МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
«ІНСТИТУТ ПРОМИСЛОВИХ ТА БІЗНЕС ТЕХНОЛОГІЙ»
УКРАЇНСЬКОГО ДЕРЖАВНОГО УНІВЕРСИТЕТУ НАУКИ І ТЕХНОЛОГІЙ

Л.І. ЛОЗОВСЬКА, Л.М. БАНДОРІНА,
Р.В. САВЧУК, К.О. УДАЧИНА

ЕКОНОМІЧНА КІБЕРНЕТИКА

Дніпро 2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
«ІНСТИТУТ ПРОМИСЛОВИХ ТА БІЗНЕС ТЕХНОЛОГІЙ»
УКРАЇНСЬКОГО ДЕРЖАВНОГО УНІВЕРСИТЕТУ НАУКИ І ТЕХНОЛОГІЙ

Л.І. ЛОЗОВСЬКА, Л.М. БАНДОРІНА,
Р.В. САВЧУК, Т.О. УДАЧИНА

ЕКОНОМІЧНА КІБЕРНЕТИКА

Друкується за Планом видань навчальної та методичної літератури,
затвердженим Вченою радою УДУНТ
Протокол № 1 від 24.01.2022

Дніпро 2022

УДК 681.518
Лозовська Л.І., Бандоріна Л.М., Савчук Р.В., Удачина К.О. Економічна кібернетика : навч. посібник. Дніпро : УДУНТ, 2022. 152 с.

В посібнику розглянуті фундаментальні структури і алгоритми обробки даних. Викладені основи асимптотичного аналізу обчислювальної складності ітеративних і рекурсивних алгоритмів. Наведені класичні методи сортування і пошуку інформації. Викладені підходи до реалізації абстрактних типів даних: списків, множин, асоціативних масивів і черг з пріоритетами. В кінці кожного розділу наведені вправи для самостійного виконання, які можуть бути використані як завдання для лабораторних робіт.

Призначений для студентів спеціальності 051 – економіка, 126 – інформаційні системи та технології.

Друкується за авторською редакцією.

Відповідальна за випуск Л.М. Бандоріна, канд. екон. наук, доц.

Рецензенти: Бакурова А.В., докт. екон. наук, професор, професор кафедри системного аналізу та обчислювальної математики НУ «Запорізька політехніка»;
Турчина В.А., канд. фіз.-мат. наук, доц., зав. каф. обчислювальної математики та математичної кібернетики Дніпровського національного університету ім. Олеся Гончара.

© Український державний університет науки і технологій, 2022
© Лозовська Л.І., Бандоріна Л.М., Савчук Р.В., Удачина К.О., 2022

ЗМІСТ

Вступ……. 6
1 Алгоритми та їх ефективність …… 7
1.1 Задача, алгоритм, програма ……..... 7
1.2 Показники ефективності алгоритмів……..... 9
1.3 Підрахунок кількості операцій алгоритму ……..... 10
1.4 Швидкість зростання функцій……..... 14
1.5 Асимптотичні позначення ……..... 18
1.6 Етапи асимптотичного аналізу ……..... 27
Завдання для самостійного виконання ……..... 32
2 Аналіз рекурсивних алгоритмів …… 34
2.1 Рекурсивні алгоритми …… 34
2.2 Сортування злиттям ……..... 36
2.3 Розв’язання рекурентних рівнянь……..…… 43
Завдання для самостійного виконання ……..... 44
3 Сортування …… 47
3.1 Задача сортування……... 47
3.2 Властивості і види алгоритмів сортування…….. 47
3.3 Сортування вставкою…….. 49
3.4 Сортування вибором…….. 52
3.5 Нижня межа часу сортування порівнянням……. 54
3.6 Швидке сортування……..…….. 56
3.7 Сортування злиттям…… 61
3.8 Пірамідальне сортування…… 62
3.9 Сортування підрахунком…… 62
3.10 Вибір алгоритму сортування…… 65
Завдання для самостійного виконання ……..... 66
4 Пошук……... 67
4.1 Задача пошуку……. 67
4.2 Лінійний пошук……..…….…… 67
4.3 Бінарний пошук……... 67
Завдання для самостійного виконання ……...……... 69

5 Абстрактні типи даних…….…….…….……..... 70
5.1 Базові типи даних……..….. 70
5.2 Структури даних……. 71
5.3 Абстрактні типи даних…… 72
5.4 Вибір абстрактного типу даних…….…… 78
Завдання для самостійного виконання……..… 85
6 Стеки…….……... 86
6.1 АТД стек……... 86
6.2 Реалізація стека на базі масиву…..…… 87
6.3 Реалізація стека на базі однозв’язного списку……...….. 89
6.4 Порівняння реалізацій…… 91
Завдання для самостійного виконання…….. 92
7 Списки……..…….. 93
7.1 АТД список……..……...…..... 93
7.2 Реалізація списків на базі масивів……..……... 94
7.3 Зв’язні списки.……..……..……..... 99
7.4 Однозв’язні списки..……..……. 101
7.5 Двозв’язні списки..…….…… 105
7.6 Реалізація АТД список…….. 108
Завдання для самостійного виконання……..... 112
8 Черги……..... 113
8.1 АТД черга…….….. 113
8.2 Реалізація черги на базі зв’язного списку…… 114
8.3 Реалізація черги на базі кільцевого буфера...……..…. 116
8.4 Порівняння реалізацій…… 120
Завдання для самостійного виконання…….…….…..... 121
9 АТД дерево…..……. 122
9.1 Кореневі дерева..……....…… 122
9.2 Бінарні дерева……..…… 123
9.3 Зберігання дерев в пам’яті..……..……. 124
9.4 Обходи бінарного дерева……...……...…….…… 126
Завдання для самостійного виконання…….…… 127

10 Бінарні дерева пошуку…...…….. 128
10.1 Структура бінарного дерева пошуку…….……..…….. 128
10.2 Створення вузла…… 129
10.3 Додавання вузла……..... 129
10.4 Пошук вузла ключем……..….. 131
10.5 Пошук мінімального і максимального вузлів…… 132
10.6 Пошук наступного і попереднього вузлів…….. 133
10.7 Обхід дерева в упорядкованій послідовності……. 135
10.8 Видалення вузла…… 136
10.9 Видалення дерева……..… 141
10.10 Висота бінарного дерева…… 142
Завдання для самостійного виконання……... 143
Контрольні запитання для самоперевірки…….. 144
Рекомендована література ……..……. 146
Додаток А ……..……..... 148

ВСТУП

Цей посібник містить частину теоретичного матеріалу курсу «Економічна кібернетика (поглиблений курс)» для студентів спеціальності 051 − економіка та «Економічна кібернетика» для студентів спеціальності 126 – Інформаційні системи і технології. При доведенні оцінок обчислювальної складності алгоритмів зведено до мінімуму апарату використання теорії ймовірності та наведені коментарі. Основні математичні формули і співвідношення, що використовуються в посібнику, винесені в додатки.

Перші два розділи присвячені класичним аспектам аналізу часової і просторової ефективності алгоритмів. В них викладені основні етапи асимтотичного аналізу обчислювальної складності алгоритмів: від підрахунку кількості операцій до застосуванні асимптотичних позначень. Розглянуті методи оцінки ефективності рекурсивних алгоритмів: аналіз дерева рекурсивних викликів і застосування основної теореми.

В наступних двох розділах розглянуті задачі сортування і пошуку. В наведені тривіальні алгоритми сортування з квадратичною складністю і основні асимптотично оптимальні алгоритми: сортування злиттям і пірамідальне сортування. Проаналізовані лінійний і бінарний алгоритми розв’язання задачі пошуку.

В посібнику також розглянуті питання реалізації найважливіших структур даних. Викладення матеріалу починається зі введення поняття абстрактних типів даних та їх класифікації. На прикладі реалізації лінійних типів даних розглянуті основні операції над зв’язними списками, статичними масивами і кільцевими буферами. Далі наведено опис бінарних дерев пошуку і хеш-таблиць, як основних засобів реалізації абстрактних типів даних множина і асоціативний масив (словник). Також в посібник включена інформація про бінарні кучі та їх реалізація на основі черги з пріоритетами.

Опис алгоритмів наводиться природною мовою та доповнено кодом на C++.

1 АЛГОРИТМИ ТА ЇХ ЕФЕКТИВНІСТЬ

1.1 Задача, алгоритм, програма

Процес розв'язання задачі (*problem solving*) на комп'ютері включає в себе наступні етапи.

1. Постановка задачі.
2. Розробка алгоритму розв'язання задачі.
3. Доведення коректності алгоритму й аналіз його ефективності.
4. Реалізація алгоритму мовою програмування.
5. Виконання програми для отримання необхідного результату.

У постановці задачі (*problem statement*) наводиться точне формулювання умов задачі з описом її вхідних (*input*) і вихідних (*output*) даних.

В процесі розробки алгоритму (*algorithm design*) формулюється покрокова процедура отримання з вхідних даних вихідних.

Алгоритм (algorithm) − це кінечна послідовність інструкцій виконавцю, в результаті виконання яких забезпечується отримання з вхідних даних необхідного вихідного результату (*розв'язання задачі*).

Алгоритм записується формальною мовою виконавця, що виключає неоднозначність тлумачення призначених йому приписів. Запис алгоритму формальною мовою виконавця називається програмою (*program*, рис. 1.1).

Вважається, що алгоритм коректний (*correct*), якщо він для будь-яких коректних значень вхідних даних видає коректні вихідні дані. Тут під коректністю даних розуміється їх відповідність умовам розв'язуваної задачі.

Після розробки алгоритму розв'язання задачі здійснюється його реалізація (*implementation*) однією з мов програмування, наприклад: C, C ++, Java, C #, Go, Python, JavaScript та ін. Мова програмування виступає в ролі мови виконавця, а виконавцем є інтерпретатор або процесор, в набір інструкцій якого компілятор транслює програму.

В ході розробки програми алгоритм може зазнавати змін, що пов'язані з архітектурою цільової системи. Наприклад, в алгоритм можуть вноситися зміни, що забезпечують ефективне використання кеш-пам'яті процесора, та ін.

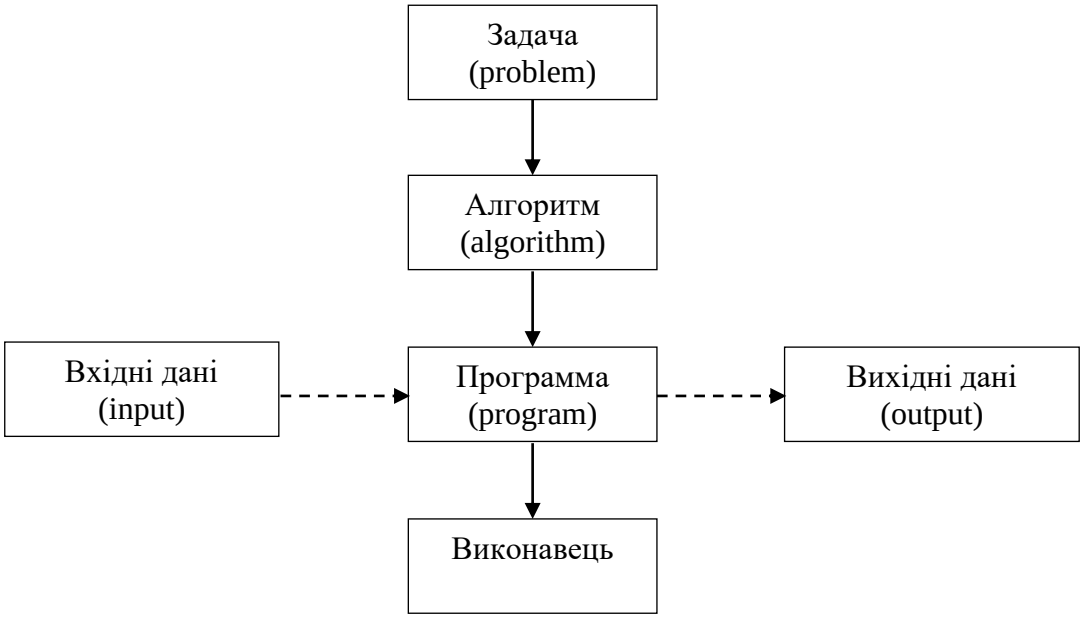


Рисунок 1.1 − Процес розв’язання задачі: задача, алгоритм, програма.

Розглянемо основні властивості, якими повинен задовольняти алгоритм.

1. ***Дискретність*** − алгоритм представляється як послідовність інструкцій виконавця. Кожна інструкція виконується тільки після того, як закінчилося виконання попередньої команди.

2. ***Кінечність*** (результативність) − алгоритм повинен закінчуватися після виконання кінечної кількості інструкцій.

3. ***Детермінованість*** − кожен крок алгоритму має бути однозначно визначено − записано формальною мовою виконавця. Детермінованість забезпечує збіг результатів, що одержуються при багаторазовому виконанні алгоритму, на одному і тому ж наборі вхідних даних.

4. ***Масовість*** − алгоритм розв’язання задачі повинен бути застосовний для деякого класу задач, що розрізняються лише значеннями вхідних даних.

Приклад. Пошук максимального елемента масиву. Розглянемо задачу пошуку в масиві номера елемента з максимальним значенням. Формальна постановка задачі виглядає наступним чином:

− вхід: послідовність з n чисел ($a_1, a_2, ..., a_n$);

− вихід: номер i елемента a_i, що має найбільше значення:

$$a_i \geq a_j, \quad j = \overline{1, n}. \qquad (1.1)$$

Якщо на вхід подається послідовність (24, 14, 53, 1, 187, 2, 78), то розв’язанням задачі є номер 5 (елемент зі значенням 187). Такий конкретний набір вхідних даних називається екземпляром задачі (*problem instance*).

Нижче наведено код лінійного алгоритму `indexOfMaxElement` розв'язання даної задачі про пошук номера максимального елемента. На вхід алгоритму надходить масив *mas* з *n* елементів.

Алгоритм 1.1 – Пошук максимального елемента масиву

```
int indexOfMaxElement(double *mas, int n)
{
    int maxi = 0;
    for (int i = 1; i < n; i++)
    {
        if (mas[maxi] < mas[i])
        {
            maxi = i;
        }
    }
    return maxi;
}
```

Алгоритм `indexOfMaxElement` є дискретним, так як він написаний мовою програмування C++. Кінечність алгоритму забезпечується, тим, що він завжди закінчує свою роботу після перегляду *n-1* елемента масиву *mas*. В процесі своєї роботи алгоритм не використовує датчик псевдовипадкових чисел для прийняття рішень про хід подальшого виконання, це забезпечує його детермінованість. Масовість алгоритму обумовлена тим, що він розв'язує задачі пошуку номера максимального елемента для довільного масиву *mas*.

1.2 Показники ефективності алгоритмів

Алгоритми, розроблені для розв'язання однієї і тієї ж задачі, можуть значно відрізнятися за ефективністю. Тому для характеристики їх якості вводять показники ефективності. Розглянемо найбільш поширені з них.

1. ***Кількість операцій*** – часова ефективність (*time efficiency*), показує наскільки швидко працює алгоритм.

2. ***Обсяг споживаної пам'яті*** – просторова ефективність (*space efficiency*), відображає максимальний об'єм пам'яті, що необхідна для виконання алгоритму.

Існують й інші показники, які має сенс розглядати, якщо вони значно впливають на процес розв'язання задачі. Наприклад, для алгоритмів, що

працюють з даними на зовнішніх носіях інформації (жорсткі диски, мережеві сховища), доцільно враховувати кількість звернень до зовнішньої пам'яті, а в алгоритмах, що використовують мережеві канали зв'язку, важливо брати до уваги кількість переданих повідомлень (мережевих пакетів).

Введення показників ефективності дозволяє проводити аналіз алгоритмів з метою порівняння їх між собою і оцінювання потреби того чи іншого алгоритму в обчислювальних ресурсах: процесорного часу, пам'яті, пропускної здатності мережі.

1.3 Підрахунок кількості операцій алгоритму

Для більшості алгоритмів кількість операцій, що виконуються ними, прямо залежить від розміру вхідних даних. Чим більше обсяг вхідних даних, тим довше працює алгоритм. Так, час виконання алгоритму `indexOfMaxElement` пошуку максимального елемента (алгоритм 1.1) залежить від n довжини масиву.

Кількість операцій алгоритму можна виразити як функцію від одного або декількох параметрів, що пов'язані з розміром вхідних даних. Наприклад:

− алгоритм сортування вибором − кількість операцій алгоритму (час його роботи) залежить від кількості n елементів в масиві;

− алгоритм множення двох матриць − час виконання залежить від кількості рядків m і стовпців n в матрицях;

− алгоритм Дейкстри пошуку найкоротшого шляху в графі − час виконання залежить від кількості n вершин і m ребер в графі.

RAM-машина. Для підрахунку кількості операцій, що виконуються алгоритмом, необхідно формально описати систему команд деякого виконавця. В якості такого виконавця будемо використовувати модель однопроцесорної обчислювальної машини з довільним доступом до пам'яті (Random Access Machine − RAM) [11,13]. Будемо вважати, що машина має необмежену пам'ять і працює за такими правилами:

− для виконання арифметичних і логічних операцій (+, -, *, /,%) потрібно один часовий крок (такт процесора);

− кожне звернення до осередку в оперативній пам'яті для читання або запису займає один часовий крок;

− виконання умовного переходу (*if-then-else*) вимагає обчислення логічного виразу і виконання однієї з гілок if-then-else;

− виконання циклу (*for, while, do*) передбачає виконання всіх його ітерацій, в свою чергу, виконання кожної ітерації вимагає обчислення умови завершення циклу і виконання його тіла.

Приклад. Знаходження суми елементів масиву. Обчислимо кількість операцій алгоритму `sumAllElement`, що реалізує обчислення суми *n* елементів масиву.

Алгоритм 1.2 − Підсумовування елементів масиву

```
1.int sumAllElement(double *mas, int n)
2.{
3.     int sum = 0;
4.     for (int i = 0; i < n; i++)
5.     {
6.         sum += mas[i];
7.     }
8.     return sum;
9.}
```

Час роботи алгоритму `sumAllElement` залежить тільки від розміру *n* масиву. У рядку 3 виконується одна операція запису в пам’ять. Далі, перед виконанням кожної з *n* ітерацій циклу відбуваються перевірка умови його закінчення $i=n$ і перехід на рядок 6 або 8. На кожній ітерації в рядку 6 виконується чотири операції: читання з пам’яті значень *sum і mas[i],* знаходження їх суми і запис результату в пам’ять. В кінці алгоритму виконується повернення результуючого значення − одна операція. Таким чином, кількість операцій $T(n)$, що виконуються алгоритмом `sumAllElement`, є

$$T(n) = 4n + 2. \tag{1.2}$$

Далі ми переконаємося, що такий точний аналіз кількості операцій алгоритму в багатьох випадках не потрібно виконувати. Досить обмежитися підрахунком лише тих операцій, сумарна кількість яких залежить від розміру вхідних даних. Так, в алгоритмі `sumAllElement` рядки 3 і 8 не мають значного впливу на підсумковий час виконання, який фактично визначається тільки операціями в рядку 6.

При аналізі обчислювальної складності алгоритмів ми будемо ігнорувати операції, що пов'язані з перевіркою умови закінчення циклу `for` і автоматичним збільшенням його лічильника.

Приклад. Лінійний пошук. Існує велика кількість алгоритмів, час виконання яких залежить не тільки від розміру вхідних даних, але і від їх значень. Як приклад розглянемо алгоритм `linearSearch` лінійного пошуку заданого значення x в масиві з n елементів.

Кількість операцій алгоритму `linearSearch` може значно відрізнятися для одного і того ж розміру вхідних даних. Розглянемо три можливі випадки.

Алгоритм 1.3 – Лінійний пошук

```
1.int linearSearch(double *mas, int n, double elem)
2.{
3.    for (int i = 0; i < n; i++)
4.    {
5.        if (mas[i] == elem)
6.        {
7.            return i;
8.        }
9.    }
10.   return -1;
11.}
```

Кращий випадок (*best case*) − екземпляр задачі (*набір вхідних даних*), на якому алгоритм виконує найменшу кількість операцій. У нашому прикладі − вхідний масив, перший елемент якого містить шукане значення x. У цій ситуації потрібно виконати

$$T_{best}(n) = 3 \tag{1.3}$$

операцій: перевірка умови закінчення циклу, умова в циклі (рядок 5) повернення знайденого значення (рядок 7). Таким чином, час роботи алгоритму в кращому випадку − теоретична нижня межа часу його роботи.

Найгірший випадок (*worst case*) – екземпляр задачі, на якому алгоритм виконує найбільшу кількість операцій. Для розглянутого алгоритму − масив, в якому відсутній шуканий елемент або він лежить у останньому осередку. У цій ситуації потрібно виконати

$$T_{worst}(n) = 2n + 1 \tag{1.4}$$

операцію: один раз перевірити умову закінчення циклу й умову в ньому, потім повернути значення -1. Час роботи алгоритму в гіршому випадку − теоретична верхня межа часу його роботи.

Середній випадок (*average case*) − «середній» екземпляр задачі, набір «усереднених» вхідних даних. В середньому випадку оцінюється математичне сподівання кількості операцій, які виконуються алгоритмом. Зазначимо, що не завжди очевидно, які вхідні дані вважати «усередненими» для задачі. Часто робиться припущення, що всі набори даних надходять на вхід алгоритму з однаковою ймовірністю.

Проведемо аналіз ефективності нашого алгоритму для середнього випадку. Позначимо ймовірність успішного пошуку елемента в масиві через $p \in [0,1]$. Тоді ймовірність відсутності значення x в масиві дорівнює $1-p$. Будемо вважати, що шуканий елемент з однаковою ймовірністю p/n може знаходиться в будь-якому з n осередків масиву.

Якщо шуканий елемент x знаходиться в осередку 1, то для його пошуку потрібно виконати 3 операції (перевірити умову закінчення циклу, умову в циклі і повернути значення 1); якщо елемент знаходиться в осередку 2, то потрібно 5 операцій, і т. д. У загальному випадку, якщо шуканий елемент x розташований в осередку i, то це вимагає виконання $2i+1$ операцій.

Обчислимо математичне очікування (середнє значення) кількості операцій, що виконуються алгоритмом. За визначенням, математичне очікування є сума добутків значення дискретної випадкової величини на ймовірність прийняття випадковою величиною цього значення. Наша випадкова величина − кількість операцій, що виконуються алгоритмом. Як ми домовилися вище, вона може приймати з однаковою ймовірністю p/n наступні значення: $3,5,...,(2i+1),...,(2n+1)$. Тому

$$T_{average} = 3\frac{p}{n} + 5\frac{p}{n} + ... + (2i+1)\frac{p}{n} + ... + (2n+1)\frac{p}{n} . \qquad (1.4)$$

У нашій оцінці ми повинні врахувати той факт, що дані значення x з імовірністю $1-p$ може бути відсутнім в масиві. Тоді вираз прийме наступний вигляд:

$$T_{average} = \frac{p}{n}[3+5+...+(2i+1)+...(2n+1)] + (1-p)(2n+1) . \qquad (1.5)$$

Неважко помітити, що в квадратних дужках записана сума членів арифметичної прогресії.

Обчислимо її:

$$T_{average} = \frac{p}{n}[n^2 + n] + (1-p)(2n+1) = p(n+2) + (1-p)(2n+1)\ . \qquad (1.6)$$

З отриманої формули можна зробити наступний висновок: якщо шуканий елемент присутній в масиві $(p=1)$, то в середньому потрібно виконати $n+2$ операції для його знаходження.

Аналіз ефективності алгоритмів для середнього випадку − більш важке завдання, ніж аналіз їх поведінки в гіршому і кращому випадках. Але для багатьох алгоритмів саме оцінки ефективності для середнього випадку дають реальну картину щодо їх практичного застосування.

Далі при аналізі алгоритмів ми будемо приділяти основну увагу часу роботи алгоритмів в гіршому випадку − максимальному часу роботи на всіх наборах вхідних даних і, по можливості, будувати оцінки ефективності для середнього випадку.

В якості синонімів поняття ***кількості операцій алгоритму*** ми будемо використовувати терміни ***час виконання алгоритму*** (*execution time*), ***обчислювальна складність алгоритму*** (*computational complexity*).

1.4 Швидкість зростання функцій

Нехай, ми маємо два алгоритми розв'язання однієї і тієї ж задачі. Відповідно до описаної вище схеми побудовані функції $T_1(n)$ і $T_2(n)$ залежно кількості операцій алгоритмів від розміру їх вхідних даних для кращого, середнього або гіршого випадку. Будемо вважати, що

$$T_1(n) = 130\,\mathrm{n}^2 + 50n + 100\,, \qquad (1.7)$$

$$T_2(n) = 3\mathrm{n}^3 + 2\,. \qquad (1.8)$$

Виникає запитання: який з алгоритмів краще використовувати на практиці?

Час виконання обох алгоритмів зростає зі збільшенням розміру n вхідних даних. На рисунку 1.2 наведені графіки функцій $T_1(n)$ і $T_2(n)$, а в табл. 1.1 − їх значення. Для даних невеликого розміру другий алгоритм виконує менше

операцій ніж перший, отже, в цьому випадку перевагу слід віддати другому алгоритму. Для великих значень n другий алгоритм починає помітно поступатися першому, тут варто вибрати перший алгоритм.

У загальному випадку ми можемо знайти таке значення n_0, при якому відбувається перетин функцій $T_1(n)$ і $T_2(n)$, і, знаючи конкретний розмір вхідних даних n і n_0, віддавати перевагу тому чи іншому алгоритму.

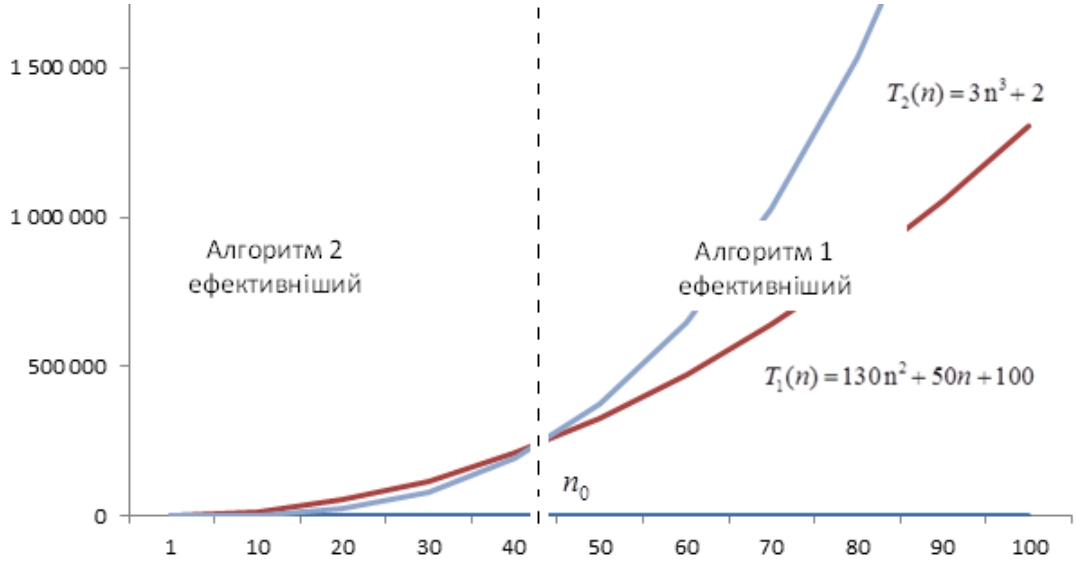


Рисунок 1.2 − Залежність кількості операцій, що виконуються алгоритмами, від розміру вхідних даних

У більшості випадків при малих розмірах n вхідних даних різниця в часі виконання алгоритмів, які розв’язують одну і ту ж задачу, як правило, незначна. Якщо апріорі відомо, що на вхід будуть надходити дані невеликих розмірів, то питання про вибір ефективного алгоритму не є першорядним: можна використовувати самий «простий» алгоритм − зрозумілий, легко реалізований мовою програмування або доступний в стандартній бібліотеці мови. Навіть неефективний алгоритм при роботі з вхідними даними невеликого розміру завершується за допустимий час.

Питання, пов’язані з просторовою і часовою ефективністю алгоритмів, набирають ваги при великих розмірах вхідних даних. Включаючи випадки, коли заздалегідь невідомо, якого розміру екземпляри задач нам доведеться розв’язувати. Отже, в першу чергу нас буде цікавити питання про те, як швидко

зростає кількість операцій $T(n)$, що виконуються алгоритмом, при збільшенні розміру вхідних даних.

Таблиця 1.1 − Кількість операцій, що виконуються алгоритмами

N	$T_1(n)$	$T_2(n)$	Порівняння алгоритмів
1	280	5	Перший алгоритм в 56,00 разів повільніший за другий
10	13600	3002	Перший алгоритм в 4,53 рази повільніший за другий
20	53100	24002	Перший алгоритм в 2,21 рази повільніший за другий
30	118600	81002	Перший алгоритм в 1,46 рази повільніший за другий
40	210100	192002	Перший алгоритм в 1,09 рази повільніший за другий
50	327600	375002	Перший алгоритм в 1,14 рази швидший за другий
60	471100	648002	Перший алгоритм в 1,38 рази швидший за другий
70	640600	1029002	Перший алгоритм в 1,61 рази швидший за другий
80	836100	1536002	Перший алгоритм в 1,84 рази швидший за другий
90	1057600	2187002	Перший алгоритм в 2,07 рази швидший за другий
100	1305100	3000002	Перший алгоритм в 2,30 рази швидший за другий
1000	1,3E+08	3E+09	Перший алгоритм в 23,07 раза швидший за другий

Для порівняння функцій за швидкістю зміни своїх значень зі збільшенням значень їх аргументів, вводять поняття швидкості росту функцій.

Швидкість зростання (*rate of growth*) або ***порядок зростання*** (*order of growth*) функції $T(n)$ визначаються її старшим, домінуючим членом. При великих значеннях n членами меншого порядку можна знехтувати. У нашому прикладі функція $T1(n) = 130\mathrm{n}^2 + 50n + 100$ зростає як n^2, а функція $T2(n) = 3\mathrm{n}^3 + 2$ як n^3. Порядок зростання функції n^3 більше порядку зростання функції n^2, так як n^3 приймає великі значення, ніж n^2 .

Один алгоритм вважається більш ефективним у порівнянні з іншим (в гіршому, середньому або кращому випадку), якщо кількість його операцій має нижчий порядок зростання.

У таблиці 1.2 показана швидкість росту деяких функцій, які часто зустрічаються в оцінках часу роботи алгоритмів. Найповільніше зростають функції $\lg(n)$ і $\log_2(n)$. Алгоритми з подібною логарифмічною залежністю кількості операцій від розміру вхідних даних працюють практично миттєво. Наприклад, бінарний пошук елемента в упорядкованому масиві або пошук вузла в АВЛ-дереві або червоно-чорному дереві. Найшвидше (з наведених в табл. 1.2) зростають функції 2^n і $n!$. Алгоритми з такою складністю на практиці майже не застосовуються, так як навіть на невеликих розмірах вхідних даних вимагають виконання колосальної кількості операцій. Прикладами можуть служити деякі оптимізаційні алгоритми, що реалізують повний перебір множини допустимих рішень задачі. Якщо розв'язання представляється у вигляді перестановки з n чисел, то кількість варіантів, які перебираються алгоритмом дорівнює кількості перестановок порядку n, а саме $n!$.

Таблиця 1.2 − Швидкість зростання функцій

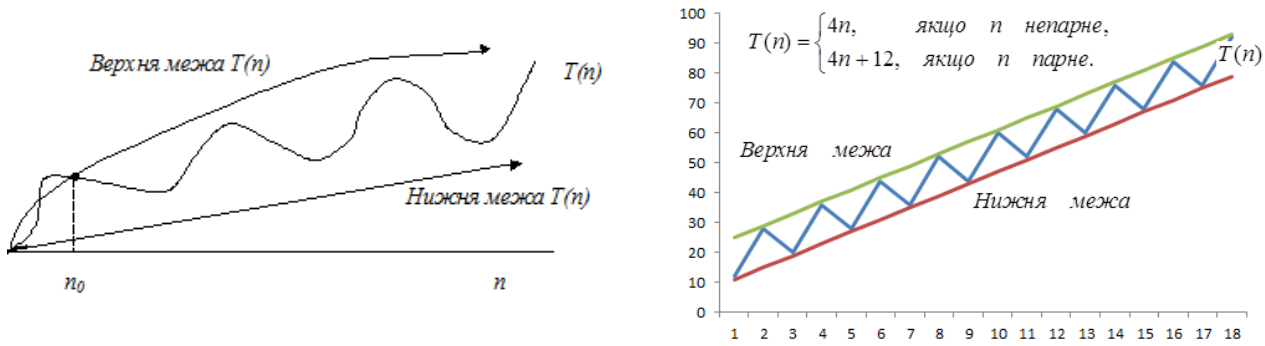
$\lg(n)$	$\log_2(n)$	n	$n\log_2(n)$	n^2	2^n	$n!$
0	0	1	0	1	2	1
0.3	1	2	2	4	4	2
0.5	1.6	3	5	9	8	6
0.6	2	4	8	16	16	24
0.7	2.3	5	12	25	32	120
0.78	2.6	6	16	36	64	720
0.85	2.8	7	20	49	128	5 040
0.90	3	8	24	64	256	40 320
0.95	3.2	9	29	81	512	362 880
1	3.3	10	33	100	1024	3 628 800
3	10	1000	9 966	1 000 000		
4	13.3	10 000	132 877	100 000 000		
5	16.6	100 000	1 660 964	10 000 000 000		
6	19.9	1 000 000	19 931 569	1 000 000 000 000		

Для порівняння і класифікації швидкостей зростання функцій, що виражають час роботи алгоритмів, нам необхідний спеціальний математичний апарат. Він повинен давати можливість робити висновки про те, що два алгоритми при великих n працюють однаковий час або один алгоритм

ефективніший іншого. Відповіді на ці питання дає ***асимптотичний аналіз*** (*asymptotic analysis*), який дозволяє оцінювати швидкість росту функцій $T(n)$ при прагненні розміру вхідних даних до нескінченності (при $n \to \infty$).

1.5 Асимптотичні позначення

Як було показано раніше, час виконання алгоритму в гіршому, середньому і кращому випадках можна представити, як функцію $T(n)$ від розміру його вхідних даних n. Однак аналізувати ефективність алгоритмів за такими функціями досить важко по ряду причин. Як правило, функція $T(n)$ часу виконання алгоритму має велику кількість локальних екстремумів − нерівний графік з виступами і западинами (рис. 1.3, а). Наприклад, можлива ситуація, коли час виконання алгоритму на вхідних масивах з непарною довжиною буде більше часу виконання на масивах з парної довжиною. У цьому випадку графік $T(n)$ буде мати пилкоподібний вигляд, як на рисунку 1.3 б. Тому набагато простіше працювати з ***верхньою*** і ***нижньою межами*** (оцінками) ***часу виконання алгоритму***. Так для прикладу на рис. 1.3 б замість пилкоподібної функції $T(n)$ можна використовувати її верхню межу $4 \cdot n + 12$ або іншу.



А Б

Рисунок 1.3 − Приклад можливих верхньої і нижньої меж часу $T(n)$ виконання алгоритмів.

Для позначення меж функцій $T(n)$ в ***теорії обчислювальної складності алгоритмів*** (*computational complexity theory*) використовують асимптотичні позначення: O (*о велике*), Ω (*омега велике*), Θ (*тета велике*), а також o (*о мале*) і ω (*омега мале*). Далі будемо вважати, що областю визначення функцій

$f(n)$ і $g(n)$, які визначають кількість операцій алгоритму, є множина невід'ємних цілих чисел ($n \in \{0,\ 1,\ 2,...\}$). Функції $f(n)$ і $g(n)$ є асимптотично невід'ємними − при великих значеннях n вони приймають значення, більші або рівні нулю. Що стосується аналізу складності алгоритмів остання вимога виконується завжди, так функції виражають кількість операцій алгоритму або обсяг споживаної ним пам'яті, які не можуть бути від'ємними.

1.5.1. O-позначення. O-позначення використовують, якщо необхідно вказати ***асимптотичну верхню межу*** (*asymptotic upper bound*) для функції $f(n)$ кількості операцій алгоритму. Говорять, що функція $f(n)$ належить множині функцій $O(g(n))$, що записується як $f(n) \in O(g(n))$, якщо існують позитивна константа $c > 0$ і $n_0 \in \{0,\ 1,\ 2,...\}$, Такі що для будь-яких $n > n_0$

$$0 \le f(n) \le cg(n). \tag{1.9}$$

Формально множина $O(g(n))$ визначається наступним чином:

$$O(g(n)) = \{ f(n) : \exists c > 0, n_0 \in \{0,1,2,...\}, \text{таких, що } 0 \le f(n) \le cg(n), \forall n > n_0 \} \tag{1.10}$$

Інакше кажучи, $O(g(n))$ − це множина всіх функцій, значення яких при великих n не перевищують значення $cg(n)$. Зазвичай факт належності функції $f(n)$ множині $O(g(n))$ записують як $f(n) = O(g(n))$. Читається як ***« f від n є о велике від g від n »***.

Для доведення $f(n) = O(g(n))$, потрібно знайти константи $c > 0$ і $n_0 \in \{0,\ 1,\ 2,...\}$, які забезпечують виконання умов (1.10). Якщо доведено, що $f(n) = O(g(n))$, то говорять: ***«функція $f(n)$ асимптотично обмежена зверху функцією $g(n)$ з точністю до постійного множника»***. Таким чином, добуток $cg(n)$ є асимптотичною оцінкою зверху часу $f(n)$ роботи алгоритму.

Приклад. Доведемо, що $3n - 15 = O(n)$. Для цього потрібно знайти відповідні константи $c > 0$ і $n_0 \in \{0,\ 1,\ 2,...\}$. Доведення: візьмемо $c = 3$ і $n_0 = 5$. Ці значення забезпечують виконання нерівності $0 \le 3n - 15 \le 3n$ для будь-яких n $n \ge 5$. На рисунку 1.4 видно, що пряма $3n$ проходить вище прямої $3n - 15$. Можна було взяти й інші значення c і n_0, головне − це те, що ми показали їх існування.

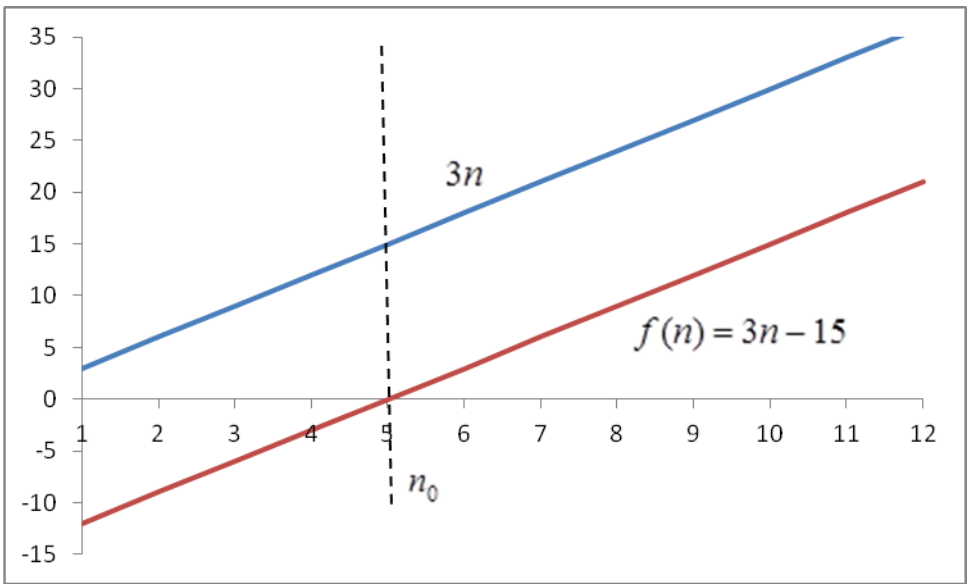


Рисунок 1.4 – Ілюстрація належності функції $3n-15$ множині $O(n)$.

Аналогічно можна довести коректність наступних тверджень.

1. $4n^2+50n-15=O(n^2)$. Для доведення візьмемо $c=5$ і $n_0=50$, при будь-яких $n\geq 50$ справедлива нерівність $0\leq 4n^2+50n-15\leq 5n^2$.

2. $4n^2+50n-15=O(n^3)$. Візьмемо $c=1$ і $n_0=3$, для будь-яких $n\geq 3$ справедливо $0\leq 4n^2+50n-15\leq n^3$.

3. $0.00004n^3\neq O(n^2)$, так як не існує констант $c>0$ і n_0, які забезпечують виконання нерівностей (1.2). Для будь-яких $c>0$ і $n\geq= c/0.00004$ має місце нерівність $0.00004n^3\geq cn^2$

На рисунку 1.5 наведені ілюстрації основних асимптотичних позначень.

1.5.2 Ω-позначення. Ω-позначення використовується для запису ***асимптотичної нижньої межі*** (*asymptotic lower bound*) для функції $f(n)$.

Говорять, що функція $f(n)$ належить множині функцій $\Omega(g(n))$, що записується як $f(n)\in\Omega(g(n))$, якщо існують позитивна константа $c>0$ і $n_0\in\{0,\ 1,\ 2,...\}$, такі що для будь-яких $n>n_0$

$$0\leq cg(n)\leq f(n). \qquad (1.11)$$

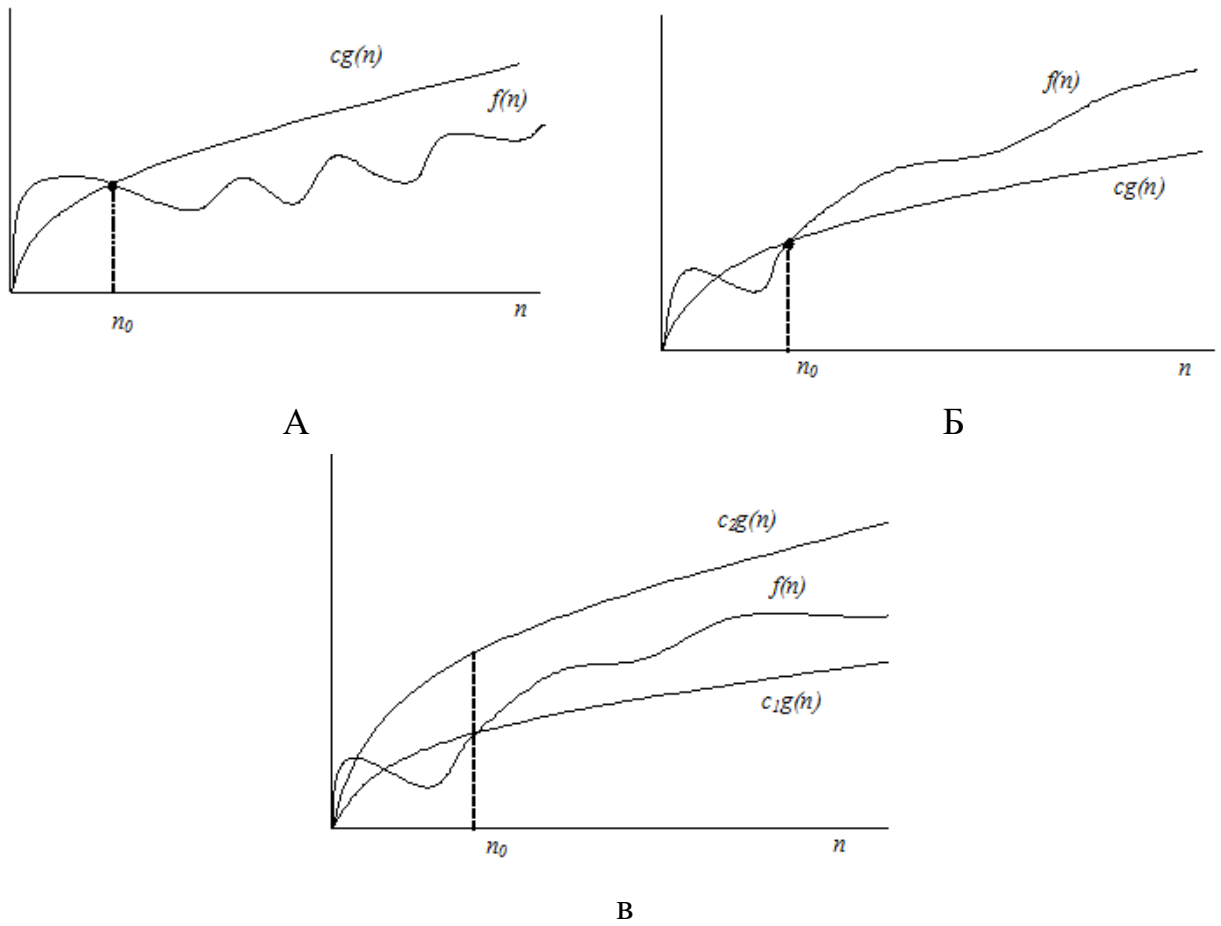


Рисунок 1.5 − Ілюстрація асимптотичних позначень.

Формально множина $\Omega(g(n))$ визначається наступним чином:

$$\Omega(g(n)) = \left\{ \begin{array}{ll} f(n): & \exists c > 0, n_0 \in \{0,1,2,...\}, \text{ такі,} \\ & \text{що} \quad 0 \le cg(n) \le f(n), \quad \forall n > n_0 \end{array} \right\} \qquad (1.12)$$

Інакше кажучи, $\Omega(g(n))$ − це множина всіх функцій, значення яких при великих n не перевищують значення $cg(n)$. Зазвичай факт належності функції $f(n)$ множині $\Omega(g(n))$ записують як $f(n) = \Omega(g(n))$. Читається як ***« f від n є омега велике від g від n »***.

Для доведення $f(n) = \Omega(g(n))$, потрібно знайти константи $c > 0$ і $n_0 \in \{0,\ 1,\ 2,...\}$, які забезпечують виконання умов (1.12). Якщо доведено, що $f(n) = \Omega(g(n))$, то говорять: ***«функція $f(n)$ асимптотично обмежена знизу функцією $g(n)$ з точністю до постійного множника»***. Таким чином, добуток $cg(n)$ є асимптотичною оцінкою знизу часу $f(n)$ роботи алгоритму.

Приклад. Доведемо справедливість відносини $3n^4+15=\Omega\left(n^2\right)$. Дотримуючись визначення, нам необхідно показати існування констант $c>0$ і $n_0\in\{0,\ 1,\ 2,...\}$, при яких $3n^4+15\geq n^2$ для будь-яких $n\geq n_0$. Для доведення досить взяти $c=1$ і $n_0=1$.

1.5.3 Θ-позначення. Θ-позначення дозволяє записати ***асимптотично точну оцінку*** (*asymptotic tight bound*) для функції $f(n)$. Функція $f(n)$ належить множини функцій $\Theta(g(n))$, записується як $f(n)=\Theta(g(n))$, якщо існують позитивні константи $c_1>0$, $c_2>0$ і $n_0\in\{0,\ 1,\ 2,...\}$, такі що для будь-яких $n\geq n_0$

$$0\leq c_1 g(n)\leq f(n)\leq c_2 g(n). \tag{1.13}$$

Читається як ***« f від n є тета велике від g від n »***. Множина $\Theta(g(n))$ визначається наступним чином

$$\Theta(g(n))=\left\{\begin{matrix} f(n):\ \ \exists c_1>0, \exists c_2>0\ i\ n_0\in\{0,\ 1,\ 2,...\},\ \text{такі,} \\ \text{що}\ \ 0\leq c_1 g(n)\leq f(n)\leq c_2 g(n),\ \ \forall n\geq n_0 \end{matrix}\right\}. \tag{1.14}$$

Для доведення того, що $f(n)=\Theta(g(n))$, потрібно знайти константи $c_1>0$, $c_2>0$ і $n_0\in\{0,\ 1,\ 2,...\}$, які забезпечують виконання нерівностей (1.13). Якщо доведено, що $f(n)=\Theta(g(n))$, то говорять: ***«функція асимптотично обмежена знизу і зверху функцією з точністю до постійного множника»***.

Позначення Θ більш сильне, ніж та Ω. Використовуючи позначення теорії множин, можна записати $\Theta(g(n))\subseteq O(g(n))$.

Твердження. Для будь-яких двох функцій $f(n)$ і $g(n)$ відношення $f(n)=\Theta(g(n))$ виконується тоді і тільки тоді, коли $f(n)=O(g(n))$ і $f(n)=\Omega(g(n))$.

З цього твердження випливає: якщо відомо, що $f(n)=\Theta(g(n))$, то це гарантує $f(n)=O(g(n))$ і $f(n)=\Omega(g(n))$. З іншого боку, якщо ми покажемо, що для деякої функції $f(n)$ одночасно виконуються $f(n)=O(g(n))$ і $f(n)=\Omega(g(n))$, то тим самим ми доведемо справедливість $f(n)=\Theta(g(n))$.

Розглянуте твердження дає нам двокрокову процедуру для побудови асимптотично точних оцінок для функцій.

Приклад. Доведемо коректність затвердження $3n^4+15=\Theta\left(n^4\right)$. За визначенням потрібно знайти константи $c_1>0$, $c_2>0$ і $n_0\in\{0,\ 1,\ 2,...\}$, які забезпечують виконання нерівностей

$$0\le c_1n^4\le 3n^4+15\le c_2n^4. \qquad (1.15)$$

Спершу покажемо, що наша функція обмежена зверху. Нехай $c_2=4$. Нерівність $3n^4+15\le 4n^4$ виконується для будь-яких $\forall n\ge 2$ − це точка перетину функцій $3n^4+15$ і $4n^4$. Таким чином $n_0=2$ і $3n^4+15=O\left(n^4\right)$. Залишилося показати виконання нерівностей $0\le c_1n^4\le 3n^4+15$ Вони справедливі при $c_1=3$ і $n_0=1$, отже $3n^4+15=\Omega\left(n^4\right)$. Далі, нам потрібно вибрати n_0, при якому виконуються як умови обмеження функції знизу, так і обмеження зверху. Тому виберемо n_0 як максимальне значення з знайдених n_0, це забезпечить виконання обох умов: $n_0=\max\{1,2\}=2$. Ми знайшли константи $c_1=3$, $c_2=4$ і $n_0=2$, тим самим довели справедливість вихідного твердження 2 $3n^4+15=\Theta\left(n^4\right)$.

1.5.4 *o*-позначення. Позначення ***o*** використовують для зазначення того, що ***верхня межа функції не є асимптотично точно***. Наприклад, запис

$$3n^4+15=o\left(n^5\right)$$

містить в собі коректну асимптотичну верхню межу функції $3n^4+15$, але не забезпечує нас вірною інформацією про асимптотично точну верхню межу функції. Дамо визначення ***o***-позначення

$$o\left(g\left(n\right)\right)=\left\{\begin{array}{l} f(n):\quad \forall c>0,\exists n_0\in\{0,1,2,...\},\text{ такі,} \\ \qquad\qquad \text{що}\quad 0\le f\left(n\right)\le cg\left(n\right),\quad \forall n>n_0 \end{array}\right\}. \qquad (1.16)$$

Таким чином $o\left(g\left(n\right)\right)$ − це множина всіх функцій, значення яких при великих n менше значення $cg\left(n\right)$ для будь-яких $\forall c>0$. Факт належності функції $f\left(n\right)$ множині $o\left(g\left(n\right)\right)$ записують як $f\left(n\right)=o\left(g\left(n\right)\right)$. Читається як ***« f від n є o мале від g від n ».*** Важливою відмінністю від ***O***-позначення є обмеження зверху функції $f\left(n\right)$ значенням $cg\left(n\right)$ для будь-яких (всіх)

позитивних констант c. Наприклад, $3n^4+15=o\left(n^5\right)$, в той же час $3n^4+15=o\left(n^4\right)$.

1.5.5 ω-позначення. Позначення подібно позначенню Ω. Дамо йому формальне визначення:

$$\omega\left(g\left(n\right)\right)=\left\{\begin{array}{ll} f(n): & \forall c>0, \exists n_0 \in \{0,1,2,...\}, \text{такі,} \\ & \text{що} \quad 0 \le cg\left(n\right) \le f\left(n\right), \quad \forall n>n_0 \end{array}\right\}. \qquad (1.17)$$

Інакше кажучи, $\omega\left(g\left(n\right)\right)$ − це множина всіх функцій, значення яких при великих n не перевищують значення $cg\left(n\right)$ для будь-яких c. Факт належності функції $f\left(n\right)$ множині $\omega\left(g\left(n\right)\right)$ записують як $f\left(n\right)=\omega\left(g\left(n\right)\right)$. Читається як ***« f від n є омега мале від g від n »***. Важливою відмінністю від ***Ω***-позначення є обмеження знизу функції $f\left(n\right)$ значеннями $cg\left(n\right)$ для будь-яких позитивних констант c. Наприклад, справедливо відношення $n^4=\omega\left(n^3\right)$, але $30n^4=\omega\left(n^4\right)$.

1.5.6 Випадок функції декількох змінних. Асимптотичні позначення можна узагальнити і на випадок функцій декількох змінних. Наприклад, для функції двох змінних матимемо наступне визначення ***O***-позначення:

$$O\left(g\left(n,m\right)\right)=\left\{\begin{array}{ll} f(n,\mathrm{m}): & \exists c>0, n_0 \in \{0,1,2,...\}, \mathrm{m}_0 \in \{0,1,2,...\}, \text{таких,} \\ & \text{що} 0 \le f\left(n,m\right) \le cg\left(n,m\right), \quad \forall n>n_0, \forall m>m_0 \end{array}\right\}. \quad (1.18)$$

Аналогічно можна ввести визначення інших асимптотичних позначень для функцій декількох змінних.

1.5.7 Порівняння порядку зростання функцій через межу. Нехай, як і раніше, ми маємо дві функції $f\left(n\right)$ і $g\left(n\right)$. Для відповіді на запитання, яка з функцій має вищий порядок зростання, досить обчислити межу їх відношення при $n \to \infty$. [4, 13]

$$\lim_{n\to\infty}\frac{f\left(n\right)}{g\left(n\right)}=\begin{cases} 0, & \text{якщо} \quad f\left(n\right)=o\left(g\left(n\right)\right), \\ c, & \text{якщо} \quad f\left(n\right)=\Theta\left(g\left(n\right)\right), \\ \infty, & \text{якщо} \quad f\left(n\right)=\omega\left(g\left(n\right)\right). \end{cases} \qquad (1.19)$$

Якщо межа дорівнює 0, то $f\left(n\right)=o\left(g\left(n\right)\right)$ і $f\left(n\right)=O\left(g\left(n\right)\right)$. Другий випадок: $f\left(n\right)=\Theta\left(g\left(n\right)\right)$, отже, $f\left(n\right)=O\left(g\left(n\right)\right)$ і $f\left(n\right)=\Omega\left(g\left(n\right)\right)$. З третього випадку слідує $f\left(n\right)=\omega\left(g\left(n\right)\right)$ і $f\left(n\right)=\Omega\left(g\left(n\right)\right)$.

Приклад. Порівняємо порядки (швидкості) зростання функцій $f(n)=n^3(n-2)/3$ і $g(n)=n^4$.

$$\lim_{n\to\infty}\frac{n^3(n-2)/3}{n^4}=\frac{1}{3}\lim_{n\to\infty}\left(1-\frac{2}{n}\right)=\frac{1}{3}.$$

Межа дорівнює позитивній константі, тоді за (1.19) обидві функції мають однаковий порядок зростання $n^3(n-2)/3=\Theta\left(n^4\right)$.

Приклад. Порівняємо порядки зростання функції $f(n)=n-2$ і функції $g(n)=\log_a n$ при $a>1$.

$$\lim_{n\to\infty}\frac{n-2}{\log_a n}=\lim_{n\to\infty}\left(\frac{1}{\frac{1}{n\ln_n a}}\right)=\lim_{n\to\infty}\left(n\ln_n a\right)=\infty.$$

Для обчислення останньої межі використано правило Лопіталя – перейшли до розгляду межі відношення похідних функцій $f(n)$ і $g(n)$. Межа дорівнює нескінченності. Це означає, що функція $f(n)$ має більший порядок зростання, ніж $g(n)$. Іншими словами, $n-2=\omega(\log n)$. Зверніть увагу, що тут навмисно опущено основу логарифма, так як при використанні спільно з асимптотичними позначеннями вона не має сенсу. За властивостями логарифмів зміна основи приводить до множення логарифма на константу, а за властивостями асимптотичних позначень константи можуть бути відкинуті:

$$\log_a n=\frac{1}{\log_c a}\log_c n. \tag{1.20}$$

1.5.8 Властивості асимптотичних відношень. Факт належності функції $f(n)$ деякій множині ми записували через знак рівності, наприклад, $f(n)=O\left(g(n)\right)$. Це позначення є зручним на практиці, але потрібно пам’ятати, що сенс його полягає в тому, що функція, що стоїть зліва від знаку рівності, належить множині, що вказана справа:

$$f(n)\in O\left(g(n)\right) \tag{1.21}$$

Тобто це не є рівність в звичайному розумінні, а несиметричне бінарне відношення між функціями $f(n)$ і $g(n)$. Наприклад, запис $f(n)=O\left(g(n)\right)$ є коректним, в той же час вираз $O\left(g(n)\right)=f(n)$ позбавлений сенсу.

Якщо асимптотичне позначення O, Ω або Θ присутнє у формулі, то замість нього можна подумки підставити будь-яку функцію з цієї множини. Наприклад, у виразі $5n^3+3n^2+55n+5=5n^3+\Theta\left(n^2\right)$, замість $\Theta\left(n^2\right)$ можна підставити будь-яку функцію з квадратичним порядком зростання. Асимптотичні бінарні відношення O, Ω та Θ мають багато властивостей бінарних відношень між дійсними числами (для визначеності позначимо числа через $\boldsymbol{f}$ і $\boldsymbol{g}$):

$$f(n)=O(g(n)) \text{ відповідає } f \le g; \tag{1.22}$$

$$f(n)=\Omega(g(n)) \text{ відповідає } f \ge g; \tag{1.23}$$

$$f(n)=\Theta(g(n)) \text{ відповідає } f = g; \tag{1.24}$$

$$f(n)=o(g(n)) \text{ відповідає } f < g; \tag{1.25}$$

$$f(n)=\omega(g(n)) \text{ відповідає } f > g. \tag{1.26}$$

Транзитивність

$$f(n)=O(g(n)) \text{ і } g(n)=O(h(n)), \text{ то } f(n)=O(h(n)); \tag{1.27}$$

$$f(n)=\Omega(g(n)) \text{ і } g(n)=\Omega(h(n)), \text{ то } f(n)=\Omega(h(n)); \tag{1.28}$$

$$f(n)=\Theta(g(n)) \text{ і } g(n)=\Theta(h(n)), \text{ то } f(n)=\Theta(h(n)); \tag{1.29}$$

$$f(n)=o(g(n)) \text{ і } g(n)=o(h(n)), \text{ то } f(n)=o(h(n)); \tag{1.30}$$

$$f(n)=\omega(g(n)) \text{ і } g(n)=\omega(h(n)), \text{ то } f(n)=\omega(h(n)). \tag{1.31}$$

Рефлексивність

$$f(n)=O(f(n)); \tag{1.32}$$

$$f(n)=\Omega(f(n)); \tag{1.33}$$

$$f(n)=\Theta(f(n)). \tag{1.34}$$

Симетричність

$$f(n)=\Theta(g(n)) \text{ тоді і лише тоді, коли } g(n)=\Theta(f(n)). \tag{1.35}$$

Перестановочна симетрія

$$f(n)=O(g(n)) \text{ тоді і лише тоді, коли } g(n)=\Omega(f(n)); \tag{1.36}$$

$$f(n)=o(g(n)) \text{ тоді і лише тоді, коли } g(n)=\omega(f(n)). \tag{1.37}$$

Розглянемо властивості O-позначення, які зручно використовувати на практиці:

– добуток – якщо $f_1(n)=O(g_1(n))$ і $f_2(n)=O(g_2(n))$, то

$$f_1(n)\cdot f_2(n)=O(g_1(n)\cdot g_2(n)); \qquad (1.38)$$

– сума – якщо $f_1(n)=O(g_1(n))$ і $f_2(n)=O(g_2(n))$, то

$$f_1(n)+f_2(n)=O(|g_1(n)|+|g_2(n)|) \text{ і}$$

$$f_1(n)+f_2(n)=O(\max\{g_1(n);g_2(n)\}); \qquad (1.39)$$

– множення на константу – якщо $f(n)=O(g(n))$ і $k>0$, то

$$k\cdot f(n)=O(g(n)). \qquad (1.40)$$

При використанні асимптотичних позначень константи в функції $T(n)$ ігноруються.

З властивостей слідує: для будь-якого полінома $p(n)$ ступеня $k\in\{0,1,2,...\}$ при $a_k>0$ справедливо

$$p(n)=a_k n^k+a_{k-1}n^{k-1}+...+a_1 n^1+a_0=\Theta(n^k). \qquad (1.41)$$

1.6 Етапи асимптотичного аналізу

Асимптотичний аналіз складності алгоритму полягає в побудові ***асимптотичних оцінок його обчислювальної складності*** (*computational complexity*) або ***обсягу необхідної йому пам'яті*** (*space complexity*) в гіршому, середньому або кращому випадку. Розглянемо основні кроки асимптотичного аналізу.

1. Визначаються параметри, від яких залежить час виконання алгоритму або обсяг необхідної йому пам'яті: встановлюється, скільки аргументів буде у функції $T(n)$. Наприклад, $T(n)$, $T(n,m)$, $T(n,m,k)$.

2. Для гіршого, середнього або кращого випадку обчислюється кількість операцій алгоритму або обсяг необхідної йому пам'яті, що записується як функція від параметрів, встановлених на попередньому кроці. Наприклад, $T(n)=4n^5+3n^2+10n+5$. У першу чергу обчислюють кількість операцій алгоритму для гіршого випадку. Якщо на практиці виникнення цього випадку малоймовірно, то підраховується кількість операцій алгоритму для середнього випадку. На даному етапі можна обмежитися підрахунком кількості тільки ***базових операцій*** (*basic operations*) − найбільш важливих операцій, від яких

залежить час виконання алгоритму. Наприклад, можна не враховувати константну кількість операцій читання значень змінних з пам'яті і операцій привласнення.

3. До побудованої функції $T(n)$ застосовуються асимптотичні позначення для класифікації порядку її зростання. За допомогою Θ-позначення будується асимптотично точна оцінка для функції $T(n)$. У випадку виникнення труднощів побудови такої оцінки, обмежуються визначенням хоча б O-позначення − асимптотичної верхньої межі функції $T(n)$.

На практиці в процесі асимптотичного аналізу алгоритмів зустрічається невеликий набір класів складності, які наведені в таблиці 1.3 (в порядку зростання швидкості росту).

Таблиця 1.3 − Класи складності алгоритмів

Позначення класу складності	Назва класу	Приклад алгоритму
$O(1)$	Константна складність	Алгоритм визначення парності числа. Час виконання і обсяг необхідної пам'яті не залежать від розміру вхідних даних.
$O(\log n)$	Логарифмічна складність	Алгоритм бінарного пошуку у впорядкованому масиві. Такі алгоритми на кожному кроці оброблюють лише частину вхідного набору даних.
$O(n)$	Лінійна складність	Алгоритм пошуку мінімального елемента в невпорядкованому масиві. Алгоритми такої складності виконують перегляд всього вхідного набору даних.
$O(n\log n)$	Лінійно-логарифмічна складність	Алгоритм сортування злиттям. Така складність характерна для алгоритмів, розроблених з використання методу декомпозиції.
$O(n^2)$	Квадратична складність	Алгоритм сортування вибором.
$O(n^3)$	Кубічна складність	Алгоритм множення квадратних матриць за визначенням.
$O(2^n)$	Експоненційна складність	Алгоритми, що оброблюють всі підмножини деякої множини з n елементів.
$O(n!)$	Факторіальна складність	Оптимізаційні алгоритми, що реалізують повний перебір множини допустимих розвозів.

Приклад. Факторіал числа. Проведемо асимптотичний аналіз обчислювальної складності алгоритму обчислення факторіала цілого числа. За визначенням факторіал числа є

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n. \tag{1.42}$$

Алгоритм 1.4 – Обчислення факторіала кількості

```
1.int factorial(int n)
2.{
3.    int fact = 1;
4.    for (int i = 1; i <= n; i++)
5.    {
6.        fact *= i;
7.    }
8.    return fact;
9.}
```

Крок 1. Час виконання алгоритму `factorial` залежить тільки від значення n, отже, функція $T(n)$ буде мати один аргумент.

Крок 2. Обчислимо кількість $T(n)$ операцій алгоритму. Найгірший, середній і кращий випадки для цього алгоритму збігаються. Одна операція йде на ініціалізацію змінної $fakt$, далі на кожній з $n-1$ ітерації циклу виконується дві операції (множення і привласнення), після чого відбувається повернення значення з функції – ще одна операція. отже,

$$T(n) = 1 + 2(n-1) + 1 = 2n. \tag{1.43}$$

Крок 3. Отримаємо асимптотично точну оцінку обчислювальної складності алгоритму. Доведемо, що $2n = \Theta(n)$. Візьмемо $c_1 = 1$, $c_2 = 2$ і $n_0 = 1$, ці константи забезпечує виконання нерівності

$$0 \le n \le 2n \le 2n, \quad \text{при} \quad n \ge 1. \tag{1.44}$$

Результатом асимптотичного аналізу обчислювальної складності алгоритму `factorial` є асимптотично точна оцінка $\Theta(n)$, яка говорить про те, що алгоритм має лінійну обчислювальну складність. Тепер виконаємо асимптотичний аналіз складності за пам’яттю алгоритму. Обчислимо кількість $M(n)$ осередків пам’яті ***RAM***-машини, яка потрібно алгоритму. У рядку 3 ініціалізується змінна *fakt* – це вимагає одного осередку пам’яті, для змінної i також потрібен один осередок пам’яті.

Остаточно отримуємо

$$M(n) = 2 = \Theta(1). \tag{1.44}$$

Алгоритм `factorial` має константну складність за пам'яттю − він не створює в пам'яті структур даних, розмір яких залежить від n.

Приклад. Сортування вибором. Проведемо асимптотичний аналіз обчислювальної складності ***алгоритму сортування вибором*** (*selection sort*).

Алгоритм 1.5 − Сортування вибором

```
1.int selectionSort(double *mas, int n)
2.{
3.    double tmp = 0;
4.    for (int i = 0; i < n - 1; i++)
5.    {
6.        int min = i;
7.        for (int j = i + 1; j < n; j++)
8.        {
9.            if (mas[j] < mas[min])
10.           {
11.               min = j;
12.           }
13.       }
14.       if (min != i)
15.       {
16.           tmp = mas[i];
17.           mas[i] = mas[min];
18.           mas[min] = tmp;
19.       }
20.   }
21.   return 0;
22.}
```

Крок 1. Час виконання алгоритму `selectionSort` залежить від розміру масиву і значень елементів в ньому.

Крок 2. Обчислимо кількість $T(n)$ операцій алгоритму, що виконуються ним в гіршому випадку. Тобто, у випадку коли необхідно виконати максимальну кількість операцій при виконанні на всіх ітераціях циклів умов в рядках 9 і 14. Найгірший випадок для алгоритму `selectionSort` − це масив, елементи якого впорядковані за спаданням − в порядку, зворотному напрямку сортування. В такому випадку зовнішній цикл `for` виконується $n-1$ раз. На

кожній ітерації цього циклу одна операція доводиться на запис значення в змінну (рядок 6), що разом дає $n-1$ операцію. На рядки 14−19 доводиться чотири операції (перевірка умови і три привласнення), що дає ще $4(n-1)$ операцій. тоді

$$T(n) = (n-1) + 4(n-1) + T_{inner_loop}(n). \quad (1.45)$$

де функція $T_{inner_loop}(n)$ визначає кількість операцій, що припадають на внутрішній цикл, за весь час виконання алгоритму.

Обчислимо, скільки операцій припадає на внутрішній цикл в рядках 7-13. Розглянемо процес виконання зовнішнього циклу:

− при $i=1$ внутрішній цикл виконується $n-1$ раз (змінна j приймає значення від 2 до n);

− при $i=2$ внутрішній цикл виконується $n-2$ рази (змінна j приймає значення від 3 до n);

− ...

− при $i=n-2$ внутрішній цикл виконується 2 рази (змінна j приймає значення від $n-1$ до n);

− при $i=n-1$ внутрішній цикл виконується 1 раз (змінна j приймає значення від n до n).

З огляду на те, що на кожній ітерації внутрішнього циклу виконується дві операції (перевірка умови і привласнення (рядки 7 −9)), отримуємо

$$T_{inner_loop}(n) = 2(n-1+n-2+...+1). \quad (1.46)$$

$$T(n) = (n-1) + 4(n-1) + 2[n-1+n-2+...+1]. \quad (1.47)$$

У квадратних дужках записана сума членів арифметичної прогресії з різницею 1. Обчислимо її

$$T(n) = (n-1) + 4(n-1) + 2[\frac{n^2-n}{2}] = \mathrm{n}^2 + 4\mathrm{n} - 5. \quad (1.48)$$

Крок 3. Отримаємо асимптотично точну оцінку обчислювальної складності алгоритму `selectionSort`. Доведемо справедливість

$$T(n) = \mathrm{n}^2 + 4\mathrm{n} - 5 = \Theta(n^2). \quad (1.49)$$

Дотримуючись визначення асимптотичних позначень, візьмемо константи $c_1 = 1$, $c_2 = 2$ і $n_0 = 2$, вони забезпечують виконання нерівності

$$0 \le n^2 \le n^2 + 4n - 5 \le 2n^2, \quad \text{при} \quad n \ge 2. \qquad (1.50)$$

Таким чином, ми показали, що алгоритм сортування вибором в гіршому випадку має квадратичну обчислювальну складність, і ця оцінка є асимптотично точною.

Діючи аналогічно, можна оцінити обчислювальну складність алгоритму сортування вибором для кращого випадку, коли вхідний масив вже упорядкований в напрямку сортування. Умови всередині циклів виконуватися не будуть, і кількість операцій алгоритму дорівнюватиме

$$T(n) = (n-1) + (n-1) + [n-1 + n-2 + ... + 1] = \Theta(n). \qquad (1.51)$$

Складність за пам'яттю алгоритму сортування вибором є константною, так як в процесі своєї роботи алгоритму потрібно лише чотири додаткові осередки під змінні *i, j, min* і *tmp* (вхідний масив *mas* не враховується).

Завдання для самостійного виконання

1. Визначте параметри, від яких залежить час роботи перерахованих нижче алгоритмів:

− алгоритм послідовного підсумовування елементів масиву;

− алгоритм складання за визначенням двох квадратних матриць;

− алгоритм Евкліда знаходження найбільшого спільного дільника двох цілих чисел.

2. Упорядкуйте наступні функції за неспаданням швидкості їх зростання n, n^2, $\sqrt{n}$, $\log_2 n$, $\log_{10} n$, $\sqrt[3]{n}$, $\log_2(\log_2 n)$, $(\log_2 n)^3$, $\frac{3}{2^n}$, 2^n.

3. Є обчислювач (процесор), який виконує $5 \cdot 10^{12}$ операцій в секунду, і три алгоритми з кількістю операцій

$$T_1(n) = 500n \cdot \log_2 n + 2048,\ T_2(n) = 4n^2 + 10n + 48,\ T_3(n) = n! \cdot \log_2 n + 2,$$

Визначте скільки часу (днів, годин, хвилин, секунд) буде виконуватися кожен алгоритм при $n = 10$, $n = 100$, $n = 500$.

4. Використовуючи визначення асимптотичних позначень O і Θ, доведіть справедливість наступних відношень:

$$n^4 + 3n^3 - 10n^2 - 50n + 3 = O(n^4);$$

$n^2 = O\left(2^n\right)$;

$2^{n-1} = \Theta\left(2^n\right)$.

5. Порівняйте порядки зростання функцій через межу

$f(n) = 2^{3n}$, $g(n) = 2^n$;

$f(n) = \ln(n)$, $g(n) = 3n$;

$f(n) = 2^{5n}$, $g(n) = n^{20}$ (використайте правило Лопіталя 20 разів);

$f(n) = \log_a(n)$, $g(n) = \sqrt[a]{n}$, $a > 1$,;

$f(n) = n!$, $g(n) = 2^n$ (використайте формулу Стиглінга).

6. Нижче наведено код функції `loopsA`. Обчисліть кількість $T(n)$ операцій у ній.

```
int loopsA(int n){
    int x = 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = i + 1; j <= n; j++)
        {
            for (int k = 1; k <= n; k++)
            {
                x += 1;
            }
        }
    } return x;}
```

7. Нижче наведено псевдокод функції `loopsB`. Обчисліть кількість $T(n)$ операцій у ній.

```
int loopsB(int n){
    int x = 1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = i + 1; j <= n; j++)
        {
            for (int k = 1; k <= j; k++)
            {
                x += 1;
            }
        }
    }
    return x;}
```