Тема 3. Метод декомпозиції

3.1. Метод декомпозиції

В методі сортування включенням використовується так званий **інкрементний** підхід: маючи відсортований підмасив $A[1\ldots j-1]$, ми розміщуємо черговий елемент $A[j]$ туди, де він повинен заходитись, в результаті чого отримуємо відсортований підмасив $A[1\ldots j]$. В цій темі буде розглянутий рекурсивний підхід, коли початкова задача розбивається на підзадачі, розв'язки яких після цього певним чином опрацьовуються для отримання загального рішення. Зрозуміло, що підзадачі початкової задачі потім так само розбиваються на власні підзадачі, і так триває до досягнення певного мінімального розміру задачі, яка сама розв'язується вже тривіальним чином.

Ця рекурсивна парадигма отримала назву «**розділяй та володарюй**» (divide and conquer). Одним з перших алгоритмів, який працював в рамках цієї парадигми, був метод Карацуби множення двох цілих чисел, який розглядався в першій темі цього курсу. Сама парадигма «розділяй та володарюй» складається з трьох етапів:

- *Розділення* задачі на декілька підзадач.
- *Рекурсивний розв'язок* цих підзадач. Коли об'єм підзадач достатньо малий, вони розв'язуються безпосередньо.
- *Комбінування* розв'язку початкової задачі з розв'язків допоміжних підзадач.

Розгляд цієї парадигми ми почнемо з алгоритму сортування **методом злиття** (merge sort). В рамках підходу «розділяй та володарюй» цей метод можна описати так:

- *Розділення*: послідовність для сортування, яка складається з n елементів, розбивається на дві менші послідовності, кожна з яких містить $n/2$ елементів.
- *Рекурсивний розв'язок*: сортування обох створених послідовностей методом злиття у випадку, якщо їх розмір перевищує один.
- *Комбінування*: злиття двох відсортованих послідовностей для отримання кінцевого результату.

Рекурсія досягає своєї нижньої межі, коли довжина послідовності для сортування дорівнює 1. В цьому випадку вся робота вже зроблена, оскільки будь-яку таку послідовність можна вважати впорядкованою.

Основна операція, яка виконується в процесі сортування за методом злиття, – це об'єднання двох відсортованих послідовностей під час комбінування (останній етап). Це робиться за допомогою додаткової процедури `Merge`(A, p, q, r), де A – це масив, а p, q та r – індекси, які нумерують елементи масиву, такі що $p \le q < r$. В цій процедурі припускається, що елементи підмасивів $A[p\ldots q]$ та $A[q+1\ldots r]$ впорядковані. Вона зливає ці два підмасиви в один відсортований, елементи якого замінюють поточні елементи підмасиву $A[p\ldots r]$.

Для виконання процедури `Merge` потрібний час $\Theta(n)$, де $n = r - p + 1$ – кількість елементів, які потрібно об'єднати. Для демонстрації роботи процедури можна знову повернутись до прикладу грального карт. Нехай на столі лежать дві стопки карт, кожна з яких є вже відсортованою. Для того, щоб об'єднати їх в одну відсортовану стопку, можна діяти наступним чином: на кожному кроці ми порівнюємо по одній карті зі стопок, які лежать згори, та обираємо в нову стопку ту з них, яка є найменшою. Цей крок повторюється до тих пір, допоки одна зі стопок не спорожніє. Тоді залишиться лише додати карти зі стопки, що лишилась, у нову стопку не порушуючи порядок карт. З обчислювальної точки зору виконання кожного основного кроку займає однакові проміжки часу, тоді як все зводиться до порівняння двох верхніх карт. Оскільки необхідно виконати принаймні n основних кроків, час роботи процедури `Merge` дорівнює $\Theta(n)$.

У наведеному нижче псевдокоді представлена процедура `Merge`. Проте для її спрощення використовуються додаткові міркування. Щоб на кожному кроці не перевіряти,

чи не спорожнів якийсь з двох підмасивів, до кожного з них додається так званий сигнальний елемент, який має значення нескінченності ∞. Таким чином, не існує елементів масивів, які були б більшими за ці сигнальні елементи. Робота процедури `Merge` продовжується до тих пір, поки поточні елементи в обох підмасивах не виявляться сигнальними. Як тільки це станеться, це буде означати, що всі несигнальні елементи розміщені у вихідний масив. Оскільки завчасно відомо, що у вихідному масиві повинен бути присутній $r - p + 1$ елемент, то виконавши таку кількість кроків можна зупинитись.

```
Merge(A, p, q, r)
1  n1 ← q - p + 1
2  n2 ← r - q
3  Створити масиви L[1..n1+1] та R[1..n2+1]
4  for i ← 1 to n1
5     do L[i] ← A[p+i-1]
6  for j ← 1 to n2
7     do R[j] ← A[q+j]
8  L[n1+1] ← ∞
9  R[n2+1] ← ∞
10 i ← 1
11 j ← 1
12 for k ← p to r
13    do if L[i] ≤ R[j]
14          then A[k] ← L[i]
15               i ← i + 1
16          else A[k] ← R[j]
17               j ← j + 1
```

Лістинг 3.1 Процедура злиття `Merge`

Детально опишемо роботу процедури `Merge`. В рядку 1 обраховується довжина n_1 підмасиву $A[p…q]$, а у рядку 2 - довжина n_2 підмасиву $A[q+1…r]$. Далі в рядку 3 створюються масиви L («лівий») та R («правий»), довжини яких відповідно $n_1 + 1$ та $n_2 + 1$. В двох циклах **for** в рядках 4-5 та 6-7 елементи масиву $A[p…q]$ та $A[q+1…r]$ копіюються у масиви L та R відповідно. В рядках 8 та 9 останнім елементам масивів L та R приписуються сигнальні значення.

Як показано на рис. 3.1, в результаті копіювання та додавання сигнальних елементів отримуємо масив L з послідовністю чисел ⟨2, 4, 5, 7, ∞⟩ та масив R з послідовністю ⟨1, 2, 3, 6, ∞⟩. Світло-сірі комірки масиву A містять кінцеві елементи, а світло-сірі комірки масивів L та R – значення, які ще тільки необхідно скопіювати в масив A. У темно-сірих комірках A містяться значення, які будуть замінені іншими, а в темно-сірих масивів L та R – значення, які вже скопійовані назад в A.

В рядках 10-17 лістингу 3.1 виконуються $r - p + 1$ основних кроків, в ході кожного з яких відбуваються маніпуляції з інваріантом циклу:

> Перед кожною ітерацією циклу **for** в рядках 12-17, підмасив $A[p…k–1]$ містить $k–p$ найменших елементів масивів L та R у відсортованому порядку. Окрім того, елементи $L[i]$ та $R[j]$ є найменшими елементами L та R, які ще не були скопійовані у A.

Необхідно показати, що цей інваріант циклу зберігається перед першою ітерацією даного циклу **for**, що кожна ітерація циклу не порушує його, і що з його допомогою можна продемонструвати коректність алгоритму, коли цикл закінчує свою роботу.

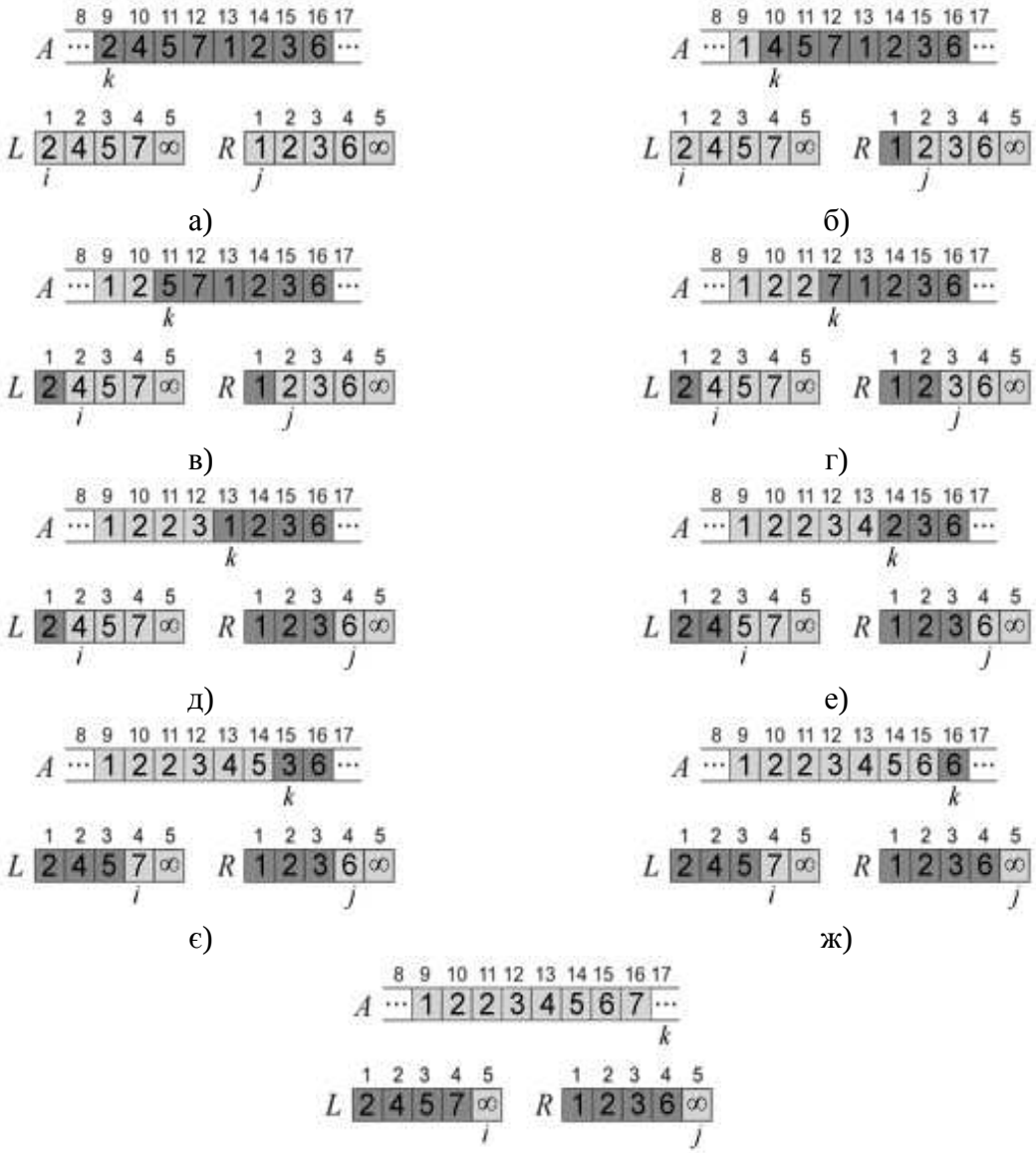

Рис. 3.1. Приклад виконання процедури `Merge`.

1. *Ініціалізація*. Перед першою ітерацією циклу $k = p$, тому підмасив $A[p…k–1]$ порожній. Він містить $k – p = 0$ найменших елементів масивів L та R. Оскільки $i = j = 1$, елементи $L[i]$ та $R[j]$ – найменші елементи масивів L та R, які ще не скопійовані у A.
2. *Збереження*. Щоб пересвідчитись, що інваріант циклу зберігається після кожної ітерації, спочатку припустимо, що $L[i] \le R[j]$. Тоді $L[i]$ – найменший елемент, який ще не скопійовано в A. Оскільки в підмасиві $A[p…k–1]$ міститься $k–p$ найменших елементів, після виконання рядку 14, в якому значення елементу $L[i]$ приписується елементу $A[k]$, в підмасиві $A[p…k]$ буде міститись $k–p+1$ найменший елемент. В результаті збільшення параметру k циклу **for** та значення змінної i (рядок 15), інваріант циклу відновлюється перед наступною ітерацією. Якщо $L[i] > R[j]$, то в рядках 16 та 17 виконуються відповідні дії, які також зберігають інваріант циклу.
3. *Завершення*. Алгоритм завершується, коли $k = r + 1$. У відповідності з інваріантом циклу, підмасив $A[p…k–1]$ (тобто підмасив $A[p…r]$) містить $k - p = r - p + 1$ найменших елементів масивів L та R у відсортованому порядку.

Сумарна кількість елементів в масивах L та R дорівнює $n_1 + n_2 + 2 = r - p + 3$. Всі вони, окрім двох найбільших, скопійовані назад в масив A, а два елементи що залишились є сигнальними.

Щоб показати, що час роботи процедури `Merge` дорівнює $\Theta(n)$, де $n = r - p + 1$, зазначимо, що кожний рядок 1-3 та 8-11 виконується протягом фіксованого часу; довжина циклів **for** у рядках 4-7 дорівнює $\Theta(n_1 + n_2) = \Theta(n)$, а в циклі **for** у рядках 12-17 виконується n ітерацій, на кожну з яких витрачається фіксований час.

Тепер процедуру `Merge` можна використовувати в якості підпрограми в алгоритмі сортування злиттям. Процедура `MergeSort`(A, p, r) виконує сортування елементів в підмасиві $A[p…r]$. Якщо виконується нерівність $p \geq r$, то в цьому масиві елементів міститься не більше одного, тому він є відсортованим. У протилежному випадку відбувається розбиття, під час якого обраховується індекс q, який розбиває масив $A[p…r]$ на два підмасиви: $A[p…q]$ з $\lceil n/2 \rceil$ елементами та $A[q+1…r]$ з $\lfloor n/2 \rfloor$ елементами.

```
MergeSort(A, p, r)
1  if p<r
2     then q ← ⌊(p+r)/2⌋
3          MergeSort(A, p, q)
4          MergeSort(A, q+1, r)
5          Merge(A, p, q, r)
```

Лістинг 3.2 Процедура сортування `MergeSort`

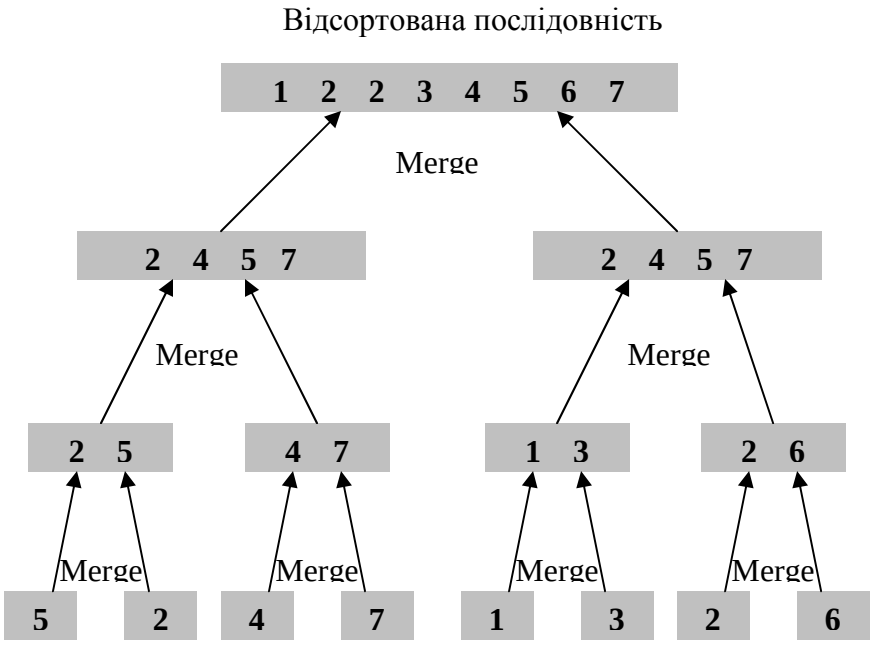

Рис 3.2. Процес сортування масиву $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$

Щоб відсортувати послідовність $A = \langle A[1], A[2], \ldots, A[n] \rangle$, викликається процедура `MergeSort`(A, 1, *length*[A]), де *length*[A] = n. На рис. 3.2 наводиться приклад роботи цієї процедури, якщо n – ступінь двійки. В ході роботи відбувається попарне об'єднання одноелементних послідовностей у відсортовані послідовності довжини 2, потім – попарне об'єднання двоелементних послідовностей у відсортовані послідовності довжини 4 і т.д., допоки не будуть отримані дві послідовності довжиною $n/2$, які об'єднаються у кінцеву відсортовану послідовність довжиною n.

3.2. Аналіз алгоритму сортування злиттям

Якщо алгоритм рекурсивно звертається сам до себе, час його роботи часто описується за допомогою **рекурентного рівняння**, в якому повний час, потрібний для розв'язку всієї задачі з об'ємом входу n, виражається через час розв'язку допоміжних підзадач. Потім дане рекурентне рівняння розв'язується за допомогою певних математичних методів (див. тему 4) і встановлюються межі ефективності алгоритму.

Отримання рекурентного співвідношення для часу роботи алгоритму, який заснований на принципі «розділяй та володарюй», базується на трьох етапах, які відповідають цій парадигмі. Позначимо через $T(n)$ час розв'язку задачі, розмір якої дорівнює n. Якщо розмір задачі достатньо малий, скажімо, $n \le c$, де c – деяка заздалегідь відома стала, то задача розв'язується безпосередньо за фіксований час, який позначається через $\Theta(1)$. Припустимо, що наша задача ділиться на a підзадач, об'єм кожної з яких дорівнює $1/b$ від об'єму висхідної задачі. Для алгоритму сортування методом злиття a та b дорівнюють 2, хоча в загальному випадку зовсім не обов'язково, щоб $a=b$. Якщо розбиття задачі на допоміжні підзадачі відбувається протягом часу $D(n)$, а об'єднання розв'язків підзадач до розв'язку початкової задачі – протягом часу $C(n)$, то ми отримуємо наступне рекурентне рівняння:

$$T(n) = \begin{cases} \Theta(1) & \text{якщо } n \le c, \\ aT(n/b) + D(n) + C(n) & \text{в протилежному випадку.} \end{cases}$$

Для спрощення аналізу швидкодії алгоритму `MergeSort` припустимо, що n дорівнює степеню двійки. Проте даний алгоритм працює однаково ефективно для загального випадку.

Сортування одного елементу методом злиття триває фіксований проміжок часу. Якщо $n>1$, час роботи алгоритму розподіляється наступним чином.

- *Розділення*. Під час розділення визначається, де знаходиться середина підмасиву. Ця операція триває фіксований час, тому $D(n) = \Theta(1)$.
- *Рекурсивний розв'язок*. Рекурсивно розв'язуються дві підзадачі, об'єм кожної з яких складає $n/2$. Час розв'язку цих підзадач становить $2T(n/2)$.
- *Комбінування*. Процедура `Merge` для n-елементного масиву виконується протягом часу $\Theta(n)$, тому $C(n) = \Theta(n)$.

Виходячи з того, що $D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$, отримуємо вираз рекурентного рівняння для методу злиття в найгіршому випадку:

$$T(n) = \begin{cases} \Theta(1) & \text{якщо } n = 1, \\ 2T(n/2) + \Theta(n) & \text{якщо } n > 1. \end{cases} \qquad (3.1)$$

Перепишемо рівняння (3.1) у вигляді:

$$T(n) = \begin{cases} c & \text{якщо } n = 1, \\ 2T(n/2) + cn & \text{якщо } n > 1, \end{cases} \qquad (3.2)$$

де константа c позначає час, який необхідний для розв'язку задачі, розмір якої дорівнює 1, а також питомий час, який потрібний для розділення та комбінування (у випадку, якщо розміри цих проміжків часу різні, то константою c можна позначити найбільший з них).

Процес розв'язку рекурентного співвідношення (3.2) показаний на рис. 3.3. В частині *а* показаний час $T(n)$, який представлений у частині *б* у вигляді еквівалентного дерева, що отримується з рекурентного рівняння (3.2). Коренем цього дерева є доданок cn, а два піддерева представляють дві менші рекурентні послідовності $T(n/2)$. В частині *в* показаний черговий крок рекурсії. Далі продовжується розкладення кожного вузла, який входить у дерево шляхом розбиття на складові частини, виходячи з рекурентного співвідношення. Так відбувається до тих пір, допоки розмір послідовності не буде дорівнювати 1, а час її виконання – константі c. Отримане дерево наводиться в частин *г*. Дерево складається з $\lg n + 1$ рівнів (адже його висота дорівнює $\lg n$), а кожний рівень дає

сумарний вклад в повний час роботи алгоритму, який дорівнює cn. В цьому легко переконатись: час роботи в корені дорівнює cn, час роботи на другому рівні – $c(n/2) + c(n/2) = cn$. В загальному випадку рівень i має 2^i вузлів, кожний з яких виконується протягом часу $c(n/2^i)$, тому повний час для всіх вузлів рівня i становить $2^i c(n/2^i) = cn$. На нижньому рівні є n вузлів, кожний з яких дає вклад c, що в сумі становить знову cn.

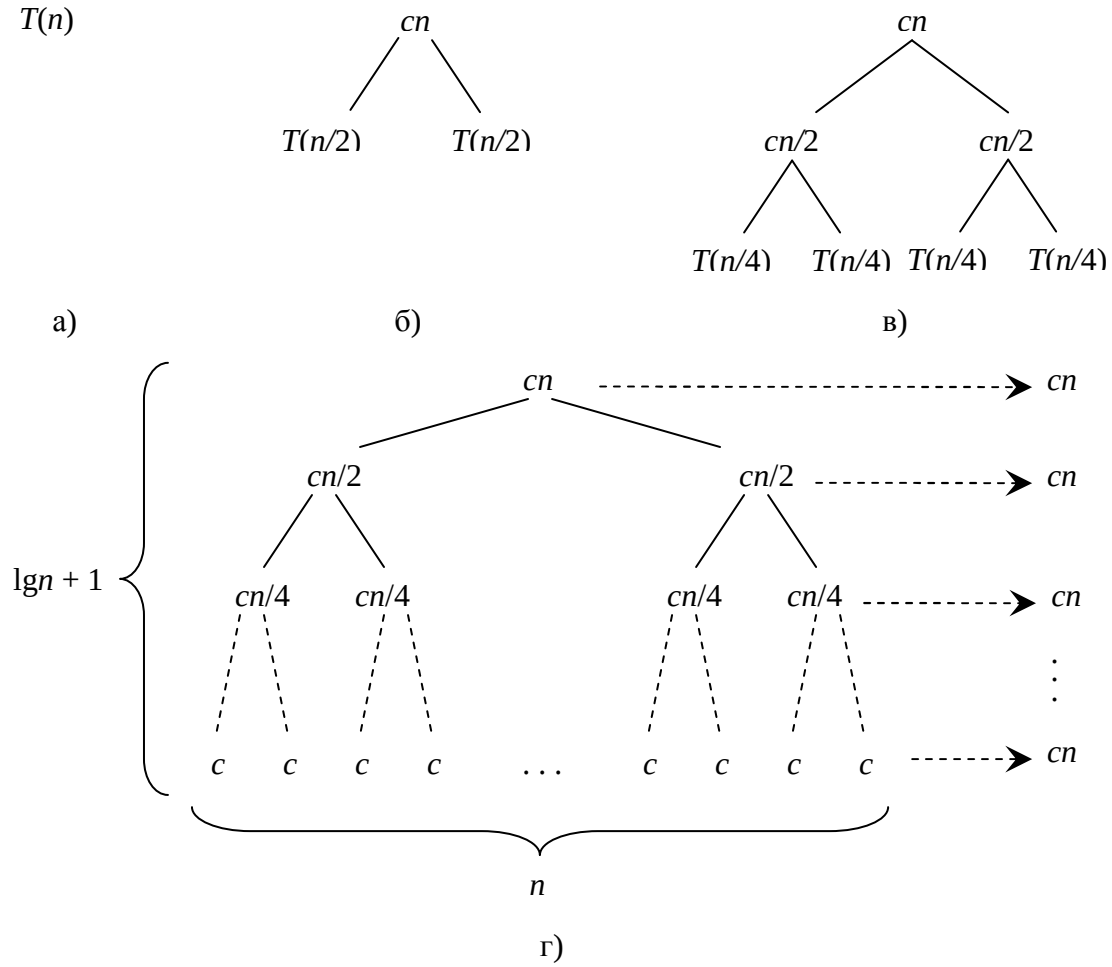

Рис. 3.3. Побудова дерева рекурсіє для рівняння $T(n) = 2T(n/2) + cn$

Щоб знайти повний час роботи алгоритму, потрібно скласти час кожного рівня. Маючи $\lg n + 1$ рівнів, кожний з яких виконується протягом часу cn, отримуємо загальний час $cn(\lg n + 1) = cn\lg n + cn$. Нехтуючи членами менших порядків та константою c, в результаті отримуємо $\Theta(n\lg n)$ – час роботи процедури `MergeSort`.

3.3. Підрахунок інверсій

Розглянемо наступну задачу. Припустимо, що $A[1\ldots n]$ – це масив, який складається з n різних чисел. Якщо $i < j$ та $A[i] > A[j]$, то пара (i, j) називається інверсією в масиві A. Задача полягає в тому, щоб знайти кількість всіх інверсій в заданому масиві A.

Наприклад, нехай заданий масив $A = \langle 2, 3, 8, 6, 1\rangle$. Тоді пара індексів $(1, 5)$ буде інверсією, адже $A[1] = 2$ та $A[5] = 1$ і $2>1$. Загалом масив A містить наступні інверсії: $(1, 5)$, $(2, 5)$, $(3, 4)$, $(3, 5)$, $(4, 5)$ і їх кількість – 5. Якщо масив відсортований (наприклад, $A = \langle 1, 2, 3, 6, 8\rangle$), то він не містить жодної інверсії. У випадку, коли масив відсортований у зворотному порядку, то кількість інверсій максимальна і дорівнює $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ (так для масиву $A = \langle 8, 6, 3, 2, 1\rangle$ кількість інверсій становитиме 10).

Одним з прикладів застосування інверсій є обрахунок того, наскільки два пріоритетні списки є подібними один до одного. Наприклад, якщо дві людини – Аліса та Богдан – вирішили дізнатись наскільки подібними є їх уподобання щодо художніх кінофільмів. Кожен з них отримує список, скажімо, з десяти фільмів, які потрібно відсортувати за власним смаком (фільм, який найбільше подобається, розташовується на першій позиції; який найменше подобається – на останній).

Для того, щоб дізнатись наскільки близькі уподобання Аліси та Богдана за допомогою інверсій, створюється новий масив пріоритетів, який заповнюється наступним чином. Наприклад, Богдан поставив на перше місце у власному списку фільм «Володар перснів», тоді як Аліса розташувала цей фільм на п'яту позицію у своєму переліку уподобань. В цьому випадку в новому масиві в першій позиції буде вписано число 5. Далі, якщо другий фільм Богдана – «Зоряні війни» – у Аліси займає третю сходинку, то другий елемент масиву пріоритетів буде містити число 3. Після заповнення таким чином всього масиву пріоритетів, лишається підрахувати кількість інверсій в ньому, яка й буде визначати ступінь подібності смаків Аліси та Богдана.

Цей підхід може використовуватись у веб сайтах для надання рекомендацій новим користувачам на основі вибору користувачів, які вже користувались даним ресурсом. Для цього потрібно знайти користувачів з найменшою кількістю інверсій між собою, що буде автоматично означати подібність смаків користувачів.

Очевидний алгоритм обрахунку кількості інверсій, який працює за принципом перебору всіх можливих пар елементів заданого масиву, буде працювати $\Theta(n^2)$ часу, де n – кількість елементів в масиві. Дійсно, ми можемо перебрати всі можливі пари елементів, яких є якраз n^2, і для кожної пари за сталий час $\Theta(1)$ визначити чи є вона інверсією, чи ні.

Чи можемо ми покращити цей результат з допомогою підходу «розділяй та володарюй»? В рамках цього підходу, за аналогією з алгоритмом методу злиття, масив можна розбивати на дві частини. Тоді всі інверсії (i, j), де $i<j$, будуть підпадати під одну з трьох категорій:

1) *ліві інверсії*: якщо $i, j \le n/2$,
2) *праві інверсії*: якщо $i, j > n/2$,
3) *розділені інверсії*: якщо $i \le n/2 < j$.

Так, у масиві $A = \langle 2, 3, 8, 6, 1\rangle$ буде жодної лівої інверсії – підмасив $A_L = \langle 2, 3, 8\rangle$ немає інверсій (зверніть увагу на розбиття масиву у випадку непарної кількості елементів), буде одна права інверсія – підмасив $A_R = \langle 6, 1\rangle$, та 4 розділені інверсії.

Тоді на етапі рекурсивного розв'язку методу «розділяй та володарюй» ми успішно повинні обрахувати кількість лівих та правих інверсій. Розділені інверсії повинні обраховуватись на етапі комбінування.

Наведемо псевдокод алгоритму обрахунку інверсій. Легко бачити, що основна робота по підрахунку інверсій буде відбуватись саме в процедурі `CountSplitInv`, яка наразі лишається нереалізованою.

```
CountInv(A)
1  n ← length(A)
2  if n=1
3     then return 0
4     else x ← CountInv(перша половина A)
5          y ← CountInv(друга половина A)
6          z ← CountSplitInv(A)
7          return x+y+z
```

Лістинг 3.3 Процедура підрахунку інверсій `CountInv`

Розглянемо тепер випадок, коли перший та другий підмасиви A відсортовані всередині, і подивимось ще раз, що відбувається при використанні процедури `Merge` методу сортування злиттям для об'єднання цих двох підмасивів. Нехай, є масив

$A = \langle 1, 3, 5, 2, 4, 6 \rangle$, в якому ліва частина $L = \langle 1, 3, 5 \rangle$ та права частина $A = \langle 2, 4, 6 \rangle$ вже відсортовані (рис. 3.4, *а*). Випадки, коли вставляється елемент лівого масиву в масив A (рядки 14-15 лістингу 3.1), відповідають тій ситуації, при якій поточний елемент для вставки $L[i]$ є меншим за $R[j]$ та у початковому масиві A знаходився лівіше за будь-який елемент підмасиву R (випадки *б*, *г* та *е* рис. 3.4). В таких випадках жодних розділених інверсій з елементом $L[i]$ не може існувати.

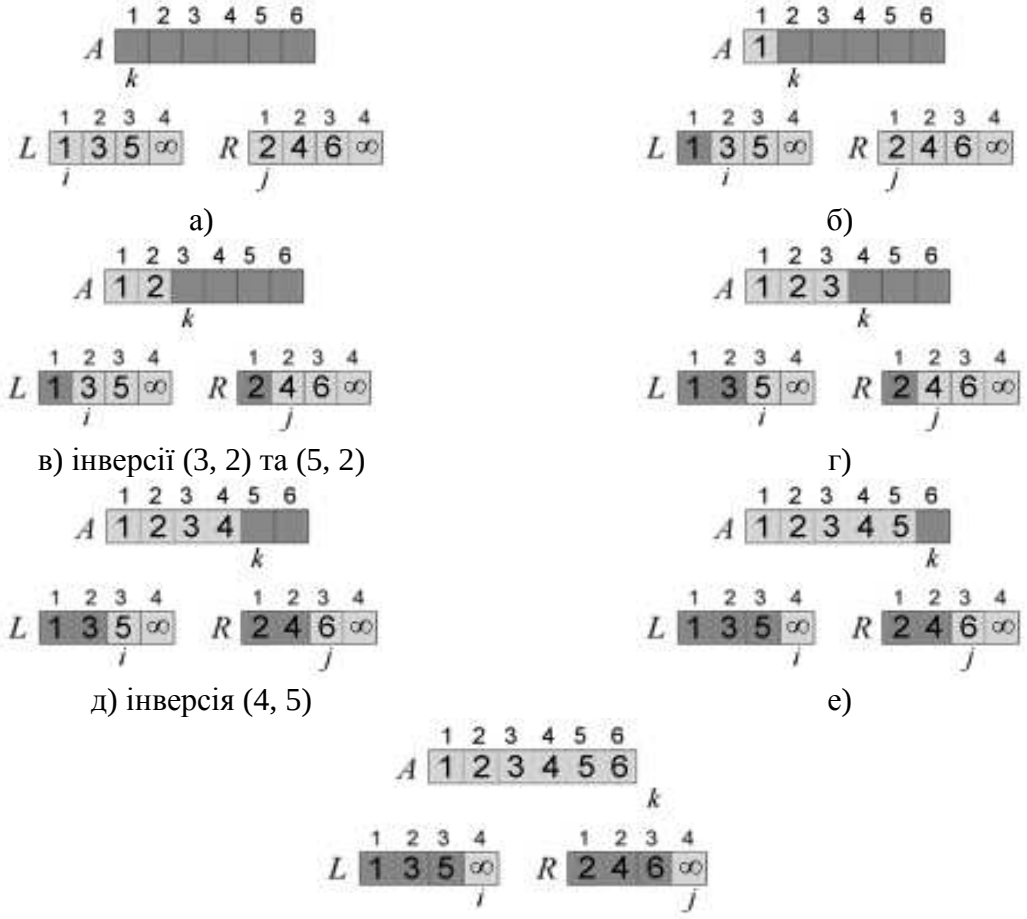

Рис. 3.4. Підрахунок інверсій під час роботи процедури `Merge` для масиву $A = \langle 1, 3, 5, 2, 4, 6 \rangle$

Коли ж вставляється елемент правого підмасиву R у масив A (рядки 16-17 лістингу 3.1), то це означає, що найменший не вставлений наразі елемент $R[j]$ є також меншим за деякі елементи, що знаходяться у лівому підмасиві L, та ще не були вставлені у масив A. Всі ці невставлені елементи з L будуть більшими $R[j]$ і, отже, будуть створювати з ним інверсії. Кількість таких інверсій буде дорівнювати кількості ще не доданих до A елементів з підмасиву L. Наприклад, коли вставляється елемент $R[1] = 2$ (рис. 3.4, *в*), то в лівому підмасиві ще є два елементи 3 та 5, які поки не були вставлені в A. Це означає, що в початковому масиві є інверсії, які відповідають парам елементів (3, 2) та (5, 2). Коли вставляється елемент $R[2] = 4$ (рис. 3.4, *д*), то в L ще лишається елемент 5, який створює з $R[2]$ одну інверсію.

Наведені вище міркування приводять нас до наступної модифікації процедури `CountInv`, яка використовує сортування злиттям всередині себе. Цю модифікацію ми позначимо як `SortAndCountInv`.

```
SortAndCountInv(A)
1  n ← length(A)
2  if n=1
3     then return (A, 0)
4     else (L, x) ← SortAndCountInv(перша половина A)
5          (R, y) ← SortAndCountInv(друга половина A)
6          (A, z) ← MergeAndCountSplitInv(A, L, R)
7          return (A, x+y+z)
```

Лістинг 3.4 Процедура підрахунку інверсій `SortAndCountInv`

Процедура `SortAndCountInv` приймає на вхід масив *A*, в якому необхідно обрахувати кількість інверсій. На виході процедури буде відсортований масив *A* та знайдена кількість інверсій в ньому. Рядок 3 процедури `SortAndCountInv` відповідає базовому випадку, коли масив *A* містить тільки один елемент. В цьому випадку масив вже є відсортованим та не містить інверсій. Якщо масив містить більше ніж один елемент, то тоді виконуються рядки 4-7. В рядку 4 рекурсивно викликається процедура `SortAndCountInv` для лівої половини масиву *A*. Результатом цього виклику буде цей відсортований підмасив, а також кількість лівих інверсій у *A*. Аналогічно результатом виконання рядку 6 буде відсортована права частина масиву *A* разом із кількістю правих інверсій в *A*. Тепер лишається об'єднати лівий *L* та правий *R* підмасиви *A* і підрахувати кількість розділених інверсій в *A*. Це робиться у процедурі `MergeAndCountSplitInv`, якій передаються відсортовані підмасиви *L* та *R*.

```
MergeAndCountSplitInv(A, L, R)
1  n1 ← length(L)
2  n2 ← length(R)
3  L[n1+1] ← ∞
4  R[n2+1] ← ∞
5  i ← 1
6  j ← 1
7  c ← 0    // лічильник розділених інверсій
8  for k ← p to r
9     do if L[i] ≤ R[j]
10          then A[k] ← L[i]
11               i ← i + 1
12          else A[k] ← R[j]
13               j ← j + 1
14               c ← c + (n2 - i + 1)
15 return (A, c)
```

Лістинг 3.5 Процедура злиття та підрахунку розділених інверсій `MergeAndCountSplitInv`

Процедура `MergeAndCountSplitInv` має схожий на процедуру `Merge` (лістинг 3.1) вигляд, хоч й приймає на вхід інший набір параметрів. У процедуру додана змінна *c*, яка на виході буде містити кількість розділених інверсій в масиві *A*. Лічильник *c* збільшується на кількість неопрацьованих елементів в масиві *L* у рядку 14 процедури.

Час роботи процедури `MergeAndCountSplitInv` становить $\Theta(n)$, адже вона повністю подібна до процедури `Merge` і в ній просто додається виконання однієї елементарної операції додавання. За аналогією з `MergeSort` процедура `SortAndCountInv` має час роботи $\Theta(n\lg n)$, що краще за алгоритм простого перебору всіх пар елементів з його часом роботи $\Theta(n^2)$.

3.4. Добуток матриць

Добутоком двох матриць X та Y розмірності $n \times n$ є матриця $Z = XY$, яка також має розмірність $n \times n$, кожен елемент (i, j) якої обчислюється за формулою:

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj} .$$

Час, потрібний на обрахунок добутку двох матриць за даною формулою, дорівнює $\Theta(n^3)$: необхідно обрахувати n^2 елементів матриці Z, на обрахунок кожного з яких витрачається час $\Theta(n)$. Тривалий час цей підхід вважався оптимальним для розв'язання задач добутку матриць, допоки в 1969 році німецький математик Штрассен (Strassen) не запропонував метод, який базувався на парадигмі «розділяй та володарюй» (аналогічно до методу Карацуби, див. тему 1 даного курсу).

Добуток матриць достатньо легко розбити на підзадачі, адже його можна виконувати по блоках в матрицях. Для цього необхідно розбити матриці X та Y на чотири блоки розмірності $n/2 \times n/2$ кожний:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Тепер добуток матриць XY можна представити в термінах цих блоків так само, якби вони були просто окремими елементами матриць:

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Отже, ми отримали стратегію методу «розділяй та володарюй»: щоб обрахувати добуток двох матриць X та Y розмірності $n \times n$, необхідно рекурсивно обрахувати 8 добутків AE, BG, AF, BH, CE, DG, CF, DH розмірності $n/2 \times n/2$, а після цього зробити ще $\Theta(n^2)$ додаткових додавань. Загальний час обрахунку за цим підходом можна виразити через рекурентне рівняння:

$$T(n) = 8T(n/2) + \Theta(n^2).$$

Його розв'язок за допомогою основного методу (див. тему 4) дає відповідь: $T(n) = \Theta(n^3)$, що не дає покращення у порівнянні зі стандартним методом добутку матриць. Проте, за рахунок певних алгебраїчних тонкощів можна отримати кращий час роботи алгоритму, аналогічно до методу Карацуби добутку двох цілих чисел. Штрассен показав, що достатньо лише сім добутків матриць розмірності $n/2 \times n/2$, щоб обрахувати початковий добуток XY.

Якщо позначити через P_i, i=1,…,7 наступні вирази:

$$P_1 = A(F - H), \qquad P_5 = (A + D)(E + H),$$
$$P_2 = (A + B)H, \qquad P_6 = (B - D)(G + H),$$
$$P_3 = (C + D)E, \qquad P_7 = (A - C)(E + F),$$
$$P_4 = D(G - E),$$

то добуток XY можна виразити як:

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}.$$

Наведемо перевірку для $P_5 + P_4 - P_2 + P_6$:

$$P_5 + P_4 - P_2 + P_6 = (A + D)(E + H) + D(G - E) - (A + B)H + (B - D)(G + H) =$$
$$= AE + AH + DE + DH + DG - DE - AH - BH + BG + BH - DG - DH = AE + BG .$$

Час роботи даного методу можна визначити за рекурентним рівнянням:

$$T(n) = 7T(n/2) + \Theta(n^2),$$

який за основним методом (див. тему 4) буде дорівнювати $O(n^{\log_2 7}) \approx O(n^{2,81})$.

Слід відзначити, що метод Штрассена був найшвидшим методом обрахунку матриць до 1987 року. Того року був опублікований метод Коперсміта-Винограда (Coppersmith–Winograd), складність якого оцінюється як $O(n^{2,375477})$. Пізніше цей результат був покращений шляхом певних модифікацій до $O(n^{2,3727})$. Проте, на противагу методу Штрассена, даний метод майже не використовується для практичних задач, адже за рахунок прихованих сталих множників він дає виграш по швидкості роботи тільки для матриць дуже великої розмірності.

Література

Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001) [1990]. Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03293-7. Глава 2, розділ 2.3.

Dasgupta, Sanjoy; Papadimitriou, Christos; Vazirani, Umesh (2006) Algorithms. McGraw-Hill Science/Engineering/Math. ISBN-13 978-0073523408. Глава 2, розділ 2.5.