ТЕМА 2:
Асимптотичний аналіз.
Оцінки складності алгоритмів

§1. Основи асимптотичного аналізу
§2. Оцінки складності алгоритмів (O, Ω, Θ)
§3. Використання границь для порівняння порядку росту двох функцій

§1. Основи асимптотичного аналізу

Асимптотичний аналіз математично оцінює час виконання алгоритму підрахунком операцій.

Метою асимптотичного аналізу є порівняння витрат часу та інших ресурсів різними алгоритмами, призначеними для рішення однієї і тієї ж задачі, при великих обсягах вхідних даних.

Оцінка функції ефективності, що використовується в асимптотичному аналізі, називається *складністю алгоритму*, і дозволяє визначити, як швидко росте трудомісткість алгоритму зі збільшенням обсягу даних.

Нотація "Big O"("нотація Ландау", "О-нотація") отримала статус суперзірки серед інших концепцій математики, оскільки програмісти люблять використовувати її в обговореннях алгоритмів. Це швидкий спосіб розповісти про складність часу алгоритму.

Хоча це математична концепція, яка застосовується до різних галузей, програмісти, ймовірно, є одним з найпоширеніших користувачів О-нотації. ЇЇ використовують як скорочення для опису швидкості та/або кількості пам'яті, яку потрібно алгоритму, щоб пройти від початку до кінця.

Що ж таке Big O?

Big O - це відносне представлення складності алгоритму.
Тут є кілька важливих і навмисно підібраних слів:

1.відносне: можна порівняти яблука лише з яблуками. Не можна порівнювати алгоритм, який здійснює арифметичне множення, з алгоритмом, який сортує список цілих чисел. Але порівняння двох алгоритмів виконання арифметичних операцій (множення та додавання) покаже релевантні результати;

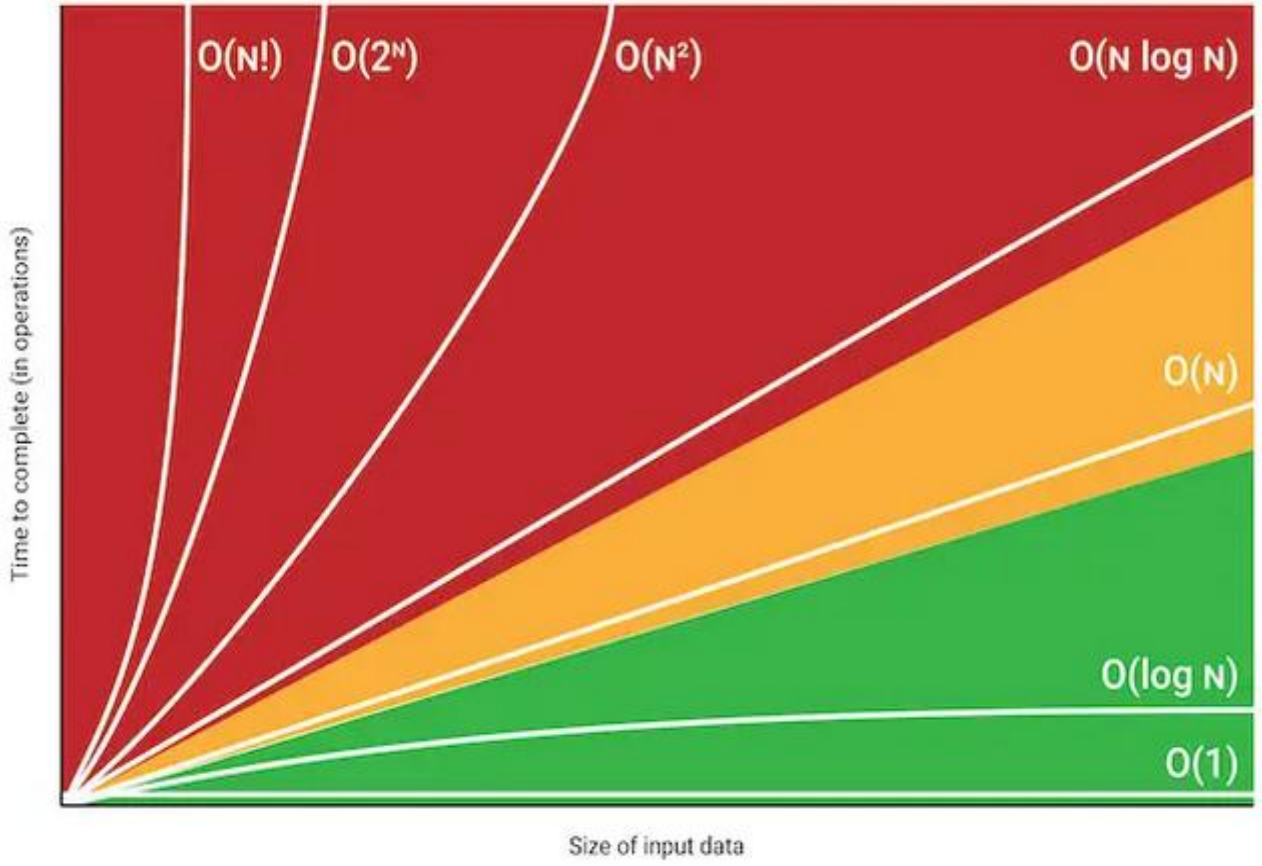
2.представлення: Big O (у своїй найпростішій формі) зменшує порівняння між алгоритмами до однієї змінної. Ця змінна вибирається на основі спостережень або припущень. Наприклад, алгоритми сортування зазвичай порівнюються на основі операцій порівняння (порівняння двох параметрів для визначення їх відносного впорядкування). При цьому передбачається, що порівняння "коштує" дорого. Але що, якщо порівняння дешеве, а перестановка дорога? Тоді треба порівнювати по іншому параметру;

2.складність: якщо мені знадобиться одна секунда для сортування 10 000 елементів, скільки часу знадобиться для сортування мільйона? У цьому випадку складність є відносною мірою до чогось іншого.

Ми частково використовуємо Big O, щоб полегшити проблему опису, наскільки «хороший» алгоритм.
Замість того, щоб говорити, що алгоритм A є "швидким навіть при дуже великому об'ємі вхідних даних" або "повільним навіть при відносно невеликому об'ємі даних", ми кажемо, що алгоритм A має часову складність O(log n), якщо він швидкий, або $O(n^2)$, наприклад, якщо повільний.
Значення в круглих дужках O() має описувати порядок величини, в якому алгоритм є швидким або повільним щодо розміру вхідних даних. Це і дало назву "Big O", але про це трохи пізніше.
Загалом, ми можемо використовувати O-нотацію для опису таких характеристик алгоритму:
•**Часова складність (time complexity)** - на скільки збільшиться час виконання алгоритму в залежності від вхідних даних;
•**Просторова складність (space complexity)** - скільки додаткової пам’яті (оперативної) буде потрібно алгоритму під час виконання, коли збільшується розмір вхідних даних;

Варто зазначити, що кількість часу та простору, які потрібні для запуску алгоритму не враховуються в Big O. Це спосіб наближено оцінити, наскільки **збільшиться** час виконання та об'єм пам'яті **при збільшені** вхідних даних.
Також хотілось би згадати, що складність алгоритму можна описувати ще двома характеристиками:
Логічна складність - кількість людино-місяців, витрачених на створення алгоритму;
Статична складність - довжина опису алгоритмів (кількість операторів).
Але до них не застосовується О-нотація, так як ці характеристики не залежать від об'єму вхідних даних і мають відношення саме до виконання алгоритму.

Класи відношень



На графіку приведено найпоширеніші функції, які зустрічаються в аналізі алгоритмів, а саме:

- $O(1)$ – константа
- $O(\log n)$ – логарифмічне
- $O(n)$ – лінійне
- $O(n \cdot \log n)$ – лінеарітмічне
- $O(n^2)$ – квадратичне
- $O(2^n)$ – експоненціальне
- $O(n!)$ – факторіальне

де $n \to \infty$

O(1)

Незалежно від того, що ви згодовуєте алгоритму, він все одно працюватиме за той самий проміжок часу.

1 запис - 1 сек 10 записів - 1 сек 100 записів - 1 сек

Наприклад: Визначення чи є число кратне двом.

O(log n)

Час роботи алгоритму ледь збільшився при збільшенні вхідних даних.

1 запис - 1 сек 10 записів - 2 сек 100 записів - 3 сек

Наприклад: бінарний пошук.

O(n)

Час розрахунку збільшується з тією ж швидкістю, що і вхідні дані.

1 запис - 1 сек 10 записів - 10 сек 100 записів - 100 сек

Наприклад: Пошук у несортованому списку.

O(n^2)

Час розрахунку збільшується зі швидкістю n^2.

1 запис - 1 сек 10 записів - 100 сек 100 записів - 10000 сек

Наприклад: бульбашкове сортування.

O(n!)

Час роботи збільшується з темпами n!, тобто, якщо n дорівнює 5, це 5x4x3x2x1 або 120. Це не так вже й погано при малих значеннях n, але при великих значеннях алгоритм стає нереально обрахувати.

1 запис - 1 сек 10 записів - 3628800 сек 100 записів - $9.332621544 \times 10^{157}$ сек

Наприклад: задача комівояжера, алгоритм сина маминої подруги :)

Визначаємо Big O

Найкращий приклад Big O, який я можу придумати, – це арифметика. Візьміть два числа (123456 і 789012). Основними арифметичними діями, які ми вивчали в школі, були: додавання, віднімання, множення та ділення.

Додавання є найпростішою операцією. Ви вирівнюєте числа (праворуч) і додаєте цифри в стовпчик, результат записуєте вкінці, а "десятки" цього переносяться до наступного стовпця.

Припустимо, що додавання цих чисел є найдорожчою операцією в цьому алгоритмі. Зрозуміло, що для додавання цих двох чисел разом ми повинні скласти разом 6 цифр (і, можливо, перенести 7-у). Якщо ми додаємо два 100-значних числа разом, ми повинні зробити 100 додавань. Якщо ми додаємо два 10 000-значних числа, ми повинні зробити 10 000 додавань. Бачите шаблон? **Складність** (будучи кількістю операцій) прямо пропорційна кількості цифр n в більшому числі. Ми називаємо це O(n) або лінійною складністю.

Операція множення буває різною. Ви вирівнюєте числа, берете першу цифру в нижньому числі і по черзі помножуєте її на кожну цифру у верхньому числі і так далі. Отже, щоб помножити два наші 6-значні числа, ми повинні зробити 36 множення. Можливо, нам потрібно буде додати до 10 або 11 стовпців, щоб отримати кінцевий результат.

Якщо у нас є два 100-цифрових числа, нам потрібно зробити 10 000 множень і 200 додавань. Для двох чисел з мільйоном цифр нам потрібно зробити один трильйон (10^{12}) множень і два мільйони додавань.

Видно, що цей алгоритм масштабується $O(n^2)$ або має квадратичну складність. Тепер варто зауважити, що існує декілька правил:

1.Завжди розглядаємо тільки найгірший варіант

Можливо ваш алгоритм і може при певних умовах завершитись за лінійних час, але нас цікавить тільки найгірший випадок, тільки хардкор! Отже, Big O стосується найгіршого сценарію виконання алгоритму.

2.Константи ігноруються

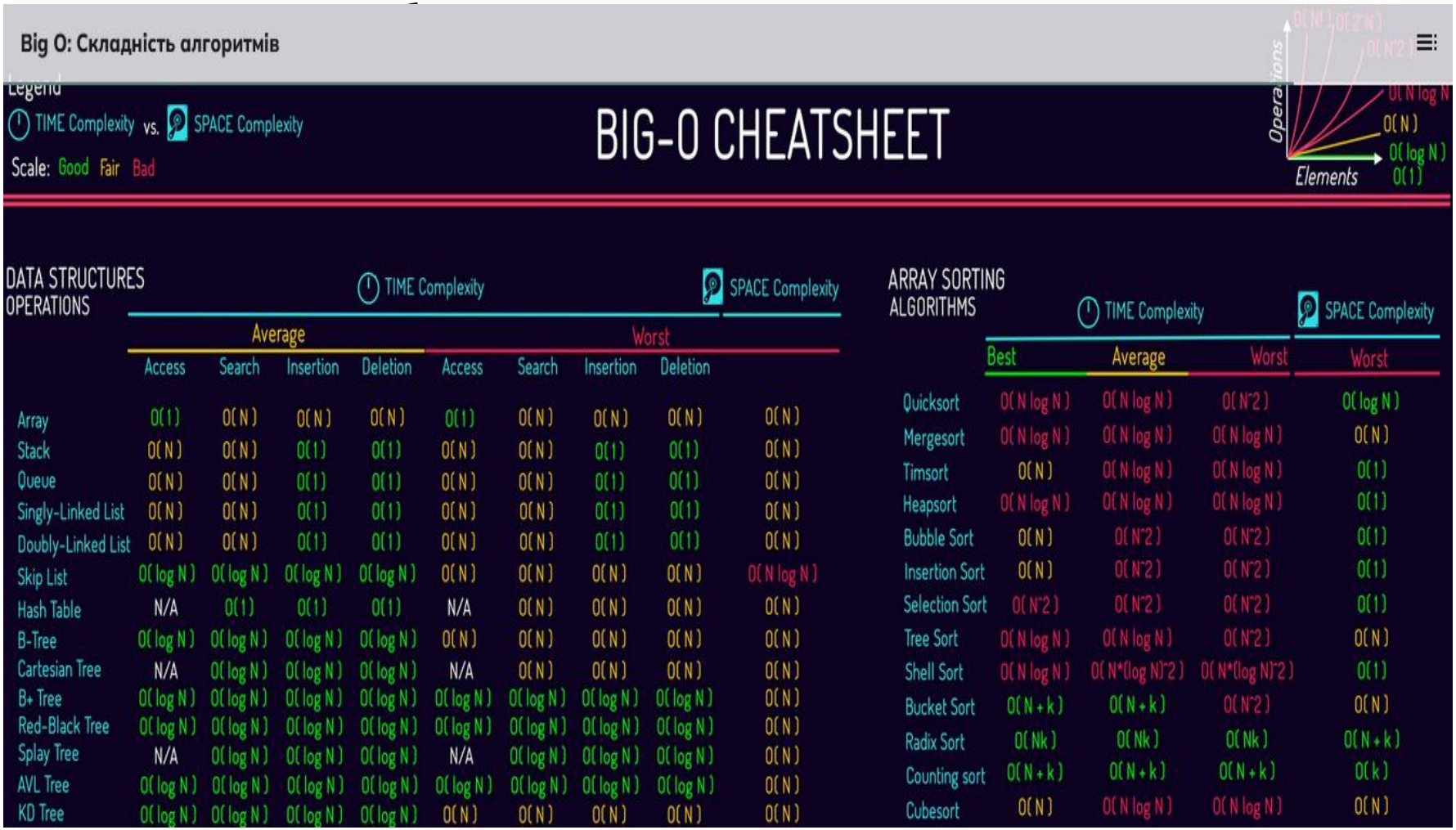
З прикладу з множенням, можливо, хтось помітить, що кількість операцій можна виразити так: $n^2+ 2n$. Але, як ви бачили з нашого прикладу з двома числами по мільйон цифр в кожному, другий доданок 2n стає незначним (на його частку припадає 0,0002% від загальної кількості операцій на цьому етапі). Розглядаючи Big O, ми відкидаємо константи.

3.Вхідні параметри спрощуються

Що станеться, якщо наш алгоритм приймає два параметри замість одного? Наприклад, якщо алгоритм описується функцією O (a * b), то її спрощують до $O(n^2)$, ми приховуємо той факт, що складність дійсно залежить від двох параметрів, а не одного. Просто зрозуміліше.

Алгоритми пошуку

Найпоширенішими алгоритмами на яких вивчають О-нотацію є алгоритми пошуку, вони найбільш наглядно демонструють різницю в складності алгоритмів.



§2. Оцінки складності алгоритмів

2.1 Оцінка O (O - велике)

$$P_n(x)=a_n x^n+a_{n-1}x^{n-1}+\ldots+a_i x^i+\ldots+a_1 x^1+a_0.$$

Алгоритм 1: для кожного доданку, крім a_0 піднести x у заданий ступінь послідовним множенням і помножити на коефіцієнт. Потім доданки скласти. Обчислення i-го доданку ($i = 1..n$) вимагає i множень.

Виходить, усього

$1 + 2 + 3 + \ldots + n = n(n+1)/2$ множень. Крім того, потрібне n додавання. Разом

$n(n+1)/2 + n = n^2/2 + 3n/2$ операцій.

Сума арифметичної прогресії

$S=\frac{a_1+a_n}{2}n$

Алгоритм 2: винесемо x за дужки й перепишемо багаточлен у вигляді

$$P_n(x)=a_0+x(a_1+x(a_2+...(a_i+...x(a_{n-1}+ +a_n x))).$$

Наприклад,

$$P_3(x) = a_3x^3 + a_2x^2 + a_1x^1 + a_0 = a_0 + x(a_1 + x(a_2 + a_3x))$$

Сама внутрішня дужка вимагає 1-го множення й 1-го додавання. Її значення використається для наступної дужки.

Отже, одне множення й одне додавання на кожну дужку, яких $n-1$ штука.

Після обчислення самої зовнішньої дужки помножити на x і додати a_0.

Всього: n (множень) + n (додавань) = $2n$ операцій.

ВИСНОВОК:

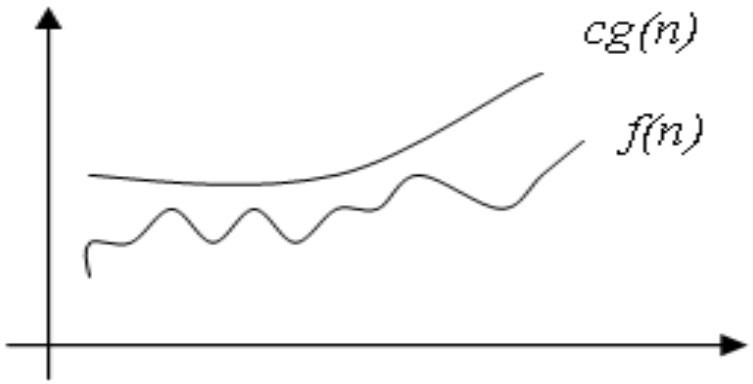
Функція $f(n)=n^2/2+3n/2$ зростає приблизно як $n^2/2$ (відкидаємо порівняно повільно зростаючий доданок $3n/2$), константним множником 1/2 також можна знехтувати. Одержимо асимптотичну оцінку для алгоритму 1, що позначається спеціальним символом $O(n^2)$ й $O(n)$ для алгоритму 2 відповідно.

Математичне тлумачення оцінки O-велике

- Нехай $f(n)$ – кількість операцій, які виконує алгоритм.
- O-символ $f(n) = O(g(n))$

$$f(n) \in O(g(n)) = \begin{Bmatrix} \exists\, c > 0, n_0 \geq 0 \\ 0 \leq f(n) \leq cg(n), \ \forall n \geq n_0 \end{Bmatrix}$$

- функція $f(n)$ не перевищувала $g(n)$ починаючи з $n > n_0$, з точністю до постійного множника.



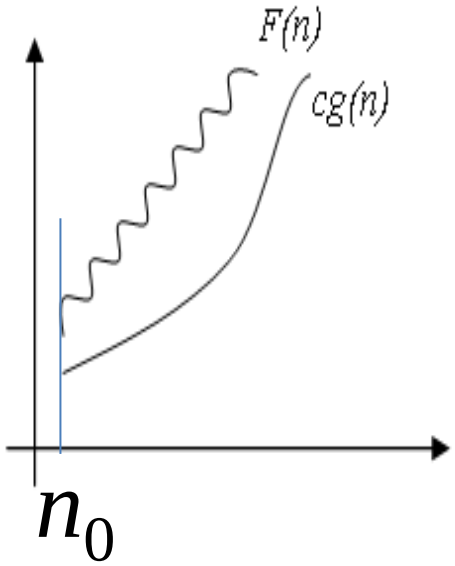
Означення. *Функція складності алгоритму f(n) має оцінку O(«O-велике») й записується як f(n) = O(g(n)), якщо існує невід'ємна функція g(n) та додатні n_0, c такі, що*
$0 \leq f(n) \leq cg(n)$,
при $n > n_0$.

2.2 Оцінка Ω (Омега-велике)

- Нехай $f(n)$ – кількість операцій, які виконує алгоритм.
- **Ω**-символ$f(n) = \mathbf{\Omega}(g(n))$

$$f(n) \in \boldsymbol{\Omega}(g(n)) = \begin{Bmatrix} \exists\, c > 0, n_0 \geq 0 \\ 0 \leq cg(n) \leq f(n), \ \forall n \geq n_0 \end{Bmatrix}$$

- функція $f(n)$ обмежена знизу $g(n)$ починаючи з $n > n_0$, з точністю до постійного множника.
- оцінка Ω означає нижню оцінку росту функції, тобто визначає клас функцій, що ростуть не повільніше, ніж $g(n)$.



Нехай кількість операцій алгоритму описує функція $f(n) = \Omega(n^2)$. Це означає, що навіть у самому вдалому випадку буде зроблено не менше ніж n^2 дій. У той час як оцінка $f(n) = O(n^3)$ гарантує, що в самому гіршому випадку дій буде близько n^3, не більше.

Означення. *Функція складності алгоритму f(n) має оцінку Ω(«омега-велике») й записується як f(n) = Ω(g(n)), якщо існує невід'ємна функція g(n) та додатні n_0, c такі, що 0 ≤, cg(n) ≤ f(n)* (14.3)
при $n > n_0$.

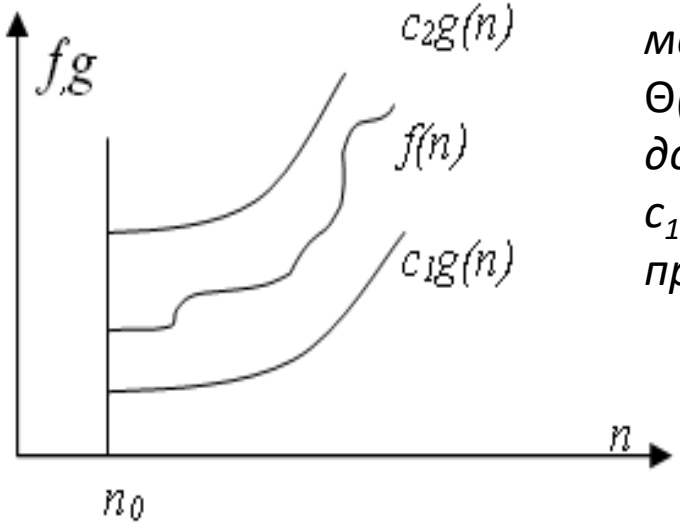
2.3 Оцінка Θ (Тетта-велике)

Нехай $f(n)$ – кількість операцій, які виконує алгоритм.

Θ -символ$f(n) = \Theta(g(n))$

$$f(n) \in \Theta(g(n)) = \begin{Bmatrix} \exists\ c1, c2 > 0, n_0 \geq 0 \\ c_1 g(n) \leq f(n) \leq c_2 g(n), \ \forall\ n \geq n_0 \end{Bmatrix}$$

Оцінка Θасимптотично обмежує функцію $f(n)$ зверху і знизу.



Означення. *Функція складності алгоритму f(n) має оцінку* Θ(тета) *й записується як f(n) = Θ(g(n)), якщо існує невід'ємна функція g(n) та додатні n_0, c_1, c_2 такі, що*
$c_1 g(n) \leq f(n) \leq c_2 g(n)$,
при $n > n_0$.

Властивості оцінок O, Ω, Θ

1) $O(f(n))+O(g(n))=O(\max(f(n),g(n)))$

Приклад: $n^3+n^2+n+1=n^3$

2) $O(c\cdot f(n))=O(f(n))$

Приклад: $O(5n^3)= O(n^3)$

3) $O(f(n))\cdot O(g(n))=O(f(n)\cdot g(n))$

Приклад: $O(n^3)\cdot O(n^2)= O(n^5)$

У такому разі говорять, що функція *g(n)* є асимптотично точною оцінкою функції *f(n),* оскільки за визначенням функція *f(n)* не відрізняється від функції *g(n)* з точністю до сталого множника.
Важливо розуміти, що
Θ *(g(n))* є не однією функцією, а множиною функцій для опису зростання *f(n)* з точністю до сталого множника. Іншими словами, функція *f(n)* належить множині Θ *(g(n)),* якщо існують додатні c_1 та c_2, що дозволяють обмежити цю функцію у рамки між функціями $c_1 g(n)$ та $c_2 g(n)$ для достаньо великих значень n.
Наприклад, для методу сортування послідовності чисел алгоритмом heapsort оцінка трудомісткості становить $f(n) = \Theta(n \log_2 n)$, тобто в цьому разі $g(n) = n \log_2 n$.
З означення $f(n) = \Theta(g(n))$ випливає, що $g(n) = \Theta(f(n))$.
Інтуїтивно зрозуміло, що при асимптотично точній оцінці асимптотично невід'ємних функцій, доданками нижчих порядків у них можна знехтувати, оскільки при великих n вони стають неістотними. Навіть невеликої частки доданка найвищого порядку достатньо для того, щоб перевершити доданки нижчих порядків.
Тому коефіцієнт при старшому доданку можна не враховувати, тому що він лише змінює зазначені константи.

2.4 Основні класи ефективності

Клас	Назва складності
1	Константа
log n	Логарифмічна
n	Лінійна
n log n	n–log–n
n^2	Квадратична
n^3	Кубічна
2^n	Експоненціальна
n!	Факторіальна

2.5 Основні недоліки підходу

1) для складних алгоритмів одержання O-оцінок, як правило, або дуже трудомістке, або практично неможливе;

2) O-оцінки занадто грубі для відображення більш тонких відмінностей алгоритмів;

3) O-аналіз дає занадто мало інформації (або зовсім її не дає) для аналізу поводження алгоритмів при обробці невеликих обсягів даних;

Нехай алгоритм A виконує деяку задачу за $0.001 \cdot N$ с, алгоритму B для цього потрібно $1000 \cdot N$.

Очевидно, що B в мільйон разів швидше, ніж A, проте A та B мають однакову часову складність O(N).

§3. Використання границь для порівняння порядку росту двох функцій

При порівнянні порядків росту двох конкретних функцій використовують метод, який полягає в обчисленні границі відношення двох функцій.

Існує три основних випадки:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\begin{cases}0, & \text{якщо } f(n) \text{ має меньший порядок росту, ніж } g(n)\\ c, & \text{якщо } f(n) \text{ має той же порядок росту, що і } g(n)\\ \infty, & \text{якщо } f(n) \text{ має більший порядок росту, ніж } g(n).\end{cases}$$

Приклад 1. Порівняти порядки росту функцій $\frac{n(n-1)}{2}$ та n^2.

Розв'язок.

$$\lim_{n\to\infty}\frac{n(n-1)/2}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2}$$

Оскільки границя дорівнює додатній константі, обидві функції мають однаковий порядок росту, що записується як $n(n-1)/2\in\Theta(n^2)$

Приклад 2. Порівняти порядки росту функцій $log_2 n$ та $\sqrt{n}$.

Розв'язок.

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{\left(\log_2 n\right)'}{\left(\sqrt{n}\right)'}=\lim_{n\to\infty}\frac{\left(\log_2 e\right)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}=2\log_2 e\lim_{n\to\infty}\frac{\sqrt{n}}{n}=0$$

Оскільки границя дорівнює нулю, функція $log_2 n$ має менший порядок росту, ніж $\sqrt{n}$, що записується як $log_2 n \in \mathrm{O}(\sqrt{n})$.

Приклад 3. Порівняти порядки росту функцій $n!$ та 2^n.

Розв'язок.

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\,\frac{n^n}{2^n e^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty$$

Функція $n!$ росте швидше, ніж 2^n, що записується як $n!\in\Omega\left(2^n\right)$