ЛЕКЦІЯ 8. ШВИДКЕ СОРТУВАННЯ

Швидке сортування (*quick sort*) – це алгоритм сортування, час роботи якого для вхідного масиву з n чисел в найгіршому випадку дорівнює $\Theta(n^2)$. Не дивлячись на таку повільну роботу в найгіршому випадку, цей алгоритм на практиці часто виявляється оптимальним завдяки тому, що в середньому час його роботи набагато кращий: $\Theta(n\lg n)$. Окрім того, сталі множники, які не враховуються у виразі $\Theta(n\lg n)$, достатньо малі за величиною (наприклад, у порівнянні з алгоритмом сортування методом злиття). Алгоритм також має суттєву перевагу в тому, що працює без використання додаткової пам'яті (на противагу, наприклад, тому самому алгоритму сортування методом злиття).

8.1. Опис швидкого сортування

Швидке сортування, аналогічно до сортування злиттям, засноване на парадигмі „*розділяй та володарюй*". Нижче описаний процес сортування підмасиву $A[p\ldots r]$, який складається з трьох етапів, як і всі алгоритми цієї парадигми.

- *Розділення*: Масив $A[p\ldots r]$ розбивається на два (можливо порожніх) підмасиви $A[p\ldots q–1]$ та $A[q+1\ldots r]$ шляхом переупорядкування його елементів. Кожний елемент масиву $A[p\ldots q–1]$ не є більшим за елемент $A[q]$, а кожний елемент підмасиву $A[q+1\ldots r]$ є більшим елементу $A[q]$. Індекс q обраховується під час процедури розбиття.
- *Рекурсивний розв'язок*: Підмасиви $A[p\ldots q–1]$ та $A[q+1\ldots r]$ відсортовуються шляхом рекурсивного виклику процедури швидкого сортування.
- *Комбінування*: Оскільки підмасиви відсортовуються на місці без застосування додаткової пам'яті, для їх об'єднання не потрібні жодні додаткові дії: увесь масив $A[p\ldots r]$ виявляється відсортованим.

Алгоритм швидкого сортування представлений наступною процедурою.

```
QuickSort(A, p, r)
1    if p<r
2         then q = Partition(A, p, r)
3              QuickSort(A, p, q-1)
4              QuickSort(A, q+1, r)
```

Лістинг 8.1. Процедура сортування `QuickSort`.

Щоб виконати сортування всього масиву *A*, виклик процедури повинен мати вигляд `QuickSort(A, 1, length[A])`.

Ключовою частиною алгоритму швидкого сортування є процедура `Partition`, яка змінює порядок елементів підмасиву $A[p \ldots r]$ без використання додаткової пам'яті.

```
Partition(A, p, r)
1    x = A[r]
2    i = p-1
3    for j = p to r-1
4         do if A[j] = x
5              then i = i+1
6                   Обміняти A[i] = A[j]
7    Обміняти A[i+1] = A[r]
8    return i+1
```

Лістинг 8.2. Процедура розбиття `Partition`.

По суті, процедура `Partition` виконує те, що описано вище в пункті „*Розбиття*" парадигми „*розділяй та володарюй*". На початку процедури обирається *опорний* (англ. *pivot*) *елемент*. Таким елементом буде останній елемент масиву $A[p \ldots r]$ – елемент $x = A[r]$. Після цього всі елементи масиву $A[p \ldots r-1]$ розбиваються на такі, що не більше опорного – вони будуть розташовані згодом ліворуч від нього, та такі, які не менше нього і будуть розташовані праворуч. В кінці елемент $A[r]$ переміщується в позицію, яка відповідає цьому розділенню: всі елементи ліворуч не більше нього та всі елементи праворуч – не менше.

В процесі роботи масив $A[p \ldots r]$ складається з чотирьох частин (деякі з них можуть бути порожніми) (рис. 8.1), які утворюють інваріант циклу:

1. якщо $p \le k \le i$, то $A[k] \le x$;
2. якщо $i+1 \le k \le j-1$, то $A[k] > x$;
3. якщо $k = r$, то $A[k] = x$;
4. якщо $j \le k \le r-1$, то елементи можуть мати довільні значення.

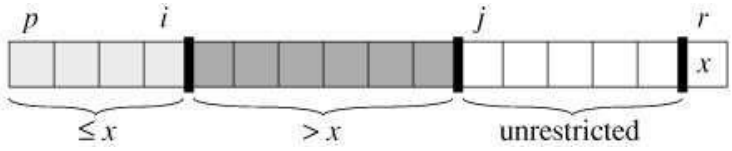

Рис. 8.1. Області масиву $A[p \ldots r]$, які підтримуються процедурою `Partition`.

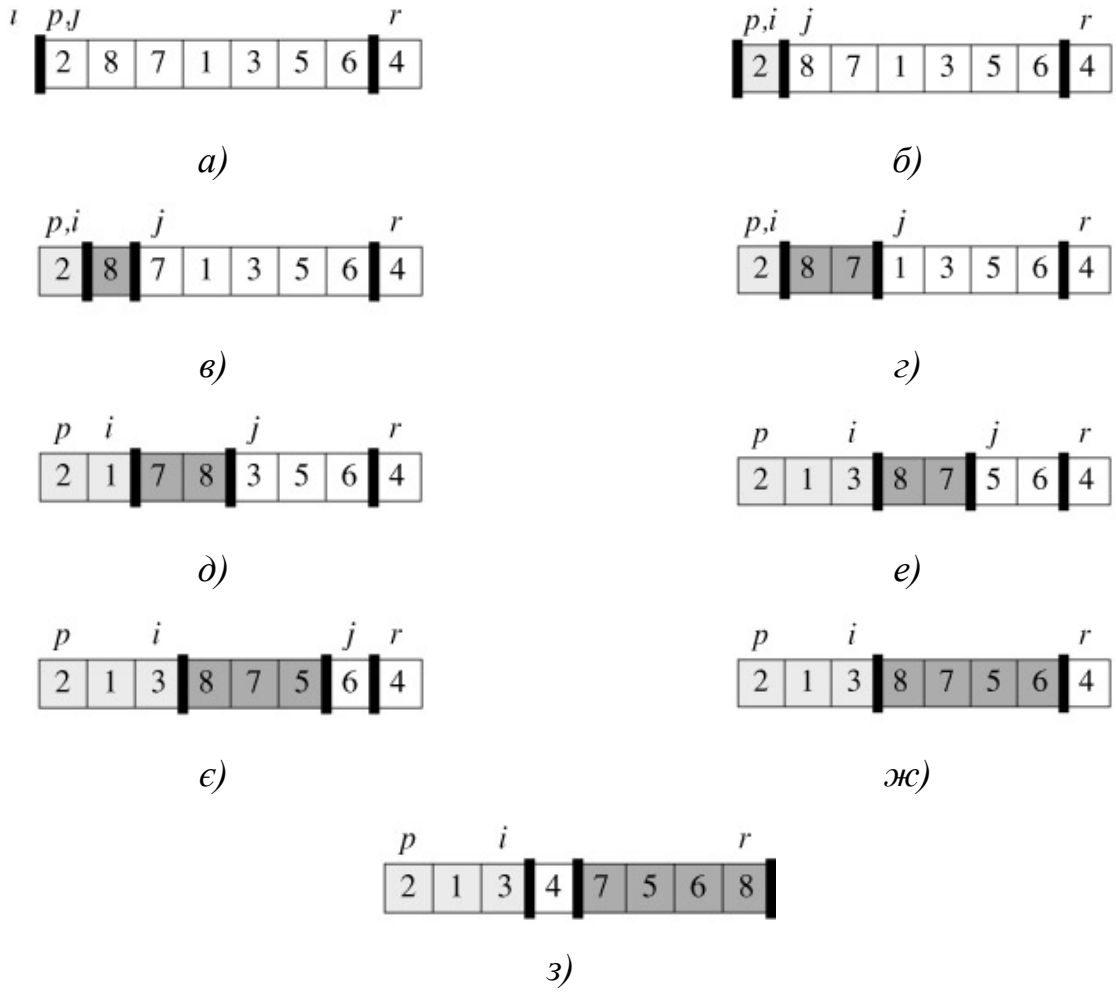

Рис. 8.2. Приклад роботи процедури `Partition`.

На рис 8.2. наведений приклад роботи процедури `Partition` для 8-елементного масиву. Світло-сірим кольором позначені елементи масиву, які потрапили в першу частину; значення всіх цих елементів не більше величини x (в прикладі $x = 4$). Елементи темно-сірого кольору утворюють другу частину; їх величина вже більша за x. Незафарбовані елементи – це такі, що наразі не

потрапили в жодну з двох частин; останній незафарбований елемент є опорним. В частині *а* рисунку показаний початковий стан масиву і значення змінних. Жоден з елементів не поміщений в жодну з перших двох частин. В частині *б* елемент 2 сам утворює ту частину, яка менша *x*. В частинах *в* та *г* елементи 8 та 7 розміщуються в другій частині. В частині *д* елементи 1 та 8 помінялись місцями, в результаті чого кількість елементів в першій частині збільшилась. В частині *е* міняються місцями елементи 3 та 7, і знову кількість елементів в першій частині збільшується. В частинах *є* та *ж* елементи 5 та 6 додаються до другої частини, після чого цикл завершується. В частині *з* демонструється дія рядку 7 процедури `Partition`, коли опорний елемент *x* переставляється на нове місце – між двома першими частинами.

Важливо зазначити, що після виконання процедури `Partition` для масиву $A[p\ldots r]$ обраний опорний елемент буде розташовуватись саме там, де йому потрібно знаходитись у відсортованому масиві. Тому в подальшому сортуванні він вже не приймає участі.

Час роботи процедури `Partition` для масиву $A[p\ldots r]$ складає $\Theta(n)$, де $n = r - p + 1$.

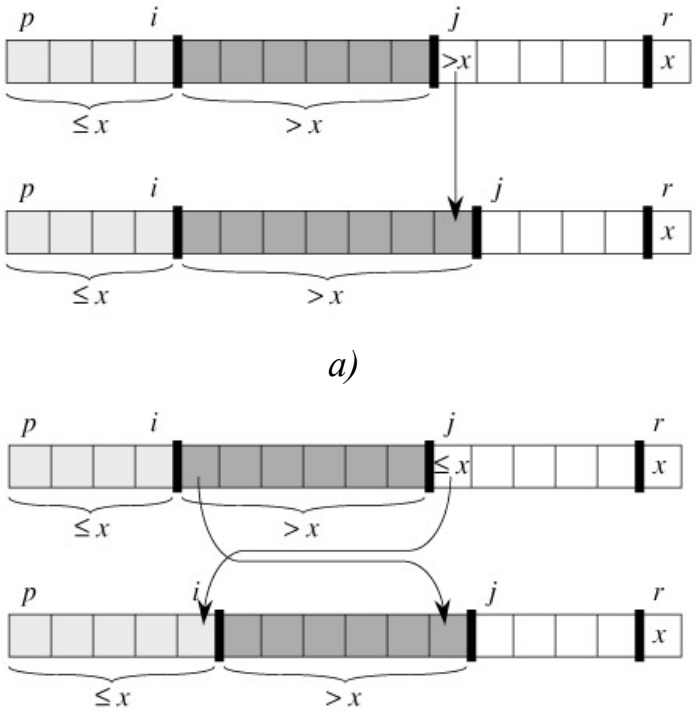

Рис. 8.3. Два варіанти ітерації процедури `Partition`.

Тепер покажемо, що вказаний вище інваріант циклу справедливий перед початком циклу, перед кожною операцією та вкінці циклу.

1. *Ініціалізація*. Перед першою ітерацією циклу **for** $i = p - 1$ та $j = p$. Між елементами з індексами p та i немає жодних елементів, як їх немає між елементами з індексами $i + 1$ та $j - 1$, тому перші дві умови інваріанту виконуються. Рядок 1 процедури `Partition` призводить до виконання і третьої умови (четверта умова виконується за визначенням).
2. *Збереження*. Як видно з рис. 8.3, необхідно розглянути два випадки, вибір кожного з яких визначається перевіркою у рядку 4 процедури. На рис. 8.3, *а* показано, що відбувається, якщо $A[j] > x$; єдина дія, яка виконується в цьому випадку в циклі – збільшення на одиницю значення j. При збільшенні значення j для елементу $A[j - 1]$ виконується умова 2, а всі інші елементи лишаються незмінними. На рис. 8.3, *б* показано, що відбувається, якщо $A[j] \le x$; в цьому випадку збільшується значення i, елементи $A[i]$ та $A[j]$ міняються місцями, після чого на одиницю збільшується значення i. В результаті перестановки $A[i] \le x$, і умова 1 виконується. Аналогічно отримуємо $A[j - 1] > x$, оскільки елемент, який був переставлений з елементом $A[j - 1]$, відповідно до інваріанту циклу, більше за x.
3. *Завершення*. По завершенню роботи алгоритму $j = r$. Тому кожний елемент масиву є членом однієї з трьох множин (четверта множина стає порожньою).

Таким чином, всі елементи масиву розбиті на три підмножини: величина яких не більше x, більша за x, та одноелементна множина, яка складається з x.

8.2. Ефективність швидкого сортування

Час роботи алгоритму швидкого сортування залежить від степеня збалансування, яким характеризується розбиття. Збалансування, в свою чергу, залежить від того, який елемент був обраний в якості опорного. Якщо розбиття збалансоване, асимптотично алгоритм працює так само швидко, як сортування злиттям. В протилежному випадку асимптотична поведінка цього алгоритму така сама повільна, як й в алгоритму сортування включенням.

Найгірша поведінка алгоритму швидкого сортування має місце в тому випадку, коли процедура розбиття породжує одну підзадачу з $n-1$ елементами, а другу – з 0 елементами. Припустимо, що таке незбалансоване розбиття виникає при кожному рекурсивному виклику. Для виконання розбиття необхідний час $\Theta(n)$. Оскільки рекурсивний виклик процедури розбиття, на вхід якої подається масив розміру 0, нічого не робить, то $T(0) = \Theta(1)$. Отже, рекурентне співвідношення, яке описує час роботи цієї процедури, записується наступним чином:

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n).$$

Інтуїтивно зрозуміло, що при підсумовуванні проміжків часу, який витрачається на кожний рівень рекурсії, отримується арифметична прогресія, що дає в результаті оцінку $\Theta(n^2)$. Це легко показати за допомогою методу підстановки для наведеного вище рекурентного співвідношення.

Таким чином, якщо на кожному рівні рекурсії алгоритму розбиття максимально незбалансоване, то час роботи алгоритму дорівнює $\Theta(n^2)$. Відповідно, ефективність такого методу не буде кращою за ефективність сортування методом включення. Більше того, за такий саме час алгоритм швидкого сортування опрацьовує масив, який вже повністю відсортований, – ситуація, яка зустрічається часто і в якій час роботи алгоритму сортування включенням дорівнює $\Theta(n)$.

У найбільш сприятливому випадку процедура `Partition` розбиває задачу розміром n на дві підзадачі, розмір кожної з яких не перевищує $n/2$. В такому випадку швидке сортування працює набагато більш ефективно, і час її роботи описується наступним рекурентним співвідношенням:

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

Це рекурентне співвідношення підпадає під випадок 2 *основного методу* (власне, воно ідентичне до рекурентного співвідношення для сортування методом злиття), так що його розв'язок – $T(n) = \Theta(n \lg n)$. Отже, розбиття на рівні частини призводить до асимптотично більш швидкого алгоритму.

Тепер розглянемо випадок, коли розбиття далеке від збалансованого (але не

таке погане, як у наведеному вище найгіршому випадку). Припустимо, наприклад, що розбиття проводиться у співвідношенні *один до дев'яти*. В цьому випадку для часу роботи алгоритму швидкого сортування отримується наступне рекурентне співвідношення:

$$T(n) = T(9n/10) + T(n/10) + \Theta(n) \text{ або } T(n) \le T(9n/10) + T(n/10) + cn.$$

На рис. 8.4 показане рекурсивне дерево, яке відповідає цьому рекурентному співвідношенню. У вузлах дерева наведені розміри відповідних підзадач, а справа від кожного рівня – час його роботи. Час роботи рівнів явним чином містить сталу *c*. Зверніть увагу, що час роботи кожного рівня цього дерева становить *cn*. Так відбувається до тих пір, доки не буде досягнути глибина $\log_{10} n = \Theta(\lg n)$. Час роботи більш глибоких рівнів не перевищує величини *cn*. Рекурсія припиняється на глибині $\log_{9/10} n = \Theta(\lg n)$; таким чином, повний час роботи алгоритму швидкого сортування дорівнює O(*n*lg*n*). Отже, хоч наведене розбиття є достатньо розбалансованим, асимптотично алгоритм поводить себе так само, як при розбитті задачі на дві підзадачі однакової розмірності.

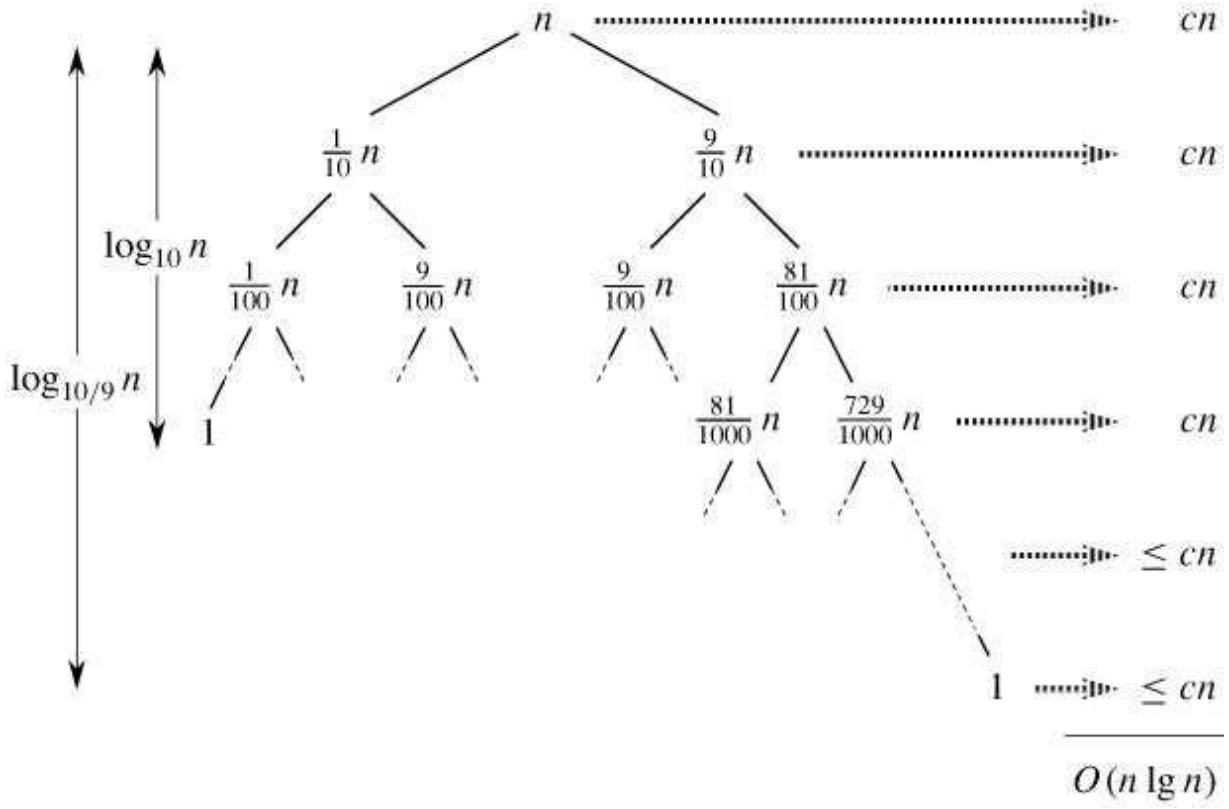

Рис. 8.4. Дерево рекурсії, яке відповідає розділенню задачі у співвідношенні 1:9.

Фактично, навіть розбиття у співвідношенні *один до дев'яносто дев'яти* приводить до того, що час роботи цього алгоритму буде $O(n\lg n)$. Причина в тому, що будь-яке розбиття, яке характеризується скінченною сталою пропорційності, приводить до утворення рекурсивного дерева висотою $\Theta(\lg n)$ і часом роботи кожного рівня $O(n)$. Таким чином, при будь-якій сталій величині пропорції повний час роботи становитиме $O(n\lg n)$.

8.3. Випадкова версія швидкого сортування

Як з'ясувалось вище, алгоритм швидкого сортування найкраще працює при збалансованому розбитті вхідного масиву на підмасиви. Для досягнути цього на практиці часто використовується підхід, коли в якості опорного елементу обирається не останній елемент вхідного масиву $A[p\ldots r]$, а випадково обраний елемент з цього масиву. Це дозволяє асимптотично наблизитись до збалансованого розбиття.

Зміни, які необхідно внести в процедури `QuickSort`та `Partition`, незначні. У новій версії процедури `Partition` безпосередньо перед розбиттям достатньо реалізувати випадкову перестановку:

```
RandomizedPartition(A, p, r)
1     i = Random(p, r)
2     Обміняти A[r] ↔ A[i]
3     return Partition(A, p, r)
```

Лістинг 8.3. Процедура випадкового розбиття `RandomizedPartition`.

У новій процедурі швидкого сортування замість процедури `Partition` викликається процедура `RandomizedPartition`.

```
RandomizedQuickSort(A, p, r)
1     if p < r
2          then q = RandomizedPartition(A, p, r)
3               RandomizedQuickSort(A, p, q-1)
4               RandomizedQuickSort(A, q+1, r)
```

Лістинг 8.4. Процедура випадкового швидкого сортування `RandomizedQuickSort`.

8.4. Аналіз швидкого сортування

Нижче наводиться аналіз оцінки часу роботи алгоритму швидкого сортування над зафіксованим випадковим масивом A.

Час роботи процедури `QuickSort` визначається здебільшого часом роботи, який витрачається на виконання процедури `Partition`. При кожному виконанні останньої відбувається вибір опорного елементу, який потім не приймає участі в жодному рекурсивному виклику процедур `QuickSort` та `Partition`. Таким чином, протягом всього часу виконання алгоритму швидкого сортування процедура `Partition` викликається не більше n разів. Робота цієї процедури здебільшого зосереджена у циклі **for** в рядках 3–6. В кожній ітерації циклу в рядку 4 опорний елемент порівнюється з іншими елементами масиву A. Тому, якщо відомо скільки разів виконувався рядок 4, то можна оцінити повний час, який витрачається на виконання циклу **for** в процесі роботи процедури `QuickSort`.

Позначимо через X – кількість порівнянь, які виконуються в рядку 4 процедури `Partition` протягом повної обробки n-елементного масиву процедурою `QuickSort`. Як зазначалось вище, процедура `Partition` викликається n разів, і при цьому виконується певний фіксований об'єм роботи. Далі певну кількість разів виконується цикл **for**, при кожній ітерації якого виконується рядок 4. Отже, час роботи процедури `QuickSort` становитиме $O(n + X)$.

Тож, щоб оцінити час роботи алгоритму швидкого сортування, необхідно обрахувати величину X. Для цього потрібно зрозуміти, в яких випадках в алгоритмі відбувається порівняння двох елементів, а в яких – ні. Для спрощення аналізу перейменуємо елементи масиву A як $z_1, z_2, \ldots z_n$, де z_i – i-й найменший елемент масиву. Окрім того, визначимо множину $Z_{ij} = \{z_i, z_{i+1}, \ldots z_j\}$, яка містить елементи, що розташовані між елементами z_i та z_j включно.

В яких випадках в алгоритмі відбувається порівняння елементів z_i та z_j? Відмітимо, що порівняння кожної пари елементів відбувається не більше одного разу. Дійсно, всі елементи порівнюються з опорним елементом, який ніколи не використовується в двох різних викликах процедури `Partition`. Таким чином, після конкретного виклику цієї процедури, елемент, який використовується в

якості опорного, більше не буде порівнюватись з іншими.

Визначимо випадкову величину X_{ij}, яка дорівнює кількості порівнянь елементів z_i та z_j. З попередніх міркувань зрозуміло, що X_{ij} може дорівнювати або 0, або 1. Тепер повна кількість порівнянь, які виконуються протягом роботи алгоритму, можна виразити наступним чином:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} .$$

Застосувавши до обох частин цього виразу операцію обчислення математичного сподівання і використовуючи властивість лінійності математичного сподівання, отримаємо:

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P\{z_i \text{ порівнюється з } z_j\}. \quad (8.1)$$

Адже, $E[X_{ij}] = 0 \cdot P[X_{ij} = 0] + 1 \cdot P[X_{ij} = 1] = P[X_{ij} = 1]$. Таким чином, все звелось до визначення $P\{z_i \text{ порівнюється з } z_j\}$. При чому вважається, що опорні елементи обираються випадковим чином, незалежно один від одного.

Спробуємо тепер розібратись, коли два елементи не порівнюються один з одним. Розглянемо, наприклад, в якості вхідних даних масив, який складається з чисел від 1 до 10 у довільному порядку, і припустимо, що в якості першого опорного елементу обрано число 7. Тоді в результаті першого виклику процедури `Partition` всі числа розпадуться на дві множини: {1, 2, 3, 4, 5, 6} та {8, 9, 10}. При чому елемент 7 порівнюється з усіма іншими елементами. Зрозуміло, що жодне з чисел, які потрапили в першу множину (наприклад, 2), більше не буде порівнюватись з жодним елементом другої підмножини (наприклад, 9).

Оскільки передбачається, що значення всіх елементів різні, то при виборі x в якості опорного елементу далі не будуть порівнюватись жодні елементи z_i та z_j, для яких $z_i < x < z_j$. З іншого боку, якщо в якості опорного елементу обраний елемент z_i, то він буде порівнюватись з кожним елементом множини Z_{ij}, окрім самого себе. Те саме стосується елементу z_i. Таким чином, елементи z_i та z_j будуть порівнюватись тоді й тільки тоді, коли першим в ролі опорного в множині Z_{ij}

обраний один з них.

Тепер обчислимо ймовірність цієї події. Перед тим як в множині Z_{ij} буде обраний опорний елемент, вся ця множина не є розділеною, і будь-який її елемент може бути обраний в якості опорного. Оскільки всього в множині $j-i+1$ елементів, а опорні елементи обираються випадково та незалежно один від одного, то ймовірність того, що конкретний елемент буде обраний першим в якості опорного, дорівнює $1/(j-i+1)$. Таким чином, виконується наступне співвідношення:

$$P\{z_i \text{ порівнюється з } z_j\} = P\{\text{Першим опорним обрано } z_i \text{ або } z_j\} =$$
$$= P\{\text{Першим опорним обрано } z_i\} + P\{\text{Першим опорним обрано } z_j\} =$$
$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1} \quad . \qquad (8.2)$$

З рівнянь (8.1) та (8.2) отримуємо:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}.$$

Цю суму можна оцінити, скориставшись заміною змінних ($k = j - i$).

$$E[X] = \sum_{i}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = 2\sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k+1} < 2\sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k}. \qquad (8.3)$$

Внутрішня сума $\sum_{k=1}^{n} \frac{1}{k}$ є сумою гармонічного ряду $1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}$ і яку можна оцінити як $\lg n + O(1)$. Таким чином, вираз (8.3) спрощується до:

$$E[X] =< 2\sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k} = 2\sum_{i=1}^{n-1} O(\lg n) = O(n \lg n).$$

Отже, можна зробити висновок, що при використанні процедури `RandomizedPartition` математичне сподівання роботи алгоритму швидкого сортування (при різних елементах в масиві) становить $O(n\lg n)$. Використовуючи оцінку для найкращого випадку, яка наводилась у розділі 8.2, приходимо до того, що математичне сподівання часу роботи алгоритму становить $\Theta(n\lg n)$.

8.5. Порядкові статистики

Означення 8.1. Будемо називати i-ою *порядковою статистикою* множини, яка складається з n елементів, i-й елемент за порядком зростання.

Наприклад, *мінімум* такої множина – це *перша порядкова статистика* ($i = 1$), а його *максимум* – це *n-а порядкова статистика* ($i = n$). *Медіана* неформально означає середину множини. Якщо n непарне, то медіана єдина, і її індекс дорівнює $i = (n + 1)/2$; якщо ж n парне, то медіани дві, і їх індекси дорівнюють $i = n/2$ та $i = n/2 + 1$.

Тут ми будемо розглядати проблему вибору i-ої порядкової статистики в множині, яка складається з n різних елементів. Формально задачу вибору можна визначити наступним чином.

Вхід: множина A, яка складається з n (різних) чисел, і число $1 \le i \le n$.

Вихід: елемент $x \in A$, який більше за величиною рівно $i - 1$ інших елементів A.

Задачу вибору можна розв'язати за час $O(n\lg n)$. Для цього достатньо виконати сортування елементів за допомогою, наприклад, сортування злиттям, а потім просто дістати елемент вихідного масиву з індексом i. Однак, згадаємо золоте правило розробників алгоритмів: чи можемо ми зробити це краще?

Спочатку розглянемо простішу задачу – пошук мінімального елементу в n-елементній множині. Скільки потрібно виконати порівнянь, щоб знайти такий елемент? Для цієї величини легко знайти верхню межу, яка буде дорівнювати $n - 1$ порівнянням: ми по черзі перевіряємо кожний елемент множини та слідкуємо за тим, який з них є мінімальним на даний момент. Очевидно, що для пошуку максимального елементу також знадобиться не більше $n - 1$ порівнянь.

Чи є наведений підхід оптимальним? Так, оскільки можна довести, що нижня межа для задачі визначення мінімуму також дорівнює $n - 1$ порівнянь. Будь-який алгоритм, який призначений для визначення мінімального елементу множини, можна представити у вигляді турніру, в якому приймають участь всі елементи. Кожне порівняння – це двобій між двома елементами, в якому перемагає елемент з меншою величиною. Важливе спостереження полягає в тому, що кожний елемент, окрім мінімального, повинен отримати поразку хоча б в одному двобої. Таким чином, для визначення мінімуму знадобиться $n - 1$ порівняння.

В деяких практичних задачах є необхідність знайти як мінімальний, так і максимальний елементи множини. Наприклад, графічній програмі може знадобитись виконати масштабування множини координат (x, y) таким чином, щоб вони зіставились за розміром з прямокутною областю екрану або іншого пристрою виведення. Для цього спочатку потрібно визначити мінімальну та максимальну координати.

Нескладно розробити алгоритм для одночасного пошуку мінімального та максимального елементів n-елементної множини, виконуючи при цьому $\Theta(n)$ порівнянь. Достатньо просто виконати незалежний пошук мінімального та максимального елементів. Для виконання кожної підзадачі знадобиться $n-1$ порівняння, що в сумі складе $2n-2$ порівнянь.

Однак насправді для одночасного пошуку мінімуму та максимуму достатньо не більше $3\lfloor n/2 \rfloor$ порівнянь. Для цього потрібно слідкувати за тим, який з перевірених на даний момент елементів є мінімальним, а який – максимальним. Замість того, щоб окремо порівнювати кожний вхідний елемент з поточним мінімумом та максимумом, ми будемо обробляти пари елементів. Утворивши пару вхідних елементів, спочатку порівняємо їх один з одним, а потім менший елемент пари будемо порівнювати з поточним мінімумом, а більший – з поточним максимумом. Таким чином, для кожної пари елементів знадобиться по 3 порівняння.

Спосіб початкового вибору поточного мінімуму та максимуму залежить від парності елементів в множині n. Якщо n непарне, ми обираємо з множини один з елементів та вважаємо його значення одночасно мінімальним та максимальним; інші елементи опрацьовуються парами. Якщо ж n парне, то обирається два перших елементи та шляхом порівняння визначаються який з них є більшим, а який меншим. Інші елементи опрацьовуються парами, як і в попередньому випадку.

Проаналізуємо, чому дорівнює повна кількість порівнянь. Якщо n непарне, то потрібно буде виконати $3\lfloor n/2 \rfloor$ порівнянь. Якщо ж n парне, то виконується одне початкове порівняння, а потім – ще $3(n-2)/2$ порівнянь, що в сумі дає $3n/2-2$ порівнянь. Таким чином, в обох випадках загальна кількість порівнянь не перевищує $3\lfloor n/2 \rfloor$.

8.6. Вибір за лінійний час

Загальна задача вибору виявляється більш складною, ніж проста задача пошуку мінімуму. Однак, як не дивно, час розв'язку обох задач асимптотично поводить себе однаково – як $\Theta(n)$. Ми розглянемо алгоритм `RandomizedSelect`для розв'язання цієї задачі. Він заснований на процедурі `RandomizedQuickSort`. Як і в алгоритмі швидкого сортування, в алгоритмі `RandomizedSelect` використовується ідея рекурсивного розбиття вхідного масиву. Однак на противагу алгоритму швидкого сортування, в якому рекурсивно обробляються обидві частини розбиття, алгоритм `RandomizedSelect` працює лише тільки з однією частиною.

В алгоритмі `RandomizedSelect` використовується процедура `RandomizedPartition`, таким чином це випадковий алгоритм, оскільки його поведінка частково залежить від роботи генератора псевдовипадкових чисел. Наведений нижче псевдокод процедури `RandomizedSelect` повертає значення i-го елементу в масиві $A[p \ldots r]$.

```
RandomizedSelect(A, p, r, i)
1    if p = r
2         then return A[p]
3    q = RandomizedPartition(A, p, r)
4    k = q - p + 1
5    if i = k
6         then return A[q]
7    elseif i < k
8         then return RandomizedSelect(A, p, q-1, i)
9    else return RandomizedSelect(A, q+1, r, i-k)
```

Лістинг 8.5. Процедура вибору `RandomizedSelect`.

Після виконання процедури `RandomizedPartition`, яка викликається в рядку 3, масив $A[p \ldots r]$ виявляється розбитим на два (можливо, порожніх) підмасиви $A[p \ldots q–1]$ та $A[q+1 \ldots r]$. При цьому величина кожного елементу $A[p \ldots q–1]$ не перевищує $A[q]$, а величина кожного елементу $A[q+1 \ldots r]$ більше за $A[q]$. Як і в алгоритмі швидкого сортування, елемент $A[q]$ називається опорним. В рядку 3 процедури `RandomizedSelect` обраховується кількість елементів k підмасиву

$A[p\ldots q]$, тобто кількість елементів, які підпадають в нижню частину розбиття плюс один опорний елемент. Потім в рядку 5 перевіряється, чи є елемент $A[q]$ i-м за порядком зростання елементом. Якщо це так, то повертається елемент $A[q]$. В протилежному випадку в алгоритмі визначається, в якому з двох підмасивів міститься i-й за зростанням елемент: в підмасиві $A[p\ldots q-1]$ чи в підмасиві $A[q+1\ldots r]$. Якщо $i < k$, то потрібний елемент знаходиться в нижній частині розбиття і він рекурсивно обирається з відповідного підмасиву в рядку 8. Якщо ж $i > k$, то потрібний елемент знаходиться у верхньому підмасиві. Оскільки нам вже відомі k значень, які менші i-го за порядком зростання у масиві $A[p\ldots r]$, то потрібний елемент є $(i-k)$-м за порядком зростання в масиві $A[q+1\ldots r]$, який рекурсивно знаходиться в рядку 9.

Час роботи алгоритму `RandomizedSelect` в найгіршому випадку дорівнює $\Theta(n^2)$, причому навіть для пошуку мінімуму. Справа в тому, що фортуна може від нас відвернутись, і розбиття завжди буде виконуватись відносно найбільшого з елементів що лишились, а саме розбиття займає час $\Theta(n)$. Однак в середньому алгоритм працює добре і зараз ми знайдемо оцінку часу його роботи.

Ідея визначення оцінки часу роботи процедури `RandomizedSelect` полягає у наступному: ми очікуємо, що розмір масиву для кожного рекурсивного виклику зменшується на сталий фактор.

Будемо вважати, що алгоритм знаходиться у фазі j, якщо розмір масиву поточного рекурсивного виклику (тобто величина $r-p+1$) лежить в проміжку між $n\left(\frac{3}{4}\right)^{j+1}$ та $n\left(\frac{3}{4}\right)^{j}$, де n – розмірність початкового масиву A. Для того щоб оцінити час роботи алгоритму протягом фази j, введемо випадкову величину X_j, яка дорівнює кількості рекурсивних викликів протягом фази j.

Як зазначалось вище, за межами рекурсивних викликів процедура `RandomizedSelect` витрачає час $\Theta(n)$ (або cn, де n – деяка стала) на випадкове розбиття за допомогою процедури `RandomizedPartition`. Таким чином, за одну ітерацію (один рекурсивний виклик) протягом фази j витрачається часу не більше ніж $cn\left(\frac{3}{4}\right)^{j}$, а за всю фазу j (враховуючи, що протягом цієї фази буде виконано

X_j рекурсивних викликів) – $X_j \cdot cn\left(\frac{3}{4}\right)^j$. Отже, ми можемо дати верхню оцінку часу роботи $T(n)$ процедури `RandomizedSelect`:

$$T(n) \le \sum_{j=0}^{\infty} X_j \cdot cn\left(\frac{3}{4}\right)^j. \qquad (8.4)$$

Введемо ще один технічний термін. Для заданої ітерації алгоритму ми будемо вважати елемент множини A *центральним*, якщо принаймні чверть елементів множини менше даного елементу і принаймні чверть елементів – більше даного елементу. Тепер уявіть, що центральний елемент буде обрано в якості опорного під час роботи процедури `RandomizedPartition`. Це означатиме, що принаймні четверта частина поточного масиву буде відкинута з подальшої роботи, тобто розмір масиву для нового рекурсивного виклику (розмір $A[p\ldots q-1]$ або $A[q+1\ldots r]$) буде складати 3/4 поточного масиву (а може навіть і менше). Легко бачити, що це призведе до закінчення поточної фази j, і алгоритм перейде до роботи в фазу $j + 1$.

Скільки ж елементів масиву є центральними? Відповідь на це питання проста – половина елементів поточного масиву. Отже, ймовірність вибору центрального елементу в якості опорного становить 1/2 (адже елементи обираються незалежно). І це означає, що ми звели задачу визначення математичного сподівання $E[X_j]$ кількості рекурсивних викликів протягом однієї фази j, до задачі підкидання ідеальної монети:

$$E\left[X_j\right] \le E[N],$$

де N – кількість підкидань монети до настання події випадіння, наприклад, орла.

Для знаходження математичного сподівання випадкової величини N скористаємось наступними міркуваннями. Для $m > 0$, маємо: $P[N = m] = (1/2)^{m-1}(1/2)$, адже для того, щоб орел випав на підкиданні за номером m, протягом попередніх $m-1$ підкидань повинні випасти решки, а на останнє – орел. Таким чином,

$$E[N]=\sum_{m=0}^{\infty} m\cdot P[N=m]=\sum_{m=0}^{\infty} m\cdot(1/2)^{m-1}\cdot(1/2)=\sum_{m=0}^{\infty} m\cdot(1/2)^{m}=\frac{1/2}{(1-1/2)^{2}}=2,$$

що означає, що $E\left[X_j\right]\le 2$.

Очікуваний час виконання процедури `RandomizedSelect` ми можемо позначити через $E[T(n)]$. Тоді за рахунок отриманих вище результатів і використовуючи вираз (8.4), отримуємо:

$$E\left[T(n)\right]\le E\left[\sum_{j=0}^{\infty} X_j\cdot cn\left(\frac{3}{4}\right)^{j}\right]=\sum_{j=0}^{\infty} E\left[X_j\right]\cdot cn\left(\frac{3}{4}\right)^{j}=$$

$$=cn\cdot\sum_{j=0}^{\infty} E\left[X_j\right]\cdot\left(\frac{3}{4}\right)^{j}\le cn\cdot\sum_{j=0}^{\infty} 2\cdot\left(\frac{3}{4}\right)^{j}=\frac{2cn}{1-3/4}=8cn.$$

Звідки отримуємо, що середній час роботи процедури `RandomizedSelect` становить $O(n)$.

Література

1. Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. – М. : Изд. дом "Вильямс", 2011. – 1296 с. (Глава 7, Глава 9, розділ 9.1, 9.2.)