Тема 2. Аналіз алгоритмів

2.1. Сортування включенням

Приступаючи до вивчення аналізу алгоритмів, ми розглянемо достатньо простий алгоритм для задачі сортування – сортування методом включення. Нагадаємо формулювання проблеми:

Вхід: послідовність n чисел $\langle a_1, a_2, \ldots, a_n \rangle$

Вихід: перестановка $\langle a'_1, a'_2, \ldots, a'_n \rangle$ вхідної послідовності таким чином, що для всіх її членів виконується співвідношення $a'_1 \le a'_2 \le \ldots \le a'_n$.

В реальних задачах часто потрібно сортувати не самі цілі числа, а певні значення, що вказують на, можливо, достатньо великі структури даних. Ці значення в такому випадку називаються **ключами**. Наприклад, в базі даних про працівників компанії таким ключем може бути номер паспорту або водійських прав.

Алгоритм сортування включенням ефективно працює при сортуванні невеликої кількості елементів. Сортування включенням нагадує спосіб, яким послуговуються гравці для сортування карт, що вони мають на своїх руках. Нехай спочатку у лівій руці немає жодної карти і всі вони лежать сорочкою догори. Далі зі столу береться по одній карті, кожна з яких розміщується в потрібне для неї місце серед вже обраних карт. Щоби визначити, куди потрібно вставити нову карту, її масть та вага порівнюються з мастю та вагою карт в руці. Порівняння може проводитись, наприклад, зліва направо. В будь-який момент часу карти в руці будуть розсортовані і це будуть ті карти, які спочатку лежали в колоді на столі.

Нижче наведений псевдокод методом включення під назвою `Insertion_sort`. На його вхід подається масив $A[1\ldots n]$, який містить послідовність n чисел для сортування (кількість елементів тут позначена як *length*[*A*]). Вхідні числа сортуються без використання додаткової пам'яті: їх перестановка відбувається всередині масиву, а об'єм використаної для цього додаткової пам'яті не перевищує деяку сталу величину. По закінченню роботи алгоритму `Insertion_sort` вхідний масив міститиме відсортовану послідовність.

```
Insertion_sort(A)
1  for j ← 2 to length[A]
2     do key ← A[j]
3        // Включення елементу A[j] у відсортовану послідовність A[1..j-1]
4        i ← j-1
5        while i>0 and A[i]>key
6           do A[i+1] ← A[i]
7              i ← i-1
8        A[i+1] ← key
```

Лістинг 2.1 Сортування включенням `Insertion_sort`

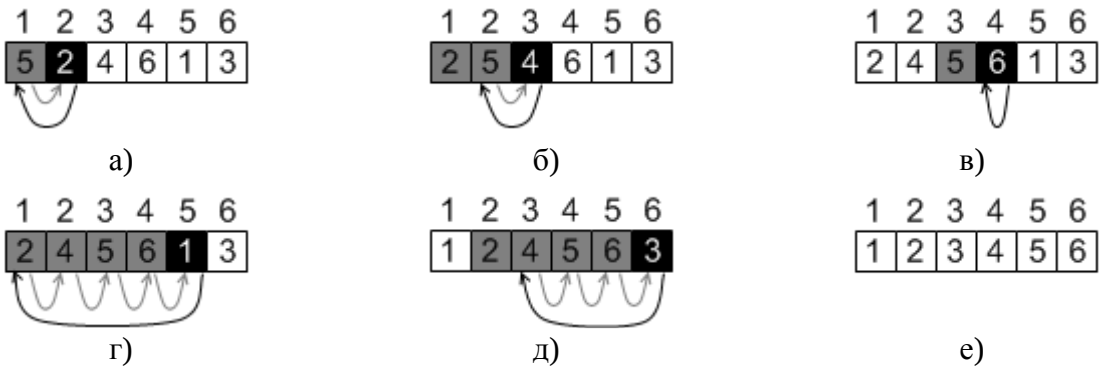

Рис. 2.1. Виконання алгоритму `Insertion_sort` над масивом A = ⟨5, 2, 4, 6, 1, 3⟩

На рис. 2.1 продемонстровано, як цей алгоритм працює для масиву $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Елементи масиву позначені квадратиками, над якими знаходяться індекси, а всередині – значення відповідних елементів. Частини *а-д* цього рисунку відповідають ітераціям циклу **for** в рядках 1-8 псевдокоду. В кожній ітерації чорний квадратик містить значення ключа, яке порівнюється зі значенням сірих квадратиків, які розташовані зліва від нього (рядок 5). Сірими стрілками позначені ті значення масиву, які зсуваються на одну позицію вправо (рядок 6), а чорною стрілкою – переміщення ключа (рядок 8). В частині *е* наводиться заключний стан відсортованого масиву. Індекс j вказує «поточну карту», яка розміщається в руці. На початку кожної ітерації зовнішнього циклу **for** з індексом j масив A складається з двох частин. Елементи $A[1...j–1]$ відповідають відсортованим картам в руці, а елементи $A[j+1…n]$ – колоді карт, які лишаються ще на столі. Відмітимо, що елементи $A[1...j–1]$ з самого початку також знаходились на позиціях від 1 до $j–1$, але в іншому порядку, а тепер вони відсортовані. Назвемо цю властивість елементів $A[1...j–1]$ **інваріантом циклу**. Іншими словами: на початку кожної ітерації циклу for з рядків 1-8 підмасив $A[1...j–1]$ містить ті самі елементи, які були в ньому з самого початку, але розташовані в несортованому порядку.

Інваріанти циклу дозволяють зрозуміти, чи коректно працює алгоритм. Необхідно показати, що інваріанти циклів володіють трьома наступним властивостями.

1. *Ініціалізація*. Вони виконуються перед першою ініціалізацією циклу.
2. *Збереження*. Якщо вони істинні перед черговою ітерацією циклу, то вони лишаються істинними й після неї.
3. *Завершення*. По закінченню циклу інваріанти дозволяють пересвідчитись у правильності алгоритму.

Якщо виконуються перші дві властивості, інваріанти циклу лишаються істинними перед кожною черговою ітерацією циклу. Можна помітити, що перші дві властивості інваріантів циклу схожі з математичною індукцією.

Розглянемо, чи зберігаються ці властивості для сортування методом включення.

Ініціалізація. Покажемо справедливість інваріанту циклу для першої ітерації, тобто при $j=2$. Таким чином, підмножина елементів $A[1...j–1]$ складається тільки з одного елементу $A[1]$, який зберігає початкове значення.

Збереження. Покажемо, що інваріант циклу зберігається після кожної ітерації. Можна сказати, що в тілі зовнішнього циклу **for** відбувається зміщення елементів $A[j–1]$, $A[j–2]$, $A[j–3]$, … на одну позицію до тих пір, поки не звільниться відповідне місце для елементу $A[j]$ (рядки 4-7), куди він і розташовується.

Завершення. Нарешті, подивимось, що відбувається по завершенню роботи циклу. При сортуванні методом включення зовнішній цикл **for** завершується, коли j більше за n, тобто коли $j = n + 1$. Підставив в формулювання інваріанту циклу значення $n + 1$, отримаємо таке твердження: у підмножині елементів $A[1…n]$ знаходяться ті самі елементи, які були в ньому на початку роботи алгоритму, але вони розташовані у відсортованому порядку. Це і підтверджує коректність алгоритму.

2.2. Машина з довільним доступом до пам’яті

Аналіз алгоритму полягає в тому, щоб передбачити потрібні для його виконання ресурси. Іноді оцінюється потреба в таких ресурсах, як пам’ять, пропускна здатність мережі або необхідне апаратне забезпечення, але найчастіше визначається час обчислення. Шляхом аналізу деяких алгоритмів, призначених для розв’язку однієї та тієї самої задачі, можна легко обрати найбільш ефективний з них. В процесі такого аналізу може також виявитись, що декілька алгоритмів приблизно рівноцінні, а всі інші варто відкинути.

З урахуванням того, що алгоритми реалізуються у вигляді комп’ютерних програм, в більшості випадків в якості технології аналізу алгоритмів буде використовуватись модель узагальненої однопроцесорної машини з довільним доступом до пам’яті (Random-

Access Machine – RAM). В цій моделі команди процесора виконуються послідовно; операції, які виконуються одночасно, відсутні.

У цю модель входять ті команди, які зазвичай можна знайти в реальних комп'ютерах: арифметичні (додавання, віднімання, добуток, ділення, обчислення остачі від ділення, наближення дійсного числа найближчим більшим чи найближчим меншим цілим числом), операції переміщення даних (завантаження значення в пам'ять, копіювання) та керуючі операції (умовне та безумовне галуження, виклик підпрограми і повернення з неї). Для виконання кожної такої інструкції потрібно певний фіксований проміжок часу. В моделі RAM є цілочисловий тип даних та тип чисел з плаваючою комою. Також передбачається, що є верхня межа розміру одного слова даних.

У моделі RAM, яка розглядається, не моделюється ієрархія пристроїв пам'яті, які сьогодні розповсюджені у звичайних комп'ютерах. Таким чином, кеш та віртуальна пам'ять відсутні у RAM. Моделі, які включають в себе таку ієрархію, набагато складніше моделі RAM, тому вони можуть ускладнити роботу. Окрім цього, аналіз, який заснований на моделі RAM, зазвичай чудово прогнозує продуктивність алгоритмів, які виконуються на реальних машинах.

Аналіз навіть простого алгоритму в моделі RAM може вимагати значних зусиль. В число необхідних математичних інструментів може увійти комбінаторика, теорія ймовірностей, алгебраїчні перетворення та здатність ідентифікувати найбільш важливі доданки в формулі. Оскільки поведінка алгоритму може відрізнятись для різних наборів вхідних значень, буде потрібна методика обліку, яка описує поведінку алгоритмів за допомогою простих та зрозумілих формул.

2.3. Аналіз алгоритму сортування методом включення

Час роботи процедури `Insertion_sort` залежить від набору вхідних значень: для сортування тисячі чисел потрібно буде більше чисел, ніж для сортування десяти. Окрім того, час сортування за допомогою цієї процедури може бути різним для послідовностей, які складаються з однакової кількості елементів, в залежності від степені їх впорядкованості до початку сортування. В загальному випадку час роботи алгоритму збільшується зі збільшенням кількості вхідних даних, тому загальноприйнята практика – представляти час роботи алгоритму як функцію, яка залежить від кількості вхідних елементів. Для цього термін «час роботи алгоритму» та «розмір вхідних даних» слід визначити детальніше.

Термін «**розмір вхідних даних**» залежить від задачі, що розглядається. В багатьох задачах, таких як сортування, це кількість вхідних елементів, наприклад, розмір n масиву для сортування. Для інших задач, таких як добуток двох цілих чисел, такою мірою може бути кількість бітів, які необхідні для представлення вхідних даних у звичайному двійковому представленні. Іноді розмір може задаватись двома числами, а не одним, так як у випадку графів, коли задається кількість вершин та кількість ребер графу.

Час роботи алгоритму для того або іншого входу вимірюється в кількості елементарних операцій, або «кроків», які необхідно виконати. Тут зручно використати поняття кроку, щоб міркування були якомога більш машино незалежними. Зараз ми будемо вважати, що для виконання кожного рядку псевдокоду потрібно фіксований час. Час виконання різних рядків може відрізнятись, але ми вважаємо, що один й той самий рядок i виконується за час c_i, де c_i – стала. Ця точка зору узгоджується з моделлю RAM та відображає особливості практичної реалізації псевдокоду на реальних комп'ютерах.

Розглянемо аналіз часу виконання алгоритму для сортування методом включення. Почнемо з того, що введемо для процедури `Insertion_sort` час виконання кожної інструкції і кількість їх повторень. Для кожного $j = 2, 3, \ldots, n$, де $n = length[A]$, позначимо через t_j кількість перевірок умови в циклі **while** (рядок 5). При нормальному закінченні циклів **for** та **while** (тобто коли перестає виконуватись умова циклу) умова перевіряється

на один раз більше, аніж виконується тіло циклу. Звісно, коментарі не є виконуваними інструкціями, а тому не збільшують час роботи алгоритму.

`Insertion_sort(A)`	час	кількість разів
1 **for** `j ← 2 to length[A]`	c_1	n
2 **do** `key ← A[j]`	c_2	$n-1$
3 `// Включення елементу A[j]`	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 **do** `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	$n-1$

Час роботи алгоритму – це сума проміжків часу, необхідних для виконання кожної інструкції, яка входить до його складу. Якщо виконання інструкції триває протягом часу c_i і вона повторюється в алгоритмі n разів, то її внесок в повний час роботи алгоритму становитиме $c_i n$. Позначимо через $T(n)$ час роботи алгоритму `Insertion_sort`:

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

Навіть якщо розмір вхідних даних є фіксованим, час роботи алгоритму може залежати від степені впорядкованості величин для сортування. Наприклад, найбільш сприятливий випадок для алгоритму `Insertion_sort` – це коли всі елементи масиву вже відсортовані. Тоді для кожного $j = 2, 3, \ldots, n$ буде виконуватись, що $A[i] \le key$ у рядку 5, ще коли i дорівнює своєму початковому значенню $j–1$. Таким чином, при $j = 2, 3, \ldots, n$ $t_j = 1$, і час роботи алгоритму в найбільш сприятливому випадку обчислюється так:

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

Цей час роботи можна записати як $an + b$, де a та b – сталі, які залежать від величин c_i; тобто цей час є **лінійною функцією** від n.

Якщо елементи масиву відсортовані в порядку, який є зворотнім до потрібного (в даному випадку в порядку зменшення), то це найгірший випадок. Кожний елемент $A[j]$ необхідно порівнювати з усіма елементами вже відсортованої підмножини $A[1..j–1]$, так що для $j = 2, 3, \ldots, n$ значення $t_j = j$. З урахуванням того, що

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

та

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2},$$

(за правилом суми арифметичної прогресії), отримуємо, що час роботи алгоритму `Insertion_sort` в найгіршому випадку визначається співвідношенням

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

Цей час роботи можна записати як $an^2 + bn + c$, де a, b, c залежать від c_i. Таким чином, це квадратична функція від n.

2.4. Порядок зростання

Вище були отримані результати аналізу щодо часу роботи алгоритму для сортування методом включення для найгіршого та найкращого випадку вхідних даних. У аналізі алгоритмів зазвичай досліджують час роботи алгоритму тільки у найгіршому випадку (тобто максимального часу роботи серед усіх вхідних даних розміром n) і на це є наступні причини.

По-перше, час роботи алгоритму в найгіршому випадку – це верхня межа цієї величини для будь-яких вхідних даних. Маючи це значення, можна точно зазначити, що для виконання алгоритму не потрібно буде більше часу.

По-друге, в деяких алгоритмах найгірший випадок зустрічається досить часто. Наприклад, якщо в базі даних відбувається пошук інформації, то найгіршому випадку відповідає ситуація, коли потрібна інформація в базі даних відсутня.

По-третє, характер поведінки «середнього» часу роботи алгоритму часто нічим не кращий поведінки часу роботи алгоритму для найгіршого випадку. Припустимо, що послідовність, для якої застосовується сортування методом включень, сформована випадковим чином. Скільки часу буде потрібно, щоби визначити, в яке місце підмасиву $A[1...j–1]$ потрібно помістити елемент $A[j]$? В середньому половина елементів підмасиву $A[1...j–1]$ менше ніж $A[j]$, а половина – більше його. Таким чином, в середньому потрібно перевірити половину елементів цього підмасиву, тому t_j приблизно дорівнює $j/2$. В результаті отримується, що середній час роботи алгоритму є квадратичною функцією від кількості вхідних елементів, тобто характер цієї залежності такий самий, що й для часу роботи в найгіршому випадку.

В деяких часткових випадках може бути цікавим середній час роботи алгоритму, тобто його математичне сподівання. Але при аналізі середнього часу роботи виникає одна проблема, яка полягає в тому, що не завжди очевидно, які вхідні дані для даної задачі будуть «середніми». Часто робиться припущення, що всі набори вхідних даних одного й того самого об’єму зустрічаються з однаковою ймовірністю. На практиці це припущення може не зберігатись, але тоді можна застосовувати **рандомізовані алгоритми**, в яких використовується випадковий вибір, і це дозволяє провести ймовірнісний аналіз.

Для полегшення аналізу процедури `Insertion_sort` були зроблені деякі припущення. По-перше, ми ігнорували фактичний час виконання кожної інструкції, представив цю величину у вигляді деякої константи c_i. Далі ми побачили, що облік всіх цих констант дає зайву інформацію: час роботи алгоритму в найгіршому випадку виражається формулою $an^2 + bn + c$, де a, b, c – деякі сталі, які залежать від вартостей c_i. Таким чином, ми ігноруємо не тільки фактичні вартості команд, але й їх абстрактні вартості c_i.

Тепер введемо ще одне абстрактне поняття, яке спрощує аналіз. Це **швидкість зростання** (rate of growth), або **порядок зростання**, часу роботи, який насправді нас і цікавить. Таким чином, до уваги приймається тільки головний член формули (тобто в нашому випадку an^2), оскільки при великих n членами меншого порядку можна знехтувати. Окрім того, сталі множники при головному члені також будуть ігноруватись, адже для оцінки обчислювальної ефективності алгоритму з вхідними даними великого об’єму вони менш важливі, аніж порядок зростання. Таким чином, час роботи алгоритму, який працює за методом включення, в найгіршому випадку дорівнює $\Theta(n^2)$ (читається «тета від n^2»). Далі Θ-позначення будуть описані детальніше.

Зазвичай один алгоритм вважається ефективнішим за інший, якщо його час роботи в найгіршому випадку має більш низький порядок зростання. Через наявність сталих множників та другорядних членів ця оцінка може бути помилковою, якщо вхідні дані невеликі. Але якщо об’єм вхідних даних значний, то, наприклад, алгоритм $\Theta(n^2)$ в найгіршому випадку працює швидше за алгоритм $\Theta(n^3)$. Таким чином, можна зазначити, що швидким алгоритмом є такий алгоритм, для якого час роботи у найгіршому випадку зростає повільно по відношенню до зростання вхідних даних.

2.5. Асимптотичні позначення

Розглядаючи вхідні дані достатньо великих розмірів для оцінки такої величини, як порядок зростання часу роботи алгоритму, ми тим самим вивчаємо **асимптотичну ефективність** алгоритмів. Це означає, що нас цікавить тільки те, як час роботи алгоритму зростає зі збільшенням розміру вхідних даних гранично, тобто коли цей розмір зростає до нескінченності. Зазвичай алгоритм, більше ефективний в асимптотичному сенсі, буде більш продуктивним для всіх вхідних даних, за винятком дуже малих.

Для простоти опису асимптотичної поведінки часу роботи алгоритму використовуються функції, область визначення яких є множина невід'ємних цілих чисел. Подібний підхід є зручним для опису часу роботи $T(n)$ в найгіршому випадку.

Вище було зазначено, що час роботи алгоритму сортування методом включення в найгіршому випадку виражається функцією $T(n) = \Theta(n^2)$. Зараз наведемо формальне означення даного позначення.

Для деякої функції $g(n)$ запис $\Theta(g(n))$ означає множину функцій:

$$\Theta(g(n)) = \left\{ \begin{array}{c} f(n) : \text{існують додатні константи } c_1, c_2 \text{ та } n_0, \text{такі що} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ для всіх } n \ge n_0 \end{array} \right\}.$$

Функція $f(n)$ належить множині $\Theta(g(n))$, якщо існують додатні константи c_1 та c_2, які дозволяють заключити цю функцію в межі між функціями $c_1g(n)$ та $c_2g(n)$ для достатньо великих n. Оскільки $\Theta(g(n))$ – це множина, то можна записати $f(n) \in \Theta(g(n))$, але також більш поширеним є запис $f(n) = \Theta(g(n))$.

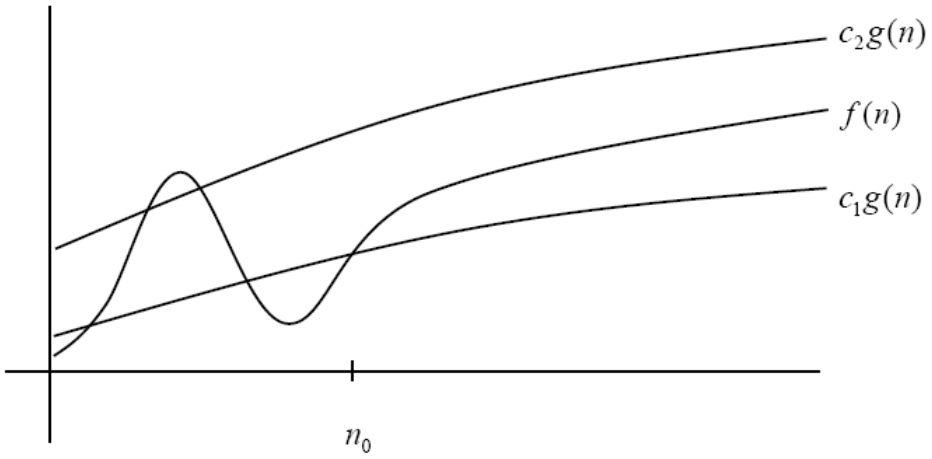

Рис. 2.2. $f(n) = \Theta(g(n))$.

На рис. 2.2 показане інтуїтивне зображення функцій $f(n)$ та $g(n)$, таких що $f(n) = \Theta(g(n))$. Кажуть, що функція $g(n)$ є **асимптотично точною оцінкою** функції $f(n)$.

Відповідно до означення множини $\Theta(g(n))$, необхідно, щоб кожний елемент $f(n) \in \Theta(g(n))$ цієї множини був асимптотично невід'ємним. Це означає, що для достатньо великих n функція $f(n)$ є невід'ємною. Відповідно, функція $g(n)$ повинна бути асимптотично невід'ємною, тому що у зворотному випадку множина $\Theta(g(n))$ виявиться порожньою.

Розглянемо приклад, в якому наводиться доведення, що $n^2/2 - 3n = \Theta(n^2)$. Для цього необхідно визначити, чому дорівнюють додатні константи c_1, c_2 та n_0, для яких виконується співвідношення

$$c_1 n^2 \le \frac{1}{2} n^2 - 3n \le c_2 n^2$$

для всіх $n \ge n_0$. Розділимо наведену нерівність на n^2:

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2 .$$

Права нерівність виконується для всіх $n \geq 1$, якщо обрати $c_2 \geq 1/2$. Аналогічно, ліва нерівність виконується для всіх $n \geq 7$, якщо обрати $c_1 \leq 1/14$. Таким чином, обравши $c_1=1/14$, $c_2=1/2$ та $n_0=7$, пересвідчуємось, що $n^2/2-3n=\Theta(n^2)$. Звісно, константи можна обрати й по іншому, але важливі не самі значення констант, а факт того, що ця можливість існує.

Продемонструємо, що $6n^3 \neq \Theta(n^2)$. Доведемо це від протилежного. Спочатку припустимо, що існують такі c_2 та n_0, що $6n^3 \leq c_2 n^2$ для всіх $n \geq n_0$. Але тоді виходить, що $n \leq c_2/6$, а ця нерівність не може виконуватись для великих n, оскільки c_2 – константа.

Інтуїтивно зрозуміло, що при асимптотично точній оцінці асимптотично додатних функцій, доданками нижчих порядків в них можна знехтувати, адже при великих n вони стають незначними. Навіть невеликої долі доданка самого високого порядку достатньо для того, щоб перебільшити доданки нижчих порядків.

Оскільки будь-яка константа – це поліном нульового ступеню, то сталу функцію можна виразити як $\Theta(n^0)$ або $\Theta(1)$. Але останнє позначення не зовсім точне, адже незрозуміло, по відношенню до якої змінної досліджується асимптотика. $\Theta(1)$ буде часто використовуватись для позначення або константи, або сталої функції деякої змінної.

У Θ-позначеннях функція асимптотично обмежується знизу та згори. Якщо ж достатньо визначити тільки асимптотичну нижню границю, то використовують O-позначення. Для заданої функції $g(n)$ позначення $O(g(n))$ (читається як «омікрон від $g(n)$» або як «велике о від $g(n)$») позначає множину функцій, таких що:

$O(g(n)) = \{f(n)$: існують додатні константи c та n_0, такі що $0 \leq f(n) \leq cg(n)$ для всіх $n \geq n_0\}$.

O-позначення використовується тоді, коли необхідно вказати верхню границю функції з точністю до сталого множника. На рис. 2.3 показане інтуїтивне представлення $f(n) = O(g(n))$.

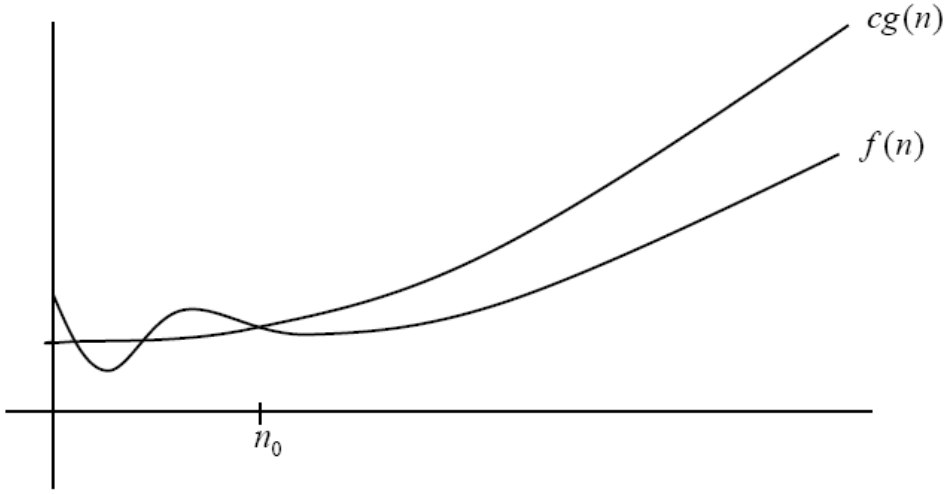

Рис. 2.3. $f(n) = O(g(n))$.

Легко помітити, що якщо $f(n) = \Theta(g(n))$, то $f(n) = O(g(n))$, оскільки Θ-позначення більш сильне, ніж O-позначення (в термінах теорії множин це можна записати як $\Theta(g(n)) \subset O(g(n))$). Таким чином, доведення того, що функція $an^2 + bn + c$, де $a>0$, належить множині $\Theta(n^2)$, одночасно доводить, що множині $O(n^2)$ належить будь-яка подібна квадратична функція. Але так само будь-яка лінійна функція $an + b$ при $a>0$ також належить множині $O(n^2)$, що легко показати, обравши $c = a + |b|$ та $n_0 = \max(1, -b/a)$.

Щоб записати час роботи алгоритму в O-позначеннях, часто достатньо просто вивчити його загальну структуру. Наприклад, наявність подвійного вкладеного циклу в структурі алгоритму сортування за методом включення вказує на те, що верхня межа часу роботи в найгіршому випадку виражається як $O(n^2)$: вартість кожної ітерації у

внутрішньому циклі обмежена згори константою O(1), індекси i та j – числом n, а внутрішній цикл виконується саме більше один раз для кожної з n^2 пар значень i та j.

Оскільки O-позначення описують верхню межу, то під час їх використання для обмеження часу роботи алгоритму в найгіршому випадку отримується верхня межа цієї величини для будь-яких вхідних даних. Таким чином, межа $O(n^2)$ для часу роботи алгоритму в найгіршому випадку може застосовуватись для часу розв'язку задачі з будь-якими вхідними даними, чого не можна сказати про Θ-позначення. Наприклад, оцінка $\Theta(n^2)$ для часу сортування включенням в найгіршому випадку не може застосовуватись для довільних даних: якщо вхідні дані вже відсортовані, то час роботи цього алгоритму оцінюється як $\Theta(n)$.

Аналогічно тому, як в O-позначеннях дається асимптотична верхня границя функції, в Ω-позначеннях дається її асимптотична нижня границя. Для заданої функції $g(n)$ позначення $\Omega(g(n))$ (читається як «омега від $g(n)$») позначає множину функцій, таких що:

$$\Omega(g(n)) = \{f(n) : \text{існують додатні константи } c \text{ та } n_0, \text{ такі що } 0 \le cg(n) \le f(n) \text{ для всіх } n \ge n_0\}.$$

Ω-позначення використовується тоді, коли необхідно вказати нижню границю функції з точністю до сталого множника. На рис. 2.4 показане інтуїтивне представлення $f(n) = \Omega(g(n))$.

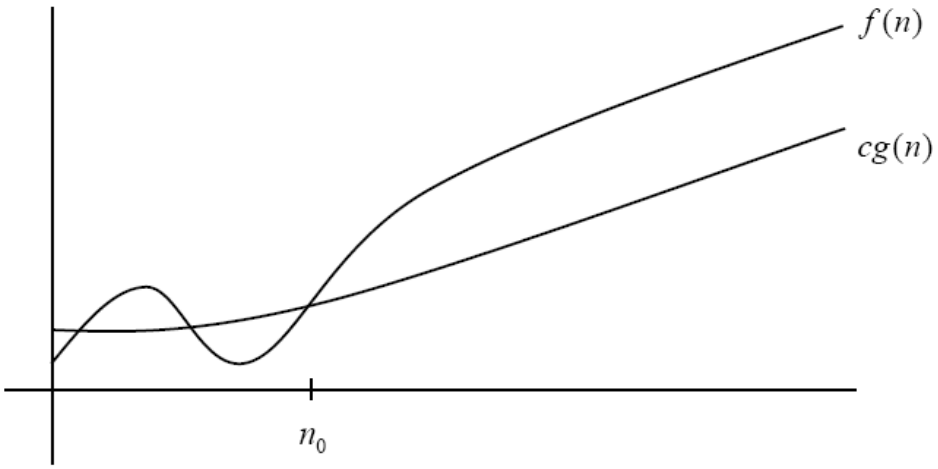

Рис. 2.4. $f(n) = \Omega(g(n))$.

Використовуючи введені означення асимптотичних позначень, легко довести наступну теорему.

Теорема 2.1. Для довільних двох функцій $f(n)$ та $g(n)$ відношення $f(n) = \Theta(g(n))$ виконується тоді й тільки тоді, коли $f(n) = O(g(n))$ та $f(n) = \Omega(g(n))$.

В якості прикладу застосування цієї теореми зазначимо, що із співвідношення $an^2 + bn + c = \Theta(n^2)$ для довільних констант a, b та c, де $a > 0$, безпосередньо слідує, що $an^2 + bn + c = O(n^2)$ та $an^2 + bn + c = \Omega(n^2)$. На практиці теорема 2.1 застосовується не для отримання асимптотичних верхніх та нижніх меж, а навпаки – для отримання асимптотичної точної оцінки за допомогою асимптотичних верхніх та нижніх меж.

Оскільки Ω-позначення використовуються для визначення нижньої границі часу роботи алгоритму в найкращому випадку, то вони також дають нижню границю часу роботи алгоритму для довільних вхідних даних.

2.6. Порівняння функцій

Асимптотичні порівняння мають деякі властивості відношень звичайних дійсних чисел. Передбачається, що функції $f(n)$ та $g(n)$ асимптотично додатні.

Транзитивність.

- Якщо $f(n) = \Theta(g(n))$ та $g(n) = \Theta(h(n))$, то $f(n) = \Theta(h(n))$,
- Якщо $f(n) = O(g(n))$ та $g(n) = O(h(n))$, то $f(n) = O(h(n))$,
- Якщо $f(n) = \Omega(g(n))$ та $g(n) = \Omega(h(n))$, то $f(n) = \Omega(h(n))$.

Рефлексивність.

- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$

Симетричність.

- $f(n) = \Theta(g(n))$ виконується тоді й тільки тоді, коли $g(n) = \Theta(f(n))$.

Перестановочна симетрія.

- $f(n) = O(g(n))$ виконується тоді й тільки тоді, коли $g(n) = \Omega(f(n))$.

Оскільки ці властивості виконуються для асимптотичних позначень, можна провести аналогію між асимптотичним порівнянням двох функцій f та g і порівнянням двох дійсних чисел a та b:

- $f(n) = O(g(n)) \approx a \leq b$,
- $f(n) = \Omega(g(n)) \approx a \geq b$,
- $f(n) = \Theta(g(n)) \approx a = b$.

Однак одна з властивостей дійсних чисел в асимптотичних позначеннях не виконується: для будь-яких дійсних чисел a та b повинно виконуватись тільки одне з співвідношень $a < b$, $a = b$ або $a > b$. Хоча будь-які два дійсних числа можна порівняти, по відношенню до асимптотичного порівняння функцій це твердження не є вірним. Для двох функцій f та g може не виконуватись ні $f(n) = O(g(n))$, ні $f(n) = \Omega(g(n))$. Наприклад, функції n та $n^{1+\sin n}$ неможна асимптотично порівняти, адже показник степеню в функції $n^{1+\sin n}$ коливається між значеннями 0 та 2 (приймаючи всі значення на цьому інтервалі).

Література

Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001) [1990]. Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03293-7. Глава 2, розділи 2.1, 2.2. Глава 3, розділ 3.1.