Київський національний університет імені Тараса Шевченка
факультет комп'ютерних наук та кібернетики

ПОБУДОВА ТА АНАЛІЗ АЛГОРИТМІВ.

ЛЕКЦІЇ

КИЇВ – 2020

УДК 004.43.421(042.3)

Рекомендовано вченою радою факультету комп'ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка (протокол №8 від 21 грудня 2020 р.).

Укладач: кандидат фіз.-мат. наук, професор, доктор габілітації Вергунова І.М.

Рецензенти: доктор фіз.-мат. наук, доцент І.О. Завадський
доктор наук, професор В.Р. Стебловська

В 52 Вергунова І.М. Побудова та аналіз алгоритмів.Лекції. – Вінниця : ТВОРИ, 2020. – 164 с.

Викладено лекційний курс з дисципліни «Побудова та аналіз алгоритмів» для студентів факультету комп'ютерних наук та кібернетики спеціальності 122 «Комп'ютерні науки» ОП «Інформатика».

ISBN 978-966-949-684-3

УДК 004.43.421(042.3)

ISBN 978-966-949-684-3 © Вергунова І.М., 2020

Лекція 1. Асимптотичні вирази. *O*-нотація. Властивості *O*-нотації

Розглядатимемо додатні функції (нехай – $f(n)$, $g(n)$, $h(n)$ та ін.) з аргументом $n \in N$.

Визначення *O*-нотації. Функція $f(n)$ є *O*-великим від $g(n)$, якщо існують сталі $c, n_0 : 0 \le f(n) \le c\varphi(n), n \ge n_0$.

Справедливе і наступне визначення: $f(n) = O(\varphi(n))$ – це множина функцій таких, що $\exists \operatorname{const} c, n_0 : 0 \le f(n) \le c\varphi(n), n \ge n_0$.

Визначаючи $f(n) = O(\varphi(n))$, вказуємо верхню границю функції з точністю до деякого множника.

Маємо,

$O(n) + O(n^2) = O(n^2), n \to \infty$. $\Rightarrow$ $\forall f, g : f(n) = O(n), g(n) = O(n^2), n \to \infty$ $f(n) + g(n) = O(n^2), n \to \infty$.

$\frac{1}{n}O(1) = O(1) + O(\frac{1}{n^2}), 0 < n < \infty$. Якщо для $f(n)$ справедливо $f(n) = O(1)$, то $n^{-1}f(n)$ можна розбити на суму функцій $g(n)$ та $h(n)$: $g(n) = O(1)$, $h(n) = O(n^{-2})$ (наприклад, $g(n) = 0, 0 \le n \le 1$, $g(n) = n^{-1}f(n), n > 1$, та $h(n) = n^{-1}f(n), 0 \le n \le 1$, $h(n) = 0, n > 1$).

Для подібних формул будь-який вираз з символом *O* розглядається як клас функцій. Наприклад, на інтервалі $0 < n < \infty$ $O(1) + O(n^{-2})$ означає клас усіх функцій виду $g(n) + h(n)$, де $g(n) = O(1)$, $h(n) = O(n^{-2})$, $0 < n < \infty$, а вираз $\frac{1}{n}O(1) = O(1) + O(\frac{1}{n^2}), 0 < n < \infty$ вказує, що клас $n^{-1}O(1)$ міститься у класі $O(1) + O(n^{-2})$. Іноді у лівій частині може знаходитися не клас, а окрема функція. Це значить, що функція зліва входить до класу, що стоїть у правій частині. Знак «=» насправді не підходить для таких виразів, тому, що він передбачає симетрію, якої тут немає (наприклад, $O(n) = O(n^2), n \to \infty$ справедливе, а навпаки – ні).

Нехай $g(n)$ та $h(n)$ дві такі функції, що $g(n) = O(h(n))$, $0 < n < \infty$ виконується, а $h(n) = O(g(n))$ ні. Якщо $f(n) = O(g(n))$, $0 < n < \infty$, то $f(n) = O(h(n))$, $0 < n < \infty$. Якщо справедливий вираз $f(n) = O(g(n))$,

то його називають **уточненням** $f(n) = O(h(n))$. Уточнення $f(n) = O(g(n))$ називають **найкращим можливим**, якщо воно не може бути уточненим, тобто, якщо знайдуться такі додатні сталі *c* та *C*, що $cg(n) \le f(n) \le Cg(n)$ для всіх достатньо великих *n*. Наприклад, $2n + n\sin n = O(n), n \to \infty$, є найкращим можливим, так як ліва частина знаходиться в межах *n* та 3*n*.

Розглянемо питання рівномірності оцінок на прикладах.

1. Нехай S – множина значень *n*, $k > 0, f(n), g(n)$ є довільними додатними функціями. Маємо $(f(n) + g(n))^k = O((f(n))^k) + O((g(n))^k)$.

Дійсно,

$$|f(n) + g(n)|^k \le (|f(n)| + |g(n)|)^k \le (2\max\{|f|, |g|\})^k \le$$
$$\le 2^k \max\{|f|^k, |g|^k\} \le 2^k(|f|^k + |g|^k).$$

Тоді знайдуться такі додатні сталі *A* та *B*, що $|f(n) + g(n)|^k \le A|f|^k + B|g|^k$. Сталі *A* та *B* залежать від *k* (точніше – не знаємо, чи існують не залежні від *k*).

2. Наприклад, $\left(\frac{k}{n^2 + k^2}\right)^k = O(n^{-k}), k > 0, 1 < n < \infty$. В наведеному виразі стала, що входить до *O*-великого може бути вибрана не залежною від *k*. Дійсно, $n^2 + k^2 = (n-k)^2 + 2nk \ge 2nk$, тому $\frac{k}{n^2 + k^2} \le \frac{1}{(2n)^k}$. Для всіх $k > 0$ $2^{-k} < 1$, тому можемо вибрати сталу *A* не залежну від *k* (наприклад, *A*=1), $\left(\frac{k}{n^2 + k^2}\right)^k \le An^{-k}, k > 0, 1 < n < \infty$. Це можемо виразити сказавши, що вираз $\left(\frac{k}{n^2 + k^2}\right)^k = O(n^{-k}), k > 0, 1 < n < \infty$ має місце рівномірно по *k*.

Рівномірність потрібна у наступних випадках. Якщо потрібно одержати *O*-оцінку для функції $f(n)$. Але в якості оцінки маємо для функції $f(n)$ деякий вираз, який розбивається на суми. Спосіб, за якого $f(n)$ розбивається на суми залежить від параметра *t*. Наприклад, одержано оцінку $f(n) = O(n^2 t) + O(n^4 t^{-2}), n > 1, t > 1$. Оцінюючи доданки за *n* та *t*

бажано вибрати t, щоб права частина стала як можна меншою. Так як оцінка рівномірна, то можна вважати t деякою функцією від n, тому получимо задачу $n^2 t + n^4 t^{-2} \to \min$ за заданого n. Мінімум досягається у точці $t = (2n^2)^{1/3}$ та в точці мінімуму обидва доданки мають один порядок $n^{8/3}$. Тому $f(n) = O(n^{8/3})$. Такий же результат можна одержати й з аналізу поведінки функції.

А, наприклад, оцінка $k^2 \left(1 + kn^2\right)^{-1} = O(n^{-1}), k > 0, n \to \infty$ не є рівномірною по k. Дійсно, якби вона була рівномірною, то знайшлися б такі додатні сталі A та B, незалежні від k, що $k^2 \left(1 + kn^2\right)^{-1} < An^{-1}, k > 0, n > B$. Узявши $k = n^2$ одержали б $A\left(1 + kn^2\right) > n^5,\ n > B$, що неможливо. З іншого боку, одну із сталих A та B можна вибрати не залежною від k:

$B = k, A = 1$, так як $k^2 \left(1 + kn^2\right)^{-1} < kn^{-2} < k^{-1}, k > 0, n > k$;

$B = 1, A = k$, так як $k^2 \left(1 + kn^2\right)^{-1} < n^{-2} < kn^{-1}, k > 0, n > 1$.

Властивості O-нотації [1].

1. $nO(1) = O(n)$.
2. $nO(f(n)) = O(nf(n))$.
3. $O(1)O(1) = O(1)$.
4. $f(n) = O(f(n))$.
5. $O(1)O(f(n)) = O(f(n))$.
6. $cO(f(n)) = O(f(n))$.
7. $O(cf(n)) = O(f(n))$.
8. $O(f(n)) + O(g(n)) = O(f(n) + g(n))$.
9. $O(f(n))O(g(n)) = O(f(n)g(n))$.
10. $f(n) - g(n) = O(h(n)) \Rightarrow f(n) = g(n) + O(h(n))$.

Приклад 1.1. Спростити заданий асимптотичний вираз:

$$(n + O(1)(n + O(\lg n) + O(1)).$$

За наведеними вище властивостями маємо:

$$(n + O(1)(n + O(\lg n) + O(1)) =$$
$$= n^2 + O(n\lg n) + O(\lg n) + O(n) + O(1) = n^2 + O(n\lg n).$$

Лекція 2. Поняття обчислювальної складності алгоритму. Принципи аналізу алгоритму. Зростання функцій. Ω-нотація. Θ-позначення

Оцінювання та аналіз побудованих алгоритмів потрібні, наприклад, для: порівняння різних алгоритмів, призначених для виконання однієї задачі; одержання наближеної оцінки продуктивності програми; встановлення значень параметрів алгоритму (можливість одержання формул для розрахунку часу виконання у реальних ситуаціях). Іноді аналіз зводиться до визначення частоти виконання декількох фундаментальних операцій (тобто шукаємо наближені оцінки цих величин. Результати аналізу та оцінювання дадуть можливість зрозуміти, що одна програма буде виконуватися ефективніше інших у конкретних ситуаціях.

Більшість оцінок алгоритмів (виражаються додатними функціями) має головний параметр, наприклад, n, який значно впливає на час їх виконання. Параметр n може бути аргументом деякої монотонно зростаючої функції (поліноміальної, логарифмічної тощо), розміром елементів, пов'язаних з виділенням пам'яті або розміром файлу, іншою абстрактною мірою задачі, що розглядається.

Якщо таких параметрів більше одного, то часто зводять аналіз до розгляду одного параметру, задаючи його як функцію інших. Найчастіше розглядають загальний час виконання в залежності від одного параметру (а саме розглядають вирази, справедливі для великих значень параметрів).

Підрахунок кількості операцій (узагальнених) дуже часто дозволяє порівняти ефективність алгоритмів. Аналіз проводять з розрахунком на достатньо великий об'єм оброблюваних даних та величину аргументу n (можливо $n \to \infty$), тому ключове значення має **швидкість зростання** функції складності, а не точна кількість операцій, а в аналізі застосовують асимптотичні позначення (наприклад, $O(f(n))$ –функції, що зростають повільніше ніж f). Використовують часто пропорційність до функцій $\lg n, n\lg n, n, n^2$ та ін. Пропорційність отримують при екстраполяції продуктивності на основі емпіричного вивчення. Порядок функції складності – при порівнянні продуктивності різних алгоритмів або при передбаченні абсолютної продуктивності.

Отже, складність алгоритмів визначається для доволі великих об'ємів даних. Тому при порівнянні деяких двох алгоритмів працеємність можна розглядати як границю відношення функції їхньої складності ($\lim_{n\to\infty} f(n)/g(n)$). За значенням границі можна зробити висновок відносно швидкостей зростання функцій (додатна стала – зростають з однаковою швидкістю, 0 – $g(n)$ зростає швидше за $f(n)$, ∞ – $f(n)$ зростає швидше за $g(n)$).

Запис $g(n) = O(f(n))$ ($f(n)$ зростає швидше за $g(n)$) вказує на належність $g(n)$ до класу $O(f(n))$, тобто функція $g(n)$ **обмежена зверху** функцією $f(n)$ для достатньо великих значень аргументу, якщо $\exists \text{const}\, c, n_0 :\ \forall n \ge n_0\ \ g(n) \le cf(n)$.

Функцію $f(n)$ називають **поліноміально обмеженою**, якщо існує така стала *k*, що $f(n) = O(n^k)$.

Обмеженість знизу функції $f(n)$ функцією $g(n)$, яку записують як $f(n) = \Omega(g(n))$, значить, що для достатньо великих значень аргументу $\exists \text{const}\, c, n_0 :\ \forall n \ge n_0\ \ cg(n) \le f(n)$. Нотації O та Ω взаємно замінюються $g(n) = O(f(n)) \Leftrightarrow f(n) = \Omega(g(n))$.

Якщо функції $f(n)$ та $g(n)$ мають **однакову швидкість зростання** ($f(n) = \Theta(g(n))$), то, це значить, що для достатньо великих значень аргументу $\exists \text{const}\, c_1, c_2 : (\exists n_0 > 0 :\ \forall n \ge n_0\ \ 0 \le c_1 g(n) \le f(n) \le c_2 g(n))$. При цьому маємо також справедливість $g(n) = \Theta(f(n))$. Кажуть, що $g(n)$ є **асимтотично точною оцінкою** функції $f(n)$, якщо $f(n) = \Theta(g(n))$. Із вище вказаного для множини $\Theta(f(n))$ необхідно, щоб кожен елемент множини $g(n)$ був **асимптотично невід'ємним** (за достатньо великих значень *n* функція $g(n) \ge 0$), інакше множина $\Theta(f(n))$ буде $\emptyset$.

Функція $f(n)$ є асимптотично додатною, якщо за достатньо великих значень *n* функція $f(n) > 0$.

З наведених означень можна одержати справедливість наступного твердження: для будь-яких функцій $f(n)$ та $g(n)$ маємо $f(n) = \Theta(g(n))$ тоді і тільки тоді, коли $f(n) = O(g(n))$ та $f(n) = \Omega(g(n))$ [1-2].

Наведене твердження є дуже важливим для конкретних застосувань, так як на практиці асимптотично точну оцінку Θ визначають за нотаціями O та Ω.

Властивості Ω-нотації та Θ є аналогічними до наведених раніше властивостей O нотації.

Часто результати аналізу не є точними, вони наближені у точному технічному змісті, а результат являє собою вираз у вигляді послідовності доданків. Нотації **дозволяють розглядати головні члени** та випускати менші доданки у роботі з наближеними виразами. O-нотацію використовують для дослідження фундаментальної асимптотичної поведінки алгоритмів.

Наприклад, $0{,}5n^2 - 3n = \Theta(n^2)$. Які повинні бути значення c_1, c_2, n_0? Маємо $c_1 n^2 \le 0{,}5n^2 - 3n \le c_2 n^2$ для $\forall n \ge n_0$. Тоді $c_1 \le 0{,}5 - 3n^{-1} \le c_2$. Виберемо $c_2 \ge 1/2$. Тоді одержимо виконання правої частини для $\forall n \ge 1$. Ліва частина виконується для $\forall n \ge 7$, якщо $c_1 \le 1/14$. Тому $c_1 = 1/14$, $c_2 = 1/2$, $n_0 = 7$, щоб мало місце $0{,}5n^2 - 3n = \Theta(n^2)$.

Розглянемо ще один приклад. $f(n) = an^2 + bn + c$, $a, b, c - const$, $a > 0$. Покажемо, що $f(n) = \Theta(n^2)$.

За визначенням $c_1 n^2 \le an^2 + bn + c \le c_2 n^2$,

$$c_1 \le a + bn^{-1} + cn^{-2} \le c_2.$$

Виберемо $n_0 = 2\max\left(\frac{|b|}{a}, \sqrt{\frac{|c|}{a}}\right)$. Тоді, якщо візьмемо наступні сталі $c_1 = a/4$, $c_2 = 7a/4$, то виконується $f(n) = \Theta(n^2)$ (для вибору c_2: $a + \frac{1}{2}a + \frac{1}{4}a = \frac{7}{4}a$, $c_2 = 7a/4$; для вибору c_1: $a - \frac{1}{2}a - \frac{1}{4}a = \frac{1}{4}a$, $c_1 = a/4$).

Коли кажуть, що час роботи алгоритму $\Omega(g(n))$, то вважають, що незалежно від того, які конкретно вхідні дані розміру n обрані, для кожного значення n час роботи для кожного із вхідних даних буде для достатньо великих n як мінімум стала, помножена на $g(n)$. Тобто дають **нижню границю** часу роботи алгоритму у *найкращому випадку*. Це маємо, коли гарантуємо продуктивність або передбачаємо її (наприк-

лад, алгоритм бінарного пошуку має границю $O(\lg n)$). Ця продуктивність може дуже відрізнятися від продуктивності випадку найгірших вхідних даних. Середня продуктивність дає припущення про час виконання. Але знання середнього часу виконання може бути недостатньо (може бути потрібним середнє відхилення або інформація про розподіл часу виконання, що важко отримати). Якщо кажуть, що час виконання алгоритму у найгіршому випадку $O(f(n))$, то мають на увазі, що $f(n)$ є **верхньою границею** складності задачі. Коли у побудові алгоритму для деякої задачі маємо, що верхня та нижня границі складності алгоритмів співпадають, то можна бути впевненим у тому, що немає смислу розробляти алгоритм, який би був фундаментально швидшим, ніж найкращий з відомих.

Розглянемо наступний приклад. Нехай середній час виконання алгоритму $2an\lg n + O(n)$. За великих n це час близький до $2an\lg n$. Покажемо, що для передбачення як зросте час виконання для $2n$ знати величину сталої a не треба. Маємо:

$$\frac{2a2n\lg 2n + O(2n)}{2an\lg n + O(n)} = \frac{2a2n\lg 2n + 2nO(1)}{2an\lg n + nO(1)} =$$

$$= \frac{2\lg 2n + O(1)}{\lg n + O(1)} = 2 + \frac{O(1)}{\lg n + O(1)} = 2 + O(\frac{1}{\lg n}).$$

Використання для аналізу тільки головного члену $O(n\lg n)$ такого б результату не дало.

Дуже часто в оцінюванні алгоритмів використовують поняття порядку функції та пропорційності. Під цим розуміють наступне. Коли функція $f(n)$ є асимптотично великою у порівнянні з функцією $g(n)$, тобто $g(n)/f(n) \to 0, n \to \infty$, приймаючи, що $f(n) = O(g(n))$ використовують **порядок** $f(n)$. Тоді за великих значень n досліджувана величина буде близькою до величини $f(n)$. Кажуть, що час виконання алгоритму **пропорційний** $f(n)$, коли можемо довести, що він дорівнює $cf(n) + g(n)$, де $g(n)$ асимптотично мале у порівнянні з $f(n)$, c – деяка стала. Але між вказаними поняттями є різниця: пропорційність використовується для екстраполяції продуктивності алгоритму на основі емпіричного вивчення, порядок – при порівнянні продуктивності різних алгоритмів або передбаченні абсолютної продуктивності.

Розглянемо поняття ***o*-малого** та $\omega-$ **малого.** Асимптотична верхня границя (*O*-велике) може описувати асимптотичну поведінку функції з різною точністю. Наприклад, $2n^2 = O(n^2)$ та $2n = O(n^2)$. Для вказівки того, що верхня границя не є асимптотичною оцінкою функції застосовують ***o*-мале**, яке визначимо наступним чином.

$f(n) = o(g(n))$, якщо для будь-якої сталої $c > 0 \, \exists n_0 : 0 \le f(n) < cg(n)$, $n \ge n_0$ (або $\lim_{n\to\infty} f(n)/g(n) = 0$).

Наприклад, $2n = o(n^2)$.

Аналогічно, $\omega-$ **мале** – це нижня границя, що не є асимптотично точною оцінкою.

$f(n) = \omega(g(n))$ якщо для будь-якої сталої $c > 0 \, \exists n_0 : 0 \le cg(n) < f(n)$, $n \ge n_0$ (або $\lim_{n\to\infty} f(n)/g(n) = \infty$).

Наприклад, $2n^2 = \omega(n)$.

Кажуть, що функція $f(n)$ асимптотично менша функції $g(n)$, якщо $f(n) = o(g(n))$. Або: функція $f(n)$ – асимптотично більша функції $g(n)$, якщо $f(n) = \omega(g(n))$. Але є функції які порівняти не можна (наприклад, n та $n + n\sin n$).

Загальні властивості функцій порівняння:

1. Транзитивність (для O, Ω, Θ, o, ω).
2. Симетричність (для Θ).
3. Рефлексивність (для O, Ω, Θ).
4. Зв'язок між функціями $g(n) = O(f(n)) \Leftrightarrow f(n) = \Omega(g(n))$

$$g(n) = o(f(n)) \Leftrightarrow f(n) = \omega(g(n)).$$

Найчастіше аналіз алгоритмів пов'язаний з аналізом їх часової складності. Його результатом є асимптотична оцінка кількості виконуваних алгоритмом операцій, як функції довжини входу, що корелює з асимптотичною оцінкою часу виконання програмної реалізації алгоритму.

Дослідимо асимптотичну поведінку, наприклад, алгоритму сортування вставками за наведеною нижче процедурою Insertion-Sort [2]. Модель обчислень за алгоритмом передбачає реалізацію для машини з довільним доступом до пам’яті (RAM). В ній множина базових операцій складається з операцій з пам’яттю, включаючи операцію адресації, арифметичних операцій, порівнянь та переходів на інші операції.

Зауважимо, що, так як асимптотичні оцінки вказують не більш ніж порядок росту функції, то результати можливого порівняння алгоритмів за такими оцінками будуть справедливі тільки за великих значень довжин входів.

Уведемо середню «вартість» базових операцій у рядку, пов'язану з часом їх виконання. Для кожного j, $j=\overline{2..n}$, n= *A.length*, позначимо t_j кількість перевірок умови в циклі **while** (рядок 5). Загальний час виконання алгоритму є сумарним часом виконання усіх операцій процедури. Тому сумарний час роботи алгоритму T(n) є сумою добутків вартостей операцій на кількість їх повторювань.

	Insertion-Sort (A)	Вартість	Кількість повторювань
1	**for** $j = 2$ **to** *A.length*	c_1	n
2	$key = A[j]$	c_2	$n-1$
3	//вставка $A[j]$ до відсортованої послідовності $A[1..j\text{-}1]$	0	$n-1$
4	$i = j - 1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	$i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8	$A[i+1] = key$	c_8	$n-1$

Маємо

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\sum_{j=2}^{n} t_j\right) + c_6\left(\sum_{j=2}^{n}(t_j - 1)\right) +$$
$$+ c_7\left(\sum_{j=2}^{n}(t_j - 1)\right) + c_8(n-1).$$

Але навіть, якщо розмір вхідних даних є фіксованою величиною, загальний час роботи алгоритму залежить і від самих вхідних даних, а саме – степені їх початкової впорядкованості. У найкращому випадку всі елементи можуть стояти вже у відсортованому порядку. Тоді для кожного j, $j=\overline{2..n}$, у рядку 5 знаходиться, що $A[i] \le key$ ще на початку (при $i = j - 1$). Тому $t_j = 1$ для $j=\overline{2..n}$, а час роботи алгоритму буде лінійною функцією від n:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) =$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

У найгіршому випадку елементи масиву знаходяться у порядку, зворотному порядку у відсортованому масиві. Тоді необхідно порівнювати кожен елемент $A[j]$ з кожним елементом всього відсортованого масиву $A[1..j\text{-}1]$. Тому $t_j = j$ для $j = \overline{2..n}$, а час роботи алгоритму буде квадратичною функцією від n:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\sum_{j=2}^{n} j\right) + (c_6 + c_7)\left(\sum_{j=2}^{n} (j-1)\right) + c_8(n-1) =$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + (c_6 + c_7)\left(\frac{n(n-1)}{2}\right) + c_8(n-1) =$$

$$= (c_5 + c_6 + c_7)n^2/2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

Лекція 3. Метод підстановки в аналізі алгоритмів

Нехай в результаті аналізу алгоритму (наприклад, алгоритму сортування злиттям або деякого іншого) одержаний рекурентний вираз для визначення граничної оцінки часу роботи алгоритму:

$$T(n) = \begin{cases} \Theta(1), \ n \le n_0, \\ aT(n/b) + D(n) + C(n), n > n_0, \end{cases}$$

де a – кількість підзадач, n/b – їх розмір, $D(n)$ – вартість поділу (час, що витрачається на поділ) задачі на підзадачі, $C(n)$ – вартість об'єднання рішень підзадач у рішення задачі. Для процедури Merge-Sort рекурентний вираз має наступний вигляд:

$$T(n) = \begin{cases} \Theta(1), \ n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n), n > 1. \end{cases}$$

Як обчислити значення такого виразу? Використаємо для цього метод підстановки. Сформулюємо його основні етапи:

1. Наводиться припущення про вигляд розв'язку.
2. За цим припущенням методом математичної індукції визначають сталі у нерівностях.
3. Розглядають граничні умови.

Нехай потрібно визначити верхню границю рекурентного співвідношення

$$T(n) = \begin{cases} \Theta(1), & n \le n_0, \\ 2T(\lfloor n/2 \rfloor) + n, & n > n_0. \end{cases}$$

Припустимо, що рішення має вигляд $T(n) = O(n lg n)$.

Покажемо, що за підходящого вибору сталої додатної c виконується нерівність $T(n) \le cnlgn$.

Припустимо справедливість нерівності для всіх додатних $m < n$, а саме для $m = \lfloor n/2 \rfloor$, тобто $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor lg \lfloor n/2 \rfloor$.

Покажемо справедливість для довільного n, підставимо вказане до рекурентного виразу, тоді

$$T(\lfloor n \rfloor) \le 2c \lfloor n/2 \rfloor lg \lfloor n/2 \rfloor + n \le cnlgn/2 + n =$$
$$= cnlgn - cnlg2 + n = cnlgn - cn + n \le$$
$$\le cnlgn,$$

де стала повинна бути $c \ge 1$.

Розглянемо граничні умови. Необхідно показати, що рішення задовольняє граничним умовам. Звичайно для цього достатньо показати, що граничні умови є підходящою базою для доведення за індукцією. Покажемо, що сталу c можна вибрати достатньо великою для того, щоб нерівність $T(n) \le cnlgn$ мала місце й для граничних умов. Нехай $T(n) = 1$ є єдиною граничною умовою. Якщо $n = n_0 = 1$, $c \ge 1$, то получимо $T(1) \le c1lg1 = 0$ і гранична умова $T(1) = 1$ не виконується. Припустимо, $n = n_0 = 2$, $c \ge 1$, тоді – $T(2) \le c2lg2 = 2c$ (з рекурентного співвідношення – $T(2) = 4$). Звідси випливає: якщо вибрати $c \ge 2$, $n_0 = 2$, то рекурентне співвідношення має місця для всіх $n \ge 2$. Тому верхньою границею рекурентного співвідношення буде $O(nlgn)$.

Але іноді зробити вірне припущення про асимптотичну поведінку рішення рекурентного співвідношення досить важко. Крім того, можливі деякі ускладнення при визначенні асимптотичної границі методом підстановки, пов’язані з тим, що було обране недостатньо сильне припущення індукції, яке не дозволяє показати точну границю. В такому випадку необхідно переглянути припущення індукції. Наприклад, необхідно визначити верхню границю рекурентного співвідношення

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1.$$

Припустимо, що рішення має вигляд $T(n) = O(n)$. Спробуємо показати, що за підходящого вибору додатної сталої c виконується $T(n) \le cn$.

Припустимо справедливість нерівності для всіх додатних $m < n$, а саме для $m = \lceil n/2 \rceil$.

Підставимо прогнозоване рішення до рекурентного виразу:

$$T(n) \le c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 = cn + 1.$$

Але звідси не маємо, що $T(n) \le cn$ для деякої додатної сталої c (доведення гіпотези індукції повинно бути у точному вигляді).

Можна зробити припущення про вищий порядок оцінки ($T(n) = O(n^2)$). Але така оцінка буде не найкращою. А з виразу бачимо, що припущення було коректним. Тому для даному випадку маємо вибрати більш сильну гіпотезу індукції.

В припущенні ми помилилися на величину меншого порядку, на сталу, що дорівнює 1. Тому спробуємо відняти у початковому припущенні член нижчого порядку, а саме прийняти гіпотезу індукції у вигляді $T(n) \le cn - d$, де стала $d \ge 0$. В результаті матимемо:

$$T(n) \le (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 = cn + 1 = cn - 2d + 1 \le cn - d$$

для $d \ge 1$.

Щодо граничних умов, то для їх виконання потрібно сталу c вибрати достатньо великою.

Розглянемо ще один приклад. Нехай потрібно визначити верхню границю рекурентного співвідношення $T(n) = 2T(\sqrt{n}) + \lg n$.

Вираз виглядає доволі складним, щоб вказати вірне припущення. Спробуємо його спростити. Для цього використаємо заміну змінних не звертаючи увагу на заокругленні $\sqrt{n}$ до цілих. Візьмемо $m = lgn$. Тоді матимемо, що $T(2^m) = 2T(2^{m/2}) + m$. Перейменуємо $S(m) = T(2^m)$. В результаті одержимо нове рекурентне співвідношення $S(m) = 2S(m/2) + m$, розв'язок якого легко одержати. Як вже відомо, його розв'язком є $S(m) = O(m \lg m)$.

Перейдемо до початкових змінних,

$$T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n).$$

Отже, верхньою границею цього рекурентного співвідношення буде $O(\lg n \lg \lg n)$.

Лекція 4. Метод дерев рекурсії в аналізі алгоритмів

Метод дозволяє одержати припущення для методу підстановки та дає загальну вартість всіх рівнів дерева рекурсії, яке відповідає повному обліку (вартості) рекурсивної складової алгоритму.

Так як у побудованому дереві рекурсії кожен вузол представляє вартість виконання відповідної підзадачі, то необхідно знайти **суму вартостей для кожного рівня та загальну суму за всіма рівнями**, що дасть **повну вартість задачі**.

Якщо метод дерев рекурсії застосовують для одержання припущення у методі підстановки, то при знаходженні сум можемо припускати деякі неточності, тому, що потім, при виконанні методі підстановки, все одно величина одержаної суми перевіряється.

При прямому доведенні коректності рішення, сумування проводиться ретельно, враховуючи усі нюанси.

Розглянемо приклад $T(n) = 3T(n/4) + cn^2$, $c = const > 0$, для демонстрації методу [1]. Знайдемо асимптотично точну границю виразу.

Нехай *n* є степенем 4. Тоді розміри усіх підзадач є цілими.

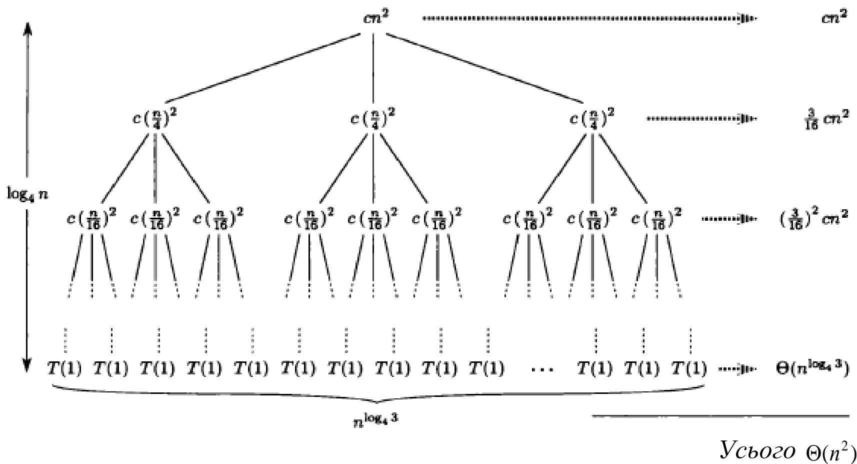


Рис. 4.1. Дерево рекурсії для прикладу $T(n) = 3T(n/4) + cn^2$, $c = const > 0$ [1].

Розташований в корені дерева член cn^2 дає час для самого верхнього рівня рекурсії, а його три піддерева – час виконання підзадач розміру $n/4 - c(n/4)^2$. Далі йдуть піддерева задач $T(n/4)$ з часом виконання $c(\frac{n}{4^2})^2$, наступні – підзадачі $T(n/4^i)$, що мають час виконання $c(n/4^i)^2$. В останніх підзадачах k-го шару маємо $(n/4^k) = 1$, $k = \log_4 n$. Тому в дереві всього $\log_4 n + 1$ рівнів $(0,1,\ldots, \log_4 n)$.
Визначимо вартість кожного шару. Для i-го шару маємо 3^i елементів з вартістю $3^i c(n/4^i)^2 = (3/16)^i cn^2$, а на самому нижньому – вузлів $3^{\log_4 n} = n^{\log_4 3}$ при загальній вартості $n^{\log_3 4}T(1)$, що є $\Theta(n^{\log_4 3})$, так як $T(1)$ вважаємо константною величиною.
Знайдемо загальну вартість усіх рівнів:

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) =$$

$$= \sum_{i=0}^{\log_4 n - 1}\left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) = \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}).$$

Одержаний вираз є не дуже підходящим для обчислення. Тому далі для одержання оцінки зверху перейдемо до розгляду нескінчено спадаючої геометричної прогресії.

$$T(n) = \sum_{i=0}^{\log_4 n - 1}\left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) =$$

$$= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) = O(n^2).$$

Тому для оцінки $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ можемо узяти припущення для вигляду розв'язку таке: $T(n)=O(n^2)$. Тут коефіцієнти при cn^2 обмежені зверху 16/13, а повний час роботи всього дерева в основному визначається часом роботи його кореня (вартість cn^2).
Далі, якщо $O(n^2)$ дійсно є верхньою границею для рекурентного співвідношення, то ця границя має бути асимптотично точною оцінкою (перший рекурсивний виклик дає вклад у загальний час роботи алгоритму $\Theta(n^2)$, тому нижня оцінка має бути $\Omega(n^2)$). Покажемо це.
Перевіримо припущення щодо верхньої границі для $T(n) = O(n^2)$. Для цього використаємо метод підстановок.
Маємо припущення методу $T(n) \le dn^2, d = const > 0$.
Тоді

$$T(n) \le 3T(\lfloor n/4 \rfloor) + cn^2 \ge 3d\lfloor n/4 \rfloor^2 + cn^2 \le 3d(n/4)^2 + cn^2 = \frac{3}{16}dn^2 + cn^2 \le dn^2$$

для $d \ge 16c/13$.

Тому $\Theta(n^2)$ є асимптотично точною оцінкою рекурентного співвідношення.

Розглянемо метод для більш складного рекурентного співвідношення. Нехай маємо таке співвідношення $T(n) = T(n/3) + T(2n/3) + O(n)$, а також n є степенем 3. Тоді розміри усіх підзадач є цілими.

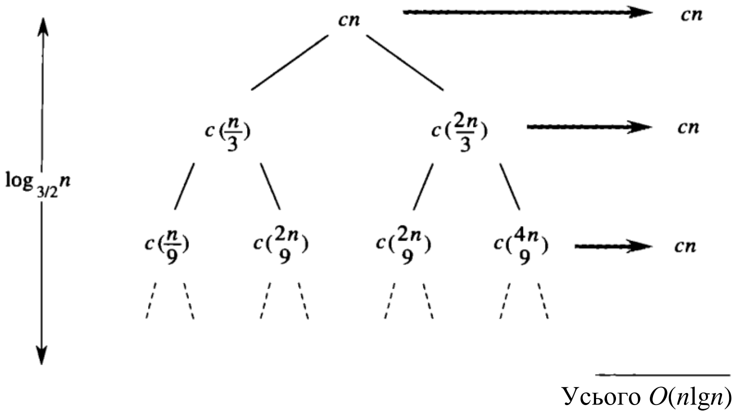


Рис. 4.2. Дерево рекурсії для прикладу $T(n) = T(n/3) + T(2n/3) + O(n)$ [2].

З рисунку бачимо, що для кожного шару одержимо вартість cn. Найдовшим шляхом від кореня до листа дерева буде шлях $n \to (2/3)n \to (2/3)^2 n \to \ldots \to 1$. Для нього листок досягаємо, коли $(2/3)^k n = 1$ та $k = \log_{3/2} n$. Отже, висота дерева: $\log_{3/2} n$.

Далі, значення $T(n)$ повинно не перевищувати сумі добутків рівнів на їхню вартість, а це $O(cn\log_{3/2} n) = O(n\lg n)$. Але в реальності не кожен рівень дає вартість cn. Дерево не є повним бінарним деревом (для повного бінарного дерева повинно бути листків $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$, їх загальна вартість $\Theta(n^{\log_{3/2} 2})$; величина $\Theta(n^{\log_{3/2} 2})$, вартість листків, має являти собою $\omega(n\lg n)$, оскільки $\log_{3/2} n$ є сталою, що більше 1), тому воно має менше листків. В ньому із збільшенням глибини зростає кількість вузлів, що зникають, тому нижні рівні мають вартість меншу за cn. Їх

можна спробувати врахувати. Але зараз, використовуючи метод підстановки, припустимо, що асимптотична верхня границя це – $O(n\lg n)$. Тоді гіпотеза індукції: $T(n) \le dn\lg n, d = const > 0$.

Припустимо справедливість нерівності для всіх додатних $m < n$.

Доведемо для довільних доволі великих n. Підставимо прогнозоване рішення до рекурентного виразу:

$$
\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \le \\
&\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn = \\
&= d(n/3)\lg n - d(n/3)\lg 3 + d(2n/3)\lg n - d(2n/3)\lg(3/2) + cn = \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn = \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn = \\
&= dn\lg n - \left(dn(\lg 3 - 2/3) - cn\right) \le dn\lg n
\end{aligned}
$$

для $d \ge c/(\lg 3 - (2/3))$.

Це звільняє від необхідності точного обрахування вартості рівнів. Маємо $O(n\lg n)$ є асимптотичною оцінкою верхньої границі рекурентного співвідношення.

Нехай маємо співвідношення $T(n) = T(n-2) + 1/\lg n, n > 2$, та $T(n) = \Theta(1), n \le 2$.

Знайдемо асимптотичні границі цього виразу.

На самому верхньому, нульовому, шарі дерева рекурсії маємо вартість $1/\lg n$, на наступному – $1/\lg(n-2)$, на i-му – $1/\lg(n-2i)$, на останньому – $\Theta(1)$. Висота дерева: $\lfloor n/2 \rfloor$.

Значення $T(n)$ є сумою добутків рівнів дерева рекурсії на їхню вартість, тому, наприклад, для n парного (враховуючи, що функція $1/i$ монотонно спадає та використовуючи нерівність $\frac{i}{1+i} \le \ln(1+i) \le i$) матимемо:

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\lfloor n/2 \rfloor - 1} \frac{1}{\lg(n-2i)} + \Theta(1) = \sum_{i=1}^{n/2} \frac{1}{\lg(2i)} + \Theta(1) = \sum_{i=1}^{n/2} \frac{1}{\lg i + 1} + \Theta(1) = \\
&= \sum_{i=2}^{n/2} \frac{1}{\lg i + 1} + \Theta(1) \le \sum_{i=2}^{n/2} \frac{1}{\lg i} + \Theta(1) \le \sum_{i=2}^{n/2} \frac{ln2}{ln(i)} + \Theta(1) \le \\
&\le ln2 \sum_{i=2}^{n/2} \frac{i}{i-1} + \Theta(1) \le ln2 \sum_{i=2}^{n/2} \left(1 + \frac{1}{i-1}\right) + \Theta(1) = ln2\left(\frac{n}{2} - 1\right) + ln2 \sum_{i=3}^{n/2} \left(\frac{1}{i-1}\right) + \Theta(1) \le \\
&\le ln2\left(\frac{n}{2} - 1\right) + ln2 \int_{2}^{n/2} \frac{1}{x-1} dx + \Theta(1) = ln2\left(\frac{n}{2} - 1\right) + ln2\left(ln\left|\frac{n}{2} - 1\right| - \ln 1\right) + \Theta(1) =
\end{aligned}
$$

$$= ln2(\frac{n}{2}-1)+ln2ln\left|\frac{n}{2}-1\right|+\Theta(1)=O(n).$$

Також

$$T(n)=\sum_{i=0}^{\lfloor n/2\rfloor-1}\frac{1}{\lg(n-2i)}+\Theta(1)=\sum_{i=1}^{n/2}\frac{1}{\lg(2i)}+\Theta(1)=\sum_{i=1}^{n/2}\frac{1}{\lg i+1}+\Theta(1)\geq$$

$$\geq\sum_{i=2}^{n/2}\frac{1}{2\lg i}+\Theta(1)\geq\frac{1}{2}\sum_{i=2}^{n/2}\frac{1}{i}+\Theta(1)\geq\frac{1}{2}\int_{2}^{(n/2)+1}\frac{1}{x}dx+\Theta(1)=$$

$$=\frac{1}{2}(ln(\frac{n}{2}+1)-\ln 2)+\Theta(1)=\Omega(lnn).$$

Використовуючи метод підстановки одержимо оцінку для верхньої границі. Припустимо за наведеним вище, що асимптотична верхня границя – $O(n)$.

Тоді гіпотеза індукції: $T(n)\ \leq cn, c=const>0$.

Припустимо справедливість нерівності для всіх додатних $m<n$.

Доведемо для довільних доволі великих n. Підставимо прогнозоване рішення до рекурентного виразу:

$$T(n)\ \leq c(n-2)+1/\lg n=cn-(2c-1/\lg n)\leq cn$$

за умови $2c-1/\lg n\geq 0$, тобто $c\geq 1/2$ для $n>2$.

Отже, асимптотичною верхньою границею є $O(n)$. Вказана границя не є асимптотично точною, так як вона не є також і асимптотичною нижньою границею.

Лекція 5. Основна теорема в аналізі алгоритмів

Розглянемо рекурентні співвідношення вигляду $T(n)=aT(n/b)+f(n)$, де a, b – деякі сталі, $a\geq 1$, $b>1$, $f(n)$ – асимптотично додатні функції, що описують час, потрібний для поділу задачі на складові підзадачі та поєднання результатів, отриманих з рішення допоміжних підзадач.

Найкраще формулювання та доведення теореми представлено в [2]. Тому й викладемо теорему, використовуючи результати [2].

Основна теорема.

Нехай $a\geq 1$, $b>1$ – сталі, $f(n)$ – асимптотично невід'ємна функція, а функція $T(n)$ визначена на множині невід'ємних цілих чисел за допомогою рекурентного співвідношення вигляду $T(n)=aT(n/b)+f(n)$, де n/b

інтерпретується або як $\lfloor n/b \rfloor$, або як $\lceil n/b \rceil$. Тоді $T(n)$ має наступні асимптотичні границі:

1. Якщо $f(n) = O(n^{\log_b a-\varepsilon})$ для деякої сталої $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$.
2. Якщо $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Якщо $f(n) = \Omega(n^{\log_b a+\varepsilon})$ для деякої сталої $\varepsilon > 0$ та $af(n/b) \leq cf(n)$ для деякої сталої $c < 1$ й усіх достатньо великих n, то $T(n) = \Theta(f(n))$.

Розглянемо спочатку дерево рекурсії, що буде відповідати вказаному рекурентному співвідношенню. Як видно з побудови дерева, виразу $T(n)=aT(n/b)+f(n)$ відповідає повне a-арне дерево рекурсії висотою $\log_b n$ з $n^{\log_b a}$ листками на нижньому рівні.

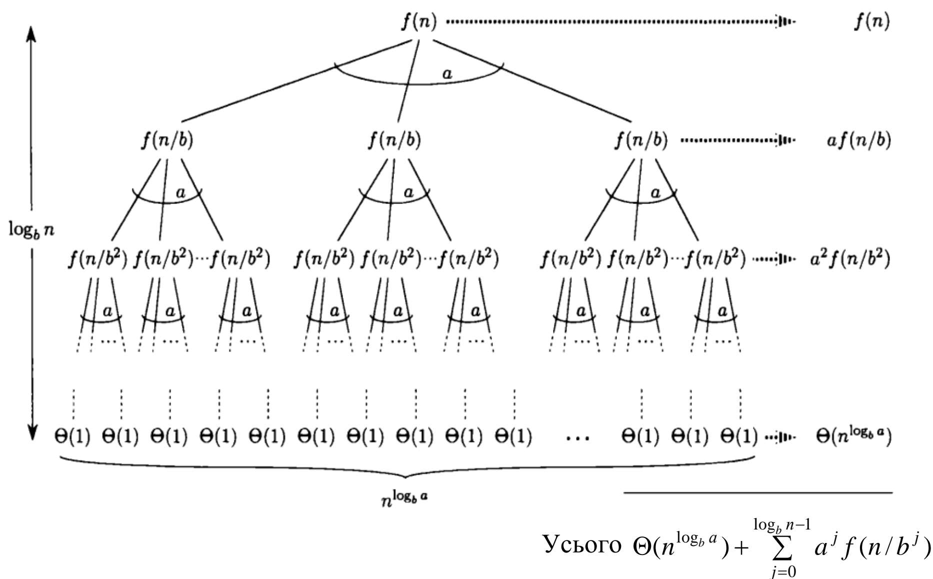


Рис. 5.1. Дерево рекурсії для співвідношення $T(n)=aT(n/b)+f(n)$ [2].

Дивлячись на формулювання теореми можна побачити, що в теоремі проводиться порівняння функції $f(n)$ з функцією, що вказує кількість листків. При порівняні (якщо функції можемо порівняти) може бути лише три випадки. З досвіду відомо, що при підрахуванні вартості всіх шарів дерева рекурсії різні шари мали різний вклад у

загальну вартість, а в загальній сумі оцінювали окремо останній шар, вартість якого складалася лише з вартості листків. Формулювання основної теореми ґрунтується на тому, що більша з функцій визначає величину рішення рекурентного співвідношення.

У **випадку 1** функція $n^{\log_b a}$ **поліноміально більша** функції $f(n)$, тому вона є визначальною і в результаті буде доведено: $T(n) = \Theta(n^{\log_b a})$. Для випадку, коли вона є **поліноміально меншою**, для **випадку 3**, найбільш вагомою є функція $f(n)$ і буде доведено: $T(n) = \Theta(f(n))$. Якщо ж вони є функціями **одного порядку**, **як у випадку 2**, матимемо $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Якщо функції не є порівнюваними, теорему застосовувати не можна.

Що значить: функція поліноміально менша? Це значить, що функція $f(n)$ має бути асимптотично меншою функції $n^{\log_b a}$ на множник n^{ε}, де $\varepsilon > 0$ – деяка стала. Аналогічно розуміється поняття поліноміально більшої функції: функція $f(n)$ має бути асимптотично більшою функції $n^{\log_b a}$ на множник n^{ε}, де $\varepsilon > 0$ – деяка стала. Але для випадку 3 в формулювання теореми додані умови регулярності $af(n/b) \le cf(n)$, що звужують можливість попадання у випадок 3. Їх необхідність стане ясною із доведення.

Дивлячись на побудоване дерево рекурсії, бачимо у всіх випадках порівняння наступне: у випадку 1 повна вартість дерева буде визначатися вартістю його листків, у випадку 2 вартість дерева буде рівномірно розподілена за всіма рівнями (шарами), у випадку 3 – визначатиметься вартістю кореня.

Отже, перейдемо до доведення теореми. Нехай спочатку маємо, що значення n є степенями b. Доведення поділимо на етапи, що будуть представлені в якості окремих тверджень.

Покажемо, що справедливе наступне твердження.

Лема. Нехай $a \ge 1$, $b > 1$ – сталі, $f(n)$ – асимптотично невід'ємна функція, в якій аргумент n є степенями b, функція $T(n)$ визначена на множині точних степенів b за допомогою рекурентного співвідношення вигляду

$$T(n) = \begin{cases} \Theta(1), \ n \le 1, \\ aT(\lfloor n/b \rfloor) + f(n), n > b^i, \end{cases}$$

де i – додатнє ціле число. Тоді

$$T(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) + \Theta(n^{\log_b a}).$$

Доведення. На дереві рекурсії маємо: $f(n)$ – вартість кореня дерева, що має a гілок з вузлами вартістю $f(n/b)$ кожен; нижче, на другому шарі, кожен вузол також має a гілок, їхня вартість буде $f(n/b^2)$, загальна кількість на шарі – a^2. На j-му шарі – a^j вузлів вартістю $f(n/b^j)$ кожен. На останньому, на глибині $\log_b n$, маємо листки вартістю $T(1)=\Theta(1)$ (визначаємо як $n \to n/b \to ... n/b^j \to ... \to 1$, листок досягаємо коли $(1/b)^k n = 1$ та $k = \log_b n$), їх – $a^{\log_b n} = n^{\log_b a}$, а їхня вартість –$\Theta(n^{\log_b a})$. Вартість всіх внутрішніх вузлів буде дорівнювати вартості всіх вузлів за кожним шаром, від 0 до $\log_b n - 1$: $\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$. Звідси і одержимо повну вартість:

$$T(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) + \Theta(n^{\log_b a}).$$

Оцінимо порядок зростання функції, що визначає вартість всіх рівнів дерева, крім останнього, листкового. За результатами сформулюємо наступне твердження.

Лема. Нехай $a \ge 1$, $b > 1$ – сталі, $f(n)$ – асимптотично невід'ємна функція, в якій аргумент n є степенями b, функція $g(n)$, визначена на множині точних степенів b за допомогою співвідношення вигляду

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j),$$

має такі асимптотичні границі для точних степенів b:

1. Якщо $f(n) = O(n^{\log_b a-\varepsilon})$ для деякої сталої $\varepsilon > 0$, то $g(n) = O(n^{\log_b a})$.
2. Якщо $f(n) = \Theta(n^{\log_b a})$, то $g(n) = \Theta(n^{\log_b a} \lg n)$.
3. Якщо $af(n/b) \le cf(n)$ для деякої сталої $c < 1$ й усіх достатньо великих n, то $g(n) = \Theta(f(n))$.

Доведення. Випадок 1. За умовою маємо $f(n) = O(n^{\log_b a-\varepsilon})$, тому $f(n/b^j) = O((n/b^j)^{\log_b a-\varepsilon})$ і тому функція $g(n)$ набуде вигляду $g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j (n/b^j)^{\log_b a-\varepsilon}\right)$.

Оцінимо цей вираз як зростаючу геометричну прогресію. Маємо:

$$\sum_{j=0}^{\log_b n-1} a^j (n/b^j)^{\log_b a-\varepsilon} = n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\varepsilon}{b^{\log_b a}} \right)^j = n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} \left(b^\varepsilon\right)^j =$$

$$= n^{\log_b a-\varepsilon} \cdot \frac{b^{\varepsilon \log_b n} - 1}{b^\varepsilon - 1} = n^{\log_b a-\varepsilon} \cdot \frac{n^\varepsilon - 1}{b^\varepsilon - 1},$$

де a, b, ε – сталі. Тому результуючу оцінку можна записати як $n^{\log_b a-\varepsilon} O(n^\varepsilon) = O(n^{\log_b a})$, що й дасть $g(n) = O(n^{\log_b a})$.

Випадок 2. За умовою $f(n) = \Theta(n^{\log_b a})$, тому $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$, а $g(n)$ виглядатиме як: $g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j (n/b^j)^{\log_b a} \right)$, де

$$\sum_{j=0}^{\log_b n-1} a^j (n/b^j)^{\log_b a} = n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}} \right)^j = n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 = n^{\log_b a} \log_b n.$$

Тоді матимемо, що $g(n) = \Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \lg n)$.

Випадок 3. Для n, що є точними степенями b, справедливе – $g(n) = \Omega(f(n))$. За умовою для деякої додатної сталої $c < 1$ та всіх доволі великих n виконується $af(n/b) \le cf(n)$. Запишемо цей вираз у вигляді $f(n/b) \le (c/a) f(n)$. Після j-ї ітерації (на j-му шарі) матимемо $f(n/b^j) \le (c/a)^j f(n)$, звідки можемо вказати про справедливість $a^j f(n) \ge c^j f(n/b^j)$ для доволі великих ітерованих значень, а також припущення, що n/b^{j-1} доволі велике. Використовуючи це, а також, що стала $c < 1$, у виразі для $g(n)$ прийдемо до наступного:

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \le \sum_{j=0}^{\log_b n-1} c^j f(n) + O(1) \le f(n) \sum_{j=0}^{\log_b n-1} c^j + O(1) =$$

$$= f(n) \frac{1}{1-c} + O(1) = O(f(n)),$$

де як O(1) вказано ті члени, що не охоплюються припущенням про доволі великі значення n. Отже, $g(n) = \Theta(f(n))$.

Тепер для значень n, що є точними степенями b, розглянемо результуюче твердження.

Лема. Нехай $a \ge 1$, $b > 1$ – сталі, $f(n)$ – асимптотично невід'ємна функція, в якій аргумент n є степенями b, функція $T(n)$ визначена на множині точних степенів b за допомогою рекурентного співвідношення вигляду

$$T(n) = \begin{cases} \Theta(1), \ n \le 1, \\ aT(\lfloor n/b \rfloor) + f(n), n > b^i, \end{cases}$$

де i – додатнє ціле число. Тоді $T(n)$ має такі асимптотичні границі для точних степенів b:

1. Якщо $f(n) = O(n^{\log_b a - \varepsilon})$ для деякої сталої $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$.
2. Якщо $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Якщо $f(n) = \Omega(n^{\log_b a + \varepsilon})$ для деякої сталої $\varepsilon > 0$ та $af(n/b) \le cf(n)$ для деякої сталої $c < 1$ й усіх достатньо великих n, то $T(n) = \Theta(f(n))$.

Доведення. Тепер просто складемо все попередньо одержане. Маємо:

- для випадку 1
  $T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a})$;
- для випадку 2
  $T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_b a} \lg n)$;
- для випадку 3, так як $f(n) = \Omega(n^{\log_b a + \varepsilon})$, то
  $T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n))$.

Отже, получимо справедливість формулювання основної теореми для значень n, що є степенями сталої b.

Лекція 6. Основна теорема для довільних значень *n*. Приклади розв'язання задач

Основна теорема [2].

Нехай $a \ge 1$, $b > 1$ – сталі, $f(n)$ – асимптотично невід'ємна функція, визначена на множині цілих невід'ємних чисел, а функція $T(n)$ визначена на множині цілих невід'ємних чисел за допомогою рекурентного

співвідношення $T(n) = aT(n/b) + f(n)$, де n/b інтерпретують як $\lfloor n/b \rfloor$ або $\lceil n/b \rceil$. Тоді $T(n)$ має такі асимптотичні границі:

1. Якщо $f(n) = O(n^{\log_b a-\varepsilon})$ для деякої сталої $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$.
2. Якщо $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Якщо $f(n) = \Omega(n^{\log_b a+\varepsilon})$ для деякої сталої $\varepsilon > 0$ та $af(n/b) \le cf(n)$ для деякої сталої $c < 1$ й усіх достатньо великих n, то $T(n) = \Theta(f(n))$.

Доведення. Зважаючи на результати, отримані для значень n, що є степенями числа b, одержимо подібне для довільних значень n. Для цього потрібно получити оцінки нижньої границі для виразу $T(n) = aT(\lceil n/b \rceil) + f(n)$ та верхньої для $T(n) = aT(\lfloor n/b \rfloor) + f(n)$.

Для визначення оцінки верхньої границі використаємо, що $\lceil n/b \rceil \ge n/b$, а дерево рекурсії модифікується як нижче вказано на рис. 6.1, де n_j :

$$n_j = \begin{cases} n, & j = 0, \\ \lceil n_{j-1}/b \rceil, & j > 0. \end{cases}$$

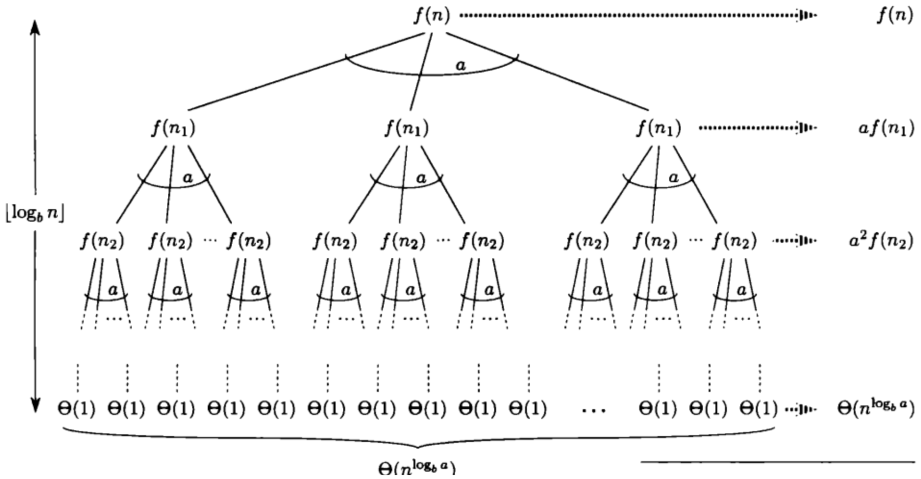
Глибина (висота) дерева обчислюється наступним чином: визначимо k таку, що n_j ціле, $k = \lfloor \log_b n \rfloor$.

Маємо (враховуючи, що $b > 1$):

$$n_0 = n,$$
$$n_1 = \lceil n/b \rceil \le n/b + 1,$$
$$n_2 = \lceil n_1/b \rceil \le n/b^2 + 1/b + 1,$$
$$n_3 = \lceil n_2/b \rceil \le n/b^3 + 1/b^2 + 1/b + 1,$$
$$\ldots,$$
$$n_j \le \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i} < \frac{n}{b^j} + \sum_{i=0}^{\infty} \frac{1}{b^i} = \frac{n}{b^j} + \frac{b}{b-1}.$$

Далі, для $j = \lfloor \log_b n \rfloor$:

$$
\begin{aligned}
&n\ ,\\
&\lceil n/b \rceil\ ,\\
&\lceil \lceil n/b \rceil /b \rceil\ ,\\
&\lceil \lceil \lceil n/b \rceil /b \rceil /b \rceil\ ,\\
&\quad\vdots
\end{aligned}
$$



$$\text{Усього}\ \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j)$$

Рис. 6.1. Дерево рекурсії для співвідношення $T(n)=aT(n/b)+f(n)$ за довільних n [2].

$$n_{\lfloor \log_b n \rfloor} < \frac{n}{b^{\lfloor \log_b n \rfloor}} + \frac{b}{b-1} < \frac{n}{b^{\log_b n - 1}} + \frac{b}{b-1} = \frac{n}{b^{\log_b n}/b} + \frac{b}{b-1} =$$

$$= \frac{n}{n/b} + \frac{b}{b-1} = b + \frac{b}{b-1} = O(1),$$

що дає також обмеженість задачі на шарі $j = \lfloor \log_b n \rfloor$ величиною $O(1)$. Кількість листків на нижньому шарі $\Theta(n^{\log_b a})$. Вартість усього дерева:

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j)\ .$$

Обчислимо вираз під знаком суми (як і раніше позначимо $g(n)$).

Випадок 3. У попередньому доведенні теореми було: з припущення леми для деякої додатної сталої $c < 1$ та всіх доволі великих n виконується $af(n/b) \le cf(n)$. Записували цей вираз у вигляді $f(n/b) \le (c/a) f(n)$.

Тут маємо: $af(\lceil n/b \rceil) \le cf(n)$ для $n > b + b/(b-1)$ та додатної сталої с < 1, тому $a^j f(n_j) \le c^j f(n)$ й суму обчислимо як і раніше, користуючись цією нерівністю. В результаті буде також $g(n) = \Theta(f(n))$.

Випадок 2. За умовами теореми $f(n) = \Theta(n^{\log_b a})$. У попередньому доведенні теореми з цього получали $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$ та знаходили значення суми. Покажемо, що тут $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$. Маємо: $j \le \lfloor \log_b n \rfloor$, тому $b^j/n \le 1$. Крім того, так як і $f(n) = O(n^{\log_b a})$, то існує додатна стала c така, що для достатньо великих n_j

$$f(n_j) \le c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} = c\left(\frac{n}{b^j}\left(1 + \frac{b^j}{n}\frac{b}{b-1}\right)\right)^{\log_b a} =$$

$$= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b^j}{n}\frac{b}{b-1}\right)^{\log_b a} \le c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} =$$

$$= O\left(\frac{n^{\log_b a}}{a^j}\right).$$

Випадок 1. За умовами теореми $f(n) = O(n^{\log_b a - \varepsilon})$, тому виконуватиметься $f(n/b^j) = O((n/b^j)^{\log_b a - \varepsilon})$ і $g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j (n/b^j)^{\log_b a - \varepsilon}\right)$. Далі оцінювали цей вираз через зростаючу геометричну прогресію. Тепер також покажемо, що $f(n_j) = O((n/b^j)^{\log_b a - \varepsilon})$.

$$\sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j O((n_j)^{\log_b a - \varepsilon}), \text{ де } n_j \le \frac{n}{b^j} + \frac{b}{b-1}.$$

$$\sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j O\left(\left(\frac{n}{b^j}\right)^{\log_b a - \varepsilon}\right) =$$

$$= n^{\log_b a - \varepsilon} O\left(\sum_{j=0}^{\lfloor \log_b n \rfloor - 1} (b^{\varepsilon})^j\right) = n^{\log_b a - \varepsilon} O\left(\frac{b^{\varepsilon \lfloor \log_b n \rfloor} - 1}{b^{\varepsilon} - 1}\right) =$$

$$= n^{\log_b a - \varepsilon} O(n^{\varepsilon}) = O(n^{\log_b a}).$$

Таким же чином для всіх трьох випадків одержується виконання нижніх границь для $T(n) = aT(\lfloor n/b \rfloor) + f(n)$.

Приклади застосування основної теореми, методу підстановки та методу дерев рекурсії.

Визначити верхню та нижню границі функції $T(n)$ для рекурентного співвідношення $T(n) = 2T(n/2) + n^4$. Вважати, що за $n \le 2$ $T(n)$ є сталою. Для зручності приймемо, що n є степенем 2.

1. Обчислимо методом підстановки. Припустимо, що верхня границя має вигляд $T(n) = O(n^4)$. Спробуємо показати, що за підходящого вибору додатної сталої c виконується $T(n) \le cn^4$.

   Припустимо справедливість нерівності для всіх додатних $m < n$, а саме для $m = \lfloor n/2 \rfloor$: $T(m) \le cm^4$.

   Покажемо справедливість для довільного $n > m$. Підставимо прогнозоване рішення до рекурентного виразу:

$$T(n) = 2c(n/2)^4 + n^4 = \frac{c}{2^3}n^4 + n^4 = c(\frac{1}{8} + \frac{1}{c})n^4 \le cn^4 \text{ для } c \ge 8/7.$$

   Для граничних умов маємо $T(n) = \Theta(1)$, $n \le 2$. Для $c \ge 8/7$ получимо $T(1) \le c1^4 = \Theta(1)$. Звідси випливає, що верхньою границею рекурентного співвідношення буде $O(n^4)$.

   Аналогічним чином показуємо, що $T(n) = \Omega(n^4)$, а тому $T(n) = \Theta(n^4)$.

2. Обчислимо методом дерев рекурсії.

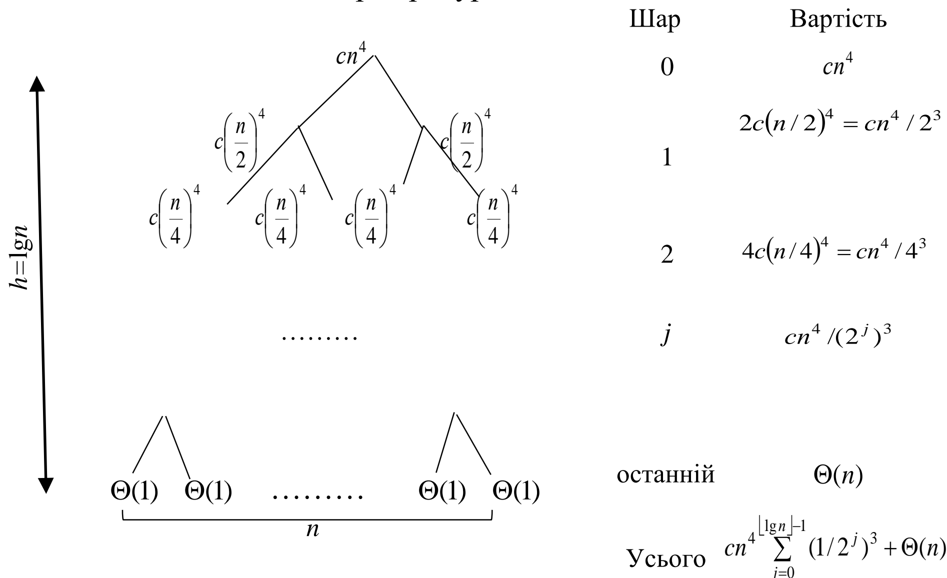


Рис. 6.2. Дерево рекурсії для співвідношення $T(n) = 2T(n/2) + n^4$.

Рекурентному співвідношенню відповідає повне бінарне дерево. Висота дерева – lg$n$, листків на нижньому шарі – $2^{\lg n} = n^{\lg 2} = n$.

З рисунку бачимо, що вартість кореневого шару cn^4, першого – $cn^4/2^3$, другого – $cn^4/(2^2)^3$, j-го – $cn^4/(2^j)^3$, а останнього – $\Theta(n)$ (вага кожного з n листків $\Theta(1)$).

Значення $T(n)$ дорівнює сумі добутків рівнів на їхню вартість, тому $T(n) = cn^4 + cn^4/2^3 + \ldots + cn^4/(2^j)^3 + \ldots + \Theta(n) = cn^4\left(\sum_{i=0}^{\lg n-1}\left(\frac{1}{8}\right)^3\right) + \Theta(n) =$

$$= cn^4\,\frac{(1/8)^{\lg n}-1}{(1/8)-1} + \Theta(n) \le cn^4\,\frac{1}{1-(1/8)} + \Theta(n) = \frac{8}{7}cn^4 + \Theta(n) = \Theta(n^4).$$

3. Застосуємо основну теорему. Маємо: $f(n)=n^4$, $n^{\log_b a} = n^{\lg 2} = n$, $f(n) = \Omega(n^{1+\varepsilon}), \varepsilon = 3$.

   Перевіримо умову регулярності: для c<1 та всіх достатньо великих n $2f(n/2) \le cf(n)$. Звідки $2(n/2)^4 \le cn^4$ справедливо для c =1/8. Отже, маємо випадок 3 основної теореми і можемо зробити висновок, що $T(n) = \Theta(n^4)$.

Визначити верхню та нижню границі функції $T(n)$ для рекурентного співвідношення $T(n) = T(7n/10)+n$. Вважати, що за $n \le 2$ значення $T(n)$ є сталою.

1. Застосуємо основну теорему. Маємо: $f(n) = n$, $n^{\log_b a} = n^{\log_{10/7} 1} = n^0 = 1$, $f(n) = \Omega(n^{0+\varepsilon}), \varepsilon = 1$.

   Перевіримо умову регулярності ($af(n/b) \le cf(n)$, c < 1 для всіх достатньо великих n): $7n/10 \le cn$, $c = 7/10 < 1$.

   Тому маємо випадок 3 основної теореми та можемо зробити висновок, що $T(n) = \Theta(n)$.

2. Обчислимо границю методом дерев рекурсії. Відповідне дерево має по одному вузлу на кожному рівні. Вартість кореневого рівня – cn, 1-го рівня – $(7/10)cn$, 2-го рівня – $(7/10)^2cn$, …, i-го рівня – $(7/10)^icn$. На останньому рівні маємо один листок вартістю $\Theta(1)$.

   Висота дерева $h = \lfloor \log_{10/7} n \rfloor$.

   Тому вартість всього дерева:

$$T(n) = cn + \frac{7}{10}cn + \left(\frac{7}{10}\right)^2 cn + \ldots + \Theta(1) = cn\sum_{i=0}^{\lfloor \log_{10/7} n \rfloor - 1}\left(\frac{7}{10}\right)^i + \Theta(n) =$$

$$= cn\frac{\left(\frac{7}{10}\right)^{\log_{10/7} n} - 1}{\frac{7}{10} - 1} + \Theta(n) \le cn\frac{1}{1 - \frac{7}{10}} + \Theta(n) =$$

$$= \frac{10}{3}cn + \Theta(n) = \Theta(n),$$

Що дозволяє зробити висновок: $T(n) = \Theta(n)$.

3. Обчислимо границі методом підстановки. Припустимо, що верхня границя має вигляд $T(n) = O(n)$.

   Спробуємо показати, що за підходящого вибору додатної сталої c виконується $T(n) \le cn$.

   Нехай справедливо для всіх додатних доволі великих $m < n$ та деякої додатної сталої c: $T(m) \le cm$.

   Покажемо справедливість гіпотези для довільного доволі великого значення $n > m$.

   $\mathrm{T}(n) \le c\frac{7}{10}n + n \le cn(\frac{7}{10} + \frac{1}{c}) \le cn,$ якщо $(\frac{7}{10} + \frac{1}{c}) \le 1$, а саме $7c + 10 \le 10c$, $10 \le 3c$, $c \ge 10/3$.

   Крайові умови, нехай, маємо $T(1) = 1$. Тоді $T(2) = T(\lfloor 14/10 \rfloor) + 2 = 3$. З іншого боку, $T(2) \le 2c$, що зважаючи на одержане вище значення c повністю відповідає вказаному, тому $c \ge 10/3$.

Лекція 7. Аналіз детермінованих алгоритмів сортування (що використовують порівняння)

Виконаємо аналіз алгоритму *сортування вставкою* масиву $A[1..n]$, який, як відомо, ефективно працює для невеликої кількості елементів. Алгоритм сортує вхідні дані на місці, обсяг використовуваної додаткової пам’яті в будь-який момент роботи не перевищує деякої сталої величини. В результаті роботи наведеної процедури Insertion-Sort [1] вхідний масив A містить вихідну відсортовану послідовність.

	Insertion-Sort (A)	Вартість	Кількість повторювань
1	**for** $j = 2$ **to** $A.length$	c_1	n
2	$key = A[j]$	c_2	$n - 1$
3	//вставка $A[j]$ до відсортованої послідовності $A[1..j\text{-}1]$	0	$n - 1$
4	$i = j - 1$	c_4	$n - 1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i + 1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i + 1] = key$	c_8	$n - 1$

Процедура містить цикл з індексом j, де елементи A[1.. j-1] відповідають відсортованій частині (до роботи процедури вони також знаходилися в цій частині), елементи A[j+1, n] – ще не відсортованій частині.

Для аналізу коректності алгоритму використаємо поняття інваріанта циклу. Для цієї процедури в якості **інваріанта циклу** виступає те, що у частині з елементами A[1.. j-1] на початку кожної ітерації циклу містилися ті елементи, що й на самому початку знаходилися у частині A[1.. j-1], але вже у відсортованому порядку. Для доведення коректності алгоритму на основі інваріанта циклу необхідно показати його наступні властивості: **ініціалізація** (справедливість інваріанта перед першою ітерацією циклу), **збереження** (те, що, якщо істинні перед першою ітерацією циклу, то залишаються справедливими й після ітерації, тобто виконуються після чергової ітерації), **завершення** (істинність по завершенню циклу, що підтвердить правильність алгоритму). Якщо виконуються перші дві властивості, то інваріанти циклу залишаються істинними перед кожною черговою ітерацією циклу, а третя дає коректність алгоритму. Покажемо виконання вказаних властивостей для циклу **for**.

Ініціалізація. Момент, коли перевіряємо справедливість інваріанта циклу перед першою ітерацією наступає одразу після початкового присвоєння значення індексу циклу та перед першою перевіркою у заголовку циклу, тому маємо присвоєне значення $j = 2$ та ще не виконану перевірку j. Продемонструємо справедливість інваріанта циклу перед першою ітерацією циклу ($j = 2$). Тоді підмасив A[1.. j-1] містить лише один елемент A[1], що зберігає своє початкове значення,

в такій пімножині всі елементи відсортовані. Тому властивість виконується.

Збереження. Продемонструємо, що інваріант циклу зберігається після кожної ітерації. Маємо, у зовнішньому циклі відбувається зсув елементів A[j-1], A[j-2], …, доки не знайдеться підходяще місце для A[j] (рядки 4-7) (вставка в це місце A[j] – рядок 8). Множина A[1.. j] після цього складається з елементів, що спочатку були в ній, але вже є відсортованою. Наступне збільшення j на новій ітерації циклу зберігає інваріант циклу.

Завершення. Розглянемо завершення циклу. Умова, що приводить до його завершення: $j > A.lenght = n$. Оскільки за кожною ітерацією циклу j збільшується на 1, то будемо мати $j = n+1$. Підставимо до формулювання інваріанта циклу таке значення, одержимо, що масив A[1.. n] складається з елементів, що спочатку знаходилися в A[1.. n], але вже відсортованих. A[1.. n] і є повним масивом, тому увесь масив відсортований та алгоритм є коректним.

Для проведення строгого аналізу алгоритму формулюють та розглядають інваріант циклу також і для внутрішнього циклу **while**.

В результаті проведеного аналізу алгоритму можна передбачити приблизну величину потрібних ресурсів (часу, пам'яті тощо) та, наприклад, порівняти ефективність алгоритмів для виконання однакової задачі. Використовуючи модель RAM, проведемо найпростіший аналіз за **часом роботи** алгоритму. Для нього застосуємо поняття **розміру вхідних даних** (в залежності від задачі, що розглядається, це може бути кількість вхідних елементів, кількість бітів для представлення вхідних даних, ін.). Час роботи алгоритму виміряють в кількості елементарних операцій (вважаємо, що одна операція – деякий фіксований час), які необхідно виконати для одержання кінцевого результату. Нехай i-й рядок виконується за деякий константний час c_i.

Час роботи усього алгоритму є сумою величин тривалості виконання кожної інструкції. Тому

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\sum_{j=2}^{n} t_j\right) + (c_6+c_7)\left(\sum_{j=2}^{n}(t_j-1)\right) + c_8(n-1).$$

Навіть якщо розмір вхідних даних є фіксованою величиною, час роботи алгоритму може залежати від самих вхідних даних (степені їхньої впорядкованості, структури тощо). В даному випадку найкращий випадок, коли елементи були спочатку у відсортованому виді. Тоді для

кожного j =2,3,…, n у рядку 5 $A[i] \le key$ ще коли i дорівнювало своєму початковому значенню j-1. Тому $t_j = 1$ для j =2,3,…, n, а час роботи для найкращого випадку є лінійною функцією від n.

У найгіршому випадку початково масив є відсортованим у порядку, оберненому до потрібного. Тут маємо порівнювати кожен A[j] з усіма елементами відсортованого підмасиву A[1..j-1], $t_j = j$ для j = 2,3,…, n, а час роботи є квадратичною функцією від n:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(\sum_{j=2}^{n} j) + (c_6+c_7)(\sum_{j=2}^{n}(j-1)) + c_8(n-1) =$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + (c_6+c_7)(\frac{n(n-1)}{2}) + c_8(n-1) =$$

$$= (c_5+c_6+c_7)n^2/2 + (c_1+c_2+c_4+ c_5/2 - c_6/2 - c_7/2 + c_8)n - (c_2+c_4+c_5+c_8) =$$

$$= a_2 n^2 + a_1 n + a_0.$$

Але час роботи алгоритму у найгіршому випадку дає верхню границю часу роботи алгоритму для довільних вхідних даних.

У середньому випадку час роботи має такий же характер, як і у найгіршому випадку. У середньому половина елементів підмасиву A[1..j – 1] менша від A[j], а половина – більша. Тоді у середньому маємо перевірити половину елементів підмасиву A[1..j – 1], звідси $t_j \approx j/2$

.

Часто досліджують середній час роботи алгоритму, тобто математичне очікування. Його будемо розглядати для рандомізованих алгоритмів.

Але наведеного алгоритму: у найгіршому випадку оцінка для часу роботи – $\Theta(n^2)$. Таку оцінку використовують, наприклад, для визначення більш ефективного алгоритму з декількох можливих алгоритмів для вирішення конкретної задачі. Зрозуміло, що ефективнішим є алгоритм, що має нижчий порядок в оцінці. Для однакових порядків необхідний більш докладний розгляд, який буде наступного семестру.

Отже, для оцінок наведеного алгоритму одразу одержали поліноміальні вирази. Рекурентні співвідношення получимо, наприклад, виконуючи сортування **методом декомпозиції** («розподіляй та володарюй»). Тут складна задача може поділятися на декілька більш простих, що подібні до початкової задачі, але меншого об’єму. Допоміжні підзадачі вирішують рекурсивно, а потім одержані рішення комбінуються для одержання рішення вихідної задачі. Тому виділяють **3 етапи**: **поділ** задачі на декілька підзадач, що є меншими екземплярами задачі;

володарювання над підзадачами шляхом їх рекурсивного вирішення; **комбінування** рішень підзадач в рішення початкової задачі.

Найпростішим прикладом застосування підходу «розподіляй та володарюй» є алгоритм *сортування злиттям*. В ньому поділяється задана послідовність на дві підпослідовності однакового розміру, далі рекурсивно сортуються ці підпослідовності (рекурсія досягає нижньої границі, коли довжина підпослідовності стає рівною 1), а потім – поєднують 2 відсортовані підпослідовності для одержання результату (для злиття використовують процедуру Merge(A, p, q, r), $p \le q < r$, елементи підмасивів $A[p..q]$ та $A[q+1..r]$ впорядковані, одержують відсортований масив $A[p..r]$, вимагає час виконання $\Theta(n), n = r - p + 1$).

Виконаємо аналіз роботи алгоритму *сортування злиттям* масиву $A[p..r]$ за наведеною нижче процедурою Merge [2]. В алгоритмі використовуються два допоміжних підмасиви L та R.

	Merge(A, p, q, r)	Вартість	Кількість повторювань
1	$n_1 = q - p + 1$	c_1	1
2	$n_2 = r - q$	c_2	1
3	$L[1..\, n_1 + 1]$, $R[1..\, n_2 + 1]$ – нові масиви		
4	**for** $i = 1$ **to** n_1	c_3	n_1
5	$L[i] = A[p + i - 1]$		
6	**for** $j = 1$ **to** n_2	c_4	n_2
7	$R[j] = A[q + j]$		
8	$L[n_1 + 1] = \infty$	c_5	1
9	$R[n_2 + 1] = \infty$	c_6	1
10	$i = 1$	c_7	1
11	$j = 1$	c_8	1
12	**for** $k = p$ **to** r		n
13	**if** $L[i] \le R[j]$		
14	$A[k] = L[i]$		
15	$i = i + 1$		
16	**else** $A[k] = R[j]$		
17	$j = j + 1$		

В рядку 1 процедури Merge обчислюється довжина підмасиву $A[p..q]$, у рядку 2 обчислюється довжина підмасиву $A[q+1..r]$. У рядку 3 створюються нові підмасиви $L[1..\, n_1 + 1]$, $R[1..\, n_2 + 1]$. У рядках 4-5 у циклі **for**

копіюється підмасив $A[p..q]$ до $L[1..n_1]$, рядках 6-7 у циклі **for** копіюється підмасив $A[q+1..r]$ до $R[1..n_2]$. У рядках 8-9 в останні комірки допоміжних масивів $L[1..n_1+1]$, $R[1..n_2+1]$ містять обмежувачі.
У рядках 10-17 виконується $n = r - p + 1$ базових кроків, що зберігають інваріант циклу. На початку кожної ітерації циклу **for** у рядках 12-17 підмасив $A[p..k-1]$ містить $k - p$ найменших елементів $L[1..n_1+1]$ та $R[1..n_2+1]$ у відсортованому порядку. Елементи $L[i]$ та $R[j]$ є найменшими елементами у своїх підмасивах, які ще не скопійовані назад до A. Сформульоване і виступає в якості інваріанту циклу.
Покажемо, що інваріант циклу виконується перед першою ітерацією циклу (рядки 12-17), що кожна ітерація його зберігає та він демонструється після закінчення роботи циклу, що дає коректність алгоритму.
Ініціалізація. Перед першою ітерацією циклу $k = p$, тому масив $A[p..k-1]$ пустий. В ньому 0 найменших елементів з $L[1..n_1+1]$ та $R[1..n_2+1]$, а так як $i = j = 1$, то $L[i]$ та $R[j]$ – найменші елементи своїх масивів, що ще не скопійовані до масиву A.
Збереження. Нехай $L[i] \le R[j]$. Покажемо, що інваріант циклу зберігається після кожної ітерації. Маємо $L[i]$ – найменший елемент, що ще не скопійований до A. Так як в масиві $A[p..k-1]$ міститься $k - p$ найменших елементів, після копіювання в рядку 14 $L[i]$ до $A[k]$ в $A[p..k]$ буде $k - p+1$ найменших елементів. Із збільшенням параметру k циклу та значення змінної i (рядок 15) інваріант циклу відновлюється перед наступною ітерацією. Якщо $L[i] < R[j]$, то отримаємо подібні висновки (рядки 16-17).
Завершення. Алгоритм завершується при $k = r + 1$. Відповідно до інваріанта циклу $A[p..k-1]$ ($A[p..r]$) містить $k - p = r - p + 1$ найменших елементів з масивів $L[1..n_1+1]$ та $R[1..n_2+1]$ у відсортованому порядку. Їх разом $n_1 + n_2 + 2 = r - p + 3$. Всі вони, крім двох найбільших, скопійовані вже до A, а два останніх є обмежувачами. Масив А[1.. n] є відсортованим та алгоритм є коректним.
Покажемо, що час роботи процедури Merge становить $\Theta(n), n = r - p + 1$. Рядки 1-3 та 8-11 виконуються за константний час, тривалість циклу у рядках 4-7 $\Theta(n_1 + n_2) = \Theta(n)$, цикл у рядках 12-17 має n ітерацій, кожна

з яких витрачає константний час. Все це разом дає $\Theta(n), n = r - p + 1$ для процедури Merge.

Процедура Merge потрібна для алгоритму сортування злиттям, що описується в процедурі Merge-Sort(A, p, r) [1] для сортування елементів підмасиву $A[p..r]$, де і відбувається поділ, в ході якого обчислюється індекс q, що поділяє $A[p..r]$ на два підмасива $A[p..q]$ (елементів $\lceil n/2 \rceil$) та $A[q+1..r]$ (елементів $\lfloor n/2 \rfloor$).

Merge-Sort(A, p, r)
1 **if** $p < r$
2 $q = \lfloor (p+r)/2 \rfloor$
3 Merge-Sort(A, p, q)
4 Merge-Sort(A, q+1, r)
5 Merge(A, p, q, r)

Проаналізуємо час роботи алгоритму. В алгоритмі задача поділяється на 2 підзадачі розміром $\lceil n/2 \rceil$ та $\lfloor n/2 \rfloor$. У загальному випадку, якщо на поділ задачі витрачаємо, наприклад, час $D(n)$, а на поєднання – $C(n)$, сортування одного елементу відбувається за константний час, то маємо наступне. На поділ витрачається $D(n) = \Theta(1)$. Далі маємо 2 підзадачі, час рішення яких $T(\lceil n/2 \rceil)$ та $T(\lfloor n/2 \rfloor)$. При комбінуванні процедура Merge при роботі з n елементним підмасивом витрачає час $C(n) = \Theta(n)$. Тому одержимо рекурентне співвідношення

$$T(n) = \begin{cases} \Theta(1), & n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n), & n > 1, \end{cases}$$

що, як вже знаємо, дає результуючу оцінку $\Theta(n \lg n)$ (загальна вартість відповідного дерева рекурсії: $cn \lg n + cn$).

Лекція 8. Побудова алгоритмів методом декомпозиції та їх аналіз. Алгоритм пошуку максимального підмасиву. Алгоритм Штрассена.Алгоритм порівняння ранжувань

В побудованому методом декомпозиції алгоритмі сортування злиттям не використовували попередніх допоміжних перетворень для побудови алгоритму. Наступний алгоритм – алгоритм пошуку максимального підмасиву, такі перетворення використовує (хоча в ньому теж відбувається поділ на дві підзадачі такого ж розміру. В якості прикладу з більшою кількістю однакових підзадач розглянемо алгоритм Штрассена.

Алгоритм пошуку максимального підмасиву. Побудова та аналіз. Розглянемо таку практичну ситуацію: можемо купувати пакети акцій в один день за деякою ціною та продавати в інший після закінчення торгів за іншою ціною, причому інформація про наступні ціни акцій відома [2]. Необхідно одержати найбільший прибуток на заданому інтервалі (конкретний інтервал із значеннями ціни наведений нижче).

День	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Ціна	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Зміна ціни		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Стратегія купівлі за найменшою ціною та продажу за найбільшою не завжди приводить до максимального прибутку. Наша задача знайти неперервну послідовність днів, для яких кінцева різниця між першим та останнім днем буде максимальною.

Якщо спробуємо одержати рішення задачі напряму, перебором, то для періоду з n днів матимемо C_n^2 пар, що дасть $\Theta(n^2)$, а якщо обчислення кожної пари займає константний час, то нижня оцінка – $\Omega(n^2)$.

Спробуємо побудувати алгоритм з оцінкою $o(n^2)$. Для цього використаємо перетворення даних. Так як шукаємо неперервну послідовність днів, для яких кінцева різниця між першим та останнім днем максимальна, то розглянемо щоденну зміну ціни, де зміна у день i є різницею між ціною торгів цього дня та попереднього. Тоді задача буде полягати в пошуку непустого неперервного підмасиву A, значення якого мають найбільшу суму. Такий масив називають **максимальним підмасивом**. Здається все одно треба перевірити C_n^2 підмасивів, але тут перебір може вимагати $\Theta(n^2)$. Для заданого інтервалу із значеннями ціни максимальним підмасивом буде підмасив $A[8..11]$ із сумою 43.

Розглянемо більш ефективне рішення задачі, яке одержимо за допомогою методу декомпозиції.

Нехай маємо підмасив *A*[*low..high*]. За технологією методу поділяємо його на дві однакові частини (за можливістю), а саме: знаходимо середню точку підмасиву, яку позначимо – *mid*, після цього розглядаємо два підмасиви *A*[*low.. mid*] та *A*[*mid+1..high*].

Будь-який неперервний підмасив *A*[*i..j*] масиву *A*[*low..high*], а тому і наш максимальний, може бути розташований лише в одному положенні з таких:

- повністю знаходитися в *A*[*low.. mid*];
- повністю знаходитися в *A*[*mid+1..high*];
- частково знаходитися в *A*[*low.. mid*], *A*[*mid+1..high*] та перетинати точку *mid.*

Максимальні підмасиви у випадках 1 та 2 можна знайти рекурсивно, так як ці підзадачі є екземплярами задачі пошуку максимального підмасиву меншого розміру.

Розглянемо як знайти максимальний підмасив у випадку 3, коли задача не є меншим екземпляром початкової внаслідок наявності обмеження щодо перетину точки *mid*.

Будь-який підмасив, який перетинає точку *mid* складається з двох підмасивів – *A*[*i.. mid*] та *A*[*mid+1.. j*], де $low \le i \le mid, mid < j \le high$. Тому будемо шукати максимальні підмасиви *A*[*i.. mid*] та *A*[*mid+1.. j*], а потім поєднаємо їх (процедура Find-Max-Crossing-Subarray [2]).

	Find-Max-Crossing-Subarray(*A*, *low*, *mid*, *high*)	Вартість	Кількість повторювань
1	*left-sum* = – ∞	c_1	1
2	*sum* = 0	c_2	1
3	**for** *i* = *mid* **downto** *low*		*mid-low*+1
4	*sum* = *sum* + *A*[*i*]		
5	**if** *sum* > *left-sum*		
6	*left-sum* = *sum*		
7	*max-left* = *i*		
8	*right-sum* = – ∞	c_3	1
9	*sum* = 0	c_4	1
10	**for** *j* = *mid* + 1 **to** *high*		*high-mid*
11	*sum* = *sum* + *A*[*j*]		
12	**if** *sum* > *right-sum*		
13	*right-sum* = *sum*		
14	*max-right* = *j*		

```
15    return (max-left, max-right, left-sum+right-sum)
```

В процедурі в рядках 1-7 відбувається пошук максимального підмасиву в лівому підмасиві *A*[*low*.. *mid*]. Рядки 1-2 виконують ініціалізацію змінних *left-sum* (зберігає найбільшу знайдену на поточний момент суму) та *sum* (зберігає суму елементів підмасиву *A*[*i*.. *mid*]). Так як працюємо з підмасивом, що містить *A*[*mid*], то у циклі **for** (рядки 3-7) робота починається з індексу *i* = *mid* та йде у напрямку до *low* й кожен підмасив, що розглядається у циклі, буде вигляду *A*[*i*.. *mid*]. Коли у рядку 5 для суми значень підмасиву *A*[*i*.. *mid*] знаходимо, що вона більша за *left-sum*, то у рядку 6 оновлюємо значення змінної *left-sum* та у рядку 7 оновлюємо змінну *max-left* (зберігає лівий індекс максимального підмасиву). У рядках 8-14 процедури Find-Max-Crossing-Subarray аналогічно відбувається пошук максимального підмасиву в підмасиві *A*[*mid+1*..*high*] від індексу *j* = *mid* + 1 до *high*, кожен підмасив, що розглядається у циклі **for**, буде вигляду *A*[*mid+1*..*j*]. Змінна *max-right* містить правий індекс максимального підмасиву. Рядок 15 повертає індекси *max-left* та *max-right*, які визначають максимальний підмасив, разом із сумою значень *left-sum+right-sum* цього підмасиву.
Покажемо: якщо підмасив *A*[*low*..*high*] містить *n* елементів, то процедура виконує роботу за час $\Theta(n)$. Дійсно, кожна ітерація кожного з циклів **for** вимагає $\Theta(1)$ часу, перший цикл (рядки 3-7) має *mid-low*+1 ітерацій, другий (рядки 20-14) – *high-mid.* Тому загальна кількість: *high-low*+1, що дорівнює *n*. Отже, маємо $\Theta(n)$.

Тепер розглянемо безпосередньо алгоритм декомпозиції, що застосовується до пошуку максимального підмасиву [2]. Рекурсивна процедура Find-Maximum-Subarray повертає кортеж, який складається з індексів, що визначають максимальний підмасив.

```
   Find-Maximum-Subarray(A, low, high)
1     if  high == low
2          return (low, high, A[low])// Базовий випадок
3     else  mid = [(low + high)/2]
4        (left-low, left-high, left-sum) = Find-Maximum-Subarray(A, low, mid)
5        (right-low, right-high, right-sum) =
                  Find-Maximum-Subarray(A, mid +1, high)
6        (cross-low, cross-high, cross-sum) =
                  Find-Maximum-Subarray(A, low, mid, high)
```

```
7      if  left-sum ≥ right-sum and left-sum ≥ cross-sum
8          return (left-low, left-high, left-sum)
9      elseif  right-sum ≥ left-sum and right-sum ≥ cross-sum
10         return (right-low, right-high, right-sum)
11     elsereturn (cross-low, cross-high, cross-sum)
```

Перевірка базового випадку (підмасив складається з одного елементу) – рядок 1, рядок 2 повертає індекси, початковий і кінцевий, разом із значенням елементу.

Обробка рекурсивного випадку – рядки 3-11, де рядки 4-5 «володарюють», рядки 6-11 дають комбінування.

Оцінимо вартість усіх рядків, враховуючи можливі повтори. Припустимо, що розмір задачі являє собою число, яке є точним степенем 2. Тоді розміри усіх підзадач є цілими числами. Маємо, перший рядок виконується за час $\Theta(1)$, значить для базового випадку $T(n)=\Theta(1)$. Якщо вимірність початкової задачі більша за 1, то маємо рекурсивне рішення задачі. Рядки 1,3 виконуються за константний час. У рядках 4,5 маємо підзадачі половинного розміру, тому для їх вирішення потрібно $T(n/2)$ часу для кожної окремо, а так як повинні розв'язати кожну з цих підзадач (лівий та правий підмасиви), то загальний час роботи для рядків 4,5 буде $2T(n/2)$ часу. У рядку 6 міститься виклик процедури Find-Max-Crossing-Subarray для вирішення 3 випадку, вже оцінений в $\Theta(n)$. Рядки 7-11 виконуються за константний час. Складаючи все разом та усуваючи припущення, що розмір задачі є числом, яке є точним степенем 2, одержимо :

$$T(n)=\Theta(1)+T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+\Theta(n)+\Theta(1)=$$
$$=T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+\Theta(n).$$

Тому повна оцінка буде:

$$T(n)=\begin{cases}\Theta(1), & n=1,\\ T(\lceil n/2\rceil)+T(\lfloor n/2\rfloor)+\Theta(n), & n>1.\end{cases}$$

Це дасть, як і попередньому алгоритмі, результуючу оцінку $\Theta(n\lg n)$.

Множення матриць, алгоритм Штрассена. Побудова та аналіз.

Стандартна процедура множення двох матриць А та В n-го порядку, де елементи результуючої матриці $c_{ij}=\sum_{k=1}^{n}a_{ik}b_{kj}$, $i,j=\overline{1,n}$, має оцінку часу

виконання $\Theta(n^3)$. Для алгоритму Штрассена оцінка часу виконання складає лише $\Theta(n^{\lg 7})$.

Розглянемо побудову алгоритму. Будемо вважати, що n є точним степенем 2. На кожному кроці поділимо матриці розміром $n \times n$ на 4 матриці розміром $n/2 \times n/2$. Нехай,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \ B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \ C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

що відповідає рівнянням:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

В кожному з цих рівнянь є два множення матриць порядку $n/2$ та додавання результатів множення цих матриць.

Рекурсивна процедура Square-Matrix-Multiply-Recursive, яка за наведеними рівняннями знаходить добуток матриць A та B [2], створює 12 матриць розміром $n/2 \times n/2$, що звичайно вимагає $\Theta(n^2)$ часу на копіювання елементів. В ній користуючись обчисленням індексів для вказівки підматриці витратимо $\Theta(1)$ часу.

Square-Matrix-Multiply-Recursive (A, B)
1 $n = A.rows$
2 C – нова матриця розміром $n \times n$
3 **if** $n == 1$
4 $c_{11} = a_{11}b_{11}$
5 **else** поділ A, B, C за формулами
6 C_{11} = Square-Matrix-Multiply-Recursive (A_{11}, B_{11}) + Square-Matrix-Multiply-Recursive (A_{12}, B_{21})
7 C_{12} = Square-Matrix-Multiply-Recursive (A_{11}, B_{12}) +

Square-Matrix-Multiply-Recursive (A_{12}, B_{22})
8 C_{21} = Square-Matrix-Multiply-Recursive (A_{21}, B_{11}) +
Square-Matrix-Multiply-Recursive (A_{22}, B_{21})
9 C_{22} = Square-Matrix-Multiply-Recursive (A_{21}, B_{12}) +
Square-Matrix-Multiply-Recursive (A_{22}, B_{22})
10 **return** *C*

Маємо, рядок 1-4 – $\Theta(1)$ часу; рядок 5 – $\Theta(1)$ часу; рядки 6-9 – 8 разів рекурсивно визиваємо процедуру, тому $8T(n/2)$ часу, та 4 рази складаємо матриці, що дасть ще $\Theta(n^2)$. Все це відповідатиме виразу

$$T(n) = \Theta(1) + 8T(n/2) + \Theta(n^2) = 8T(n/2) + \Theta(n^2) .$$

Тому повна оцінка буде:

$$T(n) = \begin{cases} \Theta(1), & n = 1, \\ 8T(n/2) + \Theta(n^2), & n > 1, \end{cases}$$

Що у результаті дасть оцінку $\Theta(n^3)$, яка не є кращою. В наведеній процедурі розтин кожної матриці за допомогою обчислення індексів вимагає $\Theta(1)$ часу, а розтинається 2 матриці. Складення двох матриць з k елементами – $\Theta(k)$, складення 4-х матриць з $n^2/4$ елементами наведеним алгоритмом – $\Theta(n^2)$. Але маючи 8 рекурсивних викликів процедури множник 8 вже не може бути поглиненим.

В алгоритмі Штрасена зменшена кількість рекурсивних викликів (їх вже 7). Ціною цього зменшення на 1 є декілька додаткових сумувань матриць $n/2 \times n/2$.

***Алгоритм Штрассена*:**

1. Поділити вхідні матриці A, B та вихідну C на підматриці розміром $n/2 \times n/2$ за вказаними вище формулами (крок можна виконати за час $\Theta(1)$ використовуючи обчислення індексів).
2. Створити 10 матриць $S_1, \ldots, S_{10}$ розміром $n/2 \times n/2$ кожна (крок можна виконати за час $\Theta(n^2)$):

$$S_1 = B_{12} - B_{22}$$
$$S_2 = A_{11} + A_{12}$$
$$S_3 = A_{21} + A_{22}$$
$$S_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{22}$$
$$S_6 = B_{11} + B_{22}$$
$$S_7 = A_{12} - A_{22}$$
$$S_8 = B_{21} + B_{22}$$
$$S_9 = A_{11} - A_{21}$$
$$S_{10} = B_{11} + B_{12}.$$

3. Використовуючи матриці $C_{ij}, A_{ij}, B_{ij},\ i, j = 1,2,$ та $S_1, \ldots, S_{10}$ рекурсивно обчислюються матричні добутки $P_1, \ldots, P_7$ (крок можна виконати за $7T(n/2)$ часу):

$$P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22}$$
$$P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22}$$
$$P_3 = S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11}$$
$$P_4 = A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11}$$
$$P_5 = S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}$$
$$P_6 = S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}$$
$$P_7 = S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}.$$

4. Обчислюються підматриці $C_{ij},\ i, j = 1,2,$ результуючої матриці C шляхом додавання та віднімання різних комбінацій матриць $P_1, \ldots, P_7$ (крок можна виконати за час $\Theta(n^2)$):

$$C_{11} = P_5 + P_4 - P_2 + P_6,$$
$$C_{12} = P_1 + P_2,$$
$$C_{21} = P_3 + P_4,$$
$$C_{22} = P_5 + P_1 - P_3 - P_7.$$

В результаті одержимо:

$$T(n) = \begin{cases} \Theta(1), & n = 1, \\ 7T(n/2) + \Theta(n^2), & n > 1, \end{cases}$$

що за основною теоремою дасть загальну оцінку $\Theta(n^{\lg 7})$.

Підрахунок інверсій. Задача порівняння двох ранжувань. Побудова та аналіз алгоритму.

Вимірювання схожості оцінок, ранжувань двох людей на числовому рівні є нетривіальним питанням. Зрозуміло, що ідентичні оцінки дуже схожі, а протилежні – сильно різняться. Тому потрібна деяка метрика для інтерполяції посеред шкали. Нехай необхідно провести ранжування деякого набору, наприклад, фільмів, помітимо їх від 1 до n. І такі ранжування надають декілька осіб. Отримані ранжування відрізняються між собою. Подивимося скільки пар порушує порядок. Отже, у ранжуванні маємо послідовність з n чисел $a_1, ..., a_n$, причому числа різні. Необхідно визначити метрику, що повідомляє наскільки список ухиляється від впорядкування за зростанням: значення метрики повинно бути 0, якщо $a_1 < a_2 < ...< a_n$, і має бути тим вищим, чим сильніше порушений порядок чисел.

Природній кількісний вираз цього поняття оснований на підрахунку інверсій (повне співпадіння 0, повне неспівпадіння – C_n^2). Можна перевірити кожну пару чисел напряму та визначити, чи утворюють вони інверсію, тоді перевірка потребує часу $O(n^2)$. Але можна за час $O(n\log n)$. Такий алгоритм має проводити підрахування інверсій без перевірки окремих інверсій. Для його побудови використаємо метод декомпозиції. Поділимо список на 2 частини: $a_1, ..., a_m$ та $a_{m+1}, ..., a_n$. Спочатку інверсії вираховуються по кожній половині окремо, а потім для двох чисел (a_i, a_j) з різних половин. Зробимо це за час $O(n)$. Під **інверсією** для пари (a_i, a_j) з різних половин розуміємо наступне: перше число з першої половини, друге з другої та $a_i > a_j$. Щоб спростити підрахунок інверсій між половинами, в алгоритмі також рекурсивно сортуються числа в двох половинах. Незначне зростання обсягу роботи на кроці рекурсії (сортування та підрахунок інверсій) спрощує частину алгоритму, де відбувається поєднання (злиття з підрахунком).

На момент проведення поєднуючого кроку маємо вже рекурсивно відсортовані обидві половини списку та підраховані інверсії в кожній половині, які позначимо A та B. Будемо поєднувати їх з побудовою відсортованого списку C, одночасно підрахувавши кількість пар (a, b), що утворюють інверсію. Це дуже схоже на сортування злиттям, але потрібно не тільки побудувати один відсортований список з A та B, а й ще підрахувати кількість пар з інверсією.

Процедура злиття Merge-and-Count з підрахунком інверсій проходить за відсортованими списками *A* та *B*, вилучаючи елементи з початку та приєднуючи їх до відсортованого списку *C*. На кожному конкретному кроці для кожного списку доступний вказівник *Current*, що позначає поточну позицію. Нехай ці вказівники вказують в деякий момент на елементи a_i, b_j. За один крок ми порівнюємо ці елементи, вилучаємо менший із списку та приєднуємо його у кінець списку *C*.
Як рахуємо кількість інверсій? Так як списки *A* та *B* відсортовані, то відслідковувати кількість інверсій дуже просто. Кожен раз, коли додаємо елемент a_i у кінець списку *C* нові інверсії не виникають, a_i менший за всі елементи, що залишилися у списку *B* та повинен передувати їм. Якщо елемент b_j приєднують до списку *C*, значить він менший за всі елементи, що залишилися у списку *A*, він повинен йти після них усіх, тому лічильник інверсій **збільшиться на кількість елементів, що залишилися в *A*** (кількість інверсій міститься в змінній *Count*).

Merge-and-Count(*A*, *B*)
Для кожного із списків *A*, *B* зберігати вказівник ***Current***, що ініціалізований вказівником на початковий елемент
Count = 0
while обидва списки *A*, *B* не пусті
    $\boldsymbol{a_i}$ та $\boldsymbol{b_j}$ – елементи, на які вказують вказівники ***Current***
    приєднати менший з них до вихідного списку *C*
    if меншим є елемент b_j
        збільшити ***Count*** на кількість елементів, що залишилися у списку *A*
    змістити вказівник ***Current*** у списку, де вибрали менший елемент
if один із списків пустий
    приєднати залишок другого списку до вихідного списку *C*
return (***Count***, *C*)

Проаналізуємо час виконання алгоритму злиття з підрахунком. Кожна ітерація алгоритму виконується за константний час. Більш того за кожної ітерації до вихідного списку додається елемент, що виключається з подальшого розгляду. Тому кількість ітерацій не може перевищити суми довжин списків *A* та *B*, звідки випливає, що загальний час – $O(n)$. Процедура Merge-and-Count використовується в рекурсивному алгоритмі (процедура Sort-and-Count), який одночасно сортує та підраховує кількість інверсій у списку *C*.

Sort-and-Count(C)
if список містить один елемент
    повертаємо список, інверсій немає
else
    поділити список на дві половини:
        А містить перші $\lceil n/2 \rceil$ елементів
        В містить останні $\lfloor n/2 \rfloor$ елементів
    (rA, A) = **Sort-and-Count**(A)
    (rB, B) = **Sort-and-Count**(B)
    (r, C) = **Merge-and-Count**(A, B)
return ($r = rA + rB + r$, відсортований список C)

Наведений алгоритм вірно сортує вхідний список та підраховує кількість інверсій. Оцінкою часу роботи такого алгоритму для списку з n елементами буде $O(n\log n)$.

Лекція 9. Алгоритм знаходження згортки векторів. Аналіз алгоритму пірамідального сортування

Швидке перетворення Фур'є. Згортка векторів. Побудова та аналіз алгоритму.

Задані вектори $a = (a_0, a_1, ..., a_{n-1})$ та $b = (b_0, b_1, ..., b_{n-1})$.
Знайти згортку цих векторів, тобто $a * b = \sum_{\substack{(i,j): i+j=k \\ i,j<n}} a_i b_j$

(у загальному випадку для векторів різної довжини $a = (a_0, a_1, ..., a_{m-1})$ та $b = (b_0, b_1, ..., b_{n-1})$: $a * b = \sum_{\substack{(i,j): i+j=k \\ i<m, j<n}} a_i b_j$).

У розгорнутому вигляді вираз для знаходження згортки можна записати як:

$$a * b = (a_0 b_0, a_0 b_1 + a_1 b_0, a_0 b_2 + a_1 b_1 + a_2 b_0, \ldots, a_{n-2} b_{n-1} + a_{n-1} b_{n-2}, a_{n-1} b_{n-1}).$$

Щоб одержати такий вираз дуже зручно користуватися наступною матрицею:

$$
\begin{array}{c}
a_0b_0 \;\; a_0b_1 \;\ldots\; a_0b_{n-2} \;\; a_0b_{n-1} \\
a_1b_0 \;\; a_1b_1 \;\ldots\; a_1b_{n-2} \;\; a_1b_{n-1} \\
a_2b_0 \;\; a_2b_1 \;\ldots\; a_2b_{n-2} \;\; a_2b_{n-1} \\
\ldots\ldots\ldots\ldots\ldots \\
a_{n-1}b_0 \;\; a_{n-1}b_1 \;\ldots\; a_{n-1}b_{n-2} \;\; a_{n-1}b_{n-1}
\end{array}
$$

Якщо розглядати багаточлени $A(x)$ та $B(x)$ з відповідники коефіцієнтами, то згортка – це добуток двох багаточленів, а саме – вектор коефіцієнтів результуючого багаточлена $C(x)$.
Для обчислення згортки напряму потрібно $\Theta(n^2)$ арифметичних операцій. Методом декомпозиції, що використовує швидке перетворення Фур'є, можна обчислити згортку за $O(n\log n)$.
Отже, нехай задані вектори $a = (a_0, a_1, ..., a_{n-1})$ та $b = (b_0, b_1, ..., b_{n-1})$. Вважаємо їх за коефіцієнти відповідних багаточленів $A(x)$ та $B(x)$. Розглянемо їх при множенні як функції змінної x.
Спершу наведемо **простий алгоритм обчислення згортки** [3].
1. Виберемо $2n$ значень $x_1, x_2, ..., x_{2n}$ та обчислимо $A(x_j)$ та $B(x_j)$ для кожного $j = 1, 2, ..., 2n$.
2. $C(x_j)$ обчислимо для кожного j як добуток двох чисел $A(x_j)$ та $B(x_j)$.
3. Відновимо C по значенням $x_1, x_2, ..., x_{2n}$. Для цього використаємо властивість: будь-який багаточлен степені d може бути поновлений за своїми значеннями для довільного набору з $d + 1$ та більше точок (поліноміальні інтерполяція). Для багаточленів A та B, степінь яких не перевищує $n - 1$, степінь добутку C не перевищує $2n - 2$, що дозволяє відновити багаточлен по значенням $C(x_1), C(x_2), ..., C(x_{2n})$, обчисленим на кроці 2.
Проаналізуємо вартість виконуваних операцій в алгоритмі. Крок 2 – вимагає $O(n)$ арифметичних операцій, оскільки задіяно множення $O(n)$ чисел. Кроки 1 та 3 мають обчислення багаточленів A і B для одного значення, що вимагає $\Omega(n)$ операцій, а потрібно зробити $2n$ таких обчислень. В результаті маємо $\Omega(n^2)$.
Для зменшення порядку оцінки необхідно знайти множину з $2n$ значень $x_1, x_2, ..., x_{2n}$, які зв'язані визначеним чином – для них робота з обчислення A і B може бути повторно використана в різних обчисленнях. Множиною, для якої це відбувається, є множина комплексних коренів з одиниці. Для чисел $x_1, x_2, ..., x_{2n}$ виберемо комплексні корені $(2n)$-ї

степені з одиниці. В результаті зможемо створити р*екурсивну процедуру побудови багаточлена A* для кожного з коренів $(2n)$-ї степені з одиниці, яка приведе до оцінки $T(n) \leq 2T(n/2) + O(n)$, де $T(n)$ – кількість операцій, необхідних для обчислення багаточлена степені $n - 1$ для всіх коренів $(2n)$-й степені з одиниці. Вважатимемо, що величина n є степенем 2.

Визначимо два багаточлена Aeven(x) та Aodd(x), що складаються з парних та непарних коефіцієнтів багаточлена A. Це, відповідно:

$$\text{Aeven}(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{(n-1)/2},$$

$$\text{Aodd}(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{(n-2)/2}.$$

Маємо, $\text{A}(x) = \text{Aeven}(x^2) + x\text{Aodd}(x^2)$.

Припустимо, що кожен з багаточленів Aeven та Aodd обчислюється для коренів n степені з одиниці. Ця задача в точності відповідає тій, з якою стикалися для багаточлену A та коренів $(2n)$-ї степені з одиниці, за виключенням того, що вхідні дані зменшилися вдвічі: степінь вже $(n - 2)/2$ замість $n - 1$, а замість $2n$ використовується n коренів. Тому ці обчислення мають оцінку за часом величина $\text{T}(n/2)$ для кожного з багаточленів Aeven та Aodd.

Залишилося лише провести обчислення A для коренів $(2n)$-ї степені з одиниці з використанням $\text{O}(n)$ додаткових операцій. Розглянемо один з коренів з одиниці $\omega_{j,2n} = e^{2\pi ji/2n}$. Величина дорівнює $(e^{2\pi ji/2n})^2 = e^{2\pi ji/n}$, а тому є коренем n-ї степені з одиниці. Таким чином, коли переходимо до обчислень $A(\omega_{j,2n}) = \text{Aeven}(e^{2\pi ji/n}) + \text{Aodd}(e^{2\pi ji/n})$, маємо, що обидва обчислення в правій частині були виконані на кроці рекурсії та $A(\omega_{j,2n})$ можна визначити з постійною кількістю операцій. Тому виконання цих операцій для всіх $2n$ коренів з одиниці означає $\text{O}(n)$ додаткових операцій після двох рекурсивних викликів, а границя для кількості операцій $T(n)$ задовольняє $T(n) \leq 2T(n/2) + O(n)$. Та ж процедура застосовується для обчислення багаточлена B для коренів $(2n)$-ї степені з одиниці, й це вже дає границю $\text{O}(n\log n)$ для кроку 1 загальної структури алгоритму.

Далі, маємо обчислення багаточленів A і B для множини коренів $(2n)$-ї степені з одиниці з використанням $O(n\log n)$ операцій та добутку $C(\omega_{j,2n}) = A(\omega_{j,2n})B(\omega_{j,2n})$ за $O(n)$ додаткових операцій.

І нарешті необхідно виконати крок 3 з використанням $O(n\log n)$ операцій для реконструкції багаточлену C із значень коренів $(2n)$-ї степені з одиниці. Задача реконструкції C вирішується визначенням відповідного багаточлена та його обчисленням для коренів $(2n)$-ї степені з

одиниці. Це вже вміємо робити за O($n$log$n$) операцій, тому ще виконуємо O($n$log$n$) операцій.

Розглянемо реконструкцію багаточлена C. Нехай, $C(x)=\sum_{s=0}^{2n-1} c_s x^s$ та його потрібно реконструювати по значенням $C(\omega_{s,2n})$ в коренях ($2n$)-ї степені з одиниці. Визначимо новий багаточлен $D(x)=\sum_{s=0}^{2n-1} d_s x^s$, де d_s = C($\omega_{s,2n}$). Розглянемо значення $D(x)$ коренів ($2n$)-ї степені з одиниці:

$$D(\omega_{j,2n}) = \sum_{s=0}^{2n-1} C(\omega_{s,2n})\omega_{j,2n}^s = \sum_{s=0}^{2n-1}(\sum_{i=0}^{2n-1} c_i \omega_{s,2n}^i)\omega_{j,2n}^s =$$
$$= \sum_{i=0}^{2n-1} c_i (\sum_{s=0}^{2n-1} \omega_{s,2n}^i \omega_{j,2n}^s).$$

Маємо $\omega_{s,2n} = (e^{2\pi i/2n})^s$. Використовуючи це та розширюючи запис до $\omega_{s,2n} = (e^{2\pi i/2n})^s$ навіть коли $s \geq 2n$, одержимо:

$$D(\omega_{j,2n}) = \sum_{l=0}^{2n-1} c_l (\sum_{s=0}^{2n-1} e^{2\pi i(sl+js)/2n}) = \sum_{l=0}^{2n-1} c_l (\sum_{s=0}^{2n-1} \omega_{l+j,2n}^s).$$

Проаналізуємо одержане. Використовуючи, що для кожного кореня ($2n$)-ї степені з одиниці $\omega \neq 1$ має місце $\sum_{s=0}^{2n-1} \omega^s = 0$ (випливає з того, що ω за визначенням є коренем рівняння $x^{2n} - 1 = 0$, а так як $x^{2n} - 1 = (x-1)(\sum_{s=0}^{2n-1} x^s)$ та $\omega \neq 1$, то маємо, що ω є коренем $(\sum_{s=0}^{2n-1} x^s)$. Тому єдина складова зовнішньої суми, що не є 0, відноситься до c_t, для якого $\omega_{t+j,2n} = 1$; а це буде, коли $t + j$ є кратними $2n$, тобто якщо $t = 2n - j$. Для цього значення $\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s = \sum_{s=0}^{2n-1} 1 = 2n$. Звідси $D(\omega_{j,2n}) = 2nc_{2n-j}$. Тому обчислення багаточлена $D(x)$ у коренях ($2n$)-ї степені з одиниці дає коефіцієнти багаточлена $C(x)$ у зворотному порядку та помножені на $2n$.

Отже, для будь-якого багаточлена $C(x)=\sum_{s=0}^{2n-1} c_s x^s$ та відповідного багаточлена $D(x) = \sum_{s=0}^{2n-1} C(\omega_{s,2n})x^s$ виконується умова $c_s = \frac{1}{2n} D(\omega_{2n-s,2n})$. Усі ж

обчислення для наведеного алгоритму з використанням методу декомпозиції виконуються за $O(n\log n)$ арифметичних операцій.

Пірамідальне сортування. Побудова та аналіз алгоритму.

Розглянемо алгоритм сортування з використанням такої структури даних, як бінарна піраміда, а саме незростаюча піраміда. В даному алгоритмі сортування відбувається без залучення додаткової пам'яті (тільки використовується для зберігання ззовні масиву деяка стала кількість елементів).

Бінарна піраміда(двійкова купа) – об'єкт-масив, який можна розглядати як майже повне бінарне дерево. Кожен вузол цього дерева відповідає певному елементу масиву. На всіх рівнях, крім, можливо, останнього, дерево є заповненим (заповнений рівень має максимально можливу кількість вузлів). Останній рівень заповнюється послідовно зліва направо доки в масиві не скінчаться елементи.

Незростаюча піраміда – бінарна піраміда, в якій для кожного вузла крім кореневого виконується: найбільший елемент масиву A знаходиться в корені дерева та значення кожного вузла з індексом i, що не є кореневим, не перевищує значення батьківського вузла, $A[Parent(i)] \geq A[i]$. Тому у незростаючій піраміді значення вузлів піддерева, яке бере початок у деякому елементі, не перевищує значення самого цього елементу.

Масив A, що представляє піраміду, є об'єктом з двома атрибутами: *A.length* (вказує кількість елементів в масиві) та *A.heap-size* (вказує скільки елементів піраміди міститься в масиві A).

Для підтримки незростаючої піраміди використаємо процедуру Max-Heapify [2], в якій на вході маємо масив A та індекс i в цьому масиві. Вважаємо, що бінарні дерева з коренями Left(i) та Right(i) є незростаючими пірамідами, але A[i] може бути меншим за значень у дочкових вузлах (може відбутися порушення властивості незростаючої піраміди). Тому процедура спускає значення A[i] вниз по незростаючій піраміді так, що піддерево з кореневим елементом з індексом i підкорюється властивості незростаючої піраміди.

Процедура Max-Heapify має наступний вигляд [2]:

```
   Max-Heapify(A, i)
1      l = Left(i)
2      r = Right(i)
```

```
3    if l ≤ A.heap-size та A[l] > A[i]
4        lagest = l
5    else lagest = i
6    if r ≤ A.heap-size та A[r] > A[lagest]
7        lagest = r
8    if lagest ≠ i
9        Обміняти A[i] та A[lagest]
10       Max-Heapify(A, lagest)
```

В наведеній процедурі на кожному кроці визначається найбільший з елементів A[i], A[Left (i)], A[Right (i)], а його індекс зберігається у змінній *largest*. Якщо найбільшим є A[i], то піддерево з коренем i вже є коректною незростаючою пірамідою та процедура завершує роботу. Інакше найбільшим буде один з двох дочкових елементів і процедура виконує обмін A[i] з A[*largest*], що приводить до того, що для вузла i та його дочкових вузлів виконується властивість незростаючої піраміди. Але тепер початкове значення A[i] вже у вузлі з індексом *largest*, тому піддерево з коренем *largest* може порушувати властивість незростаючої піраміди й тому необхідно рекурсивно викликати процедуру вже для цього піддерева.

Для роботи цієї процедури на піддереві розміром n з коренем у заданому вузлі i потрібен час $\Theta(1)$ для виправлення відношень між елементами A[i], A[Left (i)], A[Right (i)] та час роботи цієї поцедури з піддеревом, корінь якого знаходиться у одному з дочкових вузлів вузла i. Розмір таких дочкових піддерев не перевищує $2n/3$, причому найгірший випадок, якщо останній рівень заповнений наполовину. Тому час виконання процедури можна оцінити наступним виразом:

$$T(n) \leq T(2n/3) + \Theta(1),$$

що за випадком 2 основної теореми дасть $T(n)=O(\lg n)$.

Для побудови піраміди використаємо знову процедуру Max-Heapify у висхідному напрямку [2], щоб перетворити масив A[1..n], n=A.*lenght*, у незростаючу піраміду. Так як елементи в підмасиві $A[(\lfloor n/2 \rfloor + 1)..n]$ є листками дерева, то кожен з них можна вважати одноелементною пірамідою, з якої можна почати процес побудови. Для цього застосуємо процедуру Build-Max-Heap, що проходить по іншим вузлам та для кожного з них виконує процедуру Max-Heapify.

```
Build-Max-Heap(A)
1    A.heap-size = A.length
2    for i = ⌊A.lenght/2⌋ downto 1
3        Max-Heapify(A, i)
```

Покажемо коректність процедури Build-Max-Heap, для чого використаємо наступний інваріант циклу: на початку кожної ітерації циклу **for** у рядках 2-3 кожен вузол $i+1$, $i+2$, … n є коренем незростаючої піраміди.

Покажемо, що інваріант циклу справедливий перед першою ітерацією циклу, зберігається за кожної ітерації та є справедливим після завершення, що й дозволяє продемонструвати коректність алгоритму.

Ініціалізація. Перед першою ітерацією циклу $i=\lfloor n/2 \rfloor$. Всі вузли з індексами $\lfloor n/2 \rfloor+1$, $\lfloor n/2 \rfloor+2$, …, n є листками, тому кожен з них є коренем тривіальної незростаючої піраміди.

Збереження. Маємо, вузли, що є дочковими по відношенню до деякого вузлу i, мають номери більші за i. Відповідно до інваріанту циклу ці обидва вузли є коренями незростаючих пірамід. А це умова, що потрібна для виклику процедури Max-Heapify(A, i), щоб перетворити вузол з індексом i в корінь незростаючої піраміди. При її виклику зберігається властивість піраміди, тобто всі вузли з індексами $i+1$, $i+2$, … n є коренями незростаючих пірамід. Зменшення індексу i у циклі **for** забезпечує виконання інваріанту циклу для наступної ітерації.

Завершення. Після завершення циклу маємо: $i = 0$. Відповідно до інваріанту циклу всі вузли з індексами 1, 2, …, n є коренями незростаючих пірамід. Таким коренем є і вузол з індексом 1.

Одержимо верхню оцінку роботи процедури. Вище показали, що кожен виклик процедури Max-Heapify витрачає $O(\lg n)$ часу, а таких викликів маємо $O(n)$. Тому разом буде $O(n \lg n)$. Але така оцінка не є асимптотично точною.

Оцінимо більш точно. Звернемо увагу, що час виконання процедури Max-Heapify залежить від висоти вузла i, а більшість вузлів знаходиться не високо. Враховуючи, що висота піраміди є $\lfloor \lg n \rfloor$ та що на будь-якому шарі висоти h маємо не більше $\left\lceil n/2^{h+1} \right\rceil$ вузлів, час роботи Max-Heapify при роботі з вузлом, що розташований на висоті h є $O(h)$. Тому загальна вартість процедури Build-Max-Heap обмежена зверху

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil h/2^h \rceil \right).$$

Розглянемо праву частину виразу. Маємо:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil h/2^h \rceil \leq \sum_{h=0}^{\infty} \lceil h/2^h \rceil = \frac{1/2}{(1-1/2)^2} = 2 \cdot$$

А тому час роботи процедури Build-Max-Heap матиме **верхню границю O(*n*)**.

Алгоритм пірамідального сортування:

1. Виклик процедури для побудови незростаючої піраміди із вхідного масиву.
2. Кореневий елемент, як найбільший, можемо розташувати у коректній вихідній позиції у відсортованому масиві, міняємо його місцями з елементом A[n], викидаємо вузол n, зменшивши на 1 величину A.heap-size (дочкові піддерева кореня залишаються коректними незростаючими пірамідами, тільки корінь може порушувати властивість).
3. Поновлюємо властивість незростаючої піраміди викликом процедури Max-Heapify(A,1), масив A[1..n-1] стає незростаючою пірамідою.
4. Алгоритм повторюється для незростаючих пірамід розміром n - 1, n - 2, .., 2.

Наведеному алгоритму відповідає процедура Heapsort [2]:

```
   Heapsort(A)
1     Build-Max-Heap
2     for  i = A.length downto  2
3           Обміняти A[1] з A[i]
4           A.heap-size = A.heap-size – 1
5        Max-Heapify(A, 1)
```

Оцінимо час виконання наведеної процедури. Маємо: виклик процедури Build-Max-Heap вимагає часу O(n); кожен з n-1 викликів процедури Max-Heapify вимагає часу O(lg$n$); n – 1 разів константний час у рядку 4. Тому загальна оцінка часу роботи процедури Heapsort буде $O(n\lg n)$.

Лекція 10. Рандомізовані алгоритми та їх аналіз

Розглянемо задачу про найм, в якій потрібно оцінити вартість найму найкращого працівника За умовами задачі: роботодавець бажає узяти на роботу працівника, кадрова агенція надсилає кандидатів по черзі на співбесіду, після якої одразу приймається рішення наймати працівника чи ні. За кожного кандидата роботодавцю потрібно заплатити агенції деякі кошти, у випадку найму плата зростає. Крім того, йому потрібно звільнити працюючого менеджера. Так як мета роботодавця, щоб на посаді постійно знаходився найкваліфікованіший працівник, то як тільки кваліфікація кандидата є вищою, його наймають, а менеджера звільняють.

Нехай всі n кандидатів на посаду менеджера пронумеровані від 1 до n та одразу після співбесіди з i-м кандидатом можна визначити, чи є він найкращим. Відповідна найпростіша процедура має вигляд, якщо початковий кандидат (менеджер) для порівняння має номер 0:

```
   Hire-Assistant(n)
1  best = 0
2  for i =1 to n
3        Співбесіда з кандидатом i
4        if кандидат i кращий кандидата best
5             best = i
6             найняти кандидата i
```

Нехай вартість співбесіди з кандидатом – c_i, вартість найму – c_h, кількість найнятих працівників – m. Інтерв'ю проводять з усіма n кандидатами. Тоді повна вартість найму $O(c_i n + c_h m)$. В цій вартості змінною величиною є $c_h m$, яку й потрібно оцінити.

У найгіршому випадку, коли кандидати приходять в порядку зростання кваліфікації, наймають кожного кандидата, з яким була співбесіда. Тоді вартість найму буде $O(c_h n)$. Але насправді порядок невідомий і ніхто на нього не впливає. Тому розглянемо середній випадок та використаємо *ймовірнісний аналіз*, для якого потрібно знати *інформацію про розподіл вхідних даних*. В результаті такого аналізу одержимо

оцінку у середньому випадку (а для оцінки алгоритму – час **роботи у середньому випадку**).

Припустимо, що кандидати приходять на співбесіду у *випадковому порядку*. Нехай можемо порівняти кваліфікацію двох довільних кандидатів та визначити хто з них більш кваліфікований (тобто множину кандидатів можна впорядкувати). Тоді кожному кандидату можемо надати особистий ранг (число від 1 до n). Нехай ранг i-го кандидата – rank(i), причому кандидату з вищою кваліфікацією відповідає вищий ранг. Тоді впорядкована множина $\langle \text{rank}(1), \text{rank}(2), \ldots, \text{rank}(n) \rangle$ є перестановкою множини $\langle 1,2, \ldots, n \rangle$. А твердження, що кандидати приходять на співбесіду у довільному порядку еквівалентне до твердження, що ймовірність будь-якого порядку рангів однакова та усього кількість можливих перестановок є n! Тобто ранги утворюють випадкову рівномірну перестановку, а саме – кожна з усіх n! можливих перестановок з'являється з однаковою ймовірністю.

Обов'язковим моментом для розробки *рандомізованого алгоритму* є *відомості про розподіл вхідних даних*.

В даній задачі з цією метою внесемо зміни до моделі. Маємо, кадрова агенція обрала n кандидатів. Роботодавець домовляється, щоб повний список кандидатів йому надіслали наперед, а далі сам випадково обирає конкретні кандидатури для чергової співбесіди. Звичайно алгоритм вважають **рандомізованим**, якщо його поведінка визначається не тільки набором вхідних величин, а й значеннями, що видає генератор випадкових величин (нехай Random(a,b) – генератор дискретних випадкових чисел з інтервалу (a,b), всі числа рівноймовірні). Аналіз роботи такого алгоритму дає математичне очікування часу роботи. Про *час роботи рандомізованого алгоритму* кажуть як про *очікуваний час роботи* (в загальному випадку кажуть про час роботи у середньому випадку, коли випадковим чином розподілені вхідні дані алгоритму, та про очікуваний час роботи, коли випадковий вибір робить сам алгоритм).

Важливу роль для аналізу таких алгоритмів має індикаторна випадкова величина, яка дозволяє легко переходити від ймовірності до математичного очікування. Нехай маємо простір вибірки (подій) S та подію A. Тоді **індикаторна випадкова величина**, пов'язана з подією A, це

$$\mathrm{I}\{\mathrm{A}\}=\begin{cases}1, \text{подія А відбулася,}\\0, \text{подія А не відбулася.}\end{cases}$$

Основною властивістю індикаторної випадкової величини $X_A = I\{\mathrm{A}\}$ є: математичне очікування індикаторної випадкової величини дорівнює ймовірності події, пов'язаної з нею, тобто $E[X_A] = \Pr\{\mathrm{A}\}$.

Дійсно, за визначенням індикаторної випадкової величини та математичного очікування: $E[X_A] = E[I\{A\}] = 1 \cdot \Pr\{\mathrm{A}\} + 0 \cdot \Pr\{\overline{\mathrm{A}}\} = \Pr\{\mathrm{A}\}$.

Наприклад, визначимо за допомогою індикаторної випадкової величини математичне очікування того, що в результаті підкидання монети випаде орел. Простір подій тут: S={H,T}, H, T – події випадіння орла та решки, відповідно. Ймовірність кожної події: $\Pr\{\mathrm{H}\} = \Pr\{\mathrm{T}\} = 1/2$.

Індикаторна випадкова величина X_H, пов'язана з подією H, вказує наявність випадіння орла: якщо випадає орел, дорівнює 1, якщо не випадає – 0. Тобто $\mathrm{X_H} = \mathrm{I}\{\mathrm{H}\} = \begin{cases}1, \text{випав орел,}\\0, \text{орел не випав.}\end{cases}$

Математичне очікування того, що випав орел в результаті підкидання монети, дорівнює математичному очікуванню індикаторної випадкової величини X_H.

$$\mathrm{E}[\mathrm{X_H}] = \mathrm{E}[\mathrm{I}\{\mathrm{H}\}] = 1 \cdot \Pr\{\mathrm{H}\} + 0 \cdot \Pr\{\mathrm{T}\} = 1/2 .$$

Індикаторні випадкові величини корисні при аналізі процесів, в яких відбуваються повторні випробування. Це, наприклад, *n* повторних кидків монети у наведеному прикладі. Кількість випадінь орла за *n* повторних кидків монети обчислюють шляхом окремого розгляду ймовірності такої події, що орел випаде 0,1,2,… і т.д. разів. Уведемо індикаторну випадкову величину X_i, пов'язану з подією коли за *i*-го підкидання випадає орел: X_i=I{*i*-й кидок приведе до події H}. Якщо X – випадкова подія, що дорівнює загальній кількості випадінь орла за *n* кидків монети, тоді $\mathrm{X} = \sum_{i=1}^{n} X_i$. Математичне очікування випадінь орла буде:

$$\mathrm{E}[\mathrm{X}] = \mathrm{E}[\sum_{\mathrm{i}=1}^{n} X_i] = \sum_{\mathrm{i}=1}^{n} E[X_i] = \sum_{\mathrm{i}=1}^{n} 1/2 = = n/2 .$$

У задачі про найм потрібно обчислити математичне очікування події, що відповідає найму нового працівника. Мали припущення, що канди-

дати надходять на співбесіду у випадковому порядку. Нехай X – випадкова величина, значення якої дорівнює кількості наймів. Використовуючи індикаторну випадкову величину визначимо n величин, пов'язаних з найманням конкретних кандидатів: X_i – індикаторна випадкова величина, пов'язана з подією найму i-го кандидата, i = 1,2, …n. Тоді

$$X_i = I\{\text{кандидат } i \text{ найнятий}\} = \begin{cases} 1, \text{кандидат } i \text{ найнятий}, \\ 0, \text{кандидат } i \text{ не найнятий} \end{cases}$$

та

$$X = X_1 + X_2 + \ldots + X_n.$$

За основною властивістю індикаторної випадкової величини: $M[X_i] = Pr\{\text{кандидат } i \text{ найнятий}\}$.

Повернемося до аналізу процедури Hire-Assistant. У рядку 6 роботодавець наймає кандидата i, якщо він виявляється кращим усіх попередніх (від 1 до i-1). Так як мали припущення, що кандидати надходять у випадковому порядку, то й перші i кандидатів теж надходять у випадковому порядку. Крім того, будь-який з i кандидатів може бути найкращим. Тому ймовірність того, що кваліфікація i-го кандидата вище кваліфікації кандидатів від 1 до i-1 та він буде узятий на роботу, дорівнює 1/i, а за основною властивістю індикаторної випадкової величини $M[X_i] = 1/i$. Звідси:

$$M[X] = M\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} M[X_i] = \sum_{i=1}^{n} 1/i = \lg n + O(1).$$

Отже, при проведенні співбесіди з n кандидатами у середньому буде найнято лише lg$n$ кандидатів. Тому **повна вартість вартість найму при використанні алгоритму O($c_i n$+ $c_h \lg n$).**

В усіх рандомізованих алгоритмах маємо припущення про випадковий порядок надходження вхідних даних, тому додаємо генератор випадкових величин. Розглянутий вище алгоритм насправді є *детермінованим*, тобто *для будь-яких конкретних вхідних даних кількість наймів буде такою ж* при повторній роботі процедури. Ця величина є різною для різних вхідних даних та *залежить від розподілу рангів кандидатів*.

Кожен окремо узятий набір вхідних даних можна уявити у вигляді перечислення рангів кандидатів у порядку нумерації останніх,

$\langle \text{rank}(1), \text{rank}(2), ..., \text{rank}(n) \rangle$. Вартість алгоритму залежить від того, скільки разів відбувається найм працівника, але є більш дешеві та більш дорогі вхідні дані.

У випадку рандомізованого алгоритму, де спочатку відбувається випадкова перестановка вхідних даних, а потім працює алгоритм, рандомізація є складовою частиною алгоритму, а не вхідних даних. Тому для окремо узятого набору вхідних даних не можемо сказати скільки разів наймуть працівника. Ця велика буде різною за кожного запуску процедури. В рандомізованих алгоритмах жодні вхідні дані не можуть викликати найгіршу поведінку алгоритму, ми можемо лише одержати невдалу перестановку рангів. З уведенням рандомізації маємо наступну процедуру для виконання алгоритму.

```
   Randomized-Hire-Assistant(n)
1  Randomized-In-Place(A) // випадкова перестановка кандидатів
2  best = 0
3  for i =1 to n
4      Співбесіда з кандидатом i
5      if кандидат i кращий кандидата best
6          best = i
7          найняти кандидата i
```

Математичне очікування вартості (**очікувана вартість**) найму за цією процедурою **$O(c_i n + c_h \lg n)$ + час роботи Randomized-In-Place.** Тут вже *не потрібне припущення про вигляд вхідних даних* (на відміну від ймовірного аналізу, де маємо вартість у середньому випадку). Але робота генератора дискретних випадкових величин вимагає додаткового часу.

Розглянемо питання рандомізації, а саме – процесу одержання випадкової перестановки. Поширеним методом рандомізації є метод з наданням кожному елементу $A[i]$ вхідного масиву випадкового пріоритету $P[i]$ та подальшого сортування масиву у відповідності з пріоритетами.

Вважаємо, що однакових пріоритетів немає. Якщо використати *сортування порівнянням*, то потрібно не менше $\Omega(n \lg n)$ часу. Відповідна процедура може мати наступний вигляд [2].

```
   Permute-By-Sorting(A)
1  n = A.lenght
2  P[1..n] – новий масив
3  for i =1 to n
4       P[i] = Random(1, n^3)
5       Відсортувати A, використовуючи P як ключі сортування
```

При уведенні процедури рандомізації до алгоритму обов'язково потрібно показувати, що її використання дасть випадкову перестановку вхідних даних з рівномірним розподілом.
Отже, доведемо, що в результаті роботи процедури Permute-By-Sorting одержали **випадкову перестановку з рівномірним розподілом**. Доведення представимо у вигляді наступного твердження.

Твердження. У припущенні що однакових пріоритетів немає в результаті виконання процедури Permute-By-Sorting одержується випадкова перестановка вхідних значень з рівномірним розподілом.
Доведення. Розглянемо деяку перестановку, в якій кожен елемент $A[i]$ получить i-й пріоритет у порядку зростання. Покажемо, що ймовірність такої перестановки буде $1/n!$. Нехай E_i – подія: елемент $A[i]$ одержує i-й пріоритет. Обчислимо ймовірність того, що подія E_i відбувається для всіх i. Ця ймовірність дорівнює для всіх $i = n$ кандидатів

$$\Pr\{E_1 \cap E_2 \cap ... \cap E_{n-1} \cap E_n\} = \Pr\{E_1\} \cdot \Pr\{E_2 / E_1\} \cdot \Pr\{E_3 / E_1 \cap E_2\} \cdot$$
$$\cdot ... \cdot \Pr\{E_j / E_1 \cap E_2 \cap ... \cap E_{j-1}\} \cdot ... \cdot \Pr\{E_n / E_1 \cap E_2 \cap ... \cap E_{n-1}\}.$$

Маємо $\Pr\{E_1\} = 1/n$ (ймовірність того, що пріоритет обраного навмання кандидата є мінімальним), $\Pr\{E_2 / E_1\} = 1/(n-1)$ (кожен з n-1 елементів, що залишилися, має однакові шанси получити найменший пріоритет). Далі, $\Pr\{E_j / E_{j-1} \cap ... \cap E_2 \cap E_1\} = 1/(n-j+1)$ (кожен з елементів, що залишився, має однакові шанси одержати j-й найменший пріоритет). Тому:

$$\Pr\{E_1 \cap E_2 \cap ... \cap E_{n-1} \cap E_n\} = \left(\frac{1}{n}\right)\left(\frac{1}{n-1}\right) ... \left(\frac{1}{2}\right)\left(\frac{1}{1}\right) = \left(\frac{1}{n!}\right).$$

Тобто ймовірність одержання тотожної перестановки дійсно є $1/n!$.
Узагальнення доведення на випадок довільної перестановки одержимо, якщо розглянемо довільну фіксовану перестановку $\sigma = \langle \sigma(1), \sigma(2), ..., \sigma(n) \rangle$ множини $\{1,2,…,n\}$, де ранг r_i – ранг пріоритету, наданого елементу

$A[i]$. Нехай E_i – подія: елемент $A[i]$ одержує $\sigma(i)$-й пріоритет ($r_i = \sigma(i)$). Далі все виконуємо аналогічно попередньому випадку та прийдемо знову до ймовірності перестановки $1/n!$.

Використовують також метод рандомізації з одержанням випадкових перестановок шляхом *перестановок «на місці»*. Тоді потрібен час $O(n)$. Відповідна процедура може мати вигляд [2]:

```
   Randomized-In-Place(A)
1  n = A.lenght
2  for i = 1 to n
3        Обміняти A[i] та A[Random(i, n)]
```

Доведення, що в результаті виконання процедури Randomized-In-Place одержали **випадкові перестановки з рівномірним розподілом** проведемо за допомогою інваріанта циклу та k-перестановки n-елементної множини.

К-перестановкою даної n-елементної множини називають послідовність, що складається з k елементів, вибраних з n елементів початкової множини. Всього маємо $n!/(n-k)!$ можливих k-перестановок.
Доведення знову представимо у вигляді твердження.

Твердження. В результаті виконання процедури Randomized-In-Place одержується випадкові перестановки з рівномірним розподілом.
Доведення. Розглянемо наступний інваріант циклу: перед i-ю ітерацією циклу **for** (рядки 2-3) для кожної можливої (i-1)-ї перестановки n елементів ймовірність того, що підмасив A[1..i-1] містить цю (i-1)-у перестановку дорівнює $(n-i+1)!/n!$.

Покажемо, що інваріант циклу справедливий перед першою ітерацією циклу (ініціалізація), що він зберігає істинність впродовж ітерацій (збереження) та є справедливим по завершенню циклу (завершення).

Ініціалізація. Безпосередньо перед першою ітерацією циклу, маємо i =1, а за формулюванням інваріанта циклу ймовірність знаходження кожного розміщення з 0 елементів (0-розміщення) в підмасиві A[1..0] дорівнює $(n-i+1)!/n! = 1$. Підмасив A[1..0] є пустим, а 0-розміщення за визначенням не містить жодного елементу. Тому підмасив A[1..0] містить будь-яке 0-розміщення з ймовірністю 1. Звідси інваріант циклу є справедливим перед першою ітерацією.

Збереження. Нехай перед i-ю ітерацією ймовірність того, що у підмасиві А[1..i-1] маємо задане розміщення i – 1 елементів, дорівнює $(n-i+1)!/n!$ Покажемо, що після i-ї ітерації кожна з можливих i-перестановок може бути в підмасиві А[1..i-1] з ймовірністю $(n-i)!/n!$ Подальше зростання i на наступній ітерації приведе до збереження інваріанта циклу.

Розглянемо докладніше i-ту ітерацію, а саме деяке конкретне розміщення i елементів, що позначено $\langle x_1, x_2, \ldots, x_i \rangle$. Це розміщення складається з розміщення i-1 елементів, за яким йде значення x_i, що міститься при виконанні алгоритму в елемент А[i].

Нехай E_1 – подія, за якої в результаті перших i-1 ітерацій в підмасиві А[1..i-1] створюється деяке визначене розміщення i-1 елементів i-1 елементів. За інваріантом циклу $\Pr\{E_1\} = (n-i+1)!/n!$.

Нехай E_2 – подія, за якої в процесі i-ї ітерації у позицію А[i] розміщують елемент x_i. Розміщення $\langle x_1, x_2, \ldots, x_i \rangle$ сформується у підмасиві А[1..i] тільки за умови, що події E_1 та E_2 відбудуться разом. Тому знайдемо ймовірність цих двох подій одночасно, а саме: $\Pr\{E_2 \cap E_1\} = \Pr\{E_2 / E_1\} \Pr\{E_1\}$.

Так як у рядку 3 алгоритму елемент x_i вибирається випадковим чином серед $n-i+1$ значень в позиціях А[1..n], то умовна ймовірність $\Pr\{E_2 / E_1\}$ дорівнює $1/(n-i+1)$. Тоді

$$\Pr\{E_2 \cap E_1\} = \frac{1}{n-i+1} \frac{(n-i+1)!}{n!} = \frac{(n-i)!}{n!}.$$

Завершення. При завершенні алгоритму маємо: $i = n + 1$ та підмасив А[1..n] є заданою n-перестановкою з ймовірністю, що дорівнює

$$(n-(n+1)+1)!/n! = 0!/n! = 1/n!.$$

Отже, результаті виконання процедури одержимо випадкові перестановки з рівномірним розподілом.

Лекція 11. Застосування індикаторних випадкових величин, ймовірнісного аналізу в дослідженні алгоритмів

Минулої лекції було показано, що за результатами ймовірнісного аналізу алгоритму одержується *час роботи у середньому випадку* (причому наявність відомостей про розподіл вхідних даних є обов'язковою), а в результаті виконання аналізу алгоритму з використанням індикаторних випадкових величин отримують математичне очікування часу роботи (тобто *очікуваний час роботи*). Крім того, для рандомізованого алгоритму, де спочатку відбувається рандомізована перестановка вхідних даних, а потім працює алгоритм, в аналізі алгоритму не тільки додатково враховується час виконання процедур рандомізації, а й необхідно показувати, що в результаті роботи використаної процедури рандомізації одержуємо випадкову перестановку.
Ще раз на прикладі проілюструємо різницю обох варіантів аналізу алгоритму.
Розглянемо парадокс днів народження.
Проведемо ймовірнісний аналіз. Тоді постановка задачі наступна: *скільки чоловік потрібно зібрати в кімнаті, щоб імовірність співпадіння дати народження у двох з них досягла 1/2*.
Нехай у кімнаті знаходиться k чоловік (пронумеруємо їх: 1, 2,…, k), b_i – дата народження особи з номером i, $1 \le b_i \le n$, а рік вважатимемо становить $n = 365$ днів. Припустимо, що дні народження рівномірно розподілені впродовж року: $\Pr\{b_i = r\} = 1/n$ для i =1, 2, …, k та r =1, 2, …, n. Ймовірність того, що дати народження двох осіб i та j співпадають, залежить від того, чи є вибір цих дат незалежним. Вважатимемо, що дати є незалежними, тому ймовірність того, що вони народилися в один день r є $\Pr\{b_i = r \text{ і } b_j = r\} = \Pr\{b_i = r\}\Pr\{b_j = r\} = 1/n^2$.
Тоді ймовірність, що ці обидві особи народилися в один день року є

$$\Pr\{b_i = b_j\} = \sum_{r=1}^{n} \Pr\{b_i = r \text{ і } b_j = r\} = \sum_{r=1}^{n} 1/n^2 = 1/n .$$

Розглянемо яка ймовірність того, що хоча б двоє осіб народилися в один день. Ймовірність співпадання хоча б двох днів народження дорівнює 1 мінус імовірність того, що всі дні народження відрізняються. Подію, за якої всі дні народження для k осіб є різними представимо як

$$B_k = \bigcap_{i=1}^{k} A_i ,$$

де A_i – подія: день народження особи з номером i відрізняється від дня народження особи з номером j для всіх $j < i$. Так як $B_k = A_k \cap B_{k-1}$, то $\Pr\{B_k\} = \Pr\{B_{k-1}\}\Pr\{A_k | B_{k-1}\}$, де $\Pr\{B_1\} = \Pr\{A_1\} = 1$.

Якщо дні народження $b_1, b_2, ..., b_{k-1}$ різні, то умовна ймовірність того, що $b_k \neq b_i$ при i =1, 2n, …, k-1 буде $\Pr\{A_k | B_{k-1}\} = (n-k+1)/n$, так як з усіх n днів незайнятими залишаться n-(k-1) днів. Тоді

$$\Pr\{B_k\} = \Pr\{B_{k-1}\}\Pr\{A_k | B_{k-1}\} = \Pr\{B_{k-2}\}\Pr\{A_{k-1} | B_{k-2}\}\Pr\{A_k | B_{k-1}\} = ... =$$
$$= \Pr\{B_1\}\Pr\{A_2 | B_1\}\Pr\{A_3 | B_2\}...\Pr\{A_{k-1} | B_{k-2}\}\Pr\{A_k | B_{k-1}\} =$$
$$= 1\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)...\left(\frac{n-(k-2)}{n}\right)\left(\frac{n-(k-1)}{n}\right) =$$
$$= 1\left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)...\left(1-\frac{(k-2)}{n}\right)\left(1-\frac{(k-1)}{n}\right).$$

Використовуючи нерівність $1 + x \leq e^x$ маємо

$$\Pr\{B_k\} \leq e^{-1/n}e^{-2/n}...e^{-(k-1)/n} = e^{-\sum_{i=1}^{k-1} i/n} = e^{-k(k-1)/2n}.$$

Одержане буде не перевищувати ½ ($\Pr\{B_k\} \leq e^{-k(k-1)/2n} \leq 1/2$) якщо $-k(k-1)/2n \leq \ln(1/2)$. Отже, ймовірність того, що всі k днів народження різні, буде не менше ½ за умови, що $k(k-1)/ \geq 2n\ln(2)$, а саме $k \geq (1+\sqrt{1+(8\ln 2)n})/2$, що за n=365 дає $k \geq 23$.

Проведемо аналіз із застосуванням індикаторних випадкових величин. В даному випадку показуємо, що математичне очікування деякої кількості пар осіб, що народилися в один день, не менше 1.

Визначимо для кожної пари осіб (i, j) з усіх k чоловік індикаторну випадкову величину X_{ij}, $1 \leq i < j \leq k$:

$$X_{ij} = I\{\text{дні народження } i \text{ й } j \text{ співпадают ь}\} = \begin{cases} 1, \text{дні народження } i \text{ й } j \text{ співпадают ь,} \\ 0, \text{інакше.} \end{cases}$$

Ймовірність того, що у двох осіб співпадають дні народження, як було показано вище, становить 1/n. За основною властивістю індикаторної випадкової величини $E[X_{ij}] = \Pr\{\text{дні народження } i \text{ й } j \text{ співпадают ь}\} = 1/n$.

Нехай X – випадкова величина, що представляє кількість пар осіб, дні народження яких співпадають, тоді

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

Математичне очікування цієї випадкової величини:

$$E[X] = E\left[\sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}\right] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} E[X_{ij}] = \frac{k(k-1)}{2} \frac{1}{n} .$$

Звідси за $k(k-1) \geq 2n$ $(k > 0)$ математичне очікування кількості пар осіб, що народилися в один день, не менше 1. Тому, якщо в кімнаті хоча б $\sqrt{2n}+1$ людей, то можна очікувати, що хоча б у двох з них дні народження співпадають. У нас за k=28: $k(k-1)/2n \approx 1.0356$. Отже, за наведеним аналізом визначили кількість осіб, для яких *математичне очікування числа пар з одним днем народження для обох осіб буде дорівнювати 1*.

Наступний приклад – задача з *повторними* випробуваннями, а саме – задача про збирання купонів або наповнення корзин однаковими шарами. Нехай маємо b корзин (різних купонів), що пронумеровані від 1 до b. Корзини наповнюються однаковими шарами (вибираємо купони), події незалежні, тому ймовірність того, що шар опиниться у деякій корзині (оберемо деякий купон) дорівнює $1/b$ (схема Бернуллі). Тому стандартні запитання: скільки шарів попаде у визначену корзину (якщо кидали n шарів, то математичне очікування кількості шарів у корзині n/b); скільки у середньому потрібно шарів для того, щоб у конкретній корзині опинився один шар (математичне очікування кількості шарів, що необхідно кинути у корзини дорівнює b) і т.д.

Розглянемо: скільки шарів потрібно кинути у корзини, щоб у кожній з них опинився хоча б 1 шар, тобто скільки купонів потрібно зібрати, щоб мати хоча б по 1 кожного виду (еквівалентно визначенню математичного очікування кількості кидків n, необхідних для b попадань – тобто кидку шару в пусту корзину).

Проведемо ймовірнісний аналіз. Використання поняття попадань для n кидків дозволяє розбити увесь процес на b етапів: етап з номером i триває з i-1-го попадання до i-го. Якщо i = 1, то маємо перше попадання, а оскільки усі корзини пусті, то обов'язково попадемо. На подальших етапах маємо i – 1 корзин із шарами та b – (i – 1) пустих корзин (незібраних купонів). Тому, за кожного кидку на i-му етапі ймовірність попадання $(b-(i-1))/b$.

Нехай n_i – кількість кидків на i-му етапі. Тоді кількість шарів, необхідних для попадання в b корзин – $n = \sum_{i=1}^{b} n_i$. Всі значення n_i підкорю ються геометричному розподілу з імовірністю успіху $(b - (i - 1))/b$ та $M[n_i] = \frac{b}{b-i+1}$. Тоді:

$$M[n] = M\left[\sum_{i=1}^{b} n_i\right] = \sum_{i=1}^{b} M[n_i] = \sum_{i=1}^{b} \frac{b}{b-i+1} = b\sum_{i=1}^{b} \frac{1}{b-(i-1)} = b\sum_{i=1}^{b} \frac{1}{i} = b(\ln b + O(1)) \cdot$$

Тобто необхідно $b\ln b$ купонів, що випадково збираються, для того, щоб мати хоча б по 1 кожного виду (кидків, щоб у кожній з корзин опинився хоча б 1 шар).

Згадаємо задачу про найм. Нехай з метою економії не бажано проводити співбесіди з усіма кандидатами та повторювати процедуру звільнення-найму. Прагнемо знайти такого кандидата, що є найкращим з усіх. Але умова про те, що одразу після співбесіди кандидату потрібно або відмовити, або узяти його на роботу, зберігається. У такому випадку після зустрічі з кожним кандидатом кожному з них надають оцінку: для i-го кандидата – score(i). Припустимо, що всі кандидати одержали різні оцінки. Після зустрічі з j-м кандидатом стає відомим, який з цих j кандидатів одержав максимальну оцінку, але невідомо, чи є серед $n - j$ кандидатів, що залишилися, особа з більш високою кваліфікацією. Тому стратегія: обираємо ціле додатне число $k < n$, проводимо співбесіду з k особами, відмовивши усім з них, а потім наймаємо кандидата з тих, що залишилися, який буде першим з оцінкою що перевищуватиме оцінки усіх попередніх. Якщо самий найкращий був серед k перших осіб, то наймають останнього. Вказана нижче нерандомізована процедура реалізує цю стратегію та повертає номер кандидату, що наймається [2].

```
   On-Line-Maximum(k, n)
1  bestscore = – ∞
2  for i =1 to k
3      if score(i) > bestscore
4          bestscore = score(i)
5  for i =k + 1 to n
6      if score(i) > bestscore
7          return i
```

8 **return** n

Спробуємо визначити для кожного додатного фіксованого k ймовірність того, що найнятий найкращий претендент. Позначимо найвищу оцінку серед кандидатів з номерами від 1 до j як $M(j) = \max_{1 \le i \le j}\{score(i)\}$.

Нехай S – подія: вибір самого кваліфікованого кандидата; S_i – подія: найнятим на роботу виявився i-й кандидат. Так як усі події S_i є взаємовиключними, то $\Pr\{S\} = \sum_{i=1}^{n}\Pr\{S_i\} = \sum_{i=k+1}^{n}\Pr\{S_i\}$ оскільки за умовою $\Pr\{S_i\} = 0$ для i = 1, 2, … k (перших k кандидатів). Обчислимо $\Pr\{S_i\}$ для кандидатів з номеру k + 1.

Отже, щоб найняти i-го кандидата необхідно: в позиції i має бути самий кваліфікований претендент (подія B_i) та не повинен бути обраним жоден з попередніх кандидатів, а саме кандидат з k+1-ї до i-1-ї позиції (це значить, що у рядку 6 процедури для j від k +1 до i – 1 виконується умова $score(j) < bestscore$, тобто оцінка менша $M(k)$).

Нехай O_i – подія: не найнято жодного з тих, хто проходив співбесіду під номерами від k + 1 до i – 1. Події O_i та B_i є незалежними. Подія O_i залежить від порядку нумерації кандидатів у позиціях від 1 до i-1. Подія B_i залежить від того, чи перевищує оцінка кандидату i оцінки усіх інших кандидатів. Тоді справедливо:

$$\Pr\{S_i\} = \Pr\{B_i \cap O_i\} = \Pr\{B_i\}\Pr\{O_i\},$$

де ймовірність $\Pr\{B_i\} = 1/n$, оскільки кандидат з найвищою кваліфікацією може рівноймовірно знаходитися у будь якій позиції від 1 до n. Щоб відбулася подія O_i , найбільш кваліфікований кандидат для номерів від k + 1 до i – 1 має знаходитися у позиціях від 1 до k. Але він рівноймовірно може опинитися у будь-якій з i – 1 позицій, тому $\Pr\{O_i\} = k/(i-1)$. Звідки $\Pr\{S_i\} = \dfrac{k}{n(i-1)}$ та

$$\Pr\{S\} = \sum_{i=k+1}^{n}\Pr\{S_i\} = \sum_{i=k+1}^{n}\frac{k}{n(i-1)} = \frac{k}{n}\sum_{i=k+1}^{n}\frac{1}{(i-1)} = \frac{k}{n}\sum_{i=k}^{n-1}\frac{1}{i}.$$

Оцінимо цю суму зверху та знизу використавши наближення інтегралами для монотонно спадаючої функції $1/i$, а саме:

$$\int_{k}^{n}\frac{1}{x}dx \le \sum_{i=k}^{n-1}\frac{1}{i} \le \int_{k-1}^{n-1}\frac{1}{x}dx.$$

А тому

$$\frac{k}{n}\left(\ln(n)-\ln(k)\right)\le \Pr(S)\le \frac{k}{n}\left(\ln(n-1)-\ln(k-1)\right),$$

що дасть оцінку ймовірності $\Pr\{S\}$. З метою максимального збільшення ймовірності вдалого результату, виберемо значення *k*, за якого нижня границя $\Pr\{S\}$ буде максимальною. Знайдемо максимум виразу зліва (продиференціюємо по *k* та прирівняємо до 0). Маємо: нижня границя досягає максимального значення, якщо $\ln k = \ln n - 1 = \ln(n/e)$, тобто $k = n/e$. В цьому випадку знайдемо найкращого кандидата з імовірністю **не менше *1/e***.

Лекція 12. Швидке сортування. Аналіз рандомізованого алгоритму для найгіршого, середнього та найкращого випадків

В алгоритмі швидкого сортування переставляють елементи масиву *A*, який сортується, таким чином, щоб його можна було розділити на дві частини і кожний елемент з першої частини був не більший за будь-який елемент з другої. Впорядкування кожної з частин відбувається рекурсивно. Алгоритм відносяться до групи алгоритмів обміну, що містить повільні алгоритми. Але впровадження обмінів між далекими за розташуванням елементами призводить до істотного зменшення часу роботи.

Алгоритм має оцінку часу роботи для найгіршого випадку $\Theta(n^2)$, але у середньому – вже $\Theta(n\lg n)$, крім того, множник, схований у виразі для *O*-нотації, доволі малий.

Алгоритм виконує сортування на місці та використовує метод декомпозиції, три етапи якого для сортування підмасиву $A[p..r]$ виглядають як наведено нижче [2].

1. *Поділ.* Масив $A[p..r]$ розбивається на два, можливо пустих, підмасива $A[p..q-1]$ та $A[q+1..r]$, такі, що кожен елемент в $A[p..q-1]$ менший або дорівнює елементу $A[q]$, який не перевищує будь-якого елементу з $A[q+1..r]$. Індекс q обчислюється у ході процедури поділу

(процедура Partition, що змінює порядок елементів підмасиву $A[p..r]$ без залучення додаткової пам'яті).
2. *Володарювання.* Підмасиви $A[p..q-1]$ та $A[q+1..r]$ сортуються за допомогою рекурсивного виклику процедури швидкого сортування Quicksort.
3. *Комбінування.* Так як підмасиви сортуються на місці, то для їх поєднання не потрібно додаткових дій, увесь масив $A[p..r]$ стане відсортованим.
Процедуру Quicksort можна представити у наступному вигляді [2]:

```
   Quicksort(A, p, n)
1  if p < r
2       q = Partition(A, p, r)
3       Quicksort(A, p, q – 1)
4       Quicksort(A, q + 1, r)
```

Процедуру Partition можна представити у вигляді [2]:

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

У класичному алгоритмі процедура Partition завжди обирає опорним елемент $x = A[r]$ (рядок 1) і поділ підмасиву $A[p..r]$ виконується відносно цього елементу. На початку виконання процедури підмасив поділяється на 4 області, що можуть бути пустими, як вказано на рис. 12.1: всі елементи підмасиву $A[p..i]$ є не більше за x; всі елементи підмасиву $A[i+1..j-1]$ більші x; елементи підмасиву $A[j..r-1]$ мають довільні значення; елемент $A[r] = x$.

Рис. 12.1. Чотири області, виділені у підмасиві $A[p..r]$

Крім того, що на початку кожної ітерації циклу **for** (рядки 3-6) кожна з цих областей задовольняє властивостям, які можна сформулювати у вигляді **інваріанта циклу**:

на початку кожної ітерації циклу для будь-якого індексу k масиву маємо:

- якщо $p \le k \le i$, то $A[k] \le x$;
- якщо $i+1 \le k \le j-1$, то $A[k] > x$;
- якщо $k = r$, то $A[k] = x$.

Індекси між j та r-1 не підпадають до жодного з наведених випадків, а значення відповідних до них елементів не мають якогось визначеного зв'язку з опорним елементом x.

Покажемо, що сформульований інваріант циклу є справедливим перед першою ітерацією, що кожна ітерація циклу зберігає інваріант, що він є справедливим по завершенню, тобто демонструється коректність алгоритму.

Ініціалізація. Перед першою ітерацією циклу $i = p-1$ та $j = p$. Між елементами з індексами p та i немає жодних елементів, як немає їх і між індексами i+1 та j-1. Тому перші дві умови інваріанта виконуються. Присвоєння у рядку 1 приводить до задоволення третьої умови.

Збереження. Розглянемо два варіанти ітерації циклу **for**, що обумовлюються виконанням рядку 4, а саме умовою $A[j] \le x$. Якщо маємо $A[j] > x$ (випадок а, рис. 12.2), тоді виконується єдина дія у циклі – збільшується на 1 значення j. При збільшенні j виконується друга умова для елемента $A[j-1]$, усі інші елементи залишаються незмінними. Якщо $A[j] \le x$ (випадок б, рис. 12.2), то збільшується значення i, елементи $A[i]$ та $A[j]$ міняються місцями, на 1 зростає значення j. В результаті одержимо $A[i] \le x$ та умова 1 виконується. Аналогічно одержимо $A[j-1] > x$, так як елемент, що був переставлений у позицію елемента $A[j-1]$, за інваріантом циклу більший за x.

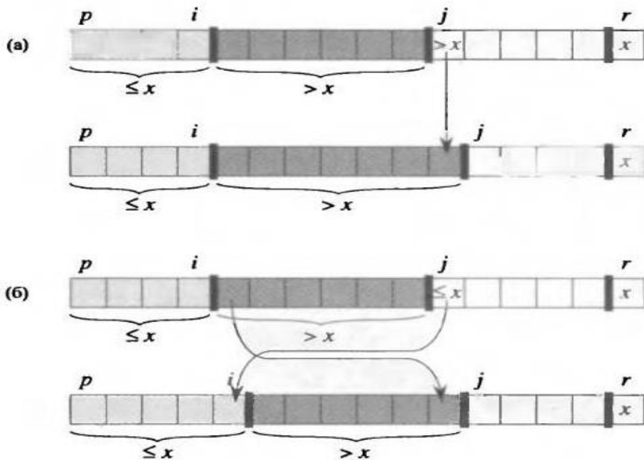


Рис. 12.2. Варіанти ітерації циклу **for** процедури Partition [2]

Завершення. По завершенню маємо $j = r$. Тому кожен елемент масиву належить до однієї з трьох множин, що описані в інваріанті. Всі елементи масиву поділені на три множини: величина яких не перевищує *x*, величина яких перевищує *x* та сам елемент *x*.

У рядку 7 процедури Partition опорний елемент *x* переміщується на місце крайнього лівого елементу, що перевищує *x*. Далі процедура повертає новий індекс опорного елементу. Вихід процедури задовольняє вимогам, накладеним кроком поділу. Він задовольняє умові: після рядку 2 процедури Quicksort елемент $A[q]$ строго менший будь-якого елементу з підмасиву $A[q+1..r]$.

На рис. 12.3 наведено на прикладі конкретних вхідних даних – масиві з чисел 2, 8, 7, 1, 3, 5, 6, 4 – виконання процедури Partition.

Час роботи процедури **Partition** над підмасивом з *n* елементів становить $\Theta(n)$. Час роботи алгоритму швидкого сортування залежить від збалансованості, якою характеризується поділ, а збалансованість залежить від того, який елемент став опорним. Якщо одержали збалансований поділ, то асимптотично алгоритм працює так же швидко, як і сортування злиттям. Якщо поділ не збалансований, то асимптотично працює з такою ж швидкістю, як і сортування вставкою.

Найгіршу поведінку алгоритму над підмасивом з *n* елементів маємо у випадку, коли процедура Partition породжує підзадачу з *n* – 1 елемента-

ми та пусту. Нехай такий незбалансований поділ відбувається за кожного рекурсивного виклику. Для виконання розбиття потрібно $\Theta(n)$ часу.

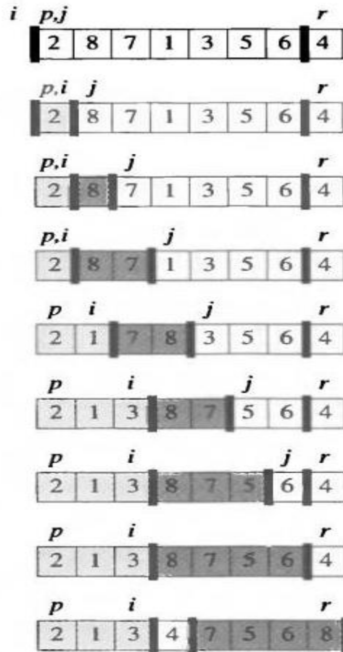


Рис. 12.3. Виконання процедури Partition на конкретних вхідних даних [2]

Рекурсивний виклик **Partition**, якщо на вході масив розміром 0, приводить одразу до повернення з процедури без виконання будь-яких операцій, тому $T(0) = \Theta(1)$. Маємо:

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2).$$

Більш того, стільки ж часу потрібно буде алгоритму у випадку обробки повністю відсортованого масиву.

Найкращу поведінку алгоритму маємо у випадку, коли процедура Partition породжує дві підзадачи, розмір яких не перевищує $n/2$ (точний розмір задач: $\lfloor n/2 \rfloor$ та $\lceil n/2 \rceil - 1$). Час роботи у цьому випадку описують як $T(n) = 2T(n/2) + \Theta(n)$, тобто має асимптотично точну оцінку $\Theta(n \lg n)$.

У середньому випадку в асимптотичний границі поведінка алгоритму набагато ближче до поведінки у найкращому випадку, ніж до поведінки у найгіршому. Це відбувається внаслідок поведінки балансу у поділі

(наприклад, якщо поділ завжди відбувається у відношенні 1 до 9 маємо $T(n) = T(9n/10) + T(n/10) + cn$, де вартість одержаного дерева (рис. 12.4) становить $O(n\lg n)$). Крім того, взагалі, поділ, що характеризується скінченою сталою пропорційності, приводить до утворення дерева рекурсії висотою $\Theta(\lg n)$ з вартістю на кожному рівні $O(n)$.

Рис. 12.4. Дерево рекурсії для $T(n) = T(9n/10)+T(n/10)+cn$ [2]

Так як поведінка алгоритму залежить від відносного розташування значень у вхідному масиві, то важливим є питання: як часто очікується поява різних вхідних даних?

Нехай всі перестановки вхідних чисел рівноймовірні. Якщо алгоритм працює з випадковим чином вибраним вхідним масивом, то малоймовірно, щоб поділ на кожному рівні відбувався в одному й тому ж відношенні. Деякі поділи будуть гарно збалансованими, деякі – ні (у середньому процедура **Partition** робить і вдалі, і погані поділи). У дереві рекурсії для середнього випадку вдалі та погані поділи рівномірно розподілені по всьому дереву. Для спрощення припустимо, що поділи чергуються, а саме: чергуються поділи як у найкращому випадку та як у найгіршому випадку. Нехай, на корені дерева вартість дорівнює n, розміри одержаних підмасивів є $n - 1$ та 0 (найгірший випадок). На наступному рівні, підмасив розміром $n - 1$ поділяється на підмасиви $((n - 1)/2) - 1$ та $(n - 1)/2$. Вважаємо, що для підмасиву розміром 0 вартість дорівнює 1. Комбінація вдалих та поганих поділів приведе до

утворення підмасивів з розмірами 0, $((n - 1)/2) - 1$ та $(n - 1)/2$ й сумарною вартістю $\Theta(n) + \Theta(n-1) = \Theta(n)$. Отже, одержана ситуація буде відповідати збалансованому поділу. І час роботи також буде $O(n\lg n)$, але схована під O-нотацією буде зовсім інша стала.

Розглянемо рандомізований випадок. Додамо до алгоритму рандомізацію, щоб одержати добру очікувану продуктивність для всіх вхідних даних. Вказаний варіант часто приймають в якості оптимального для обробки достатньо великих масивів. Для виконання рандомізації використаємо метод *випадкової вибірки*. Замість того, щоб опорним елементом завжди обирати $A[r]$, тепер випадково обирається елемент з усього масиву $A[p..r]$. Така модифікація забезпечує будь-якому з елементів підмасиву однакову ймовірність бути опорним, а завдяки випадковому вибору опорного елементу можемо очікувати, що поділ масиву у середньому виявиться збалансованим.

Відповідні процедури наведемо у вигляді [2]:

Randomized-Partition(A, p, r)
1 i = Random(p, r)
2 Обміняти $A[i]$ та $A[r]$
3 **return** Partition(A, p, r)

Randomized-Quicksort(A, p, n)
1 **if** $p < r$
2 q = Randomized-Partition(A, p, r)
3 Randomized-Quicksort(A, p, $q - 1$)
4 Randomized-Quicksort(A, $q + 1$, r)

Проаналізуємо час виконання алгоритму.

Найгірший випадок. Для детермінованого випадку мали, що за самого невдалого поділу на кожному рівні рекурсії час роботи алгоритму є найгіршим – він складає $\Theta(n^2)$. Покажемо це більш ґрунтовно. Нехай $T(n)$ – найгірший час роботи Quicksort на вхідних даних вимірності n. Тоді маємо рекурентне співвідношення:

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n),$$

де параметр q змінюється від 0 до $n-1$. Припустимо, що $T(n) \le cn^2$ для деякої сталої c. Тоді

$$T(n) = \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) = c \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n).$$

Перший доданок справа досягає максимуму на обох кінцях інтервалу у точках 0 та $n-1$ (друга похідна по q є додатною). Тому

$$\max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1.$$

Звідси

$$T(n) \le cn^2 - c(2n-1) + \Theta(n) \le cn^2,$$

оскільки сталу c можна вибрати такою, щоб другий доданок домінував над доданком $\Theta(n)$. Отже, у найгіршому випадку оцінка часу роботи Quicksort складає $\Theta(n^2)$.

Очікуваний час роботи. Припустимо, що всі елементи масиву різні. Проведемо аналіз математичного очікування часу роботи процедури Randomized-Quicksort.

Так як процедури Quicksort та Randomized-Quicksort відрізняються тільки вибором опорного елементу, то аналіз процедури Randomized-Quicksort можна виконати розглядаючи початкові процедури у припущенні, що опорні елементи вибираються випадковим чином з наданого Randomized-Partition підмасиву.

Час роботи процедури Quicksort переважно визначається часом роботи, витраченим на виконання Partition. При виконанні Partition вибирається опорний елемент, що далі не бере участі в жодному рекурсивному виклику обох процедур. Тому впродовж усього часу виконання алгоритму процедуру Partition викликають не більше n разів. Один виклик цієї процедури виконується за $O(1)$ часу та час, пропорційний кількості ітерацій циклу **for** у рядках 3-6. В кожній ітерації циклу у рядку 4 опорний елемент порівнюють з іншими елементами масиву A. Тому, якщо відома загальні кількість разів виконання рядку 4, можна оцінити увесь час, витрачений на виконання циклу у процесі роботи процедури Quicksort.

Справедливе наступне твердження.

Твердження. Якщо X – кількість порівнянь, що виконуються у рядку 4 процедури Partition за час роботи процедури Quicksort над n-елементним масивом, то час роботи процедури Quicksort буде $O(n + X)$.

Доведення. Оцінимо повну кількість порівнянь, що виконуються за всіх викликів процедури Partition. Розглянемо в яких випадках відбувається порівняння двох елементів масиву, а в яких вони не порівнюються. З цією метою перенумеруємо елементи масиву A як $z_1,\ldots,z_n$, де z_i – i-й найменший по порядку елемент та визначимо множину $Z_{ij} = \{z_i,\ldots,z_j\}$.

Коли скільки разів в алгоритмі виконується порівняння елементів z_i та z_j? Порівняння цих елементів виконується не більше 1 разу, так як елементи порівнюються з опорним, який не використовується більше 1 разу у викликах процедури Partition. Цей елемент взагалі більше не буде порівнюватися з іншими. Визначимо індикаторні випадкові величини $X_{ij} = \{z_i \text{ порівнюють із } z_j\}$. Так як кожна пара порівнюється не більше одного разу, то повна кількість порівнянь впродовж роботи алгоритму буде:

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

Тоді

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ порівнюють із } z_j\}.$$

Спробуємо обчислити $\Pr\{z_i \text{ порівнюють із } z_j\}$ за припущення, що опорні елементи вибирають випадково та незалежно один від одного.

Подивимося коли два елементи не порівнюються. Нехай обрано опорний елемент. В результаті виклику процедури Partition все елементи розбиваються на дві множини. Причому всі елементи, що попали до першої множини більше не будуть порівнюватися з елементами із другої множини. Оскільки вважали, що всі елементи різні, то в ситуації коли обрано опорний елемент x, що $z_i < x < z_j$, елементи z_i та z_j порівнюватися не будуть. З іншого боку, якщо в якості опорного елемент z_i обраний до будь-якого іншого елементу множини Z_{ij}, то він буде порівнюватися з кожним елементом цієї множини крім себе. Аналогічне заключення можна зробити про елемент z_j. Отже, елементи z_i та

z_j порівнюються тільки тоді, коли першим в якості опорного у множині Z_{ij} обраний один з них.

Обчислимо ймовірність цієї події. Перед тим як у множині Z_{ij} буде обраний опорний елемент, множина не є поділеною та будь-який її елемент з однаковою ймовірністю може стати опорним. Так як в цій множині $j - i + 1$ елементів, опорні елементи вибираються випадково та незалежно один від одного, ймовірність того, що якийсь фіксований елемент буде першим обраний в якості опорного є $1/(j - i + 1)$. Тому:

$$\Pr\{z_i \text{ порівнюють із } z_j\} = \Pr\{z_i \text{ або } z_j - \text{перший опорний елемент, вибраний із } Z_{ij}\} =$$

$$= \Pr\{z_i - \text{перший опорний елемент, вибраний із } Z_{ij}\} +$$

$$+ \Pr\{z_j - \text{перший опорний елемент, вибраний із } Z_{ij}\} =$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}.$$

Звідси

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

Тобто очікуваний час роботи алгоритму швидкого сортування при використанні процедури Randomized-Partition є $O(n\lg n)$.

Лекція 13. Аналіз алгоритмів сортування порівнянням. Аналіз алгоритмів сортування підрахунком та порозрядного сортування

Всі розглянуті на попередніх лекціях алгоритми сортування відносять до класу алгоритмів сортування порівнянням. В алгоритмах сортування порівнянням використовували попарні порівняння елементів для визначення взаємного порядку елементів (наприклад, порівняння виду $a_i \le a_j$). Для таких алгоритмів теоретично доведено замкненість класу задач щодо нижньої границі, а це значить, що неможливо побудувати алгоритм сортування порівнянням, який би мав кращу нижню границю, ніж її теоретично встановлене значення $\Omega(n\lg n)$.

Для узагальненого розгляду таких алгоритмів використаємо дерево рішень (повне бінарне дерево, в якому представлені усі порівняння, що виконуються алгоритмом, усі інші операції ігноруються). Тоді виконання алгоритму – це проходження шляху від кореня до одного з листків. При досягненні листка алгоритм встановлює відповідну цьому листку впорядкованість елементів. Наприклад, дерево рішень, наведене на рис. 13.1 може представити сортування вставкою усього трьох елементів. Для вхідної послідовності $\langle 4,6,2 \rangle$ на дереві виділений шлях, що вказує рішення, прийняті при сортуванні цієї послідовності. Всього дерево має шість листків, що відповідає 3! можливих перестановок з трьох вхідних елементів.

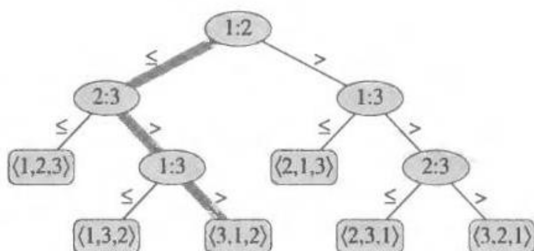


Рис. 13.1. Дерево рішень для вхідної послідовності з трьох елементів [2]

Але на дереві рішень відображені й надлишкові порівняння. Так, наприклад, немає потреби порівнювати c_1 та c_3, якщо $c_1<c_2$ та $c_2<c_3$. В цьому випадку ця частина алгоритму ніколи не буде виконуватися. Якщо вважатимемо, що надлишкові порівняння не виконуються, то будемо мати справу із структурою дерева, в якому кожному зовнішньому вузлу відповідає деяка перестановка.

Необхідною умовою коректності алгоритму сортування порівнянням є те, що в листках дерева рішень повинні міститися усі n! перестановок n елементів, а також, що з кореня можна прокласти шлях до кожного листка, відповідний одному з реальних варіантів роботи алгоритму, тобто кожен листок є досяжним. Використовуючи дерево рішень оцінимо *нижню границю часу роботи у найгіршому випадку*, що одержується як довжина самого довгого шляху від кореня до будь-якого із досяжних листків (відповідає кількості порівнянь, що виконуються алгоритмом у найгіршому випадку та дорівнює висоті дерева рішень

алгоритму). Нижня границя висоти для всіх дерев, в яких всі перестановки представлені досяжними листками, є нижньою оцінкою часу роботи для будь-якого алгоритму сортування порівнянням. Відповідний результат представимо у вигляді наступної теореми.

Теорема. Будь-який алгоритм сортування порівнянням у найгіршому випадку вимагає $\Omega(n\lg n)$ порівнянь [2, 4].
Доведення. Так як нижня границя висоти для всіх дерев, в яких всі перестановки представлені досяжними листками, є нижньою оцінкою часу роботи для будь-якого алгоритму сортування порівнянням, то достатньо визначити висоту дерева. Розглянемо дерево рішень деякої висоти h з l досяжними листками, що відповідає сортуванню порівнянням n елементів. Оскільки кожна з $n!$ перестановок вхідних елементів відповідає деякому листку, а бінарне дерево висотою h має не більше 2^h листків, то справедливо: $n! \le l \le 2^h$. Звідси $h \ge \lceil \lg n! \rceil$, де за формулою Стірлінга $\lceil \lg n! \rceil = n \lg n - n / \lg 2 + (1/2) \lg n + O(1)$. Отже, необхідно виконати $\Omega(n\lg n)$ порівнянь.

З доведеного випливає асимптотична оптимальність пірамідального сортування та сортування злиттям, так як верхні границі $O(n\lg n)$ часу роботи співпадають з нижньою границею $\Omega(n\lg n)$ для найгіршого випадку.
Розглянемо більш швидкі алгоритми сортування, які не використовують порівнянь.
Аналіз алгоритму сортування підрахунком. Припустимо, що кожен з n елементів є цілим числом з інтервалу від 0 до k, $k \in Z$. Всі елементи різні (надалі від цього відмовимося). Якщо $k = O(n)$, то час роботи алгоритму буде $\Theta(n)$. Ідея алгоритму: для кожного вхідного елементу x масиву $A[1..n]$ ($A.length = n$) визначити кількість елементів, що менше x. Це дасть можливість розташувати елемент x у тій позиції вихідного масиву, де він повинен буде знаходитися. В алгоритмі використовується додаткова пам'ять: масив $B[1..n]$ для відсортованих вихідних даних та масив $C[0..k]$ як тимчасовий робочий простір. Відповідну процедуру представлено нижче [2].

Counting-Sort(A, B, k)

```
1   C[0..k] – новий масив
2   for i = 0 to k                                        Θ(k)
3       C[i] = 0
4   for j = 1 to A.length                                 Θ(n)
5       C[A[j]] = C[A[j]] + 1
6   // Зараз C[i] містить кількість елементів, що дорівнює i
7   for i = 1 to k                                        Θ(k)
8       C[i] = C[i] + C[i-1]
9   // Зараз C[i] містить кількість елементів, що дорівнює i
10  for j = A.length downto 1                             Θ(n)
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

В процедурі Counting-Sort після ініціалізації масиву C у циклі **for** (рядки 2-3) у наступному циклі **for** (рядки 4-5) переглядається кожен вхідний елемент. Якщо значення вхідного елементу дорівнює i, збільшується $C[i]$. Тому після рядку 5 елемент $C[i]$ містить кількість вхідних елементів, що дорівнює i для кожного $i = 0, 1, \ldots, k$. В рядках 7-8 для кожного $i = 0, 1, \ldots, k$ додаванням елементів масиву C визначається кількість елементів, що не перевищує i. У циклі **for** (рядки 10-12) кожен елемент $A[j]$ розміщується у відповідній позиції вихідного масиву B. Якщо всі n елементів різні, то за першого переходу до рядку 10 для кожного елементу $A[j]$ у змінній $C[A[j]]$ зберігається коректний індекс кінцевого положення цього елементу у вихідному масиві, оскільки одержано $C[A[j]]$ елементів, що менші або дорівнюють $A[j]$. А так як різні елементи можуть мати однакові значення, то відправляючи значення $A[j]$ до масиву B необхідно щоразу зменшувати $C[A[j]]$ на 1. Тоді наступний вхідний елемент із значенням $A[j]$, якщо такий є, розташовується безпосередньо перед елементом $A[j]$.

Загальний час, потрібний для виконання алгоритму оцінюється як $\Theta(k+n)$, а для випадку $k = O(n)$ – як $\Theta(n)$.

Отже, сортування підрахунком дає оцінку вищу, ніж нижня границя $\Omega(n\lg n)$ для алгоритмів сортування порівнянням. Але, як бачили, в алгоритмі не відбувається порівняння вхідних елементів, замість цього

використовували їхні значення, за допомогою яких і співставляли елементам у відповідність індекси.

Важливою властивістю подібного алгоритму є його **стійкість**, а саме: **елементи з одним і тим же значенням знаходяться у вихідному масиві у тому ж порядку, що й у вхідному**.

Ця властивість є важливою не тільки за наявності супутніх даних для сортованих елементів, а й для побудови алгоритму порозрядного сортування.

Аналіз алгоритму порозрядного сортування. Для розуміння побудови алгоритму звичайно наводять приклади із сортування перфокарт, що мали 80 стовпців, у яких було 12 позицій для отворів, сортування колоди гральних карт та сортування за трьома ключами: роком, місяцем, днем. У загальному вигляді, для побудови алгоритму приймається, що число, яке складається з d цифр, займає поле з d стовпців. Сортувальник може за один раз обробити тільки 1 стовпчик, тому для сортування n елементів у порядку зростання записаних на них d-значних чисел і розроблений алгоритм. Сортування виконується за молодшою цифрою (ключем), потім все об'єднують в одну послідовність (колоду), далі знову сортують, вже за передостанньою цифрою (ключем), складають все разом і т.д. до останнього поєднання після сортування за найвищим розрядом (останнім ключем). В результаті вся послідовність буде відсортована у порядку зростання d-значних чисел і для цього потрібно всього лише d проходів послідовності (колоди). Принциповим моментом алгоритму є наявність стійкості при сортуванні за цифрами окремого розряду (ключа).

Нехай кожен з n елементів масиву A є числом, що має d цифр, де перша цифра стоїть у молодшому розряді. Тоді процедура сортування може мати наступний вигляд [2]:

```
   Radix-Sort(A, d)
1    for i = 1 to d
3        Виконати стійке сортування масиву A за цифрою i
```

Проаналізуємо алгоритм та покажемо справедливість наведеного твердження.

Твердження. Якщо маємо *n* *d*-значних чисел, в яких кожна цифра приймає одне з *k* можливих значень, то алгоритм Radix-Sort дозволяє виконати коректне сортування цих чисел за час $\Theta(d(n+k))$, якщо використане стійке сортування має час роботи $\Theta(n+k)$.

Доведення. Коректність алгоритму випливає з коректності сортування за стовпцями по порядку. Використовуючи алгоритм сортування підрахунком як стійкий проміжний алгоритм сортування, маємо: якщо кожна цифра належить до інтервалу від 0 до $k-1$ (*k* не дуже велике), то для обробки кожної з *d* цифр усіх *n* чисел буде потрібно часу $\Theta(n+k)$, а всі цифри будуть оброблені за час $\Theta(d(n+k))$. Коли ж $k=O(n)$, матимемо $\Theta(dn)$.

Більш того, покажемо, що справедливе наступне [2]:

Лема. Якщо маємо *n* *b*-бітових чисел та довільне натуральне число $r \le b$, то алгоритм Radix-Sort дозволяє виконати коректне сортування цих чисел за час $\Theta((b/r)(n+2^r))$, якщо використане стійке сортування має час роботи $\Theta(n+k)$ для вхідних даних в діапазоні 0 до *k*.

Доведення. Для значення $r \le b$ кожен ключ можна розглядати як число, що складається з $d = \lceil b/r \rceil$ цифр по r біт. Всі цифри є цілими числами від 0 до $2^r - 1$, тому можна використати алгоритм сортування підрахунком, в якому $k = 2^r - 1$ (так 32-бітове слово можна розглядати як число, що складається з чотирьох 8-бітових цифр, тоді: $b = 32$, $r=8$, $k = 255$, $d = 4$). Кожен прохід сортування підрахунком займає час $\Theta(n+k) = \Theta(n+2^r)$, всього таких проходів *d*. Тому час роботи алгоритму буде $\Theta(d(n+2^r)) = \Theta((b/r)(n+2^r))$.

Спробуємо вказати для двох заданих значень *n* та *b* таке значення $r \le b$, яке б мінімізувало вираз $(b/r)(n+2^r)$. Якщо $b < \lfloor \lg n \rfloor$, то для будь-якого значення $r \le b$ одержимо $n + 2^r = \Theta(n)$, а тому вибір $r=b$ дає асимптотично оптимальний час роботи: $(b/b)(n+2^b) = \Theta(n)$. Якщо $b \ge \lfloor \lg n \rfloor$, то вибір $r = \lfloor \lg n \rfloor$ дасть час роботи алгоритму $\Theta(bn/\lg n)$. Якщо значення r буде збільшуватися та перевищить $\lfloor \lg n \rfloor$, то у виразі

$(b/r)(n+2^r)$ член 2^r буде зростати швидше, ніж r, тому одержимо час $\Omega(bn/\lg n)$. Тому найкращий вибір: $r = \lfloor \lg n \rfloor$.

Отже бачимо, що асимптотично алгоритм порозрядного сортування є швидшим за алгоритми сортування порівнянням. Але, навіть за наявності оцінки $\Theta(n)$ для даного алгоритму та, наприклад, $\Theta(n \lg n)$ для алгоритму швидкого сортування слід брати до уваги, що в наведених $\Theta-$ виразах зовсім різні сталі множники. Не зважаючи на те, що для порозрядного сортування *n* ключів може бути потрібним менше проходів, ніж для швидкого сортування, сам прохід може тривати суттєво більше. Крім того, тут враховують і конкретну машину, на якій відбувається сортування, а також вигляд вхідних даних. Аналізують і обсяг додаткової пам'яті, що буде потрібна.

Розглянемо та проаналізуємо ще один алгоритм сортування, що не використовує порівнянь та у середньому випадку має час роботи $O(n)$.

Алгоритм кишенькового сортування. В даному алгоритмі використовується припущення, що вхідні дані підкоряються рівномірному закону розподілення, та уводяться наступні припущення про вхідні дані: вхідні числа генеруються випадковим процесом та рівномірно розподілені в інтервалі $[0,1)$. В алгоритмі розбивають інтервал $[0,1)$ на *n* однакових інтервалів (кишень), в які розподіляють *n* вхідних чисел. А так як ці числа є рівномірно розподіленими в інтервалі $[0,1)$, то можна припустити, що в кожну з кишень попаде не дуже багато елементів. Щоб одержати вихідну послідовність, потрібно відсортувати елементи в кожній кишені, а потім послідовно вказати елементи кожної кишені.

Отже, вважаємо, що на вхід подаємо масив *A*, що складається з *n* елементів та для величини кожного елементу $A[i]$ виконується $0 \le A[i] < 1$. В якості додаткової пам'яті використаємо допоміжний масив $B[0..n-1]$ (масив зв'язаних списків).

Процедуру, що відповідає даному алгоритму, наведемо у наступному вигляді [2].

Bucket-Sort(*A*)
1   *n* = *A.length*
2   *B*[0..*n* – 1] – новий масив
3   **for** *i* = 1 **to** *n* – 1
4   Зробити *B*[*i*] пустим списком

5 **for** $i = 1$ **to**
6 Вставити $A[i]$ до списку $B[\lfloor nA[i] \rfloor]$
7 **for** $i = 1$ **to** $n - 1$
8 Відсортувати список $B[i]$ сортуванням вставкою
9 Поєднати списки $B[0]$, $B[1]$,…, $B[n-1]$ у вказаному порядку

Проаналізуємо алгоритм. З цією метою розглянемо елементи $A[i]$ та $A[j]$. Нехай $A[i] \le A[j]$. Так як $\lfloor nA[i] \rfloor \le \lfloor nA[j] \rfloor$, то елемент $A[i]$ розміщується або у кишеню разом з елементом $A[j]$, або у кишеню з меншим індексом. У першому випадку елементи $A[i]$ та $A[j]$ розташовуються у потрібному порядку завдяки циклу **for** (рядки 7-8). Якщо вони попали до різних кишень, то після виконання рядка 9 вони розташуються у потрібному порядку. Отже, алгоритм коректний.
Для виконання алгоритму у найгіршому випадку за всіма рядками, крім рядку 8, потрібен час O(n). Оцінимо повний час, що потрібен для n викликів алгоритму сортування вставкою (рядок 8). Для цього уведемо випадкову величину n_i – кількість елементів, які попали до кишені $B[i]$. Тоді час роботи алгоритму:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

Розглянемо виконання алгоритму у середньому випадку. Обчислимо математичне очікування часу роботи, де усереднення відбувається за вхідним розподілом:

$$E[T(n)] = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] = \Theta(n) + \sum_{i=0}^{n-1} E\left[O(n_i^2)\right] =$$
$$= \Theta(n) + \sum_{i=0}^{n-1} O\left(E\left[n_i^2\right]\right)$$

Покажемо, що для $i = 0, 1, \ldots, n - 1$ будемо мати $E[n_i^2] = 2 - 1/n$, так як до кожної i-ої кишені елементи вхідного масиву можуть попасти з однаковою ймовірністю. З цією метою визначимо для кожного $i = 0, 1, \ldots, n - 1$ та $j = 1, 2, \ldots, n$ індикаторну випадкову величину $X_{ij} = I\{ A[j]$ попаде до кишені $i\}$. Тоді

$$n_i = \sum_{j=1}^{n} X_{ij} .$$

$$E[n_i^2]=E\left[\left(\sum_{j=1}^{n}X_{ij}\right)^2\right]=E\left[\sum_{j=1}^{n}\sum_{k=1}^{n}X_{ij}X_{ik}\right]=$$

$$=E\left[\sum_{j=1}^{n}X_{ij}^2+\sum_{1\le j\le n}\sum_{\substack{1\le k\le n\\ k\ne j}}X_{ij}X_{ik}\right]=\sum_{j=1}^{n}E\left[X_{ij}^2\right]+\sum_{1\le j\le n}\sum_{\substack{1\le k\le n\\ k\ne j}}E[X_{ij}X_{ik}].$$

Окремо обчислимо кожну суму. Індикаторна випадкова величина X_{ij} дорівнює 1 з імовірністю $1/n$, тому:

$$E[X_{ij}^2]=1^2\frac{1}{n}+0^2\left(1-\frac{1}{n}\right).$$

При $k\ne j$ величини X_{ij} та X_{ik} незалежні, тому:

$$E[X_{ij}X_{ik}]=E[X_{ij}]E[X_{ik}]=\frac{1}{n}\cdot\frac{1}{n}=\frac{1}{n^2}.$$

Звідки

$$E[n_i^2]=\sum_{j=1}^{n}\frac{1}{n}+\sum_{1\le j\le n}\sum_{\substack{1\le k\le n\\ k\ne j}}\frac{1}{n^2}=n\frac{1}{n}+n(n-1)\frac{1}{n^2}=1+\frac{n-1}{n}=2-1/n.$$

Використовуючи одержане, получимо, що час роботи алгоритму кишенькового сортування у середньому випадку: $\Theta(n)+nO(2-1/n)=\Theta(n)$. Така ж оцінка буде й для випадку, коли сума піднесених до квадрату розмірів кишень лінійно залежить від кількості вхідних елементів.

Лекція 14. Аналіз алгоритмів пошуку *i*-ї порядкової статистики

І-ю порядковою статистикою множини з n елементів називають *i*-й елемент у порядку зростання (тому мінімальний елемент – це перша порядкова статистика, максимальний – *n*-та, медіана множини – для *n* парного має два елементи з індексом *n*/2 та *n*/2+1, для *n* непарного – один елемент з індексом (*n*+1)/2).

Формальна задача вибору *i*-ї порядкової статистики – це вибір елементу множини, що перевищує *i*-1 інших елементів цієї множини. Загально вживана асимптотична оцінка для часу вирішення такої задачі

O($n$lg$n$). За цей час виконується сортування елементів та вибір елементу вихідного масиву з індексом i.
Розглянемо найпростіший алгоритм та оцінимо кількість необхідних порівнянь, щоб знайти першу та n-ту порядкові статистики. Для кожного з них верхньою границею є $n - 1$. Процедуру пошуку першої порядкової статистики наведемо у наступному вигляді [2].

```
   Minimum(A)
1    min = A[1]
2    for i = 2 to A.length
3        if min > A[i]
4            min = A[i]
5    return min
```

Для даного випадку нижня границя має оцінку O($n$lg$n$), тому представлений алгоритм є оптимальним **по відношенню** до кількості порівнянь, що потрібно провести.
Розглянемо *одночасний пошук 1-ї та n-ої порядкових статистик*. Очевидною оцінкою кількості порівнянь для множини з n елементів є $\Theta(n)$. Відповідний алгоритм буде асимптотично оптимальним. З такою оцінкою достатньо виконати незалежний пошук вказаних елементів, що дасть в загальній сумі $2n - 2$ порівнянь. Але сталий коефіцієнт, схований в $\Theta(n)$, можна отримати і суттєво кращим (його дадуть $3\lfloor n/2 \rfloor$ порівнянь, якщо слідкувати, який з елементів є мінімальним, а який – максимальним).
Якщо не порівнювати окремо кожен вхідний елемент з поточним мінімумом та максимумом, а утворивши пару елементів, спершу порівняти їх один з одним, а потім менший порівняти з поточним мінімумом, більший – з максимумом, то для кожної пари елементів потрібно лише 3 порівняння.
Початковий вибір поточного мінімуму та максимуму залежить від парності кількості елементів вхідної множини. Для непарної кількості обирають 1 перший елемент, що вважається одночасно і мінімальним, і максимальним. Для парної кількості обирають 2 перші елементи, порівнюють їх, щоб визначити, який стане поточним мінімумом, а який – максимумом. Далі елементи завжди обробляють парами. Повна кількість порівнянь: для непарного n становить $3\lfloor n/2 \rfloor$; для парного n

маємо 1 початкове порівняння та ще $3(n-2)/2$ (разом $3n/2-2$). Тому для довільного n повна кількість порівнянь не перевищує величини $3\lfloor n/2 \rfloor$.

Розглянемо загальну задачу **вибору**. Вона є більш складною, ніж задача пошуку мінімуму або максимуму, але в асимптотичній границі має схожу поведінку – $\Theta(n)$. Проаналізуємо рандомізований алгоритм, що використовує метод декомпозиції, для розв'язання задачі вибору i-ї порядкової статистики. В ньому вхідний масив поділяється рекурсивно, але подальша робота відбувається лише з однією частиною розбиття.

```
   Randomized-Select(A, p, r, i)
1  if p == r
2      return A[p]
3      q = Randomized-Partition(A, p, r)
4      k = q – p + 1
5      if i == k     // відповіддю є опорне значення
6              return A[q]
7   elseif i < k
8      return Randomized-Select(A, p, q – 1, i)
9   else return Randomized-Select(A, q + 1, r, i – k)
```

В процедурі у рядку 1 перевіряється базовий випадок рекурсії. Якщо виконується базовий випадок, то елемент $A[p]$ повертається як i-та порядкова статистика (рядок 2). Інакше відбувається виклик процедури Randomized-Partition (рядок 3), яка розтинає масив A на два підмасиви (можливо пусті) відносно опорного елементу $A[q]$. Елементи першого підмасиву $A[p..q-1]$ не перевищують за величиною значення $A[q]$, а значення елементів другого підмасиву $A[q+1..r]$ більше ніж $A[q]$. У рядку 4 обчислюється кількість елементів підмасиву $A[p..q]$, а у рядку 5 перевіряється чи є елемент $A[q]$ шуканою i-ою порядковою статистикою. Якщо елемент $A[q]$ є i-ою порядковою статистикою, то повертається його значення (рядок 6). Інакше визначається в якому з двох підмасивів $A[p..q-1]$ або $A[q+1..r]$ міститься i-та порядкова статистика. Якщо $i < k$, то шуканий елемент знаходиться у нижній частині – у підмасиві $A[p..q-1]$, й рекурсивно вибирається з відповідного підмасиву (рядок 8); якщо $i > k$, потрібний елемент буде у верхній частині – у

підмасиві $A[q + 1..r]$, а так як вже відомі k значень, які є меншими значення i-ої порядкової статистики всього масиву $A[p..r]$, то шуканий елемент буде вже $(i - k)$-м у порядку зростання елементом підмасиву $A[q + 1..r]$, який шукаємо рекурсивно (рядок 9).

Час роботи процедури Randomized-Select у найгіршому випадку становить $\Theta(n^2)$, причому така оцінка справедлива навіть для пошуку мінімуму (найгірший випадок – якщо поділ завжди відбувається відносно найбільшого з елементів, що залишилися).

Оцінимо очікуваний час роботи процедури. Для цього розглянемо величину часу роботи над масивом $A[p..r]$ з n елементів, яку позначимо $T(n)$, як випадкову та одержимо верхню границю $E[T(n)]$. Маємо, процедура рівноймовірно повертає будь-який елемент в якості опорного. Тому для кожного k $(1 \le k \le n)$ підмасив $A[p..q]$ містить k елементів, які не перевищують опорний, з імовірністю $1/n$. Визначимо для $k = 1, 2,\ldots,n$ індикаторні випадкові величини X_k:

$$X_k = I\{\text{підмасив } A[p..q] \text{ містить рівно } k \text{ елементів}\}.$$

Якщо всі елементи різні, то $E[X_k] = 1/n$ (за основною властивістю індикаторної випадкової величини).

В момент виклику процедури та вибору в якості опорного елементу $A[q]$ наперед невідомо, чи одержимо відповідь і робота алгоритму припиниться, чи відбудеться рекурсивне звернення до підмасиву $A[p..q-1]$ або $A[q+1..r]$ (результат залежить від того, де буде розташований шуканий елемент відносно елементу $A[q]$). У припущенні монотонного неспадання функції $T(n)$ час, необхідний для рекурсивного виклику процедури Randomized-Select, можна обмежити зверху часом, необхідним для виклику цієї процедури з вхідним масивом максимально можливого розміру. Тобто, щоб одержати верхню границю, припускаємо, що шуканий елемент завжди попаде в ту частину поділу, що містить більше елементів. При конкретному виклику процедури індикаторна випадкова величина X_k приймає значення 1 лише за одного значення k, за всіх інших вона буде дорівнювати 0. Якщо ж $X_k = 1$, то розміри двох підмасивів, до яких може бути рекурсивне звернення, будуть $k - 1$ та $n - k$. Звідси одержимо:

$$T(n) \le \sum_{k=1}^{n} X_k \big(T(\max(k-1, n-k)) + O(n)\big) = \sum_{k=1}^{n} X_k T(\max(k-1, n-k)) + O(n).$$

Тому

$$E[T(n)] \le E\left[\sum_{k=1}^{n} X_k T(\max(k-1, n-k)) + O(n)\right] = \sum_{k=1}^{n} E[X_k T(\max(k-1, n-k))] + O(n) =$$

$$= \sum_{k=1}^{n} E[X_k] E[T(\max(k-1, n-k))] + O(n) = \sum_{k=1}^{n} \frac{1}{n} E[T(\max(k-1, n-k))] + O(n).$$

Розглянемо вираз $\max(k-1, n-k)$:

$$\max(k-1, n-k) = \begin{cases} k-1, \ k > \lceil n/2 \rceil, \\ n-k, \ k \le \lceil n/2 \rceil. \end{cases}$$

Якщо n парне, то кожний доданок від $T(\lceil n/2 \rceil)$ то Т(n-1) до суми входить двічі. Якщо n непарне, то двічі входять до суми всі доданки крім доданка $T(\lfloor n/2 \rfloor)$, який входить лише 1 раз. Отже, маємо:

$$E[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n).$$

Одержимо результуючу оцінку методом підстановки. Припустимо, що $E[T(n)] \le cn$ для деякої сталої c, що задовольняє початковим умовам рекурентного співвідношення. Нехай для n, що є меншими значення деякої поки що невідомої сталої, маємо $T(n) = O(1)$. Для складової $O(n)$ також виберемо сталу. Нехай це деяка стала a. Звідси маємо обмеження зверху величиною an для всіх $n > 0$. Тоді

$$E[T(n)] \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an = \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k\right) + an =$$

$$= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2}\right) + an \le$$

$$\le \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2}\right) + an =$$

$$= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an = c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an \le$$

$$\le c\left(\frac{3n}{4} + \frac{1}{2}\right) + an = cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right).$$

Покажемо, що для доволі великих значень n останній вираз не перевищує cn, а саме вираз у дужках не від'ємний. Розглянемо за яких умов $cn/4 - c/2 - an \ge 0$. Маємо,

$$c(n/4 - a) \ge c/2.$$

Якщо стала c вибрана такою, що $c > 4a$, то $n \geq 2c/(c - 4a)$. Тому, якщо припустити, що для всіх $n < 2c/(c - 4a)$ справедливе $T(n) = O(1)$, то

$$E[T(n)] \leq O(n).$$

Значить у припущенні, що всі елементи є різними, очікуваний час лінійно залежить від вимірності вхідних даних.

Проаналізуємо *алгоритм вибору i-ї порядкової статистики з лінійним часом роботи у найгіршому випадку.* Як і в попередньому алгоритмі вибір i-ї порядкової статистики відбувається шляхом рекурсивного поділу вхідного масиву. Але в цьому алгоритмі вже намагаються гарантувати гарний поділ масиву. Тут використовують таким чином модифіковану процедуру Partition, щоб одним з її вхідних параметрів був елемент, відносно якого відбувається поділ. У випадку $n = 1$ алгоритм повертає єдине вхідне значення.

З метою подальшого аналізу алгоритм вибору i-ї порядкової статистики **Select** сформулюємо у такому вигляді [2]:

1. Розбити всі n елементів вхідного масиву на $\lfloor n/5 \rfloor$ груп по 5 елементів та одну групу, що містить елементи, які залишилися (n mod 5 елементів) (рис. 14.1).
2. Відсортувати методом сортування вставкою кожну з $\lceil n/5 \rceil$ груп, у кожному відсортованому списку вибрати медіану.
3. Шляхом рекурсивного використання процедури Select визначається медіана x множини з $\lceil n/5 \rceil$ медіан на кроці 2 (якщо їх парна кількість, то медіана x одержує значення нижньої медіани).
4. За допомогою модифікованої процедури Partition вхідний масив поділяється відносно медіани медіан x. Якщо величина k на 1 перевищує кількість елементів, що попали до нижньої частини розбиття, тоді x є k-м у порядку зростання елементом, а до верхньої частини розбиття попаде $n - k$ елементів.
5. Якщо $i = k$, то повертається значення x. Інакше викликається рекурсивно процедура Select та з її допомогою виконується пошук i-ої порядкової статистики в нижній частині, якщо $i < k$, або, якщо $i > k$, то $(i - k)$-ої порядкової статистики у верхній частині.

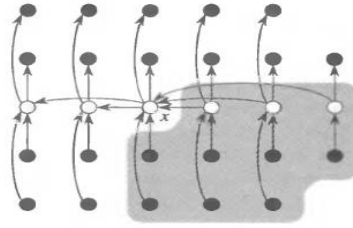


Рис. 14.1. Розбиття n елементів вхідного масиву – елементи, що перевищують x, виділено сірим [2].

Визначимо нижню границю елементів, що перевищують величину опорного елементу x. З рисунку 14.1 можна побачити, щонайменше половина медіан більше або дорівнює медіані медіан x. Далі, як мінімум $\lceil n/5 \rceil$ груп містять по 3 елементи, що перевищують величину x, виключенням є група, де міститься менше 5 елементів, та група, що сама містить елемент x. Тому, не враховуючи ці дві групи, маємо, що кількість елементів, величина яких перевищує x, буде не менше

$$3\left(\left\lceil \frac{1}{2}\lceil n/5 \rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6 .$$

Такий же висновок маємо для кількості елементів, що менші за x. Таким чином, процедуру рекурсивно викликають на кроці 5 не більше ніж для $7n/10$ елементів.

Нехай $T(n)$ – час роботи алгоритму. Тоді у найгіршому випадку для виконання кроків 1, 2 (має $O(n)$ викликів процедури сортування вставкою в групі, де щоразу оцінка часу $O(1)$), кроку 4 потрібно $O(n)$ часу. Виконання кроку 3 займе $T(\lceil n/5 \rceil)$ часу, кроку 5 – не більше $T(7n/10+6)$. Зробимо наперед припущення для граничних умов. Тоді

$$T(n) \leq \begin{cases} O(1), & n < 140, \\ T(\lceil n/5 \rceil) + T(7n/10+6) + O(n), & n \geq 140. \end{cases}$$

Методом підстановки одержимо величину оцінки часу роботи для наведеного виразу. Припустимо, що для деякої доволі великої сталої c та всіх додатних n виконується: $T(n) \leq cn$. Припустимо, що нерівність виконується для граничних умов, тобто для достатньо великої сталої c та всіх $n < 140$. Також виберемо сталу a таку, щоб функція, яка відповідає доданку $O(n)$ для всіх додатних n буда обмежена величиною an. Тоді

$$T(n) \le c\lceil n/5 \rceil + c(7n/10+6) + an \le cn/5 + c + 7cn/10 + 6c + an =$$
$$= 9cn/10 + 7c + an = cn + (-cn/10 + 7c + an).$$

Покажемо, що для доволі великих n вираз не перевищує cn. Якщо так, то $-cn/10+7c+an \le 0$.

Тоді $c \ge 10a \ (n/(n-70))$, де $n > 70$. Так як припустили, що $n \ge 140$, то $n/(n-70) \le 2$, тому, якщо узяти $c \ge 20a$, то маємо $T(n) \le cn$. Тому у найгіршому випадку час роботи алгоритму лінійно залежить від кількості вхідних елементів.

Як і в попередніх алгоритмах сортування порівнянням, у розглянутих алгоритмах Randomized-Select та Select інформація про взаємне розташування елементів одержується шляхом їх порівняння. Як доводили раніше, сортування виконується за час $\Omega(n \lg n)$ навіть у середньому. В алгоритмі, де час роботи лінійно залежить від кількості елементів, що сортуємо, додатково робилися припущення відносно вхідних даних. У наведених алгоритмів час роботи також лінійно залежить від розміру вхідних даних, додаткових припущень не потрібно, а задача вибору в них вирішується без сортування. Тому бачимо, що сама найперша наведена процедура є асимптотично неефективною.

Лекція 15. Побудова та аналіз алгоритмів динамічного програмування

Будуючи алгоритми динамічного програмування, як і при використанні методу декомпозиції, розв’язуємо задачу, комбінуючи розв’язки допоміжних підзадач. Але, якщо раніше мали декілька незалежних підзадач, що вирішувалися рекурсивно, а з їхніх розв’язків формували розв’язок початкової задачі, при побудові алгоритмів динамічного програмування маємо задачі, що перекриваються (різні підзадачі використовують для рішення одних і тих же підзадач вищого рівня). Ці підзадачі вирішуються тільки один раз, далі їх відповідь зберігають у таблиці та використовують за необхідності. Передумовами використання підходу є наявність оптимальної підструктури (коли в оптимальному рішенні містяться оптимальні рішення підзадач) та підзадач, що перекриваються.

Як відомо, динамічне програмування звичайно застосовують до задач оптимізації. В таких задачах з кожним варіантом рішення можна співставити деяке значення, для рішення задачі необхідно знайти розв'язок з оптимальним у деякому сенсі значенням. Такий розв'язок називають одним з можливих оптимальних розв'язків. Сам процес побудови алгоритмів динамічного програмування поділяють на **етапи**:

- опис структури оптимального рішення;
- визначення значення, що відповідає оптимальному рішенню, з використанням рекурсії;
- обчислення значення, що відповідає оптимальному рішенню, звичайно методом висхідного аналізу;
- можливе складання оптимального рішення на основі попередньо одержаної інформації.

Для демонстрації побудови алгоритмів динамічного програмування розглянемо найпростіший приклад – *задачу про переріз прутка*. В задачі необхідно визначити як краще розрізати пруток довжиною n, щоб одержати максимальний прибуток r_n, якщо переріз прутка безкоштовний, задані ціни p_i шматків довжиною i. Приклад конкретних значень довжини прутків та їх ціни наведені у таблиці 15.1.

Таблиця 15.1. Значення довжин прутків та їх ціни

Довжина, i	1	2	3	4	5	6	7	8	9	10
Ціна, p_i	1	5	8	9	10	17	17	20	24	30

Якщо, наприклад, довжина прутку $n = 4$. За наведеними у таблиці цінами маємо всі варіанти перерізу такого прутка. Найкращим буде переріз прутка на 2 шматки із довжиною 2, що дасть прибуток $p_2 + p_2 = 10$. Взагалі пруток довжиною n можна розрізати 2^{n-1} різними способами (можна незалежно вибирати різати чи ні на відстані i від лівого краю, $i = 1,2,\ldots n - 1$). Нехай пруток розрізаємо на k частин ($1 \le k \le n$), коли оптимальне розбиття $n = i_1 + i_2 + \ldots + i_k$ дає максимальний прибуток $r_n = p_{i_1} + p_{i_2} + \ldots + p_{i_k}$.

Величину прибутку r_n запишемо через оптимальні прибутки від більш коротких прутків:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1).$$

Оскільки наперед невідомо, яке значення $r_i + r_{n-i}$ оптимізує прибуток, маємо розглянути всі можливі значення i та вибрати серед них те, що максимізує дохід.

Для розв'язання початкової задачі вирішуємо менші задачі такого ж виду. Після розрізу обидві частини прутка можемо розглядати як **незалежні екземпляри задачі** розрізу прутка.

Загальний оптимальний розв'язок включає оптимальні розв'язки двох пов'язаних підзадач. Тобто, задача має **оптимальну підструктуру:** оптимальне рішення задачі включає оптимальні рішення підзадач, що можуть бути розв'язані незалежно. У найбільш простому способі організації рекурсивної підструктури задачі розглядаємо розріз прутка як такий, що складається з першої частини довжиною i (далі не розрізається) та частини довжиною n-i. Рішення без розрізів припускається. Тому останній вираз можна переписати у вигляді, що включає лише одну зв'язану підзадачу:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}) .$$

Використовуючи вказане наведемо **рекурсивну низхідну** реалізацію розв'язання задачі у вигляді наступної процедури [2]:

```
   Cut-Rod(p, n)
1  if n == 0
2      return 0
3      q = –∞
4  for i = 1 to n
5      q = max(q, p[i] + Cut-Rod(p, n – i))
6  return q
```

У процедурі вхідними даними є масив цін p та довжина прутка n. У рядках 1-2 перевіряється базовий випадок, за якого немає прибутку та повертається значення 0. У рядку 3 ініціалізується максимальний прибуток q значенням $-\infty$, а у рядках 4-5 цикл **for** обчислює значення $\max(q, p[i] + \text{Cut-Rod}(p, n - i))$, яке повертається у рядку 6.

Але наведена процедура є неефективною, так як багаторазово вирішує одні й ті ж підзадачі. Маємо, що загальна кількість викликів процедури з другим параметром рівним n, буде дорівнювати числу вузлів у під-

дереві з коренем з міткою n в дереві рекурсії. Це число включає і початковий виклик в корені. Наприклад, навіть для n = 4 (рис. 15.1) одержимо 2^4 вузлів та 2^3 листків. Тому

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j),$$

де $T(0)=1$, а $j = n - i$.

Методом підстановки одержимо, що $T(n) = 2^n$.

При використанні динамічного програмування одержані рішення підзадач зберігаємо, щоб забезпечити одноразове рішення цих підзадач.

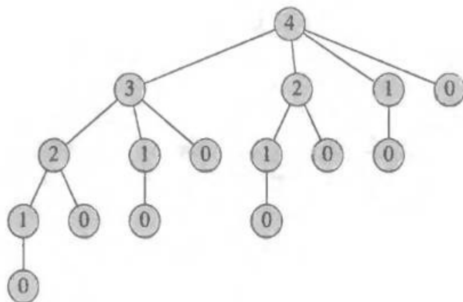


Рис. 15.1. Дерево рекурсії для n =4, шлях від кореня до листку відповідає одному з 2^{n-1} способів розрізу прутка [2].

Це вимагає додаткової пам’яті, але компроміс в результаті дає велику економію часу: рішення з експоненціальним часом роботи можна перетворити на рішення з поліноміальним часом (якщо кількість різних підзадач поліноміально залежить від розміру вхідних даних, то можна вирішити кожну з них за поліноміальний час).

Побудуємо рішення задачі розрізу прутка методом динамічного програмування **низхідним із запам’ятовуванням**. Метод має рекурсивну процедуру, але вона модифікується так, щоб запам’ятовувати рішення кожної підзадачі, а також перевіряти, чи вирішена вже була ця підзадача. Якщо вона не вирішувалася, то процедура обчислює звичайним чином значення, що повертається.

Також далі побудуємо рішення задачі й **висхідним** методом. В цьому випадку сортують підзадачі за розмірами у порядку зростання. При рішенні деякої визначеної підзадачі необхідно вирішити всі підзадачі

меншого розміру (від яких вона залежить) та зберегти одержані розв'язки. Кожну підзадачу вирішують одноразово, на момент, коли з нею стикаються, всі необхідні для її вирішення підзадачі вже мають бути розв'язані.

Для обох підходів одержуються *алгоритми з однаковим асимптотичним часом роботи* за виключенням ситуацій, коли низхідний підхід не виконує рекурсивного вивчення всіх можливих підзадач. Але звичайно оцінки часу для висхідного підходу мають кращі константні множники за рахунок менших накладних витрат, пов'язаних з викликами функцій.

Низхідна процедура із запам'ятовуванням (масив $r[0..n]$ – додатковий):

```
   Memoized-Cut-Rod(p, n)
1  r[0..n] – новий масив
2      return 0
3  for i = 1 to n
5      r[i] = – ∞
6  return Memoized-Cut-Rod-Aux(p, n, r)
```

```
   Memoized-Cut-Rod-Aux(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5   else q = – ∞
6      for i = 1 to n
7          q = max(q, p[i] + Memoized-Cut-Rod-Aux(p, n – i, r))
8   r[n] = q
9  return q
```

Висхідна процедура із запам'ятовуванням (масив $r[0..n]$ – додатковий, використовується природнє впорядкування підзадач i та j, $i < j$):

```
   Bottom-Up-Cut-Rod(p, n)
1  r[0..n] – новий масив
```

```
2    r[0] = 0
3    for j = 1 to n
4        q = −∞
5        for i = 1 to j
6            q = max(q, p[i] + r[j − i])
7        r[j] = q
8    return r[n]
```

Як бачимо, обидві версії мають однаковий асимптотичний час роботи. Вкладена структура циклів у висхідній процедурі дає оцінку часу роботи $\Theta(n^2)$. Оцінка часу роботи низхідної процедури теж буде $\Theta(n^2)$. В низхідній процедурі кожна підзадача вирішується одноразово, для рішення підзадачі розміром *n* цикл for у рядках 6-7 виконує *n* ітерацій, загальна кількість ітерацій і буде становити $\Theta(n^2)$.

Так як розв'язок задачі розрізу прутка за допомогою динамічного програмування повертає лише значення оптимального рішення, а не сам розв'язок, що є списком розміру частин, то необхідний етап відновлення розв'язку. Для цього можна записувати не тільки обчислене оптимальне значення кожної підзадачі, а й вибір, що приводить до цього значення. Це й дасть можливість легко одержати оптимальний розв'язок. В такому варіанті, наприклад, висхідна процедура може набути наступного вигляду (для кожного розміру *j* прутка обчислюють не тільки відповідний максимальний прибуток r_j, а й розмір першої відрізаної частини s_j (додатковий масив $s[1..n]$)) [2]:

```
   Extended-Bottom-Up-Cut-Rod(p, n)
1    r[0..n] та s[0..n] – нові масиви
2    r[0] = 0
3    for j = 1 to n
4        q = −∞
5        for i = 1 to j
6            if q < p[i] + r[j − i]
7                q = p[i] + r[j − i]
8                s[j] = i
```

```
9          r[j] = q
10   return r, s
```

Оцінка часу роботи процедури Extended-Bottom-Up-Cut-Rod теж буде $\Theta(n^2)$.

Додаткова процедура одержує таблицю цін *p* та розмір прутка *n* й викликає процедуру Extended-Bottom-Up-Cut-Rod для обчислення масиву $s[1..n]$ оптимальних розмірів перших частин, а потім виводить повний список розмірів частин для оптимального розрізання прутка довжиною *n*. Процедура, може мати, наприклад, вигляд:

```
   Print-Cut-Rod-Solution(p, n)
1     (r, s) = Extended-Bottom-Up-Cut-Rod(p, n)
2     while n > 0
3        print s[n]
4           n = n − s[n]
```

При вирішенні задачі розрізу прутка у рекурентному виразі мали лише одну зв'язану задачу, наступна задача – *задача про множення ланцюга матриць*, буде мати вже дві зв'язані задачі різного розміру.

Нехай маємо послідовність *n* матриць $\langle A_1, A_2, ..., A_n \rangle$, необхідно обчислити їх добуток. Його можна обчислити використовуючи стандартний алгоритм множення пар матриць. Але внаслідок асоціативності множення матриць, а тому й різних варіантів розставлення дужок, час обчислення добутку може суттєво відрізнятися. Наприклад, маємо для множення послідовність з трьох матриць $\langle A_1, A_2, A_3 \rangle$, де розміри першої матриці – 10×100, другої – 100×5, а третьої – 5×50. Якщо будемо множити матриці як $((A_1 A_2) A_3)$, то повинні виконати $10 \cdot 100 \cdot 5 = 5000$ скалярних добутків, щоб знайти добуток матриць у внутрішній дужці, що дасть матрицю розміром 10×5, та ще $10 \cdot 5 \cdot 50 = 2500$ скалярних добутків, щоб знайти добуток у зовнішній дужці. Разом це 7500 скалярних добутків. Якщо будемо множити матриці як $(A_1 (A_2 A_3))$, то маємо виконати $100 \cdot 5 \cdot 50 = 25000$ скалярних добутків, щоб знайти добуток матриць у внутрішній дужці, що дасть матрицю вже розміром 100×50, та ще $10 \cdot 100 \cdot 50 = 50000$ скалярних добутків, щоб знайти добуток вже

всіх трьох матриць. Разом це 75000 скалярних добутків, що в 10 разів більше попередньої кількості, а тому і в 10 разів більше вимагатиме часу на обчислення.

Тому задачу множення ланцюга матриць формулюють так: для заданої послідовності n матриць $\langle A_1, A_2, ..., A_n \rangle$ за допомогою дужок визначити порядок множень у матричному добутку, за якого кількість скалярних добутків буде мінімальною.

Для $n \geq 2$ добуток послідовності матриць, в якому порядок множення визначений дужками, є добутком двох таких добутків підпослідовностей, в яких порядок множення також визначений дужками. Розтин на підпослідовності може відбуватися на границі k та $k+1$ матриць для будь-якого $k = 1, 2, \dots, n-1$. Нехай $P(n)$ – кількість різних способів розстановки дужок у послідовності $\langle A_1, A_2, ..., A_n \rangle$. Тоді

$$P(n) = \begin{cases} 1, & n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k), & n \geq 2. \end{cases}$$

Методом підстановки можемо одержати, що розв'язок цього рекурентного співвідношення – $\Omega(2^n)$ (більш точно: $\Omega(4^n / n^{3/2})$), тому метод прямого перебору всіх варіантів і в цій задачі не підходить для розв'язання.

Побудуємо алгоритм динамічного програмування. Для цього виконаємо таку послідовність етапів: опишемо структуру оптимального рішення; визначимо значення, що відповідають оптимальному рішенню, з використанням рекурсії; обчислимо значення, що відповідають оптимальному рішенню, з використанням методу висхідного аналізу; складемо оптимальне рішення на основі вже отриманої інформації.

Отже, перше, спробуємо знайти *оптимальну підструктуру*, щоб з її допомогою побудувати оптимальне рішення задачі по оптимальним рішенням підзадач. Позначимо результат добутку матриць $A_i A_{i+1} ... A_j$ як $A_{ij}, i \leq j$. Якщо задача нетривіальна, то будь-який спосіб розстановки дужок у добутку $A_i A_{i+1} ... A_j$ розбиває його між матрицями k та $k+1$, $i \leq k < j$. Тому маємо спочатку обчислення добутку A_{ik}, $A_{k+1\,j}$

, а потім множення одна на одну, що дасть A_{ij}. Вартість буде рівною вартості обчислення A_{ik} плюс обчислення $A_{k+1\,j}$ та вартості обчислення їх добутку.

Покажемо наявність оптимальної підструктури задачі. Нехай в результаті оптимального розставлення дужок для $A_i A_{i+1} ... A_j$ поділ відбувся між матрицями A_k та A_{k+1}, $i \le k < j$. Ствердимо, що розставлення дужок у підпослідовності $A_i A_{i+1} ... A_k$ також є оптимальним. Дійсно, якби існував кращий спосіб розставлення дужок у підпослідовності $A_i A_{i+1} ... A_k$, то його застосування дозволило б перемножити матриці ще ефективніше, а це протирічить припущенню про оптимальність. Аналогічним чином робимо висновок про оптимальність розставлення дужок у підпослідовності $A_{k+1} ... A_j$.

Покажемо, що *одержана оптимальна підструктура дозволяє побудувати оптимальне рішення* з оптимальних рішень підзадач. Маємо, що для вирішення задачі всю послідовність необхідно розбити на підпослідовності, та кожен оптимальний розв'язок містить в собі оптимальні розв'язки підзадач. Тому розв'язок повної задачі можна побудувати шляхом розбиття на дві підзадачі – для $A_i A_{i+1} ... A_k$ та $A_{k+1} ... A_j$. А після їх вирішення оптимальний розв'язок повної задачі складається з цих розв'язків. Але необхідно впевнитися, що враховуємо всі можливі варіанти поділу. Якщо так, то знайдений розв'язок буде глобально оптимальним.

Побудуємо *рекурсивне рішення.* Для цього визначимо рекурсивно вартість оптимального рішення в термінах оптимальних рішень підзадач. Маємо послідовність *n* матриць $A_i A_{i+1} ... A_j$, $1 \le i \le j \le n$. Нехай $m[i, j]$ – мінімальна кількість скалярних добутків, що необхідні для обчислення A_{ij} (для повної задачі це $m[1,n]$). Визначимо рекурсивно величину $m[i, j]$. Якщо $i = j$, то задача є тривіальною і $m[i,i] = 0$. При $i < j$, використовуючи властивість підструктури оптимального рішення, припустимо, що в результаті оптимального розставлення дужок послідовність $A_i A_{i+1} ... A_j$ розбивається між матрицями A_k та A_{k+1}, $i \le k < j$. Тоді $m[i, j]$ дорівнює мінімальній вартості обчислення добутків A_{ik} та

$A_{k+1\,j}$, а також вартості обчислення їхнього добутку для одержаних двох матриць. Нехай матриця A_i має розміри $p_{i\text{-}1} \times p_i$, A_k – $p_{k\text{-}1} \times p_k$ і т.д. Тоді для обчислення добутку $A_{ik} A_{k+1\,j}$ потрібно $p_{i\text{-}1} p_k p_j$ скалярних добутків. Звідки, вважаючи, що k відоме,

$$m[i, j] = m[i,k] + m[k+1, j] + p_{i\text{-}1} \times p_k \times p_j .$$

Але насправді k невідоме. Для вибору цього значення маємо $j - i$ можливостей (а саме $k = i$, $i + 1$, …, $j - 1$). Так як в оптимальному розставленні потрібно використати одне із цих значень k, то необхідно перевірити всі можливості та вибрати найкращу. Тоді рекурсивне визначення оптимального розставлення дужок у добутку $A_i A_{i+1} ... A_j$ має вигляд:

$$m[i, j] = \begin{cases} 0, & i = j, \\ \min\limits_{i \le k < j} \{ m[i,k] + m[k+1, j] + p_{i\text{-}1} \times p_k \times p_j \}, & i < j. \end{cases}$$

Для побудови оптимального рішення позначимо $s[i, j]$ значення k, в якому послідовність $A_i A_{i+1} ... A_j$ мала розбиття на дві підпослідовності для оптимального розставлення дужок, тоді $s[i, j] = k$ такому, що $m[i, j] = m[i,k] + m[k+1, j] + p_{i\text{-}1} \times p_k \times p_j$.

Обчислення оптимальних вартостей для одержання добутків. Спочатку звернемо увагу на наші підзадачі. Їх кількість не дуже велика – по одній для кожного вибору j та i, де $1 \le i \le j \le n$ ($C_n^2 + n = \Theta(n^2)$). В рекурсивному алгоритмі, як бачили і в попередній задачі, кожна підзадача може неодноразово зустрічатися в різних гілках рекурсивного дерева. Така властивість перекриття підзадач, як базова властивість застосовуваності динамічного програмування, разом з оптимальною підструктурою, приводить до побудови таблиці для підзадач. Тобто, замість того, щоб постійно рекурентно розв'язувати підзадачі, обчислимо оптимальну вартість шляхом побудови таблиці у висхідному напрямку. За вказаних вище позначень для розмірів матриць послідовності маємо послідовність розмірів $p = \langle p_0, p_1, ..., p_n \rangle$ довжиною n+1. Використаємо допоміжні таблиці: $m[1..n, 1..n]$ для зберігання

вартостей $m[i, j]$ та $s[1..n,1..n]$ – для індексів k, за яких досягається оптимальна вартість $m[i, j]$.

Побудова висхідної процедури. Визначивши за допомогою яких саме записів таблиці обчислюються величини $m[i, j]$, зможемо записати відповідну коректну процедуру. Маємо, що вартість $m[i, j]$ обчислення добутку послідовності $j - i + 1$ матриць залежить тільки від вартості обчислення послідовностей матриць, які мають менше $j - i + 1$ матриць. При $k = i, i+1, \ldots, j - 1$ матриця $A_{i\,k}$ є добутком $k - i + 1 < j - i + 1$ матриць, а матриця $A_{k+1\,j}$ – добутком $j - k < j - i + 1$ матриць. Тому потрібно забезпечити заповнення таблиці $m[1..n,1..n]$ у порядку, що відповідає рішенню задачі в послідовностях матриць зростаючої довжини. Для ланцюга $A_i A_{i+1} \ldots A_j$ розглядаємо розмір підзадачі рівний довжині ланцюга $j - i + 1$. Висхідну процедуру з обчисленням вартостей $m[i, j]$ та запам'ятовуванням індексів k, за яких досягається оптимальна вартість наведемо у наступному вигляді [2].

```
   Matrix-Chain-Order(p)
 1   n = p.length – 1
 2   m[1..n, 1..n], s[1..n – 1, 2..n] – нові таблиці
 3   for i = 1 to n
 4       m[i, i] = 0
 5   for l = 2 to n  // l – довжина ланцюга
 6       for i = 1 to n – l + 1
 7           j = i + l – 1
 8           m[i, j] = ∞
 9           for k = i to j – 1
10               q = m[i, k] + m[k + 1, j] + p_{i-1} × p_k × p_j
11               if q < m[i, j]
12                   m[i, j] = q
13                   s[i, j] = k
14   return m,s
```

В процедурі Matrix-Chain-Order(p) у рядках 3-4 відбувається ініціалізація у таблиці $m[i, i] = 0$ для $i = 1, \ldots, n$, що є мінімальними вартостями для послідовностей довжиною 1. У циклі **for** (рядки 5-13) у першій ітерації рекурентно обчислюються $m[i, i+1]$ при $i = 1, \ldots, n - 1$, що є

мінімальними вартостями для послідовностей довжиною 2; у другій ітерації рекурентно обчислюються $m[i, i+1]$ при $i = 1, \ldots, n - 2$, мінімальними вартостями для послідовностей довжиною 3 і т.д. У рядках 10-13 обчислюються значення $m[i, j]$, що залежать від вже занесених до таблиці $m[i,k]$ та $m[k+1, j]$.

Оскільки величини $m[i, j]$ визначені для $i \le j$, то використовується лише частина таблиці, що розташована над головною діагоналлю. В процедурі рядки обчислюються знизу доверху, а елементи у кожному рядку – зліва направо; $m[i, j]$ – за допомогою добутків $p_{i\text{-}1} p_k p_j$ для $k = i, i+1, \ldots, j - 1$ та всіх величин унизу зліва та унизу справа від $m[i, j]$.

Час роботи процедури (за рядками та циклами) складає $O(n^3)$. А для зберігання таблиць m та s потрібно $\Theta(n^2)$ додаткового об'єму пам'яті.

Побудова оптимального розв'язку. В процедурі не показано як саме перемножуються матриці. Оптимальний розв'язок можна побудувати за допомогою інформації з таблиці $s[1..n-1, 2..n]$. Як пам'ятаємо, в кожному елементі цієї таблиці зберігається значення k, в якому послідовність $A_i A_{i+1} \ldots A_j$ мала розбиття на дві підпослідовності для оптимального розставлення дужок. Тому відомо, що оптимальне розбиття виглядає як $A_{1,s[1,n]}\ A_{s[1,n]+1,n}$. Усі попередні добутки матриць можна обчислити рекурсивно, оскільки елемент $s[1, s[1, n]]$визначає матричний добуток, що виконується останнім при обчисленні $A_{1,s[1,n]}$, а $s[s[1, n]+1, n]$ – останнім при обчисленні $A_{s[1,n]+1,n}$.

Наведений приклад рекурсивної процедури дозволяє виводити оптимальне розставлення дужок за таблицею $s[1..n-1, 2..n]$ та індексами j, i.

```
   Print-Optimal-Parens(s, i, j)
1     if i == j
2          print "A"i
3     else print "("
4          Print-Optimal-Parens(s, i, s[i, j])
5          Print-Optimal-Parens(s, s[i, j] + 1, j)
6           print ")"
```

Наприклад, для послідовності з шести матриць з послідовністю розмірів $p = \langle p_0, p_1, \ldots, p_6 \rangle = \langle 30, 35, 15, 5, 10, 20, 25 \rangle$ в результаті роботи цієї процедури для $i = 1$, $j = 6$ одержимо: $((A_1(A_2A_3))((A_4A_5)A_6))$.

Низхідне рекурсивне рішення. З цією метою знову звернемося до оптимальної підструктури. На етапі її виявлення та характеризування повинні впевнитися, що до розглянутих підзадач входять всі, що використовуються в оптимальному рішенні. Якби ще не виявляли оптимальної підструктури для низхідного розв'язання, то обов'язково мали б зробити наступне:

1. Показати, що в процесі рішення потрібно робити вибір. Після вибору потрібно розв'язувати одну або декілька підзадач.
2. Довести, що зроблений вибір веде до оптимального рішення (але для цього як саме потрібно робити вибір не розглядається).
3. Виходячи із зробленого вибору визначити, які підзадачі одержуються та як краще охарактеризувати одержаний в результаті простір задач.
4. Показати, що рішення підзадач, які виникають в ході оптимального рішення задачі, самі повинні бути оптимальними (звичайно робиться методом «від супротивного»).

При виконання цієї послідовності дій для конкретних задач необхідно прагнути, щоб простір підзадач був як можна простішим, далі його розширюють до потрібних розмірів. Так, в задачі про множення ланцюга матриць обмежуємо простір підзадач: в нашій задачі це $A_i A_{i+1} \ldots A_j$ (з i, а не з 1, так як без зміни обох індексів підзадачі можуть не бути підзадачами, що мають 1 зліва в індексі), розбиття між матрицями A_k та A_{k+1}, $i \le k < j$.

Кількість підзадач, що використовується в оптимальному рішенні, та варіантів вибору може бути різною. В нашій задачі маємо 2 підзадачі та $j - i$ варіантів вибору. Але час роботи алгоритму динамічного програмування залежить від добутку двох множників: загальної кількості підзадач та кількості варіантів вибору, що виникають у кожній підзадачі (це сума вартостей рішень підзадач та вартості витрат на визначення вірного вибору. Так, для розрізу прутків мали $\Theta(n)$ підзадач та не більне ніж n варіантів вибору (вартість, що відноситься до самого вибору – член p_i), що дає $O(n^2)$ для часу роботи. Для множення матриць всього виникало $\Theta(n^2)$ підзадач, а в кожній з них

не більше n-1 варіантів вибору (вартість, що відноситься до самого вибору – член $p_{i-1}p_kp_j$), тому $\Theta(n^3)$. Крім того, необхідно звернути увагу на незалежність підзадач. Якщо вони не будуть незалежними, то оптимальної підструктури не виникне.

При побудові висхідної процедури звертали увагу на важливість *перекриття підзадач*. У зв'язку з цим знову зауважимо, що простір підзадач повинен бути невеликим (одні й ті ж підзадачі вирішуються знову і знову, нових не виникає). Звичайно для можливого застосування динамічного програмування повна кількість підзадач – поліноміальна функція від вимірності вхідних даних. Якщо ж рекурсивний алгоритм знову і знову звертається до однієї й тієї ж підзадачі, то значить, що задача оптимального розставлення дужок містить підзадачі, що перекриваються. В нашій задачі, наприклад, для ланцюга з чотирьох матриць різних підзадач, що будуть занесені до таблиці буде всього 10: [1..1], [2..2], [3..3], [4..4], [1..2], [2..3], [3..4], [1..3], [2..4], [1..4]. А всього у процесі розв'язання такої задачі ([1..4]) звертаємося до 26 підзадач меншого розміру.

Зважаючи на те, що вже виконані дослідження оптимальної підструктури задачі та одержано рекурсивну залежність, що включає зв'язані підзадачі, наведемо неефективну рекурсивну процедуру (без запам'ятовування) [1], яку далі вдосконалимо.

```
   Recursive-Matrix-Chain(p, i, j)
1    if i == j
2        return 0
3    m[i, j] = ∞
4    for k = i to j – 1
5          q = Recursive-Matrix-Chain(p, i, k)
                 +Recursive-Matrix-Chain(p, k+1, j)+ p_{i-1} p_k p_j
6          if q < m[i, j]
7              m[i, j] = q
8    return m[i, j]
```

Покажемо, що час обчислення $m[1,n]$ процедурою Recursive-Matrix-Chain як мінімум експоненціально залежить від n. Нехай $T(n)$ – час, що потрібен процедурі для оптимального розставлення дужок у ланцюгу з

n матриць. Рядки 1, 2, 6, 7 вимагають константного часу, одноразово рядок 5 у циклі також, тому

$$T(n) \geq \begin{cases} 1, & n = 1, \\ 1 + \sum_{k=1}^{n-1} (T(k) + N(n-k) + 1), & n > 1. \end{cases}$$

У цьому виразі при аргументі k = 1, 2, …, n – 1 кожний доданок $T(i)$ з'являється двічі (при наближеннях зліва та справа), тому

$$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n.$$

Покажемо методом підстановок, що T(n) $=\Omega(2^n)$, а саме: для всіх $n \geq 1$ має місце T(n) $\geq 2^{n-1}$. Маємо T(1) $\geq 1 = 2^0$. Далі, для всіх $n > 1$:

$$T(n) \geq 2\sum_{i=1}^{n-1} 2^{i-1} + n = 2\sum_{i=0}^{n-2} 2^i + n = 2(2^{n-1} - 1) + n \geq 2^{n-1}.$$

Звідси випливає, що низхідний рекурсивний алгоритм (простий, без запам'ятовування) як мінімум експоненційно залежить від n.

Побудова оптимального рішення. Якщо у таблиці зберігаємо інформацію про те, який був зроблений вибір (який оптимальний розв'язок) в кожній підзадачі, то не потрібно додатково вирішувати задачу про поновлення цієї інформації. Використання таблиці $s[i, j]$ надає таку можливість.

Для реалізації *запам'ятовування* використовують таблицю з розв'язками підзадач. Для наведеного простого рекурсивного алгоритму підтримка запам'ятовування означає підтримку відповідного елементу таблиці. Спочатку в кожному елементі таблиці міститься спеціальне значення, що вказує на незаповненість даного елементу. Якщо в процесі виконання алгоритму підзадача зустрічається вперше, її розв'язують та рішення заносять до таблиці. Якщо вона вже зустрічалася, виконується пошук рішення в таблиці. В цьому процесі використовується також таблиця значень $m[i, j]$ мінімальної кількості скалярних добутків для обчислення $A_{i\,j}$. Відповідний низхідний алгоритм із запам'ятовуванням представимо, наприклад, у вигляді [2]:

```
   Memoized-Matrix-Chain(p)
1     n = p.length – 1
2     m[1..n, 1..n] – нова таблиця
```

```
3     for i = 1 to n
4         for j = i to n
5             m[i, j] = ∞
6     return Lookup-Chain(m, p 1, n)
```

```
   Lookup-Chain(m, p, i, j)
1     if m[i, j] = ∞
2         return m[i, j]
3     if i == j
4         m[i, j] = 0
5     else for k = i to j – 1
6             q = Lookup-Chain(m, p, i, k)
              + Lookup-Chain(m, p, k +1, j) + p_{i-1} p_k p_j
7             if q < m[i, j]
8                 m[i, j] = q
9     return m[i, j]
```

Час роботи низхідного алгоритму буде, як і у висхідному випадку, $O(n^3)$: рядок 5 виконується $\Theta(n^2)$ разів; рекурсивні виклики процедури, в яких $m[i, j] < \infty$, $O(n^3)$ разів з часом $O(1)$; рекурсивні виклики процедури, в яких $m[i, j] = \infty$, щоразу час $O(n)$ плюс час, витрачений на рекурсивне звернення в даному виклику, що дасть $O(n^3)$. Але, якщо маємо вирішувати всі підзадачі хоча б одноразово, висхідний алгоритм динамічного програмування звичайно працює швидше, ніж низхідний рекурсивний із запам'ятовуванням (за рахунок відсутності накладних витрат на рекурсію, менших витрат при підтримці таблиці).

Лекція 16. Приклади побудови та аналізу алгоритмів динамічного програмування

Задача про найдовшу спільну підпослідовність. Нехай маємо дві послідовності елементів S_1, S_2 (2 рядки символів), що складаються із скінченої множини елементів (символів). Необхідно визначити степінь їх схожості. Для цього будемо шукати таку послідовність S_3, основи

якої є як в S_1, так і в S_2 та вони йдуть в одному й тому ж порядку, але необов'язково одна за одною. Чим довша підпослідовність S_3, тим більше схожі послідовності S_1 та S_2.

Сформулюємо це ж саме дещо іншими словами. Підпослідовність даної послідовності – це дана послідовність, з якої вилучили 0 або більше елементів. Тобто послідовність $Z = \langle z_1, z_2, ..., z_k \rangle$ є підпослідовністю послідовності $X = \langle x_1, x_2, ..., x_m \rangle$, якщо існує строго зростаюча послідовність $\langle i_1, i_2, ..., i_k \rangle$ індексів X, така, що для всіх $j = 1, 2, \ldots, k$ виконується співвідношення $x_{i_j} = z_j$. Послідовність Z є *загальною підпослідовністю послідовностей* X та Y, якщо Z є підпослідовністю X та Y підпослідовністю. Тому в задачі про найдовшу спільну підпослідовність задають дві послідовності $X = \langle x_1, x_2, ..., x_m \rangle$ та $Y = \langle y_1, y_2, ..., y_n \rangle$. Шукають спільну підпослідовність цих двох послідовностей максимальної довжини (LCS).

Побудуємо алгоритм динамічного програмування для розв'язання задачі. Для цього: опишемо структуру оптимального рішення; визначимо значення, що відповідають оптимальному рішенню, з використанням рекурсії; наведемо приклад процедури для обчислення значень, що відповідають оптимальному рішенню, на основі методу висхідного аналізу; складемо оптимальне рішення на основі вже отриманої інформації.

Отже, спершу опишемо *простір підзадач* та покажемо *наявність оптимальної підструктури.* Якщо перераховувати всі підпослідовності послідовності X, то їх маємо 2^m, тому, зрозуміло, що напряму вирішувати задачу не ефективно. Але розглянемо всю множину підзадач та вивчимо її властивості. Для цього використаємо наступні поняття.

***I*-м префіксом послідовності** $X = \langle x_1, x_2, ..., x_m \rangle$ для $i = 1, 2, \ldots, m$ є підпослідовність $X_i = \langle x_1, x_2, ..., x_i \rangle$.

Використовуючи це поняття можна встановити наявність оптимальної підструктури в задачі. Для цього доведемо наступне твердження.

Твердження. Якщо маємо дві послідовності $X = \langle x_1, x_2, ..., x_m \rangle$, $Y = \langle y_1, y_2, ..., y_n \rangle$, а $Z = \langle z_1, z_2, ..., z_k \rangle$ – їхня найдовша спільна підпослідовність, то справедливо наступне.

1. Якщо $x_m = y_n$, то $z_k = x_m = y_n$ та Z_{k-1} – LCS послідовностей X_{m-1} та Y_{n-1}.
2. Якщо $x_m \neq y_n$, тоді із $z_k \neq x_m$ випливає, що Z є LCS послідовностей X_{m-1} та Y.
3. Якщо $x_m \neq y_n$, тоді із $z_k \neq y_n$ випливає, що Z є LCS послідовностей X та Y_{n-1}.

Доведення. Розглянемо перший пункт твердження. Якби виконувалося, що $z_k \neq x_m$, то до послідовності Z можна було б додати елемент $x_m = y_n$, що дало б спільну підпослідовність послідовностей X та Y вже довжиною $k + 1$. А це протиречить припущенню, що Z є LCS послідовностей X та Y. Тому повинно виконуватися $z_k = x_m = y_n$. Отже, префікс Z_{k-1} – спільна підпослідовність послідовностей X_{m-1} та Y_{n-1}. Покажемо «від супротивного», що Z_{k-1} є LCS послідовностей X_{m-1} та Y_{n-1}. Нехай маємо спільну підпослідовність H послідовностей X_{m-1} та Y_{n-1}, довжина якої більша за $k - 1$. Якщо додано до H елемент $x_m = y_n$, то получимо спільну підпослідовність послідовностей X та Y, довжина якої перевищує k, що буде протиріччям. Отже, Z_{k-1} є LCS послідовностей X_{m-1} та Y_{n-1}.

Розглянемо другий пункт твердження. Якщо $z_k \neq x_m$, то Z є спільна підпослідовність послідовностей X_{m-1} та Y. Якби існувала спільна підпослідовність H послідовностей X_{m-1} та Y, довжина якої більша за k, то вона була б і спільною підпослідовністю послідовностей X_m та Y. А це протиречить припущенню, що Z є LCS послідовностей X та Y.

Третій пункт твердження доводиться аналогічним чином.

З наведеного випливає, що найдовша спільна підпослідовність двох послідовностей містить в собі найдовшу спільну підпослідовність їхніх префіксів. Тому задача *має оптимальну підструктуру*, що дозволяє побудувати оптимальне рішення з оптимальних рішень підзадач.

Наявність *перекриття підзадач* в даній задачі проявляется в тому, що у процесі пошуку найдовшої спільної підпослідовності (LCS) X та Y можемо шукати LCS послідовностей X та Y_{n-1} та LCS послідовностей X_{m-1} та Y. Але в кожній з цих підзадач буде виникати підзадача LCS послідовностей X_{m-1} та Y_{n-1}.

Для *побудови рекурентної залежності* використаємо, що при знаходженні найдовшої спільної підпослідовності послідовностей X та Y виникає одна підзадача, якщо $x_m = y_n$ (тоді необхідно знайти найдовшу спільну підпослідовність (LCS) X_{m-1} та Y_{n-1}). Далі необхідно додати елемент $x_m=y_n$, щоб одержати найбільшу спільну послідовність послідовностей X та Y. У випадку, якщо $x_m \neq y_n$, виникають 2 підзадачі (необхідно знайти LCS послідовностей X та Y_{n-1} та LCS послідовностей X_{m-1} та Y). Яка з цих підпослідовностей виявиться довшою, та й буде LCS для X та Y.

Позначимо $c[i, j]$ довжину найдовшої спільної підпослідовності послідовностей X_i та Y_i. Якщо $i = 0$ або $j = 0$, довжина одної з цих послідовностей є 0, тому LCS також дорівнює 0. Рекурентну залежність опишемо наступним чином:

$$c[i, j] = \begin{cases} 0, & i = 0 \text{ або } j = 0, \\ c[i-1, j-1]+1, & i > 0, j > 0, x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & i > 0, j > 0, x_i \neq y_j. \end{cases}$$

Висхідний алгоритм обчислення довжини LCS. При використанні простого рекурсивного алгоритму одержимо експоненціальний час роботи. Але в даній задачі маємо всього $\Theta(mn)$ різних підзадач. Тому можна побудувати алгоритм динамічного програмування. Побудуємо висхідний алгоритм, прикладом якого є процедура LCS-Length [2]. В ній величини $c[i, j]$ зберігаються у таблиці $c[0..m, 0..n]$. Елементи цієї таблиці обчислюються по рядках зліва направо з першого рядка. Додаткова таблиця $b[1..m, 1..n]$ використовується при побудові оптимального рішення, в елементі $b[i, j]$ вказується елемент таблиці, що відповідає оптимальному рішенню підзадачі, вибраної при обчисленні елементу $c[i, j]$. Елемент $c[m, n]$ зберігає довжину LCS для X та Y.

```
   LCS-Length(X, Y)
1     m = X.length
2     n = Y.length
3     b[1..m, 1..n], c[0..m, 0..n] – нові таблиці
4     for i = 1 to m
5          c[i, 0] = 0
6     for j = 0 to n
7          c[0, j] = 0
```

```
8     for i = 1 to m
9        for j = 1 to n
10          if x_i == y_j
11             c[i, j] = c[i − 1, j − 1] + 1
12             b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14                c[i, j] = c[i − 1, j]
15                b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17                b[i, j] = "←"
14    return c, b
```

Час роботи процедури, за умови, що кожен елемент таблиці обчислюється за $\Theta(1)$, буде $\Theta(mn)$ (якщо порахувати по рядках процедури та за довжинами циклів).

В процедурі LCS-Length в елементі $c[m,n]$ зберігається довжина LCS для послідовностей X та Y, але в ній немає *побудови LCS*. Додаткова таблиця $b[1..m,1..n]$, що повертається процедурою, дозволяє це зробити. Для цього починаємо з елементу $b[m,n]$ та проходимо таблицю за стрілками до першого зліва (обернений порядок). Якщо значенням $b[i,j]$ є стрілка ↖, то елемент $x_m = y_n$ належить LCS. Приклад наступної рекурсивної процедури виводить елементи LCS X та Y [2]. Оцінка часу роботи процедури: O($m + n$).

```
   Print-LCS-(b, X, i, j)
1     if i == 0 or j == 0
2        return
3     if b[i, j] = "↖"
4          Print-LCS-(b, X, i − 1, j − 1)
5          print x_i
6      elseif b[i, j] = "↑"
7          Print-LCS-(b, X, i − 1, j)
8     else Print-LCS-(b, X, i, j − 1)
9     return c, b
```

Реалізацію наведеного алгоритму пошуку LCS можна покращити за рахунок відмови від таблиці $b[1..m,1..n]$ (для <u>даного</u> алгоритму). Так як в нашій задачі кожен елемент $c[i, j]$ залежить тільки від трьох інших

($c[i-1, j-1]$, $c[i-1, j]$, $c[i, j-1]$), для нього за час $O(1)$ можна визначити, яке з трьох значень було використане для обчислення $c[i, j]$ без таблиці $b[1..m,1..n]$. В цьому випадку LCS можна поновити за час $O(m + n)$ та зекономити час $\Theta(mn)$ (але асимптотично все одно залишиться загальна оцінка $\Theta(mn)$).

Розглянемо задачу *зваженого інтервального планування* та побудуємо алгоритм динамічного програмування для її розв'язання. Маємо n заявок з мітками 1, ..., n; для кожної заявки i вказується початковий час s_i та кінцевий час f_i. З кожним інтервалом i зв'язується його вага v_i. Два інтервали називають сумісними, якщо вони не перекриваються. Потрібно знайти підмножину $S \subseteq \{1,..., n\}$ взаємно сумісних інтервалів, що надають максимум суми v_i вибраних інтервалів [3].

Нехай усі заявки відсортовані в порядку неспадання кінцевого часу: $f_1 \leq f_2 \leq ... \leq f_n$. Заявка i є передуючою до заявки j, якщо $i < j$. Це визначає природній порядок, у якому будуть розглядатися інтервали. Визначимо $p(j)$ для інтервалу j як найбільший індекс i, $i < j$, за якого інтервали i та j не перекриваються. Тобто, i – крайній лівий інтервал, що завершується до початку j. Маємо $p(j) = 0$, якщо не існує заявки $i < j$, що не перекривається з j.

Покажемо наявність *оптимальної підструктури* в задачі, що дозволить побудувати оптимальне рішення з оптимальних рішень підзадач, та наявність *перекриття підзадач*.

Нехай маємо деякий екземпляр задачі, розглянемо оптимальне рішення, яке позначимо O (але насправді не знаємо яке воно). Інтервал n (останній) може належати O, або ні. Якщо n належить O, то очевидно, що жоден інтервал, що індексується строго між $p(n)$ та n, не може належати O, тому що за визначенням $p(n)$ знаємо, що інтервали $p(n)+1$, $p(n) + 2$, ..., $n - 1$ – всі вони перекривають інтервал n. Більш того, якщо n належить O, то рішення O має включати оптимальне рішення задачі, що складається із заявок $\{1, ..., p(n)\}$, тому, що інакше вибір заявок O з $\{1, ..., p(n)\}$ можна замінити кращим вибором без ризику перекриття заявки n. Якщо n не належить O, то O просто співпадає з оптимальним рішенням задачі, яка складається із заявок $\{1, ..., n - 1\}$.

Припустимо, що O не включає заявки n; якщо воно не вибирає оптимальну множину заявок з $\{1, ..., n-1\}$, то його можна замінити кращим вибором.
Приходимо до висновку, що в процесі пошуку оптимального рішення для інтервалів $\{1, 2, ..., n\}$ буде відбуватися пошук оптимальних рішень менших задач у формі $\{1, 2, ..., j\}$.
Спробуємо побудувати рекурсивну залежність, що включає зв'язані підзадачі. Нехай для кожного значення j от 1 до n оптимальне рішення задачі, що складається із заявок $\{1, ..., j\}$, позначимо O_j, а значення сумарної ваги цього рішення − OPT(j) (OPT(0) = 0). Шукане оптимальне рішення являє собою O_n із значенням OPT(n). Для оптимального рішення O_j по інтервалам $\{1, 2, ..., j\}$ з наведених вище міркувань (узагальнених для $j = n$) випливає: або $j \in O_j$, і тоді $\text{OPT}(j) = v_j + \text{OPT}(p(j))$, або j не належить O_j та $\text{OPT}(j) = \text{OPT}(j-1)$. Тоді одержуємо рекурентний вираз:

$$\text{OPT}(j) = \max(v_j + \text{OPT}(p(j)), \text{OPT}(j-1)).$$

Заявка j належить оптимальному рішенню для множини $\{1, 2, ..., j\}$ тоді і тільки тоді, коли $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$.
Це і дає рекурентне рівняння, яке виражає оптимальне рішення (або його значення) у контексті оптимальних рішень менших підзадач. Запишемо скорочено просту рекурсивну процедуру:

```
Compute-Opt(j)
if j = 0
    return 0
else
    return max(v_j + Compute-Opt(p(j)), Compute-Opt(j − 1))
end if
```

Процедура Compute-Opt(j) вірно обчислює OPT(j) для всіх $j = 1, 2, ..., n$ (легко доводиться індукцією по j). Але, як можна побачити за деревом викликів, асимптотично її час роботи є експоненціальним. Додавання запам'ятовування приведе до алгоритму з поліноміальним часом роботи. Так як для даної задачі маємо всього $n + 1$ різних підзадач, що перекриваються, то алгоритм динамічного програмування (що використовує додаткову таблицю M[0..n], в якій кожен елемент M[j] містить оптимальне рішення підзадачі) може представляти така процедура:

```
M-Compute-Opt(j)
if j = 0
    return 0
else if M[j] не пусте
      return M[j]
      else
         Визначити M[j]=max(v_j+M-Compute-Opt(p(j)),M-Compute-Opt(j −1))
         return M[j]
end if
```

Наведена скорочено процедура має асимптотичну оцінку часу роботи $O(n)$. Але вона лише обчислює значення оптимального рішення. Щоб одержати набір інтервалів уведемо додатковий масив S, в елементі $S[i]$ якого зберігається оптимальна множина інтервалів з $\{1, 2, ..., i\}$. Просте розширення процедури M-Compute-Opt(j) для збереження рішень в масиві S збільшить час виконання з додатковим множником в $O(n)$: хоч позиція в масиві M може оновлюватися за час $O(1)$, запис множини у додатковий масив S займає час $O(n)$. Щоб уникнути зростання $O(n)$ можна замість явного збереження S поновити оптимальне рішення по значеннях, що збережені у масиві M після обчислення оптимального значення. Користуючись тим, що j належить оптимальному рішенню для множини інтервалів $\{1, ..., j\}$ тоді і тільки тоді, коли

$$v_j + \mathrm{OPT}(p(j)) \geq \mathrm{OPT}(j - 1),$$

одержимо наведену скорочено процедуру, що в якій проходимо у зворотному напрямі по масиву M, щоб знайти множину інтервалів оптимального рішення.

```
Find-Solution(j)
if j = 0
  нічого не виводити
else
   if v_j + M[p(j)] ≥ M[j − 1]
        вивести j разом з результатом Find-Solution(p(j))
   else
        вивести результат Find-Solution(j – 1)
   end if
end if
```

Процедура має асимптотичну оцінку часу виконання $O(n)$ (рекурсивно викликає себе тільки для строго менших значень, всього $O(n)$ рекурсивних викликів, а на один виклик витрачається константний час).

Розглянемо *сегментовану задачу найменших квадратів* та побудуємо алгоритм динамічного програмування для її розв'язання. В цій задачі основним питанням є пошук «лінії найкращої відповідності» для дослідних даних. Нехай дані являють собою множину P з n точок на площині: $(x_1, y_1), ..., (x_n, y_n)$, де $x_1 < ... < x_n$. Для лінії L, що визначається, наприклад, рівнянням $y = ax + b$, мінімальна похибка L відносно P – це:

$$e(L,P) = \sum_{i=1}^{n}(y_i - \bar{a}x_i - \bar{b})^2,\ \bar{a} = \frac{n\sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2},\ \bar{b} = \frac{\sum_{i=1}^{n} y_i - \bar{a}\sum_{i=1}^{n} x_i}{n}.$$

Але дослідні точки можуть належати декільком лініям. І тоді похибка може буде набагато меншою. Тобто, якщо шукати набір ліній (обмежений), що забезпечують мінімальну похибку, то результат буде набагато кращий. Тому, враховуючи виявлення змін у заданій послідовності точок, тобто переходу до іншої лінійної апроксимації, наведемо наступну постановку задачі.

Маємо множину точок $P = \{(x_1, y_1), ..., (x_n, y_n)\}$, в якій $x_1 < ... < x_n$. Позначимо $p_i = (x_i, y_i)$. Необхідно розбити множину P на деяку кількість сегментів. Кожен сегмент є підмножиною P – неперервним набором координат $\{p_i, p_{i+1}, ..., p_j\}$ для деяких $i \leq j$. Для кожного сегменту S у розбитті P обчислити лінію, що мінімізує похибку у відношенні точок S для наведених вище формул. Штраф розбиття визначається як сума наступних доданків: кількості сегментів, на які розбивається множина P, помножена на фіксований множник $C > 0$; для кожного сегменту – значення похибки для оптимальної лінії через цей сегмент. Метою задачі є пошук розбиття з мінімальним штрафом.

Кількість можливих варіантів розбиття зростає дуже швидко й одразу не зрозуміло, чи можливий взагалі пошук оптимального розбиття. Але, насправді, маємо поліноміальну кількість підзадач, рішення яких дадуть рішення поставленої задачі та будуються з використанням рекурентного відношення. Тут важливим спостереженням є: остання точка p_n належить одному сегменту оптимального розбиття, цей сегмент починається в деякій більш ранній точці p_i. Це й підкаже вірну множину

підзадач: знаючи останній сегмент p_i, ..., p_n можна виключити ці точки з розгляду й рекурсивно розв'язувати задачу для точок p_1, ..., p_{i-1} (що вказує також на *оптимальну підструктуру* задачі).
Припустимо, що OPT(i) – оптимальне рішення для точок p_1, ..., p_i, а $e_{i,j}$ – мінімальна похибка для будь-якої лінії по відношенню p_i, ..., p_j (OPT(0) = 0) за наведеним вище формулами. Тоді вказане вище означає: якщо останній сегмент оптимального рішення складається з точок p_i, ..., p_n, то значення оптимального рішення дорівнює

$$\mathrm{OPT}(n) = e_{i,n} + C + \mathrm{OPT}(i - 1).$$

Для підзадачі, що складається з точок p_1, ..., p_j, для одержання OPT(j) необхідно знайти кращий спосіб одержання завершуючого сегменту p_i, ..., p_j (із сплатою похибки та доданку C для цього сегменту) у сполученні з оптимальним рішенням OPT(i − 1) для інших точок. Отже, обґрунтували *рекурентне* співвідношення. Маємо, для підзадачі з точками p_1, ..., p_j

$$\mathrm{OPT}(j) = \min_{1 \le i \le j} (e_{i,j} + C + \mathrm{OPT}(i - 1)),$$

а сегмент p_i, ..., p_j використовується в оптимальному рішенні підзадачі тоді і тільки тоді, коли мінімум досягається за використання індексу i.
Наведемо скорочено *висхідний рекурсивний алгоритм* (у порядку зростання i), де масив M [0... n] зберігає оптимальні рішення підзадач.

Segmented-Least-Squares(n)
M[0... n] – новий масив
M[0] = 0
Цикл у порядку зростання i
  if $i \le j$
  обчислити похибки $e_{i,j}$ для сегмента *pi, ..., pj*
кінець циклу
for $j = 1, 2, \dots, n$
    $M[j] = \min_{1 \le i \le j} (e_{i,j} + C + M[i - 1])$
кінець **for**
return M[n]

Оптимальне розбиття відслідковується у зворотному порядку за M.
Find-Segments(j)
if $j = 0$
  нічого не виводити

else
    знайти значення i, що мінімізує $e_{i,j} + C + M[i - 1]$
    вивести сегмент $\{p_i, ..., p_j\}$ та результат Find-Segments($i - 1$)
кінець **else**

Час виконання процедури Segmented-Least-Squares складається переважно з часу обчислення значень всіх похибок $e_{i,j}$ методом найменших квадратів (для визначення часу їх обчислення потрібно звернути увагу, що існує $O(n^2)$ пар (i, j), для яких ці обчислення необхідні; для кожної пари (i, j) можна використати вираз для обчислення $e_{i,j}$ за час $O(n)$). Тому загальний час – $O(n^3)$.
Алгоритм має n ітерацій для значень $j = 1, ..., n$. Для кожного значення j необхідно визначити мінімум у рекурентному співвідношенні для заповнення елементу масиву $M[j]$. Це займає $O(\text{n})$ часу для кожного j, що в сумі дає $O(n^2)$. Після визначення всіх значень $e_{i,j}$ час виконання буде $O(n^2)$.

Розглянемо *задачу про побудову вторинної структури РНК* та побудуємо алгоритм динамічного програмування для її розв'язання.
Одноланцюгова молекула РНК утворює вторинну структуру (як показано на рис. 16.1) за визначеними правилами. Молекула РНК розглядається як послідовність з n символів (основ) абетки {A, C, G, U}. Нехай B = $b_1b_2...b_n$ – одноланцюгова молекула РНК, в якій всі b_i з множини {A, C, G, U}. Основа A утворює пари з U, а основа C – з G. Кожна основа може утворювати пару з не більше ніж одною іншою основою, тобто множина пар основ утворює паросполучення. Вторинні структури також не утворюють вузлів.
Вказане сформулюємо більш докладно та іншими словами: вторинна структура B являє собою множину пар $S = \{(i, j)\}$, де $i, j \in \{1, 2, ..., n\}$, що задовольняють:
1) умові відсутності крутих поворотів – кінці кожної пари в S розділяють не менше 4 проміжних основ: якщо $(i, j) \in S$, то $i < j - 4$;
2) будь-яка пара в S складається з елементів {A, U} або {C, G} (у довільному порядку);
3) S є паросполученням; жодна основа не входить більш ніж в одну пару.
4) умові відсутності перетинів – якщо (i, j) та (k, l) – дві пари в S, то неможливо щоб $i < k < j < l$.

Формулювання задачі передбачення вторинної структури РНК: знайти для заданої B = $b_1b_2...bn$ вторинну структуру S з максимально можливою кількістю пар основ [3].

Дослідимо простір оптимальних розв’язків та покажемо наявність *оптимальної підструктури*. Всього маємо $O(n^2)$ підзадач для вирішення. Нехай OPT(i, j) – максимальна кількість пар основ у заданій вторинній структурі b_ib_{i+1} ... b_j (але яка саме не знаємо). Заборона різких поворотів дозволяє ініціювати OPT(i, j) = 0 для всех $i \geq j - 4$ (дозволимо посилання OPT(i, j) навіть за $i > j$ й тоді значення буде дорівнювати 0).

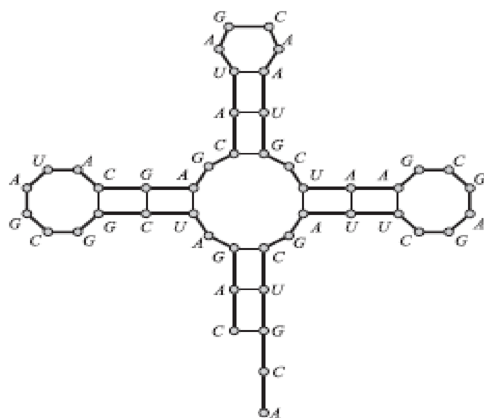


Рис. 16.1. Приклад вторинної структури РНК [3].

В оптимальній вторинній структурі b_ib_{i+1} ... b_j маємо ті ж альтернативи: або j не бере участі в парі (тоді OPT(i, j) = OPT(i, j − 1)); або j знаходиться в парі з t для деякого $t < j - 4$ (тоді породжуються підзадачі OPT(i, t − 1) та OPT(t + 1, j − 1), що ізолюються умовою відсутності перетинів). Звідси маємо *рекурсивне співвідношення*:

$$\text{OPT}(i, j) = \max(\text{OPT}(i, j-1), \max_t (1 + \text{OPT}(i, t-1) + \text{OPT}(t+1, j-1))),$$

де максимум визначається по t, для яких b_t та b_j утворюють припустиму пару основ.

Вказане дає можливість побудувати *висхідний рекурсивний алгоритм*, скорочено наведений нижче, де таблиця OPT з OPT[i, j], $i < j - 4$ (інакше OPT(i, j) = 0), зберігає оптимальні рішення підзадач. В ньому рішення підзадач завжди викликаються для більш коротких інтервалів: тих, для яких $k = j - i$ менші.

```
Proc-РНК(n)
Ініціалізувати OPT(i, j) = 0 для всех i ≥ j − 4
for k = 5, 6, ..., n − 1
    for i = 1,2, ..., n − k
        j = i + k
        OPT(i, j) = max(OPT(i, j – 1), max_t(1 + OPT(i, t – 1) + OPT(t + 1, j – 1)))
return OPT(1, n)
```

Час виконання процедури буде становити $O(n^3)$ за рахунок: підзадач для вирішення всього $O(n^2)$, а обчислення рекурентного виразу для кожної підзадачі відбувається за $O(n)$ часу.

Вторинну структуру, що відповідає одержаному рішенню, можна відновити збереженням інформації про досягнення мінімуму в рекурентному виразі та зворотного відслідковування за обчисленнями.

На рис. 16.2, що наведений приклад ітерацій алгоритму (заповнення таблиці) для РНК вигляду ACCGGUAGU для елементів, що відповідають парам $[i, j]$ з $i < j - 4$, оскільки тільки вони можуть бути відмінними від нуля.

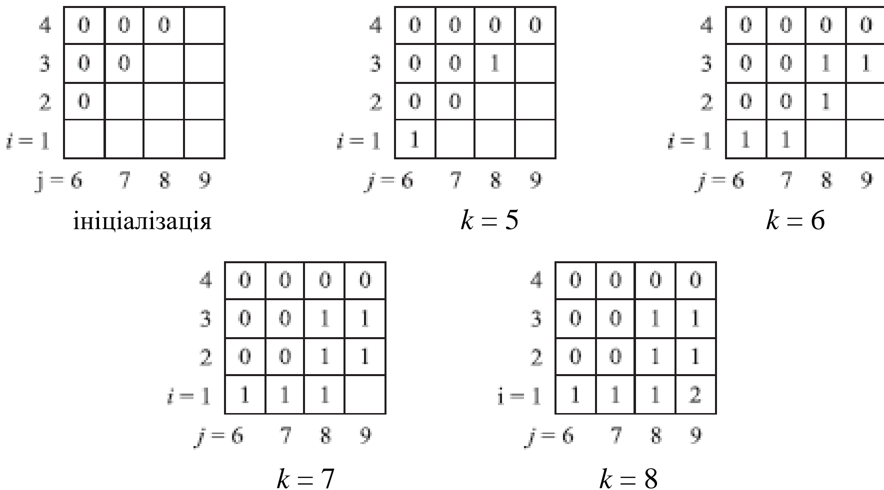


Рис. 16.2. Вторинна структура ACCGGUAGU, заповнення таблиці алгоритмом для k = 5, 6, 7, 8 [3].

Лекція 17. Побудова та аналіз жадібних алгоритмів

Жадібні (градієнтні) алгоритми, як і алгоритми динамічного програмування, використовуються для розв'язання оптимізаційних задач. В них, як відомо, роблять вибір, що здається найкращим на даний момент, причому сподіваються, що цей вибір має привести до оптимального рішення задачі. Але не завжди жадібний підхід має привести до одержання оптимального рішення.

Розглянемо на прикладах основні моменти побудови та оцінювання жадібних алгоритмів.

Задача про вибір процесів [2]. Розглянемо побудову розкладу для множини конкуруючих процесів, які повністю використовують один спільний ресурс (тому ресурс може використовуватися одночасно лише одним процесом). Результатом побудови розкладу є набір взаємно сумісних процесів, що утворюють множину максимального розміру. Позначимо множину з n процесів як $S = \{a_1,...,a_n\}$. В ній кожен процес a_i характеризується початковим моментом s_i та кінцевим f_i, де $0 \le s_i < f_i < \infty$ (процес виконується впродовж інтервалу $[s_i, f_i)$), а процеси a_i та a_j сумісні, якщо інтервали $[s_i, f_i)$ та $[s_j, f_j)$ не перекриваються.

Розглянемо множину S процесів, що відсортовані у порядку зростання моментів закінчення $f_1 \le ... \le f_i \le ... f_n$.

Побудуємо для порівняння алгоритм динамічного програмування розв'язання цієї задачі. За принципами динамічного програмування оптимальне рішення початкової задачі одержували шляхом комбінування оптимальних рішень підзадач. Спершу розглянемо питання *оптимальної підструктури* в задачі. Покажемо, що вона у цій задачі існує. Позначимо S_{ij} множину процесів, що запускаються після завершення процесу a_i та завершуються до запуску процесу a_j. Нехай необхідно знайти максимальну множину взаємно сумісних процесів в S_{ij} та такою максимальною множиною є A_{ij}, що включає деякий процес a_k. Включаючи a_k до оптимального рішення виникають 2 підзадачі: для множини S_{ik} та для множини S_{kj}. Нехай $A_{ik} = A_{ij} \cap S_{ik}$ та $A_{kj} = A_{ij} \cap S_{kj}$. Тоді множина A_{ik} містить процеси в A_{ij}, що завершуються до початку a_k, а множина A_{kj} містить процеси в A_{ij}, що починаються після того, як закінчиться a_k.

Тому A_{ij}= $A_{ik} \cup \{a_k\} \cup A_{kj}$ та множина максимального розміру A_{ij} взаємно сумісних процесів в S_{ij} складається з $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ процесів.

Як звичайно, наступним необхідно показати, що оптимальне рішення A_{ij} має включати оптимальні рішення для підзадач A_{ik} та A_{kj}. Зробимо це методом «від супротивного». Якби могли знайти множину A'_{kj} взаємно сумісних процесів у S_{kj} таку, що $|A'_{kj}| > |A_{kj}|$, то б використали її замість A_{kj} в рішенні підзадачі для S_{ij}. Тоді б побудували іншу множину $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ взаємно сумісних задач, що є протиріччям припущенню про оптимальність рішення A_{ij}. Аналогічні міркування мають місце для процесів у S_{kj}.

Позначимо $c[i, j]$ розмір оптимального рішення для множини S_{ij}. Тоді з міркувань, наведених вище:

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

Крім того, якщо не знаємо, що оптимальне рішення для S_{ij} включає процес a_k, то необхідно перевірити всі процеси в S_{ij}, щоб знайти, який з них має бути вибраний, тому

$$c[i, j] = \begin{cases} 0, & S_{ij} = \emptyset, \\ \max\limits_{a_k \in S_{ij}} \{[i, k] + c[k, j] + 1\}, & S_{ij} \neq \emptyset. \end{cases}$$

Одержаний *рекурентний вираз* дозволяє перейти до побудови, наприклад, *висхідного рекурсивного алгоритму* із запам'ятовуванням.

Але здійснення бажання мати можливість додавати процес в оптимальне рішення без попереднього розв'язання підзадач дозволяє не розглядати всі вибори, що виникають у рекурентному співвідношенні. Як такого можемо досягти у цій задачі? Інтуїтивно відчувається, що такий процес – це процес, що залишає ресурси доступними як можна більшій кількості процесів, тобто це процес, що завершується першим (якщо таких процесів декілька, то виберемо довільно будь-який з них). Нехай це a_1. Всі сумісні з a_1 процеси мають починатися після його завершення.

Вище було встановлено, що задача має оптимальну підструктуру. Нехай $S_k = \{a_i \in S : s_i \geq f_k\}$ є множиною процесів, що починаються після завершення процесу a_k. Якщо робимо жадібний вибір процесу a_1, то

маємо вирішити єдину підзадачу S_1. Зважаючи наявність оптимальної підструктури, маємо: якщо a_1 є оптимальним рішенням, то оптимальне рішення всієї задачі складається з процесу a_1 та всіх процесів в оптимальному рішенні S_1. Доведемо цей факт для будь-якої непустої підзадачі та процесу в ній, що завершується раніше інших, який вказує на можливість жадібного вибору.

Теорема. Якщо S_k – довільна непуста підзадача, a_m – процес в S_k, що завершується раніше інших, то a_m входить в деяку підмножину взаємно сумісних процесів в підзадачі S_k максимального розміру [3].
Доведення. Нехай: A_k – підмножина взаємно сумісних процесів в S_k, що має максимальний розмір; a_j – процес в A_k, що завершується раніше інших. Якщо $a_j = a_m$, то доведення завершене, оскільки вже показали, що a_m належить до деякої підмножини взаємно сумісних процесів в S_k, що має максимальний розмір. Нехай $a_j \neq a_m$ та множина $A'_k = A_k - \{a_j\} \cup \{a_m\}$ є множиною A_k, в якій a_m замінено на a_j. Процеси в A'_k не перекриваються, що випливає з того, що процеси в A_k не перекриваються, a_j є процесом з A_k, що завершується раніше інших, та $f_m \leq f_j$. З того, що $|A'_k| = |A_k|$, приходимо до висновку, що A'_k – підмножина взаємно сумісних процесів S_k максимального розміру та воно включає a_m.

Одержали можливість та *коректність* жадібного вибору і тепер не потрібно перевіряти наявність підзадач, що перекриваються. Тепер не потрібно вирішувати підзадачі, а потім виконувати вибір. Можна просто багаторазово робити вибір процесу, що завершується першим, далі залишати тільки процеси, що є сумісними із вже обраним і робити це доки не залишиться жодного процесу. В такому випадку ми розглядаємо кожен процес у порядку зростання моментів завершення лише один раз. Одержаний алгоритм працює у низхідному напрямі, робиться вибір, а потім вирішується підзадача.
Рекурсивний жадібний алгоритм. Нехай всі процеси відсортовані у порядку зростання часу їх закінчення. Масиви s та f містять значення початкових та кінцевих моментів процесів, індекс k визначає підзадачу S_k, що потрібно розв'язати.
Наведена нижче рекурсивна процедура повертає множину максимального розміру, що складається з взаємно сумісних процесів у S_k [2].

Кожен процес перевіряється в циклі у рядку 2 лише 1 раз (скільки б не було викликів).

```
   Recursive-Activity-Selector(s, f, k, n)
1      m = k + 1
2      while m ≤ n and s[m] < f[k]  // знаходимо 1-й процес в Sk, що підходить
3            m = m + 1
4      if m ≤ n
5           return {a_m} ∪ Recursive-Activity-Selector(s, f, m, n)
6      else return ∅
```

В процедурі в циклі **while** (рядки 2-3) відбувається пошук в S_k першого процесу, що завершується. В циклі процеси a_{k+1}, a_{k+2}, …, a_n перевіряються доки не буде знайдено першого процесу a_m, сумісного з процесом a_k (для такого процесу $s[m] \geq f[k]$). Якщо цикл завершується у випадку того, що такий процес не знайдений, то в рядку 5 відбувається повернення з процедури поєднання $\{a_m\}$ з підмножиною максимального розміру для підзадачі S_m, що повертається рекурсивним викликом процедури з параметрами (s, f, m, n). Якщо досягнуто $m > n$, то відбувається завершення – всі процеси перевірені в S_k, але серед них не знайдено сумісного з a_k. Тоді у рядку 6 повертається $\emptyset$ (так як $S_k = \emptyset$).

Час виконання процедури Recursive-Activity-Selector(s, f, 0, n) за умови попередньої відсортованості процесів у порядку зростання часу закінчення складає $\Theta(n)$.

Нехай конкретна, вже відсортована, множина S задана як:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

В результаті виконання процедури над такою множиною матимемо наступне. Фіктивній процес a_0 завершується в момент 0 й початковий виклик Recursive-Activity-Selector(s, f, 0, 11) вибирає процес a_1. Далі перевіряються процеси: процес, що починається після закінчення процесу a_1 є процес a_4. Виклик Recursive-Activity-Selector(s, f, 4, 11) вибирає процес a_4. Знову перевіряються процеси й першим, що починається після закінчення процесу a_4 буде процес a_8. Відбувається виклик

Recursive-Activity-Selector(s, f, 8, 11). І так далі, доки не відбудеться останній рекурсивний виклик Recursive-Activity-Selector(s, f, 11, 11), який повертає $\emptyset$. Тому максимальна множина процесів буде $\{a_1, a_4, a_8, a_{11}\}$.

Ітеративний жадібний алгоритм. Процедуру Recursive-Activity-Selector легко перетворити на ітеративну.

```
   Greedy-Activity-Selector(s, f)
1      n = s.lenght
2      A = {a_1}
3      k = 1
4      for m = 2 to n
5           if s[m] ≥ f[k]
6                A = A ∪ {a_m}
7                k = m
8      return A
```

В ітеративній процедурі змінна k індексує останнє додавання до множини A, що відповідає процесу a_k. Так як процеси розглядаємо у порядку монотонного зростання моментів закінчення, то значення f_k завжди є максимальним часом закінчення всіх процесів, що належать множині A. У рядках 2-3 вибирається процес a_1 та ініціалізується множина A, що містить тільки процес a_1. Змінній k надається індекс цього процесу. У рядках 4-7 циклу відбувається пошук процесу задачі S_k, що закінчується раніше інших, по черзі розглядається кожен процес, додається до множини A, якщо він є сумісним з усіма раніше вибраними процесами (вже містяться в A).

Час виконання цієї процедури також дорівнює $\Theta(n)$.

Сформулюємо *загальні етапи побудови жадібних алгоритмів.* Як бачили, при побудові жадібного алгоритму важливими моментами є не тільки здійснення виборів, що в кожній точці прийняття рішення виглядають найкращими. Щоб побудувати алгоритм, який приведе до глобального оптимуму задачі необхідно виконати такі етапи:

- визначення оптимальної підструктури задачі після попереднього приведення її до вигляду, коли в результаті зробленого вибору маємо для вирішення тільки 1 підзадачу;
- одержання рекурсивного рішення (співвідношення);

- доведення, що за жадібного вибору одержується лише одна підзадача, причому об'єднання оптимального рішення підзадачі із зробленим жадібним вибором приведе до оптимального рішення початкової задачі;
- обґрунтування, що за жадібного вибору завжди прийдемо до оптимального рішення, тобто існує таке оптимальне рішення задачі, яке можна одержати жадібним вибором, і такий вибір завжди припустимий (часто називають: безпека жадібного вибору);
- побудова рекурсивного алгоритму, що реалізує стратегію;
- перетворення рекурсивного алгоритму в ітеративний.

Розглянемо окремо властивість *жадібного вибору*. Така властивість притаманна не всім оптимізаційним задачам. Сутність властивості – глобальне оптимальне рішення можна одержати, роблячи локально оптимальний (жадібний, найкращий для поточної задачі) вибір. На відміну від динамічного програмування цей вибір не залежить від рішень підзадач. Жадібний вибір виглядає найкращим для поточного моменту, підзадача вирішується *після* вибору за його результатами. Сам поточний жадібний вибір може залежати від попередніх виборів, але ніколи від виборів або рішень наступних підзадач. Тому *жадібна стратегія має низхідний напрямок*, тобто кожен екземпляр поточної задачі зводиться до меншого. Важливим моментом є *обов'язковість доведення, що жадібний вибір на кожному етапі приводить до глобально оптимального рішення*. Для цього звичайно досліджується глобально оптимальне рішення деякої підзадачі, демонструється що це рішення можна перетворити так, щоб замість деякого іншого в ньому використовувався жадібний вибір, який приведе до аналогічної підзадачі меншого розміру.

Для ефективного виконання жадібного вибору можливі попередні перетворення в задачі (прикладам такого бачили сортування у порядку зростання моментів закінчення процесів). Подібні перетворення (обробка вхідних даних тощо) дозволяють прискорити процес жадібного вибору.

В жадібних алгоритмах, як і в алгоритмах динамічного програмування, необхідно показувати наявність *оптимальної підструктури* в задачі (в попередній задачі: якщо оптимальне рішення підзадачі S_{ij} містить процес a_k, то воно також містить оптимальні рішення підзадач S_{ik} та S_{kj}).

Цей момент відносять до *основної ознаки* застосовуваності динамічного програмування та жадібного підходу. За наявності встановленої конкретної оптимальної підструктури показується: якщо відомо який саме елемент входить до оптимального рішення (процес a_k), то оптимальне рішення можна побудувати шляхом вибору цього елементу та його поєднання з усіма елементами в оптимальних рішеннях утворених підзадач). А так як в жадібному алгоритмі утворюється по одній підзадачі, то обґрунтовують, що оптимальне рішення підзадачі в об'єднанні із зробленим жадібним вибором приведе до оптимального рішення початкової задачі. Це дасть можливість побудувати рекурентне співвідношення для опису оптимального рішення.

Аналіз жадібного алгоритму для задачі планування з мінімізацією затримки процесу [3]. Маємо майже попередні умови для n заявок, що використовують один ресурс впродовж деякого інтервалу часу. Вважають, що ресурс є доступним, починаючи з моменту s. Але в цій задачі замість початкового та кінцевого моментів для виконання заявки вказаний граничний час d_i та неперервний інтервал часу виконання t_i, тобто заявка може починати працювати в будь-який час до свого граничного часу. Кожній i-й заявці має бути виділений інтервал часу довжиною t_i, різним заявкам призначаються інтервали, що не перекриваються.

В задачі сподіваються задовольнити кожну заявку, але деякі заявки можуть бути відкладені на більш пізній час. Тому, починаючи із спільного початкового часу s, кожній заявці i виділяють інтервал часу t_i. Позначимо цей інтервал $[s(i), f(i)]$, $f(i) = s(i) + t_i$. Але на відміну від попередньої задачі визначимо початковий час (а тому і кінцевий час) для кожного інтервалу. Заявку i називають затриманою, якщо вона не встигає завершитися до граничного часу (тобто якщо $f(i) > d_i$). Затримка такої заявки i визначається як $l_i = f(i) - d_i$. Вважаємо, що якщо заявка не є відтермінованою, то $l_i = 0$. Ціллю нової задачі буде планування всіх заявок з інтервалами, що не перекриваються, які забезпечують мінімізацію максимальної затримки $L = \max_i l_i$.

Відсортуємо процеси у порядку зростання граничного часу d_i та плануватимемо в такому порядку. Це правило основане на принципі: завдання з більш раннім граничним часом завершаються в першу чергу. В той же час алгоритм взагалі не розглядає довжину завдань, але правило

першочергового вибору самого раннього граничного часу дає оптимальні рішення. Доведемо це.
Нехай завдання помічені в порядку слідкування їх граничного часу, тобто виконується $d_1 \leq ... \leq d_n$. Завдання плануються в цьому порядку. Нехай s є початковим часом для всіх завдань. Завдання 1 починається в час $s = s(1)$ та закінчується в час $f(1) = s(1) + t_1$; завдання 2 починається в час $s(2) = f(1)$ та завершується в час $f(2) = s(2) + t_2$ і т.д. Час завершення останнього спланованого завдання буде позначатися f.

Наведемо скорочено *жадібний алгоритм* для розв'язання задачі.

Впорядкувати завдання за граничним часом: $d_1 \leq ... \leq d_n$.
В початковому стані: $f = s$.
Розглянути завдання $i = 1, ..., n$ у такому порядку:
  Призначити завдання i часовому інтервалу від $s(i) = f$ до $f(i) = f + t_i$.
  Присвоїти $f = f + t_i$.
Кінець
Повернути множину спланованих інтервалів $[s(i), f(i)]$ для $i = 1, ..., n$.

Час виконання процедури, відповідної наведеному алгоритму, за умови попередньої відсортованості завдань за граничним часом буде складати $\Theta(n)$.
Аналіз алгоритму. Відзначимо, що алгоритм не залишає інтервалів, коли машина простоює, хоча ще залишилися завдання. Час, що проходить в таких інтервалах, називають часом простою. У розкладі A, побудованому за алгоритмом, час простою відсутній, але легко зрозуміти, що існує оптимальний розклад, що має таку властивість. Покажемо це.
Твердження. Існує оптимальний розклад без часу простою.
Доведення. Покажемо, що наш розклад A оптимальний, тобто його максимальна затримка L настільки мала, наскільки це можливо. Розглянемо оптимальний розклад O. Будемо поступово модифікувати O так, щоб зберігати оптимальність на кожному кроці, але у кінцевому результаті перетворимо його в розклад, ідентичний розкладу A, знайденому жадібним алгоритмом. Такий <u>метод аналізу</u> називають <u>*заміною*</u>.
Охарактеризуємо одержані розклади. Будемо казати, що розклад A' містить інверсію, якщо завдання i з граничним часом d_i передує завданню j з більш раннім граничним часом $d_j < d_i$. Розклад A, побудований нашим алгоритмом, не містить інверсій. Якщо існує декілька

завдань з однаковими значеннями граничного часу, то це значить, що може існувати декілька різних розкладів без інверсій. Тим не менш можна показати, що все ці розклади мають однакову максимальну затримку L.

Твердження. Всі розклади, що не містять інверсій та простоїв, мають однакову максимальну затримку.

Доведення. Якщо два різних розклади не містять ні інверсій, ні простоїв, то завдання в них можуть йти в різному порядку, але при цьому різнитися буде тільки порядок завдань з однаковим граничним часом. Розглянемо такий граничний час d. В обох розкладах завдання з граничним часом d плануються послідовно (після всіх завдань з більш раннім граничним часом та до всіх завдань з більш пізнім граничним часом). Серед усіх завдань з граничним часом d останнє має найбільшу затримку, причому вона не залежить від порядку завдань.

Головним моментом демонстрації оптимальності алгоритму є встановлення наявності оптимального розкладу, що не містить інверсій та простоїв. Для цього почнемо з будь-якого оптимального розкладу, не маючого часу простою, далі перетворимо його в розклад без інверсій без збільшення максимальної затримки. Розклад, отриманий після цього перетворення, також буде оптимальним.

Теорема. Існує оптимальний розклад, що не має інверсій та часу простою.

Доведення. За попереднім твердженням існує оптимальний розклад O без часу простою (але який саме не знаємо). Доведення буде складатися з серії *тверджень*.

(а) Якщо O містить інверсію, то існує така пара завдань i та j, для яких j йде в розкладі одразу після i та $d_j < d_i$.

Розглянемо інверсію, в якій завдання a стоїть у розкладі десь до завдання b та $d_a > d_b$. При переміщенні в порядку планування завдань від a к b десь зустрінеться точка, в якій граничний час зменшиться вперше. Вона відповідає парі послідовних завдань, що утворюють інверсію. Припустимо, що O містить як мінімум одну інверсію, та відповідно до (а) нехай i та j утворюють пару інвертованих запитів, що займають послідовні позиції в розкладі. Міняючи місцями заявки i та j в розкладі O, ми зменшимо кількість інверсій в O на 1. Пара (i, j)

створювала інверсію в O, перестановка цю інверсію усунула, й нові інверсії при цьому не виникають. Тому,

(b) Після перестановки i та j утворюється розклад, що містить на 1 інверсію менше.

Тут трудніше всього обґрунтувати факт, що розклад після усунення інверсії також є оптимальним.

(c) Максимальна затримка в новому розкладі, отриманому в результаті перестановки, не перевищує максимальної затримки O.

Якщо доведемо твердження (c), то задачу можна вважати виконаною. Початковий розклад O може мати не більше C_n^2 інверсій, а тому, після максимум C_n^2 перестановок одержимо оптимальний розклад без інверсій.

Доведемо твердження (c) та продемонструємо, що перестановка пари послідовних інвертованих завдань не приведе до зростання максимальної затримки L розкладу. Для цього уведемо наступне позначення для опису розкладу O: припустимо, що кожна заявка r планується на інтервал часу $[s(r), f(r)]$ та має затримку l'_r. Нехай $L' = \max_r l'_r$ означає максимальну затримку цього розкладу. Позначимо розклад з перестановкою $\overline{O}$; позначення $\bar{s}(r)$, $\bar{f}(r)$, $\bar{l}_r$ та $\bar{L}$ будуть представляти відповідні характеристики розкладу з перестановкою.

Повернемося до двох суміжних інвертованих завдань i та j. Ситуація така: час завершення j до перестановки у точності дорівнює часу завершення i після перестановки. Тому всі завдання, крім i та j, в двох розкладах завершуються одночасно. Крім того, завдання j в новому розкладі завершується раніше, а тому перестановка не збільшує затримку завдання j. Звідки маємо хвилюватися тільки про завдання i: його затримка могла зрости. Після перестановки завдання i завершується в момент $f(j)$, в який завершувалося завдання j в розкладі O. Якщо завдання i «запізнюється» у новому розкладі, його затримка складає $\bar{l}_i = \bar{f}(i) - d_i = f(j) - d_i$. Але тут принципово, що i не може затримуватися в розкладі більше, ніж затримувалося завдання j в розкладі O. Тому з припущення $d_i > d_j$ випливає, що

$$\bar{l}_i = f(j) - d_i < f(j) - d_j = l'_j \cdot$$

Так як затримка розкладу O була $L' \geq l'_j > \bar{l}_i$, то перестановка не збільшує максимальну затримку розкладу.

З доведеного випливає оптимальність жадібного алгоритму. Тому має місце наступне.

Твердження. Розклад *A*, побудований жадібним алгоритмом, забезпечує оптимальну максимальну затримку *L*.

Доведення. Приведене вище показує, що оптимальний розклад без інверсій існує. Крім того, було також доведено, що всі розклади без інверсій мають однакову максимальну затримку, звідси розклад, побудований жадібним алгоритмом, буде оптимальним.

Лекція 18. Приклади розв'язання задач з використанням жадібних алгоритмів

Аналіз жадібного алгоритму для задачі планування з декількома ресурсами [3].

У попередній лекціїї розглядали задачу інтервального планування з єдиним ресурсом та заявками у формі інтервалів часу. Зараз розглянемо варіант наявності декількох однакових ресурсів, для яких необхідно спланувати всі заявки з використанням мінімально можливої кількості ресурсів. Так як в цій задачі інтервали мають розподілятися за декількома ресурсами, то часто її називають задачею інтервального розбиття (або інтервального розфарбування). На практиці інтервальними заявками можуть бути: завдання, що мають бути оброблені за деякий відрізок часу деякими машинами; лекції, проведення яких плануються у заданий період в деякій множині аудиторій; запити, що повинні розподілятися у відповідності з пропускною здатністю кабелю тощо.

У загальному випадку розв'язання з *k* ресурсами можна представити як задачу розподілення заявок у *k* рядків з інтервалами, що неперекриваються: перший рядок містить всі інтервали, що призначені першому ресурсу, другий – інтервали, призначені другому ресурсу, й т. д.

Глибину множини інтервалів визначають як максимальне число інтервалів, що проходять через одну точку шкали часу.

При побудові жадібного алгоритму для вирішення даної задачі продемонструємо ще один загальний підхід до обгрунтування та доведення

його оптимальності: спочатку знаходимо просту «структурну» границю, яка показує, що для будь-якого довільного рішення використана метрика має бути не нижче визначеного значення, а далі доводимо, що алгоритм завджи забезпечує цю границю.
Доведемо справедливість наступного твердження.
Твердження. Для будь-якої задачі інтервального розбиття кількість необхідних ресурсів не менше глибини множини інтервалів [3].
Доведення. Припустимо, що множина інтервалів має глибину d, а інтервали I_1, ..., I_d проходять через одну спільну точку шкали часу. Всі ці інтервали повинні бути розподіленими за різними ресурсами, тому рішенню в цілому необхідно не менше d ресурсів.

Звідси випливає, що жодне рішення не може використовувати кількість ресурсів, меншу за глибину множини інтервалів.
Спробуємо побудувати алгоритм, який планує всі інтервали з мінімально можливою кількістю ресурсів. Але чи завжди існує розклад з кількістю ресурсів, що дорівнює глибині? Якщо так, тоді маємо шукати множину інтервалів, що займають одну точку.
Побудуємо простий жадібний алгоритм, що розподіляє всі інтервали за кількістю ресурсів, що дорівнює глибині. Такий алгоритм буде оптимальним, оскільки за доведеним вище жодне рішення не може використовувати кількість ресурсів, що менша глибини інтервалів.
Нехай d – глибина множини інтервалів, а кожному інтервалу призначена мітка з множини чисел $\{1, 2, ..., d\}$ так, щоб інтервали, що перекриваються, одержували різні мітки. Кожне число може інтерпретуватися як назва ресурса, мітка кожного інтервала – назва ресурсу, якому він буде призначений.
Використаємо жадібну стратегію за впорядкування інтервалів за початковим моментом, переберемо інтервали й надамо кожному виявленому інтервалу мітку, яка ще не була надана жодному з попередніх інтервалів, що перекриваються з поточним. Але спершу доведемо справедливість наступного твердження, яке показує коректність алгоритму.
Твердження. При використанні простого жадібного алгоритму кожному інтервалу буде призначена мітка й жодним двом інтервалам, що перекриваються, не буде надана одна й та ж мітка.

Доведення. Спершу покажемо, що жоден інтервал не залишається не поміченим. Розглянемо один з інтервалів I_j та припустимо, що в порядку сортування існує t інтервалів, які починаються раніше й перекривають його. Ці t інтервалів у сполученні з I_j утворюють множину з i інтервалів, що всі проходять через спільну точку шкали часу (а саме початковий час I_j). Тому $t + 1 \le d$. Звідси $t \le d - 1$. А з цього випливає, що хоча б одна з міток d не буде виключена з цієї множини інтервалів t, тому існує мітка, яка може бути призначена I_j.

Тепер покажемо, что жодним двом інтервалам, що перекриваються, не будуть призначені однакові мітки. Візьмемо два інтервали I та I', що перекриваються, й припустимо, що I передує I' у порядку сортування. Тепер при розгляді алгоритмом інтервалу I', інтервал I належить множині інтервалів, мітки яких виключаються з розгляду та тому алгоритм не призначає I' мітку, яка використовувалася для I.

Твердження. Отриманий за допомогою алгоритму розв'язок є оптимальним, тобто він використовує мінімально можливу кількість міток.

Доведення. Нехай алгоритм використовує d міток, d – глибина множини інтервалів. За доведеним вище при переборі інтервалів зліва направо кожному виявленому інтервалу буде призначена мітка й жодним двом інтервалам, що перекриваються, не буде надана одна й та ж мітка та не виникне ситуації, коли всі мітки вже задіяні. Так як алгоритм використовує d міток, то він використовує мінімально мінімально можливу кількість міток, тобто є оптимальним.

Наведемо скорочено *жадібний алгоритм* для розв'язання задачі за попередньо відсортованих інтервалів за початковим часом, з довільним порядком співпадання.

$I_1, I_2, ..., I_n$ – відсортовані інтервали
for $j = 1$ **to** n
    для кожного I_i, що передує I_j в порядку сортування й перекриває його
        виключити мітку I_i з розгляду для I_j
    end
    if існує мітка з множества $\{1,2, ..., d\}$, яка ще не виключена
        присвоїти невиключену мітку I_j
    else
        залишити I_j без мітки

end else
end for

Час виконання процедури, відповідної даному жадібному алгоритму, за умови попередньої відсортованості інтервалів за початковим часом буде складати $\Theta(n)$.

Аналіз алгоритму віддаленого використання для задачі оптимального хешування [3].

Розглянемо задачу, в якій обробляють послідовність заявок різної форми та при проведенні аналізу застосуємо метод заміни.

Нехай при роботі з аєрархіями пам’яті існує невеликий об’єм даних, де до даних можна звертатися дуже швидко, та великий об’єм даних, де для звернення потрібно значно більше часу. Тому необхідно вирішити, які дані потрібно тримати так, щоб можна було швидко звертатися до них. Вказане можна інтерпретувати як дані в основній пам'яті та на жорсткому диску, у вбудованому кеші та в основній пам'яті, коли при роботі в браузері диск виконує функції кешу для часто відвідуваних веб-сторінок та просто виконується завантаження веб-сторінок. Тобто під загальним терміном «кешування» в задачі розуміється процес зберігання малих обсягів даних у швидкій пам’яті, щоб зменшити витрати часу на взаємодію з повільною пам’яттю. Щоб кешування було ефективним, необхідно, аби за можливості при зверненні до даних інформація вже була в кеші. Тому алгоритм керування кешем визначає, яка інформація повинна зберігатися, а яку можна вилучити з кешу, якщо вноситимуться нові дані.

В задачі розглядається абстрактне представлення ситуації: маємо множину U з n фрагментів даних, що зберігаються в основній пам’яті; кеш може у будь-який момент часу зберігати $k < n$ фрагментів даних. Вважаємо, що кеш на початку містить множину з k елементів. Одержуємо послідовність елементів данных $D = d_1, d_2, ..., d_m$ з U – тобто послідовність звернень до пам’яті, яку потрібно обробити. При обробці необхідно постійно приймати рішення: які k елементів мають зберігатися у кеші.

Після запиту елемент d_i дуже швидко читається, якщо він знаходиться в кеші, інакше його потрібно копіювати з основної пам’яті в кеш та, якщо кеш наповнений, відбувається витиснення з нього деякого фрагменту, щоб звільнити місце для d_i (кеш-промах). Необхідно, щоб кеш-

промахів була мінімальна кількість. Тому для конкретної послідовності звернень алгоритм керування кешем визначає план витиснення, тобто які елемента та в яких точках послідовності повинні будуть втиснені з кешу, щоб мінімізувати частоту промахів.

Розглянемо простий приклад, Нехай маємо елемент*a* $\{a, b, c\}$, розмір кеша $k = 2$, послідовність a, b, c, b, c, a, b. В початковому стані кеш містить a, b. Бачимо, що для третього елементу послідовності c маємо витиснути a, а для шостого елементу a потрібно витиснути c. Отже вся послідовність включає 2 промахи.

В умовах реальних обчислень алгоритм керування кешем повинен обробляти звернення до пам'яті d_1, d_2, ... не маючи інформації щодо конкретних звернень у майбутньому. Але якщо відома послідовність S звернень до пам'яті, який план витеснення забезпечить мінімальну кількість кеш-промахів? Було показано, що наступне правило завжди приводить до мінімальної кількості промахів: коли елемент d_i має бути внесений до кешу, потрібно витиснути елемент, який буде використовуватися пізніше за всіх. Часто це жадібне правило називають «алгоритмом віддаленого використання».

Але чому саме потрібно витиснути елемент, що буде використаний пізніше за всіх, замість, наприклад, елементу, який буде використовуватися найменше з усіх? Доволі легко мати ситуації, в яких плани, побудовані за іншими правилами, також будуть оптимальними. То може відступ від правила віддаленого використання забезпечить реальний виграш десь на кінці послідовності?

В реальності несуттєво який саме з елементів буде витиснутий на деякому кроці. Отримавши план, в якому деякий елемент витискають першим, можна поміняти місцями витиснення з наступним без зміни ефективності. Ця причина (заміна одного рішення іншим) є основою методу заміни, який і використаємо для доведення оптимальності правила віддаленого використання.

Всі алгоритми керування кешем будують плани, що включають елемент d до кешу на кроці i, якщо дані d запитують на кроці i та d ще не знаходиться у кеші. Називають такий план скороченим – він виконує мінімальний обсяг роботи, необхідний для заданого кроку. В загальному випадку можна представити собі алгоритм, який будує нескорочені плани, а елементи включають до кешу на тих кроках, на яких вони

не запитуються. Покажемо, що для будь-якого нескороченого плану існує скорочений план, який не гірший.
Нехай S – план, який може не бути скороченим. Новий план – скорочення S – визначається як: на кожному кроці i, на якому S включає до кешу незапитуваний елемент d, алгоритм побудови «прикидається», що він це робить, а в дійсності залишає d в основній пам'яті.
Фактично d попадає до кэшу на наступному кроці j після того, як елемент d запитувався. В цьому випадку кеш-промах, ініційований на кроці j, може бути віднесений на рахунок більш ранньої операції з кешем, виконаною S на кроці i. Наведене обгрунтовує наступне твердження.
Твердження. Скорочений план заносить до кешу не більше елементів, ніж план S, який може не бути скороченим.
А, як бачили, для кожного скроченого плану кількість елементів, що включається до кешу, точно співпадає з кількістю промахів.
Покажемо оптимальність алгоритму віддаленого використання за допомогою методу заміни.
Розглянемо довільну послідовність звернень до пам'яті D. Нехай SFF – план, побудований алгоритмом віддаленого використання, а S^* – план з мінімально мінімально можливою кількістю промахів. Поступово перетворимо S^* в SFF, обробляючи одне рішення з витиснення за раз, без збільшення кількості промахів.
Лема. Нехай S – скорочений план, який приймає ті ж рішення з витиснення, що й SFF, в перших j елементах послідовності для деякого j. Тоді існує скорочений план S', який приймає ті ж рішення, що й SFF, у перших $j + 1$ елементах та створює не більше промахів ніж S.
Доведення. Розглянемо $(j + 1)$-ше звернення до елементу $d = d_j$+1. Так як S та SFF до цього моменту були узгодженими, вміст кешу в них не відрізняється.
Якщо d знаходиться у кеші в обох планах, то рішення з витиснення не потрібні (обидва плани є скороченими), отже, S узгоджується з SFF на кроці $j + 1$ та тому $S' = S$. Аналогічно, якщо елемент d має бути включеним до кешу, але S та SFF витискають один і той же елемент, щоб вивільнити місце для d, тому знову $S' = S$.
Якщо ж d необхідно додати до кешу, причому для цього S витискає елемент f, а SFF витискає елемент $e \neq f$. Тепер S та SFF вже не узгоджуються до кроку $j + 1$, тому що у S в кеші знаходиться e, а у SFF

в кеші – елемент f. А це значить, що для побудови S' потрібно виконати деякі нетривіальні дії. Спочатку потрібно зробити так, щоб план S' витискав e замість f. Далі маємо впевнитися, що кількість промахів у S' не більше S.
Необхідно спробувати привести кеш S' до такого ж стану, як у S, не створюючи непотрібних промахів. Коли вміст кешу буде співпадати, можна буде завершити побудову S', просто повторюючи поведінку S. Точніше, від запиту $j + 2$ та далі S' веде себе як S, доки вперше не буде виконана одна з таких умов:
1) Відбувається звернення до елементу $g \neq e, f$, який не знаходиться у кеші S, та S витискує e, щоб звільнити місце для g. Так як S' та S різняться тільки в e та f, це значить, що g також не знаходиться у кеші S'; тоді з S' витискається f та кеші S, S' співпадають. Таким чином, у частині послідовності, що залишилася, S' веде себе так же як S.
2) Відбувається звернення до елементу f та S витискає елемент e'. Якщо $e' = e$, тоді: S' просто звертається до f з кешу й після цього кеші S та S' співпадають. Якщо $e' \neq e$, то S' також витискує e' й додає e з основної пам'яти. В результаті S, S' теж мають однаковий вміст кешу. Але так як S' вже не є скороченим планом, елемент e переноситься до кешу до того, як у ньому виникне необхідність. Далі перетворюємо S' у скорочену форму, причому, за доведеним кількість елементів, що включаються до S' не зросте та S' знову узгоджується з SFF на кроці $j + 1$.
Отже, в обох випадках одержано новий скорочений план S', який узгоджується з SFF на перших $j + 1$ елементах та дає не більшу кількість промахів ніж S. Й принциповим є те, що один з цих двох випадків виникає до звернення до e (у відповідності з визначаючою властивістю алгоритму віддаленого використання): на кроці $j + 1$ алгоритм витиснув елемент e, що буде потрібний у найвіддалений майбутній момент, а тому зверненню до e має передувати звернення до f та виникне ситуація 2.

Використовуючи доведене та оптимальний план S^* побудуємо план S_1, який узгоджується з SFF на першому кроці. Продовжуючи застосовувати лему для $j = 1, 2, 3, ..., m$ будуємо плани S_j, що узгоджуються з SFF на перших j кроках. Кожен план не збільшує кількості промахів у порівнянні з попереднім, а за визначенням S_m = SFF, так як плани узгоджуються на всій послідовності.

Звідси одержуємо, що SFF породжує не більше промахів, ніж будь-який інший план S^*, а тому, він є оптимальним.
Наведемо в узагальненому вигляді *жадібний алгоритм* для розв'язання задачі керування кешем.

$d_1, d_2, ..., d_n$ – послідовність елементів
for $i = 1$ **to** n
  if елемент d_i має бути внесений до кешу
    витиснути елемент, який буде використовуватися пізніше за всіх.
end for

Час виконання процедури, відповідної алгоритму, буде складати $\Theta(n)$.

Лекція 19. Матроїди та жадібні (градієнтні) алгоритми

Матроїдні структури є дуже ефективними у побудові та обґрунтуванні жадібних алгоритмів.
Матроїдом називають впорядковану пару M=$(S,\aleph)$, що задовольняє таким умовам:

1. Множина S є скінченою множиною.
2. $\aleph$ – непуста родина підмножин множини S (які називають **незалежними** підмножинами), таких, що якщо $B \in \aleph$ та $A \subseteq B$, то й $A \in \aleph$ (умову називають спадковою, за нею випливає, що пуста множина обов'язково належить $\aleph$).
3. Якщо $A \in \aleph$, $B \in \aleph$ та $|A| < |B|$, то існує такий елемент $x \in B - A$, що $A \cup \{x\} \in \aleph$ (умову називають властивістю заміни, за виконання умови кажуть, що структура M задовольняє властивості заміни).

Графовим матроїдом M$_G = (S_G, \aleph_G)$ є неорієнтований граф $G = (V, E)$, для якого множина S_G є множиною E ребер графу та виконується умова: якщо A є підмножиною множини E, то $A \in \aleph_G$ тоді й тільки тоді, коли множина A є ациклічною (тобто множина ребер A є незалежною тоді й тільки тоді, коли підграф $G_A = (V, E)$ утворює ліс).

Приклади матроїдів [5]: матричний матроїд – матриці, в яких рядки є лінійно незалежними (елементами множини S є рядки матриці, вони задовольняють усім умовам визначення матроїду); дискретний матроїд М=$(S,\aleph)$, де $\aleph$ – множина всіх підмножин множини S (в такому матроїді всі множини є незалежними); матроїд розрізів графу $G=(V,E)$, де залежними являються ті підмножини E, які є розділяючими множинами; матроїд циклів графу $G=(V,E)$, де незалежні є ті підмножини E, які складаються з ребер деякого лісу. Всі ці матроїди задовольняють всім умовам визначення матроїду.
Нехай маємо деякий матроїд М = $(S,\aleph)$. Елемент $x\notin A$, $A\in\aleph$ називають **розширенням** множини A, якщо його можна додати до A без порушення незалежності (x – розширення множини A, якщо $A\cup\{x\}\in\aleph$).
Наприклад, М$_G$ =$(S_G,\aleph_G)$ – графовий матроїд. Якщо A – незалежна множина ребер, то ребро e є розширенням множини A тоді і тільки тоді, коли воно не належить множині A та його додавання до множини A не приведе до утворення циклу.
Нехай A – незалежна підмножина з $\aleph$ матроїду М. Якщо множина A не має розширень, то її називають **максимальною множиною**. Множина А є максимальною, якщо вона не міститься в жодній більшій незалежній підмножині $\aleph$ матроїду М.
Базами матроїда називають максимальні по включенню незалежні множини. Підмножини S, які не належать $\aleph$, називають **залежними** множинами. В силу скінченності множини S в матроїді існує хоча б одна база. **Ранг** матроїда – кількість елементів у будь-якій його базі. Максимальна за потужністю незалежна множина є й максимальною за включенням, тобто базою. Тому будь-яка незалежна множина в матроїді потужності рівній його рангу є базою. Узявши довільну незалежну множину В та деяку базу A можна у множині A вибрати A-B елементів, при додаванні яких до множини B одержимо незалежну множину потужністю $|A|$. Тому будь-яку незалежну множину можна доповнити до бази.
Наприклад, в дискретному матроїді всі множини є незалежними, тому є єдина база – це сама множина S. Ранг такого матроїда дорівнює $|S|$.

Теорема. Всі максимальні незалежні підмножини деякого матроїду мають однаковий розмір.
Доведемо методом «від супротивного». Припустимо, що A – максимальна незалежна підмножина матроїду *M*, але існує інша максимальна незалежна підмножина *B* матроїду *M*, розмір якої більше розміру підмножини A. Тоді з властивості обміну випливає, що множину А можна розширити до деякої більшої незалежної множини $A \cup \{x\}$ за рахунок деякого елементу $x \in B - A$, що протирічить припущенню про максимальність множини A.

Нехай маємо деякий графовий матроїд $M_G = (S_G, \aleph_G)$, де $G = (V, E)$ – зв'язний неорієнтований граф. Кожна максимальна незалежна підмножина має представляти вільне дерево, що містить $|V| - 1$ ребер, які поєднують всі вершини графу *G*. Таке дерево – остовне дерево графу *G*.
Матроїд *M*=$(S, \aleph)$ називають **зваженим**, якщо з ним зв'язана вагова функція *w* ($w : S \to R^+$), що надає кожному елементу *x* ($x \in S$) строго додатну вагу *w*(*x*). Вагову функцію *w* узагальнюють на підмножини S за допомогою сумування, а саме:

$$w(A) = \sum_{x \in A} w(x)$$

для будь-якої підмножини $A \subseteq S$.
Для графового матроїда *w*(*A*) – це сумарна вага всіх ребер з множини A.
Багато задач, в яких жадібний підхід дозволяє одержати оптимальне рішення, можна сформулювати в термінах пошуку незалежної підмножини з максимальною вагою у зваженому матроїді. В таких задачах задають зважений матроїд *M*=$(S, \aleph)$ та вирішують задачу пошуку такої незалежної множини $A \in \aleph$, для якої величина *w*(*A*) буде максимальною. Максимальну незалежну підмножину з максимально можливою вагою називають **оптимальною підмножиною матроїду**. Оптимальна підмножина завжди є максимальною незалежною підмножиною, що дозволяє зробити множину *A* настільки більшою, наскільки це можливо. Наприклад в задачі про мінімальне остовне дерево задають зв'язний неорієнтований граф $G = (V, E)$ та функцію довжин *w*, в якій *w*(*e*) – довжина ребра *e*. Шукають підмножину ребер, які з'єднують всі вершини та мають мінімальну спільну довжину. З цією метою у задачі

пошуку оптимальної підмножини матроїду, розглядають зважений матроїд M_G з ваговою функцією $w'(e)$, $w'(e) = w_0 - w(e)$, де w_0 перевищує максимальну довжину будь-якого ребра. Тепер будь-яка вага буде додатною, а оптимальна підмножина – це остовне дерево у початковому графі, що має мінімальну загальну довжину. Точніше: кожна максимальне незалежна підмножина *A* відповідає остовному дереву з $|V| - 1$ ребрами, а так як для будь-якої максимальної незалежної підмножини *A* має місце

$$w'(A) = \sum_{e \in A} w'(e) = \sum_{e \in A} (w_0 - w(e)) =$$
$$= (|V| - 1)w_0 - \sum_{e \in A} w(e) = (|V| - 1)w_0 - w(A),$$

то незалежна підмножина, що максимізує $w'(A)$, має мінімізувати $w(A)$. Тому будь-який алгоритм, що дозволяє знайти оптимальну підмножину *A* деякого матроїда, дозволяє також вирішити задачу про мінімальне остовне дерево. Для загального випадку жадібний такий алгоритм (алгоритм сортування та послідовного відбору) для довільного зваженого матроїда можна представити у загальному вигляді:

```
   Greedy(M, w)
1    A = ∅
2    Відсортувати M.S у незростаючому порядку ваги w
3    for кожен x ∈ M.S у незростаючому порядку ваги w(x)
4        if A ∪ {x} ∈ M.ℵ
5            A = A ∪ {x}
6    return A
```

Конкретна реалізація алгоритму залежить від того, що являє собою $\aleph$. В алгоритмі кожна ітерація циклу підтримує незалежність множини *A*, а тому і повертає в результаті незалежну підмножину деякого *A*, яка є підножиною з максимальною можливою вагою, тобто є оптимальною підмножиною. Час роботи наведеної процедури за умови, що $|S| = n$ та сортування відбувається за $O(n \lg n)$ буде оцінюватися наступним чином. Рядок 4 виконується *n* разів (одноразово для кожного елементу

множини S), при його виконанні відбувається перевірка на незалежність множини $A \cup \{x\}$, що, наприклад, вимагає $O(f(n))$ часу. Загальний час роботи оцінюється як $O(n \lg n + nf(n))$ Але ще необхідно показати, що ця процедура повертає оптимальну підмножину, тобто показати, що *матроїдам властивий жадібний вибір*. Доведемо це [2].

Лема. Нехай: $M = (S, \aleph)$ – зважений матроїд з ваговою функцією w; множина S відсортована у незростаючому порядку ваги; x – перший елемент множини S, такий, що множина $\{x\}$ є незалежною. Якщо такий елемент x існує, то існує й оптимальна підмножина A множини S, що містить елемент x.

Доведення. Якщо такого елементу $\{x\}$ не існує, то єдиною незалежною підмножиною є пуста множина. Якщо ж такий елемент x існує, то припустимо, що B – довільна непуста оптимальна підмножина, а також, що $x \notin B$ (інакше вважаємо, що $A = B$ дає оптимальну підмножину S, що містить x).

Жоден з елементів множини B не має ваги, більшої за $w(x)$. З того, що деякий $y \in B$ випливає, що множина $\{y\}$ незалежна (так як $B \in \aleph$, що є спадковою родиною). Тому завдяки вибору елементу x забезпечується виконання нерівності $w(x) \geq w(y)$ для будь-якого елементу $y \in B$.

Побудуємо множину A. Почнемо з $A = \{x\}$. У відповідності з вибором елементу x множина A є незалежною. Використовуючи властивість заміни будемо багаторазово здійсняти пошук нового елементу множини B, що можна додати до множини A, доки не досягнемо $|A| = |B|$, а множина A залишиться незалежною. У момент, коли $|A| = |B|$ маємо: множина A містить елемент x, множина B – деякий інший елемент y. Значить $A = B - \{y\} \cup \{x\}$ для деякого елементу $y \in B$, тому

$$w(A) = w(B) - w(y) + w(x) \geq w(B).$$

Так як множина B – оптимальна, то множина A, що містить x, також має бути оптимальною.

Покажемо, що, якщо який-небудь елемент не може бути доданий в даний момент, то він не зможе бути доданим і потім.

Лема. Нехай $M = (S, \aleph)$ – матроїд. Якщо x – елемент з S, що є розширенням деякої незалежної підмножини А, $A \subset S$, то x також є розширенням пустої множини $\emptyset$.
Доведення. Оскільки x – елемент з S, що є розширенням деякої незалежної підмножини A, $A \subset S$, то множина $A \cup \{x\}$ також незалежна. Тому x – розширення пустої множини.
Наслідок. Нехай $M = (S, \aleph)$ – матроїд. Якщо x – елемент множини S, що не є розширенням пустої множини $\emptyset$, то цей елемент також не є розширенням будь-якої незалежної підмножини A множини S.
Звідси й випливає, що будь-який елемент, що не може бути доданий одразу, не зможе бути доданим і потім. Тому в наведений вище процедурі не може бути помилки, що внесена пропущенням деякого початкового елементу з множини S, який не є розширенням пустої множини, так як такі елементи ніколи не можуть бути використаними.
Тепер зможемо показати, що матроїдам властива оптимальна підструктура.
Лема. Нехай x – перший елемент множини S, що вибраний жадібною процедурою Greedy для зваженого матроїду $M = (S, \aleph)$. Далі відбувається задача пошуку незалежної підмножини з максимальною вагою, що містить елемент x, яка зводиться до пошуку незалежної підмножини з максимальною вагою для зваженого матроїду $M' = (S', \aleph')$, в якому $S' = \{y \in S : \{x, y\} \in \aleph\}$, $\aleph' = \{B \subseteq S - \{x\} : B \cup \{x\} \in \aleph\}$ та вагова функція матроїда M' співпадає з ваговою функцією матроїда M, обмеженою множиною S' (в цьому випадку матроїд M' називають звуженням матроїда M на елемент x).
Доведення. Якщо A – довільна незалежна підмножина з максимальною вагою матроїду M, що містить елемент x, то $A' = A - \{x\}$ є незалежною підмножиною матроїду M'. Вірне й навпаки: з будь-якої незалежної підмножини A' матроїду M' можна одержати незалежну підмножину $A = A' \cup \{x\}$ матроїду M. А так як в обох випадках виконується $w(A) = w(A') + w(x)$, то розв'язок з максимальною вагою для матроїду M, що містить елемент x, дозволяє одержати розв'язок з максимальною вагою для матроїду M' й навпаки.

Покажемо коректність наведеного жадібного алгоритму для матроїду.

Теорема. Якщо $M=(S,\aleph)$ – зважений матроїд з ваговою функцією $w:S\to R^{+}$, то процедура Greedy повертає оптимальну підмножину.
Доведення. За доведеним вище будь-який елемент, що не може бути доданий Greedy одразу, не може бути доданим і потім. Тому про всі пропущені процедурою елементи, що не є розширеннями пустої множини , можемо забути, такі елементи ніколи не можуть бути використаними. Коли вибраний перший елемент *x,* то, за доведеним, процедура не припускає похибки, додаючи цей елемент до множини *A*, так як існує оптимальна підмножина, що містить елемент *x*. А далі з останньої леми випливає, що у задачі, що залишилася, потрібно знайти оптимальну підмножину матроїду *M'*, який називають звуженням матроїду *M* на елемент *x*. Після того як у процедурі Greedy множина *A* стане $\{x\}$, наступні дії процедури можна вважати діями над матроїдом $M'=(S',\aleph')$ (завдяки тому, що будь-яка множина $B\in\aleph'$ є незалежною підмножиною матроїду *M'* тоді й тільки тоді, коли множина $B\cup\{x\}$ є незалежною в матроїді *M*). Тому далі процедурою буде знайдена незалежна підмножина з максимальною вагою для матроїду *M'*, а в за результатом повного виконання процедури знайдемо незалежну підмножину з максимальною вагою для матроїду *M*.

Теорема Радо-Едмондса. Якщо $M=(S,\aleph)$ – матроїд, то для будь-якої вагової функції $w:S\to R^{+}$ множина *A*, знайдена алгоритмом сортування та послідовного відбору, буде множиною найбільшої ваги з *S*.
Отже, у випадку матроїдної структури підмножин жадібний алгоритм проводить до оптимального рішення. Але справедливе й зворотне твердження, що виникає з припущення про замкненість системи множин відносно включення.
Теорема. Нехай $\aleph$ – непуста система підмножин множини *S* така, що виконується спадкова умова (якщо $B\in\aleph$ та $A\subseteq B$, то й $A\in\aleph$). Якщо для будь-якої вагової функції $w:S\to R^{+}$ жадібний алгоритм знаходить підмножину *S* найбільшої ваги, то $(S,\aleph)$ – матроїд.
Використання матроїдів для розв'язання задачі планування процесів [2]. На відміну від попередніх задач про виконання завдань з одним процесором, в даній задачі завдання характеризуються одиничною тривалістю виконання. Для них задані кінцеві терміни виконання та штрафи за пропущення цих термінів.

Нехай задана кінцева множина S таких одиничних завдань $S = \{a_1, ..., a_n\}$. Розклад для множини S – це перестановка елементів цієї множини, що вказує порядок виконання завдань. Перше завдання розпочинається у нульовий момент часу, друге – в перший і т.д. Нехай: $d_1, ..., d_n$ – множина кінцевих термінів виконання відповідних завдань, $1 \le d_i \le n$, d_i – цілі числа; $w_1, ..., w_n$ – множина невід'ємних штрафів за невиконання відповідних завдань вчасно. Метою задачі є пошук розкладу для множини завдань S, що мінімізує сумарний штраф за всі протерміновані завдання.

Розглянемо довільний розклад. Завдання в ньому є протермінованим, якщо воно завершується пізніше кінцевого терміну виконання. Довільний розклад завжди можна привести до вигляду з першочерговими своєчасними завданнями (тобто виконаними вчасно), коли своєчасні завдання виконуються перед протермінованими (причому коли деяке своєчасне завдання a_i йде після деякого протермінованому a_j, то ці завдання можна поміняти місцями, в результаті завдання a_i залишиться своєчасним, а завдання a_j – протермінованим). Тому легко показати, що має місце наступне: довільний розклад можна привести до канонічного вигляду (всі своєчасні завдання передують протермінованим та розташовані в порядку неспадання кінцевих термінів виконання). Вказане дозволяє звести пошук оптимального розкладу до визначення множини A, що складається із своєчасних завдань в оптимальному розкладі. Коли знайдемо таку множину, то фактичним розкладом будуть елементи множини A у порядку неспадання кінцевих термінів виконання та протерміновані завдання з $S - A$ у довільному порядку.

Множина завдань A називається *незалежною*, якщо для неї існує розклад, в якому відсутні протерміновані завдання. Множина своєчасних завдань розкладу утворює незалежну множину завдань. Позначимо $\aleph$ – родину всіх незалежних множин завдань.

Розглянемо задачу, метою якої є визначення чи є задана множина завдань A незалежною. Позначимо $N_t(A)$ кількість завдань у множині A, для яких кінцеві терміни виконання є моменти часу, що не перевищують t, $t = \overline{0, n}$, $N_0(A) = 0$ для будь-якої множини A.

Доведемо наступне твердження [2].

Твердження. Для будь-якої множини завдань A вказані вирази є еквівалентними:

1. Множина A незалежна.
2. $N_t(A) \le t$ справедлива для всіх $t = \overline{0, n}$.
3. Якщо в розкладі завдання з множини A розташовані у порядку неспадання кінцевих термінів виконання, то жодне з них не є протермінованим.

Доведення. Покажемо, що з висловлення 1 випливає висловлення 2. Для цього покажемо справедливість: якщо $N_t(A) > t$ для деякого t, то неможливо скласти розклад так, щоб у множині A не виявилося протермінованих завдань. Вказане виконується, так як тоді до настання моменту t залишається більше t незавершених завдань. Якщо ж виконується висловлення 2, то $i – 1$ по порядку термін завершення завдання не перевищує i, тому при розстановці завдань в цьому порядку всі терміни будуть дотримані. Отже, з 2 буде випливати 3. А з виконання висловлення 3 тривіальним чином випливає виконання 1.

Саме висловлення 2 дозволяє просто визначити, чи є незалежною задана множина завдань.

Так як задача мінімізації суми штрафів за протерміновані завдання є еквівалентною до задачі максимізації суми штрафів, яких вдалося уникнути завдяки своєчасному виконанню завдань, то використаємо жадібний алгоритм (послідовного пошуку та відбору) щоб знайти незалежну множину завдань A з максимальною сумою штрафів.

Теорема. Якщо S – множина одиничних завдань з кінцевим терміном виконання, а $\aleph$ – родина всіх незалежних множин завдань, то відповідна система $(S, \aleph)$ є матроїдом [2].

Доведення. Кожна підмножина незалежної множини завдань також незалежна. Щоб довести, що виконується властивість обміну, припустимо, що B та A – незалежні множини завдань та $|B| > |A|$. Нехай k – найбільше значення t, таке, що $N_t(B) \le N_t(A)$ (таке значення t існує, оскільки $N_0(A) = N_0(B) = 0$). Так як $N_n(B) \le |B|$, $N_n(A) \le |A|$, але $|B| > |A|$, то маємо, що, $k < n$, та для всіх j в діапазоні $k + 1 \le j \le n$ повинно виконатися $N_j(B) > N_j(A)$. Тому в множині B міститься більше завдань з кінцевим

терміном виконання $k + 1$, чим у множині A. Нехай a_i – завдання з множини $B - A$ з кінцевим терміном виконання $k + 1$ та $A' = A \cup \{a_i\}$.

А тепер, використовуючи пункт 2 наведеної вище леми, можна показати, що множина A' має бути незалежною. Оскільки множина A незалежна, для будь-якого $0 \leq t \leq k$ виконується $N_t(A') = N_t(A) \leq t$. Для $k < t \leq n$ (оскільки B – незалежна) $N_t(A') \leq N_t(B) \leq t$. Тому множина A' незалежна, що й дає: $(S, \aleph)$ – матроїд.

Звідси маємо, що для пошуку незалежної множини завдань A з максимальною вагою можна використати жадібний алгоритм. Далі можна буде створити оптимальний розклад, в якому елементи множини A будуть в якості своєчасних завдань. Час роботи такого алгоритму з використанням наведеної процедури буде $O(n^2)$, оскільки кожна з $O(n)$ перевірок незалежності вимагатиме часу $O(n)$.

Наприклад, на послідовності з $n = 7$ одиничних завдань з кінцевими термінами виконання d_i та штрафами w_i, що наведені нижче, побудований жадібний алгоритм виконає такі дії.

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Спершу алгоритм послідовно вибирає завдання a_1, a_2, a_3, a_4, далі не вибирає завдання a_5, a_6 (так як $N_4(\{a_1, a_2, a_3, a_4, a_5\}) = 5$, $N_4(\{a_1, a_2, a_3, a_4, a_6\}) = 5$), вибирає завдання a_7. Получили оптимальний роклад $\langle a_1, a_2, a_3, a_4, a_7, a_5, a_6 \rangle$ із загальною сумою штрафів $w_5 + w_6 = 50$.

Лекція 20. Використання амортизаційного аналізу

Розглянемо ще один тип оцінок, які використовують в аналізі алгоритмів – амортизаційні оцінки. Такі оцінки одержуються в методі обчислювальної складності алгоритму, що називають амортизаційним аналізом, який використовують у випадках, якщо час виконання одного кроку алгоритму (верхня оцінка часу виконання одної операції), помножений на кількість кроків (операцій), дає надто завищену оцінку

для часу виконання всього алгоритму (всіх операцій послідовності) у порівнянні з реальним. Іноді тривалий час виконання деякої чергової операції тягне за собою короткий час виконання наступних операцій. Це може статися, наприклад, якщо при виконанні деякої операції йде підготовка до більш ефективного виконання наступної. В таких випадках важливим стає вивчення асимптотичної поведінки гарантованої оцінки для середнього часу виконання одної операції.
За результатами амортизаційного аналізу час, що потрібний для виконання послідовності операцій над структурою даних, усереднюють за всіма операціями. Результати також дають можливість показати, що, якщо деяка з операцій виконуваної послідовності є доволі витратною (ймовірність, пов'язана з такою операцією не розглядається), то за умов усереднення за всією послідовністю середні витрати для операцій будуть невеликими. Виконання такого аналізу показує середню продуктивність операцій у найгіршому випадку, а в його процесі витрати можуть розглядатися тільки для аналізу алгоритму і не потребувати наявності коду. Можливості такого роду аналізу та розуміння суті використовуваної структури даних може дозволити оптимізувати її використання (за рахунок демонстрації процесу витрат).

Груповий (агрегуючий) аналіз. В ході аналізу показують, що у найгіршому випадку сумарний час виконання усіх n операцій послідовності дорівнює деякій величині $T(n)$. Тому для найгіршого випадку середня (амортизована) вартість, що відповідає одній операції, визначається як $T(n)/n$. Така амортизована вартість застосовувана до всіх n операцій, навіть, якщо до послідовності входить декілька різних типів операцій.
Розглянемо виконання групового аналізу на *прикладі аналізу послідовності операцій над стеком*. Нехай наявні дві основні стекові операції, що виконуються за час $O(1)$: Push (S, x) – додає об'єкт у стек S, Pop(S) – знімає об'єкт з вершини стеку S та повертає його (виклик операції з пустим стеком генерує помилку). Приймемо, що загальні витрати для послідовності з n операцій дорівнюють n, а фактичний час виконання n таких операцій – $\Theta(n)$. Додамо ще операцію Multipop(S, k), що знімає k об'єктів або всі, якщо їх менше ніж k, з вершини стеку S:

Multipop(S, k)

```
1   while  Stack-Empty(S) == false  and  k > 0
2        Pop(S)
3        k = k – 1
```

Проаналізуємо скільки часу потрібно для виконання операції Multipop(S, k) над стеком з s об’єктів. Маємо, що час роботи лінійно залежний від кількості виконуваних операцій Pop, тому проаналізуємо процедуру Multipop за абстрактними одиничними витратами операцій Pop та Push. Кількість ітерацій циклу while дорівнює величині min(s, k), тому фактичний час роботи є лінійною функцією від цієї величини.
Розглянемо послідовність операцій Push, Pop, Multipop, що діють на початково пустий стек, та проведемо аналіз витрат. Витрати операції Multipop у найгіршому випадку складають $O(n)$, оскільки стек може містити не більше n об’єктів. Можемо сказати, що час роботи будь-якої стекової операції у найгіршому випадку складає $O(n)$, а послідовності n операцій – $O(n^2)$. Але це надто грубий аналіз.
Груповий аналіз дозволяє одержати більш точну верхню границю. За його результатами витрати довільної послідовності n операцій Push, Pop, Multipop, що діють на початково пустий стек, не перевищують $O(n)$. Розуміння суті використовуваної структури даних, а саме те, що розміщений у стеку об’єкт можна дістати з нього не більше одного разу, і приведе до такого висновку. Тому число викликів операцій Pop, Multipop не може перевищувати кількості виконаних операцій Push яких буде не більше n). Звідси середні витрати операції – $O(n)/n= O(1)$, а тому (за груповим аналізом) **амортизована вартість** кожної операції приймається рівній її середній вартості (витратам) і всі три операції мають амортизовану вартість $O(1)$.
Приклад виконання групового аналізу виконання операцій k-бітовим бінарним лічильником, що веде лічбу у висхідному напрямку. В якості лічильника використаємо бітовий масив $A[0..k-1]$, $A.lenght=k$. Молодший біт збереженого у лічильнику бінарного числа x знаходиться в елементі $A[0]$. Маємо $x=\sum_{i=0}^{k-1} A[i]2^i$. На початку $x = 0$ ($A[i]=0, i=\overline{0,k-1}$).
Для більшення показань лічильника на 1 використаємо процедуру [2]:

```
 Increment(A)
1   i = 0
2   while  i < A.lenght  and  A[i] == 1
```

```
3        A[i] = 0
4        i = i + 1
5   if  i < A.lenght
6      A[i] = 1
```

В процедурі на початку кожної ітерації циклу **while** (рядки 2-4) додається 1 до біту в позиції i. Якщо $A[i]=1$, то додавання 1 викликає обнулення біту на позиції i та додавання 1 до біту в позиції i+1 на наступній ітерації циклу. Інакше цикл закінчується. Якщо ж по його закінченні $i<k$, то буде $A[i]=0$ та змінюється значення i-го біту на 1 (рядок 6). Вартість кожної операції Increment лінійно залежить від кількості змінених бітів.

Поверхневий аналіз, як і в попередньому прикладі, для найгіршого випадку, коли масив A складається тільки з 1, дасть оцінку виконання операції Increment $\Theta(k)$, а для послідовності з n таких операцій для початково обнуленого лічильника матимемо $\Theta(nk)$.

Груповий аналіз знову дозволяє одержати більш точну верхню границю. За його результатами витрати послідовності n операцій Increment для найгіршого випадку становлять $O(n)$. Одержати такий результат дозволяє врахування структури лічильника та розуміння його роботи. Для прикладу розглянемо 8-бітовий лічильник, значення якого зростають від 0 до 16 за допомого 16 операцій Increment. На рис. 20.1 вказані результати дії цих операцій та вартість зміни бітів після кожної операції. З рисунку видно, що не за кожного виклику процедури змінюються значення всіх бітів.

Значення лічильника | Сумарна вартість

Значение счетчика	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Общая стоимость
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Рис. 20.1. Дія 16 операцій Increment на 8-бітовий лічильник [2].

Наймолодший елемент $A[0]$ змінюється щоразу, наступний $A[1]$ – через раз, $A[2]$ – один раз на 4 виклики і т.д. Тому й послідовність з n операцій Increment над початково нульовим лічильником дає зміни елементу $A[0]$ щоразу, $A[1]$ – $\lfloor n/2 \rfloor$ разів, $A[2]$ – $\lfloor n/4 \rfloor$ і т.д. Загалом біт $A[i], i = \overline{0,k\text{-}1}$, змінюється $\lfloor n/2^i \rfloor$ разів. Біти в позиціях $i \geq k$ не змінюються. Тому оцінка для загальної кількості змінених бітів

$$\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n,$$

а тому й час виконання послідовності з n операцій Increment над початково нульовим лічильником для найгіршого випадку буде мати оцінку $O(n)$. Тоді середня вартість кожної операції, а тому й амортизована вартість операції, буде $O(n)/n = O(1)$.

Метод бухгалтерського обліку (передоплати). В ньому операції різного типу оцінюють по різному, залежно від їхньої фактичної вартості. Нараховані величини називають **амортизованими вартостями** операції. Якщо амортизована вартість операції перевищуватиме її фактичну

вартість, то відповідна різниця присвоюється деяким об'єктам структури даних як **кредит**, який можна пізніше використати для компенсації витрат на виконання деякої операції, амортизована вартість якої менша за її фактичну. Тому в методі приймають, що амортизована вартість операції складається з її фактичної вартості та кредиту, який або накопичується, або витрачається.

При проведенні аналізу за допомогою методу передоплати необхідно, щоб **повна амортизована вартість послідовності операцій була верхньою границею повної фактичної вартості для цієї послідовності**. Більш того, таке співвідношення має виконуватися для всіх послідовностей операцій. Якщо позначимо c_i фактичну вартість i-ї операції, $\hat{c}_i$ – її амортизовану вартість, то вказана вимога для всіх послідовностей, що складаються з n операцій, виглядатиме:

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i .$$

Загальний кредит, що зберігатиметься у структурі даних, є різницею між повною амортизованою вартістю та повною фактичною вартістю: $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$. Повний кредит, пов'язаний із структурою даних, увесь час має бути невід'ємним. Якби це було не так, то повна амортизована вартість в той момент стала б менше відповідної фактичної вартості, тобто не була б вже верхньою границею повної фактичної вартості.

Аналіз стекових операцій методом бухгалтерського обліку. За вказаним маємо таку фактичну вартість операцій:

Push – 1,
Pop – 1,
Multipop – min(s, k).

Надамо наступні амортизовані вартості:

Push – 2,
Pop – 0,
Multipop – 0.

Зауважимо, що в даному випадку всі три амортизовані вартості мають оцінку $O(1)$, але у загальному асимптотична поведінка амортизованих вартостей може бути різною.

Доведемо, що будь-яку послідовність стекових операцій можна сплатити шляхом нарахування таких амортизованих вартостей. Маємо початково пустий стек. Нехай, при додаванні елементів 1 сплачується за саме додавання, а ще 1 залишається у запасі (в якості кредиту). Використати 1 у запасі можна для сплати вартості вивільнення елементу із стеку (тому на операцію Pop нічого не нараховується, її фактична вартість сплачується за рахунок кредиту, що зберігається у стеку). На операцію Multipop також не потрібно нічого нараховувати, так як нарахованої наперед суми завжди достатньо для сплати операції. Тому для довільної послідовності n операцій Push, Pop, Multipop, що діють на початково пустий стек, повна амортизована вартість є верхньою границею повної фактичної вартості. Звідси повна амортизована вартість є $O(n)$, а повна фактична вартість також визначається цим значенням.
Аналіз методом бухгалтерського обліку для виконання послідовності операцій у k-бітовому бінарному лічильнику. Маємо послідовність n операцій Increment, що виконуються над бінарним початко нульовим лічильником; час виконання кожної операції пропорційний кількості бітів, що змінюється (це і буде вартість операції).
Нарахуємо на операцію, за якої біту присвоюється значення 1, амортизовану вартість 2. Тепер, коли біт встановлюється нараховану 1 можна використати на сплату за встановлення, а ще 1 – залишити як кредит для наступного використання із встановлення 0 в біті. А для обнулення біту немає необхідності нараховувати жодної суми.
Визначимо амортизовану вартість операції Increment. В процедурі Increment встановлюється не більше одного біту (рядок 6), тому амортизована вартість операції не перевищує 2. Вартість обнулення бітів сплачується за рахунок кредитів, пов'язаних з цими бітами. Кількість одиниць у бінарному показнику лічильника не може бути від'ємною, тому й сума кредитів завжди невід'ємна. Отже, повна амортизована вартість n операцій Increment буде $O(n)$, що й буде оцінкою повної фактичної вартості.

Метод потенціалів. В даному методі використовують поняття *потенціальної енергії* (*потенціала*), що можна вивільнити для сплати наступних операцій. Але цей потенціал пов'язаний із структурою даних в цілому, а не з окремими об’єктами. За методом починаємо з початкової структури даних D_0, над якою виконується n операцій. Для всіх $i = \overline{1,n}$

позначимо: c_i – фактична вартість i-ї операції; D_i – структура даних, що одержується в результаті застосування i-ї операції до структури D_{i-1}.

Функцією потенціалу Φ називають функцію, що відображає кожну структуру даних D_i на дійсне число $\Phi(D_i)$, що є **потенціалом**, пов'язаним із структурою даних D_i. Амортизовану вартість $\hat{c}_i$ i-ї операції визначають як:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

Тобто – як фактичну вартість операції плюс приріст потенціалу в результаті виконання цієї операції. Тоді **повна амортизована вартість** n операцій:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

Якщо функцію потенціалі Φ можна визначити так, щоб виконувалося $\Phi(D_n) \geq \Phi(D_0)$, то **повна амортизована вартість є верхньою границею повної фактичної вартості**. На практиці не завжди відомо скільки операцій може бути виконано, тому, якщо накласти умову

$$\Phi(D_i) \geq \Phi(D_0) \text{ для всіх } i,$$

то, як і раніше, буде забезпечено передоплату. Часто приймають $\Phi(D_0) = 0$, а потім показують, що $\Phi(D_i) \geq 0$ для всіх i. Різниця потенціалів $\Phi(D_i) - \Phi(D_{i-1})$ може бути як додатною величиною (тоді амортизована вартість переоцінює i-ту операцію та потенціал структури даних зростає), так і від'ємною (амортизована вартість недооцінює i-ту операцію та потенціал структури даних спадає, фактична вартість операції виплачується за рахунок накопиченого потенціалу). Конкретні амортизовані вартості залежать від вибору функції потенціалу Φ, а амортизовані вартості, що відповідають різним функціям потенціалу, можуть бути різними, проте вони все одно залишаються верхніми границями фактичних вартостей.

Аналіз стекових операцій методом потенціалу. Визначимо функцію потенціалу Φ для стеку як кількість об'єктів у цьому стеку (для пустого

– маємо структуру D_0 та $\Phi(D_0)=0$). Оскільки кількість об'єктів у стеку не може бути від'ємною, стеку D_i, що одержується в результаті виконання i-ї операції, відповідає невід'ємний потенціал $\Phi(D_i)\geq 0$. Тоді повна амортизована вартість n операцій, пов'язана з функцією Φ, є верхньою границею фактичної вартості.

Обчислимо амортизовані вартості різних стекових операцій. Якщо i-та операція над стеком, що містить s об'єктів, є операція Push, то різниця потенціалів $\Phi(D_i)-\Phi(D_{i-1})=(s+1)-s=1$, а амортизована вартість операції $\hat{c}_i=c_i+\Phi(D_i)-\Phi(D_{i-1})=1+1$.

Якщо i-тою операцією над стеком, що містить s об'єктів, є операція Multipop (S, k) та дістаємо k'=min(k, k'=s) об'єктів, то фактична вартість такої операції буде k', а приріст потенціалу $\Phi(D_i)-\Phi(D_{i-1})=-k'$.

Тому амортизована вартість цієї операції:

$$\hat{c}_i=c_i+\Phi(D_i)-\Phi(D_{i-1})=k'-k'=0.$$

Якщо i-та операція над стеком, що містить s об'єктів, є операція Pop, то різниця потенціалів $\Phi(D_i)-\Phi(D_{i-1})=s-(s+1)=-1$, а амортизована вартість $\hat{c}_i=1-1=0$.

Отже, амортизована вартість кожної з трьох операцій буде $O(1)$, тому повна амортизована вартість послідовності з n операцій буде $O(n)$, а так як повна амортизована вартість n операцій є верхньою границею повної фактичної вартості, то для найгіршого випадку вартість n операцій буде $O(n)$.

Аналіз методом потенціалу виконання послідовності операцій у k-бітовому бінарному лічильнику. Обчислимо амортизовану вартість операції Increment. Визначимо потенціал лічильника після виконання i-ої операції Increment як кількість b_i одиниць, що містяться у лічильнику після цієї операції. Нехай i-та операція Increment обнуляє t_i біт. Тоді фактична вартість цієї операції не перевищує t_i+1 біт, так як разом з обнуленням t_i біт значення 1 надається не більше ніж 1 біту.

Якщо $b_i=0$, то в процесі виконання i-ї операції обнуляємо всі k біт. Тому $b_{i-1}=t_i=k$.

Якщо $b_i>0$, то в процесі виконання i-ї операції $b_i=b_{i-1}-t_i+1$.

У будь-якому випадку маємо $b_i\leq b_{i-1}-t_i+1$. Тоді різниця потенціалів:

$$\Phi(D_i) - \Phi(D_{i-1}) \le (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i .$$

Звідси амортизована вартість операції

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le (t_i + 1) + (1 - t_i) = 2 .$$

Якщо спочатку значення лічильника є 0, то $\Phi(D_0) = 0$. Так як для всіх i $\Phi(D_i) \ge 0$, то повна амортизована вартість послідовності з n операцій Increment є верхнею границею повної фактичної вартості. Звідси у найгіршому випадку вартість виконання послідовності з n операцій Increment буде $O(n)$.

Нехай спочатку значення лічильника не є 0, початкове показання лічильника містить b_0 одиниць, а після виконання n операцій Increment показання – b_n одиниць, причому $0 \le b_0, b_n \le k$. Тоді

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) .$$

При всіх $i = \overline{1,n}$ виконується $\hat{c}_i \le 2$. Тому, враховуючи, що $\Phi(D_0) = b_0$ та $\Phi(D_n) = b_n$, повна фактична вартість n операцій Increment:

$$\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} 2 - b_n + b_0 = 2n - b_n + b_0 .$$

При $k = O(n)$ повна фактична вартість буде $O(n)$ (інакше: якщо виконується не менше $n = \Omega(k)$ операцій Increment, то повна фактична вартість незалежно від початкового показання лічильника є $O(n)$).

Амортизаційний аналіз для динамічних таблиць. Для ситуації, коли для таблиці необхідно змінювати розмір, що викликає копіювання елементів до нової таблиці з іншим розміром, логічним чином виникає про динамічне розширення та стиснення таблиці. Використання амортизаційного аналізу не тільки надає амортизовану вартість вставки та вилучення ($O(1)$), а й показує як забезпечити дотримання умови, за якою невикористаний простір динамічної таблиці не перевищує фіксованої долі її повного простору.

Нехай в динамічній таблиці підтримуються операції Table-Insert, Table-Delete. Результатом виконання операції Table-Insert є додавання до таблиці елементу, що займає 1 комірку (простір одного елементу). Результатом виконання операції Table-Delete є вилучення з таблиці елементу, що займає 1 комірку (робить вільною 1 комірку).

Без врахування технічних подробиць застосуємо концепцію, що використовують при аналізі хеш-таблиць. Для цього визначимо **коефіцієнт заповнення** $\alpha(T)$ непустої таблиці *T* як кількість елементів, що знаходяться в таблиці, поділену на її розмір. Розмір пустої таблиці (без елементів) вважатимемо рівним 0, а її коефіцієнт заповнення – 1. Якщо коефіцієнт заповнення динамічної таблиці обмежений зверху сталою, то її невикористаний простір ніколи не перевищує фіксованої частини її повного розміру.

Розглянемо процес розширення таблиці. Нехай місце для збереження таблиці виділяється у вигляді масиву комірок. Таблиця стає заповненою, якщо заповнюються всі її комірки (коли її коефіцієнт заповнення стає рівним 1). Нехай наше програмне середовище має систему керування пам'яттю. Тобто, коли до заповненої таблиці додається елемент, її можна розширити, виділивши місце для нової таблиці, що містить більше комірок. Оскільки таблиця завжди має розміщуватися у неперервній області пам'яті, для таблиці більшого розміру необхідно виділити новий масив, а потім скопіювати елементи із старої таблиці до нової. Нехай у новій таблиці в 2 рази більше комірок, ніж у старій. Як до таблиці тільки вставляють елементи, то значення її коефіцієнту заповнення буде не менше ½, а обсяг незаповненої частини ніколи не перевищить половини повного розміру таблиці.

У наведеній нижче процедурі [2]: *T* – об'єкт, що представляє таблицю, атрибут *T.table* містить вказівник на блок пам'яті, що представляє таблицю, *T.num* містить кількість елементів у таблиці, *T.size* – повну кількість комірок у таблиці. На початку таблиця є пустою.

```
 Table-Insert(T, x)
 1    if T.size == 0
 2          виділити T.table з 1 коміркою
 3          T.size = 1
 4    if T.num == T.size
 5          виділити new-table з 2 T.size комірками
 6          вставити всі елементи з T.table до new-table
 7          звільнити T.table
 8          T.table = new-table
 9          T.size = 2 T.size
 10   вставити x до T.table
 11   T.num = T.num + 1
```

У процесі розширення мали два вида вставки (рядок 6 та рядок 10). Проаналізуємо час роботи наведеної процедури у термінах кількості елементарних вставок (вставок окремих елементів), вважаючи вартість кожної такої операції за 1. Припустимо, що фактичний час роботи процедури лінійно залежить від часу вставки окремих елементів. Накладні витрати на виділення початкової таблиці у рядку константні, накладні витрати на виділення та вивільнення пам'яті у рядках 5,7 набагато менші за вартість переносу елементів у рядку 6. Подія, за якої виконуються рядки 5-9, називається *розширенням*.

Нехай виконується n операцій Table-Insert над початково пустою таблицею. Визначимо вартість c_i i-ої операції. Якщо на поточний момент у таблиці є вільне місце, то $c_i = 1$, оскільки достатньо виконати лише одну елементарну операцію у рядку 10. Якщо таблиця заповнена та маємо виконати її розширення, то $c_i = i$ (1 елементарна вставка у рядку 10 та копіювання i-1 елементу із старої таблиці до нової у рядку 6). Якщо виконується n операцій, то вартість останньої у найгіршому випадку буде $O(n)$. Звідки можна зробити висновок, що верхня границя часу виконання n операцій – $O(n^2)$. Але, як і у попередніх прикладах, це надто неточна оцінка, оскільки потреба в розширенні таблиці виникає не дуже часто: i-та операція приводить до розширення, тільки якщо величина i-1 дорівнює степені 2.

Покажемо, що амортизована вартість дорівнює $O(1)$. Використаємо метод *групового аналізу*. Вартість c_i i-ої операції дорівнює i, якщо i-1 є точним степенем 2, або 1 в іншому випадку. Тому повна вартість виконання n операцій

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n,$$

оскільки вартість не більше n операцій дорівнює 1, а вартості інших операцій утворюють геометричну прогресію.

Так як повна вартість виконання n операцій обмежена величиною $3n$, то амортизована вартість однієї операції не буде перевищувати 3, що й дасть $O(1)$.

Одержимо амортизаційну вартість операції Table-Insert методом *бухгалтерського обліку*. Зрозуміло, що для кожного елементу приходиться тричі платити за елементарну вставку (за саму вставку до таблиці, за переміщення цього елементу при розширенні таблиці та за

переміщення ще одного елементу, що вже колись був переміщений у ході розширення таблиці). Припустимо, що одразу після розширення таблиці її розмір стає рівним m. Тоді кількість елементів у ній – $m/2$, а таблиці буде відповідати нульовий кредит. Далі за кожну вставку нараховуємо до сплати 3. Елементарна вставка використає 1 з нарахованого. Ще 1 піде як кредит за елемент, що вставили. Остання 1 буде кредитом за один з розташованих раніше в таблиці $m/2$ елементів. Для заповнення таблиці потрібно ще $m/2 – 1$ додаткових вставок. Коли ж у таблиці стане m елементів і вона буде заповненою, кожному її елементу буде відповідати 1 з кредиту для сплати за переміщення до нової таблиці при розширенні.

Виконаємо аналіз послідовності n операцій Table-Insert над початково пустою таблицею *методом потенціалів*. Визначимо функцію потенціалу Φ, що стає рівною 0 одразу після розширення та досягає значення, рівного розміру таблиці на момент, коли таблиця стає заповненою. Тоді розширення можна повністю сплатити за рахунок потенціалу. Такою функцією може бути, наприклад,

$$\Phi(T) = 2\, T.num - T.size,$$

Одразу після розширення матимемо $T.num = T.size/2$, тому, як і потрібно, $\Phi(T) = 0$. Перед розширенням матимемо $T.num = T.size$, тому $\Phi(T) = T.num$, як і потрібно. Початкове значення потенціалу $\Phi(0) = 0$. Оскільки таблиця завжди заповнена не менш ніж на половину, виконується $T.num \geq T.size/2$, тому функція потенціалу $\Phi(T)$ невід'ємна. Звідки сумарна амортизована вартість n операцій Table-Insert над початково пустою таблицею є верхньою границею сумарної фактичної вартості.

Проаналізуємо амортизовану вартість i-ої операції Table-Insert. Нехай: кількість елементів, що зберігається в таблиці після цієї операції, – num_i, розмір таблиці після операції – $size_i$, потенціал після операції – Φ_i. На початку: $num_0 = 0$, $size_0 = 0$, $\Phi_0 = 0$. Якщо ця операція не приводить до розширення, то $size_i = size_{i-1}$, а амортизована вартість операції

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) =$$
$$= 1 + (2num_i - size_i) - (2(num_i - 1) - size_i) = 3.$$

Якщо вона приводить до розширення, то $size_i = 2size_{i-1}$, $size_{i-1} = num_{i-1} = num_i - 1$, тому $size_i = 2(num_i - 1)$. Амортизована вартість операції

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) =$$
$$= num_i + (2num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) =$$
$$= num_i + 2 - (num_i - 1) = 3.$$

Розглянемо аналіз виконання операцій для випадку, коли маємо також операцію Table-Delete. При її реалізації елемент просто вилучається з таблиці. Але, якщо коефіцієнт заповнення стає малим, бажано стиснути таблицю. Стиснення таблиці виконаємо аналогічно розширенню: якщо кількість елементів досягає деякого критичного значення, виділяється місце для меншої таблиці та елементи із старої таблиці копіюються до нової, звільняється місце, використовуване для старої таблиці (псевдокод процедури буде схожим до псевдокоду процедури Table-Insert). Припустимо ще: якщо кількість елементів таблиці зменшується до нуля, то виділений для неї простір звільняється, якщо $T.num = 0$, то і $T.size = 0$).

Отже, коефіцієнт заповнення динамічної таблиці має бути обмежений знизу деякою додатною сталою, а амортизована вартість операцій – обмежена зверху деякою сталою.

Якщо при виконанні розширення та стиснення будемо подвоювати розмір таблиці при вставці елементу в заповнену таблицю та вдвічі зменшувати, коли коефіцієнт заповнення спадає нижче ½, то гарантовано матимо, що величина коефіцієнту заповнення є не менше ½. Але, якщо над таблицею виконуємо n операцій, де n – степінь двійки, і перші $n/2$ операцій – операції вставки. Їхня вартість – $\Theta(n)$. В результаті отримаємо $T.num = T.size = n/2$. Далі виконаємо $n/2$ таких операцій: вставка, вилучення, …., вставка, вилучення. Перша операція викликає розширення таблиці до таблиці розміром n. Друга операція викличе її стиснення до розміру $n/2$ (вартість $\Theta(n)$). Потім знову розширення. І т.д. Тому повна вартість n операцій буде $\Theta(n^2)$, амортизована вартість одної операції буде $\Theta(n)$. Проблема виникла з-за того, що після розширення не встигли виконати достатньої кількості вилучень, щоб накопичити кредитів (або потенціалу) для сплати стиснення, а після стиснення не встигли виконати достатньої кількості вставок, щоб сплатити розширення. Але, якщо дозволити, щоб коефіцієнт заповнення таблиці ставав нижче ½, а саме при вилученні обмежити знизу 1/4, то й при

розширенні, й при стисненні зможемо сплатити копіювання всіх елементів до нової таблиці.

Методом потенціалів проаналізуємо вартість послідовності *n* операцій Table-Insert та Table-Delete.

Визначимо функцію потенціалу *Ф*, що дорівнює 0 одразу після розширення або стиснення та зростає із збільшення коефіцієнту заповнення до 1 або спадає до ¼. Коефіцієнт заповнення таблиці (непустої) $\alpha(T)$ = $T.num/T.size$. Так як для пустої таблиці $T.num = T.size = 0$ та $\alpha(T)$ = 1, рівність $T.num = \alpha(T)\, T.size$ завжди має місце. Візьмемо в якості функцію потенціалу функцію *Ф*(*T*) вигляду

$$\Phi(T) = \begin{cases} 2T.num - T.size, & \alpha(T) \geq 1/2, \\ T.size/2 - T.num, & \alpha(T) < 1/2. \end{cases}$$

Маємо *Ф*(0) = 0 та функція *Ф*(*T*) завжди є невід'ємною. Тому повна амортизована вартість послідовності *n* операцій для функції *Ф* є верхньою границею фактичної вартості цієї послідовності.

Розглянемо як змінюються значення функції потенціалу в залежності від значення коефіцієнту заповнення таблиці. Якщо коефіцієнт заповнення є ½, то потенціал дорівнює 0. Якщо коефіцієнт заповнення 1, то маємо $T.size = T.num$, звідки $\Phi(T) = T.num$ та потенціалу буде достатньо для сплати розширення у випадку додавання елементу. Якщо коефіцієнт заповнення ¼, то $T.size = 4T.num$, звідки $\Phi(T) = T.num$, а тому потенціалу достатньо для сплати стиснення в результаті вилучення елементу.

Позначимо c_i фактичну вартість *i*-ої операції, $\hat{c}_i$ – її амортизовану вартість. Нехай знову: num_i – кількість елементів, що зберігається в таблиці після *i*-ої операції, $size_i$ – розмір таблиці після *i*-ої операції, α_i – коефіцієнт заповнення після виконання *i*-ої операції, Φ_i – потенціал після *i*-ої операції. На початку маємо $num_0 = 0$, $size_0 = 0$, $\Phi_0 = 0$.

Якщо *i*-ою виконуваною операцією є Table-Insert, то маємо наступні ситуації. За $\alpha_{i-1} \geq 1/2$ – аналіз повністю ідентичний до наведеного раніше для розширення таблиці. Для кожної з можливостей (розширення або ні) амортизована вартість $\hat{c}_i$ не перевищуватиме 3. Якщо $\alpha_{i-1} < 1/2$, таблиця не може бути розширеною в результаті виконання операції, так

як для розширення необхідно мати $\alpha_{i-1} = 1$. При $\alpha_i < 1/2$ амортизована вартість *i*-ої операції буде:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) =$$
$$= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) = 0.$$

Якщо $\alpha_{i-1} < 1/2$, але $\alpha_i \geq 1/2$, то

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (2num_i - size_i) - (size_{i-1}/2 - num_{i-1}) =$$
$$= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) =$$
$$= 3num_{i-1} - \frac{3}{2}size_{i-1} + 3 = 3\alpha_{i-1}size_{i-1} - \frac{3}{2}size_{i-1} + 3 <$$
$$< \frac{3}{2}size_{i-1} - \frac{3}{2}size_{i-1} + 3.$$

Тому амортизована вартість *i*-ої виконуваної операції Table-Insert не перевищує 3.

Якщо *i*-ою виконуваною операцією є Table-Delete, то $num_{i-1} = num_i - 1$.

Якщо $\alpha_{i-1} < 1/2$, то можливе стиснення таблиці. Якщо такого не станеться, то $size_i = size_{i-1}$, амортизована вартість операції

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) =$$
$$= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) = 2.$$

Якщо ж стиснення відбудеться, то фактична вартість операції буде $c_i = num_i + 1$ (один елемент вилучається та num_i елементів переміщується). Тоді $size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$, амортизована вартість операції

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) =$$
$$= (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) =$$
$$= (num_i + 1) + ((num_i + 1) - num_i) - ((2num_i + 2) - (num_i + 1)) = 1.$$

Якщо $\alpha_i \geq 1/2$, то стиснення не відбувається, а амортизована вартість також обмежена сталою.

В результаті маємо, що так як амортизована вартість кожної операції обмежена зверху сталою, фактична вартість виконання довільної послідовності з *n* операцій над динамічною таблицею буде $O(n)$.

Рекомендована література

[1]. Седжвик Р. Алгоритмы на С++. Фундаментальные алгоритмы и структуры данных / Р. Седжвик. – М. : ИД "Вильямс", 2011. – 1056 с.
[2]. Кормен Т. Алгоритмы. Построение и анализ. 3-е изд. / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. – М. : ИД "Вильямс", 2013. – 1328 с.
[3]. Клейнберг. Дж. Алгоритмы: разработка и применение / Дж. Клейнберг, Е. Тардос. – СПб.: Питер, 2016. – 800 с.
[4]. Кнут Д. Искусство программирования, том 3. Сортировка и поиск. 3-е изд. / Д. Кнут. – М.: Вильямс, 2006. – С. 822.
[5]. Уилсон Р.Дж. Введение в теорию графов. 5-е изд./ Р.Дж. Уилсон. – СПб.: Диалектика, 2019. – 240 с.

ЗМІСТ

Лекція 1. Асимптотичні вирази. O-нотація. Властивості O-нотації 2
Лекція 2. Поняття обчислювальної складності алгоритму. Принципи аналізу алгоритму. Зростання функцій. Ω-нотація. Θ-позначення 5
Лекція 3. Метод підстановки в аналізі алгоритмів 11
Лекція 4. Метод дерев рекурсії в аналізі алгоритмів 14
Лекція 5. Основна теорема в аналізі алгоритмів 18
Лекція 6. Основна теорема для довільних значень n. Приклади розв'язання задач 24
Лекція 7. Аналіз детермінованих алгоритмів сортування (що використовують порівняння) 29
Лекція 8. Побудова алгоритмів методом декомпозиції та їх аналіз. Алгоритм пошуку максимального підмасиву. Алгоритм Штрассена. Алгоритм порівняння ранжувань 36
Лекція 9. Алгоритм знаходження згортки векторів. Аналіз алгоритму пірамідального сортування 45
Лекція 10. Рандомізовані алгоритми та їх аналіз 53
Лекція 11. Застосування індикаторних випадкових величин, ймовірнісного аналізу в дослідженні алгоритмів 60
Лекція 12. Швидке сортування. Аналіз рандомізованого алгоритму для найгіршого, середнього та найкращого випадків 66
Лекція 13. Аналіз алгоритмів сортування порівнянням. Аналіз алгоритмів сортування підрахунком та порозрядного сортування 75
Лекція 14. Аналіз алгоритмів пошуку i-ї порядкової статистики 83
Лекція 15. Побудова та аналіз алгоритмів динамічного програмування 90
Лекція 16. Приклади побудови та аналізу алгоритмів динамічного програмування 105

Лекція 17. Побудова та аналіз жадібних алгоритмів 117
Лекція 18. Приклади розв'язання задач з використанням жадібних алгоритмів 128
Лекція 19. Матроїди та жадібні (градієнтні) алгоритми 135
Лекція 20. Використання амортизаційного аналізу 144
Рекомендована література 160

Навчальне видання

Вергунова І.М.,

Побудова та аналіз алгоритмів.

Лекції

Підписано до друку 21.12.2020.
Формат 60х84/16. Папір офсетний.
Друк різографічний.
Друк. арк. 10,25. Умов. друк. арк. – 9,53.
Наклад 100 прим. Зам. № 10.

Віддруковано з оригіналів замовника.
ФОП Корзун Д.Ю.

Видавець ТОВ «ТВОРИ».