сложность комбинаторных алгоритмов. Киб. Сборник. Н.с. вып. 17. - М. Мир. 1980.
33. Теорія алгоритмів (конспект лекцій) / Укладач: С.М. Прийма - Мелітополь: МДПУ, 2004.-32 с.:іл.
34. Трахтенброт Б.А. Сложность алгоритмов и вычислений: спецкурс для студентов НГУ, Новосибирск, 1967.
35. Тьюринг А. Может ли машина мыслить? Пер.и примечания Ю.В.Данилова. М.: ГИФМЛ, 1960.
36. Успенський В.А., Семенов А.Л. Теория алгоритмов: основные открытия и приложения. – М.: Наука, 1987. – 288 с.
37. Шенфилд Дж. Математическая логика. – М.: Наука, 1975. – 528 с.
38. Шкильняк С.С. Исследование программных алгебр функций натуральных аргументов і значений. – Модели и системы обработки информации. Вып. 8. – К., 1989. – С. 9–16.
39. Шкільняк С.С. Математична логіка: приклади і задачі. – Київ: ВПЦ "Київський університет", 2002. – 56 с.
40. Яворський Б.І. Теорія алгоритмів / Конспект лекцій. - Тернопіль: ТДТУ імені Івана Пулюя, 2000. - 32 с.

ПРЕДМЕТНИЙ ПОКАЗЧИК

Для нотаток

Прийма Сергій Миколайович

ТЕОРІЯ АЛГОРИТМІВ

Навчальний посібник
для здобувачів вищої освіти
спеціальності 122 Комп'ютерні науки

Друкується в авторській редакції

Підписано до друку 26.12.2018. Формат 60x84/16.
Друк цифровий. Папір офсетний. Гарнітура Times New Roman.
Ум. друк. арк. 7,03. Наклад 300 прим.
Зам. № 2789.

Видано та надруковано ФО-П Однорог Т.В.
72313, м. Мелітополь, вул. Героїв Сталінграда, 3а
Тел. (098) 243 96 51
Свідоцтво про внесення суб'єкта видавничої справи до
Державного реєстру видавців, виробників і розповсюджувачів
видавничої продукції від 29.01.2013 р. серія ДК № 4477