АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Структура курсу

1. Поняття алгоритму. Складність алгоритму
2. Рекурентні співвідношення та рекурсія
3. Пошукові техніки
4. Повний перебір. Метод «розділяй і володарюй»
5. Алгоритми сортування
6. Елементарні структури даних (стек, черга, зв’язаний список)
7. Дерева і графи як структури даних
8. Жадібні алгоритми
9. Динамічне програмування
10. Алгоритми на графах.

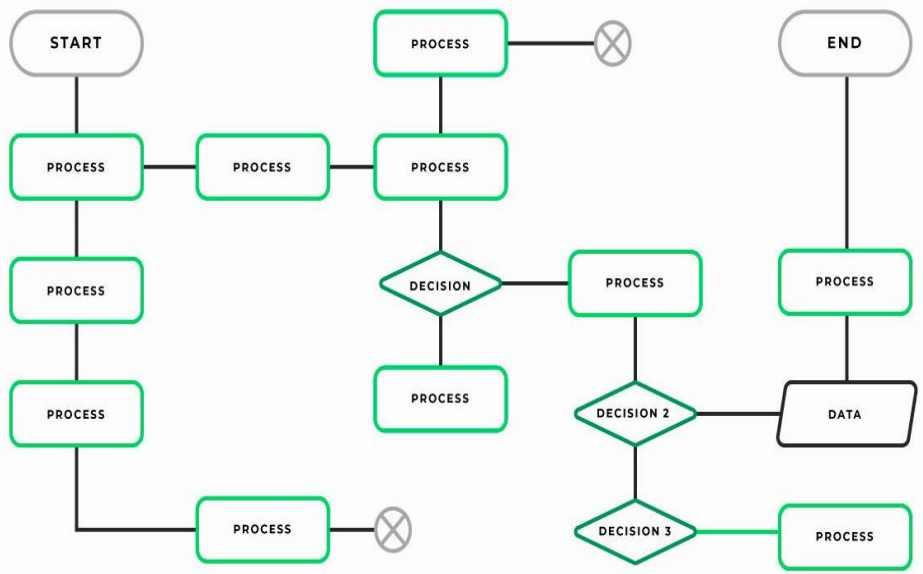
ПОНЯТТЯ АЛГОРИТМУ
СКЛАДНІСТЬ АЛГОРИТМУ

Лекція 1

Дисципліна «Алгоритми та структури даних»

Визначення алгоритму

- **Алгоритм (algorithm)** – це будь-яка коректно визначена обчислювальна процедура, на вхід (input) якої подається певна величина чи набір величин, а результатом виконання якої є вихідна (output) величина чи набір значень



- **Алгоритм** – інструмент, призначений для вирішення коректно поставленого обчислювального завдання (computational problem)
- **Алгоритм** – скінченна чітко сформульована сукупність вказівок, які визначають послідовність дій виконавця, спрямованих на досягнення мети або розв’язання задач певного класу за скінченний проміжок часу

Властивості алгоритмів

1. **Визначеність (детермінованість) алгоритму**. Алгоритм визначений, якщо він складається з допустимих команд виконавця, які можна виконати для деяких вхідних даних
2. **Скінченність**. Алгоритм повинен бути скінченним - послідовність команд, які треба виконати, має бути скінченною
3. **Дискретність.** Алгоритм складається з окремих елементарних кроків або дій, і кожна така дія має бути виконана ще до виконання наступної дії

Властивості алгоритмів

4. **Зрозумілість**. Алгоритм може бути виконаний, якщо цілком зрозуміла кожна дія (команда), і вона може бути виконана строго за її призначенням
5. **Результативність**. Зупинка алгоритму після кінцевого числа кроків (що залежить від даних) з отриманням результату
6. **Масовість**. Кожен алгоритм має мати властивість масовості, що передбачає, що такий алгоритм придатний для розв’язання будь-якої задачі з деякого класу задач

Методи запису алгоритмів

- **Словесний опис** – це опис послідовності дій за допомогою звичайної мови; достатньо компактний і зручний для подальшого кодування програми але важко сприйнятливий при високій складності алгоритму

```
Алгоритм A1

початок
  ввести a,b,c
  m присвоїти a
  якщо m>b тоді m присвоїти b
  якщо m>c тоді m присвоїти c
  вивести m
кінець
```

- **Мова програмування** – мова, зрозуміла для комп'ютера

```
Program A1;
var  a,b,c,m:integer;
BEGIN
  readln(a,b,c);
  m:=a;
  if m>b then m:=b;
  if m>c then m:=c;
  writeln(m);
END.
```

Методи запису алгоритмів

Псевдокод – напівформалізований опис алгоритмів на умовній алгоритмічній мові, що включає в себе як елементи мови програмування, так і фрази природної мови, загальноприйняті математичні позначення тощо

У псевдокоді не прийняті строгі синтаксичні правила для записи команд

Є службові слова, зміст яких визначено раз і назавжди

Службові слова виділяються в друкованому тексті жирним шрифтом, а в рукописному тексті підкреслюються

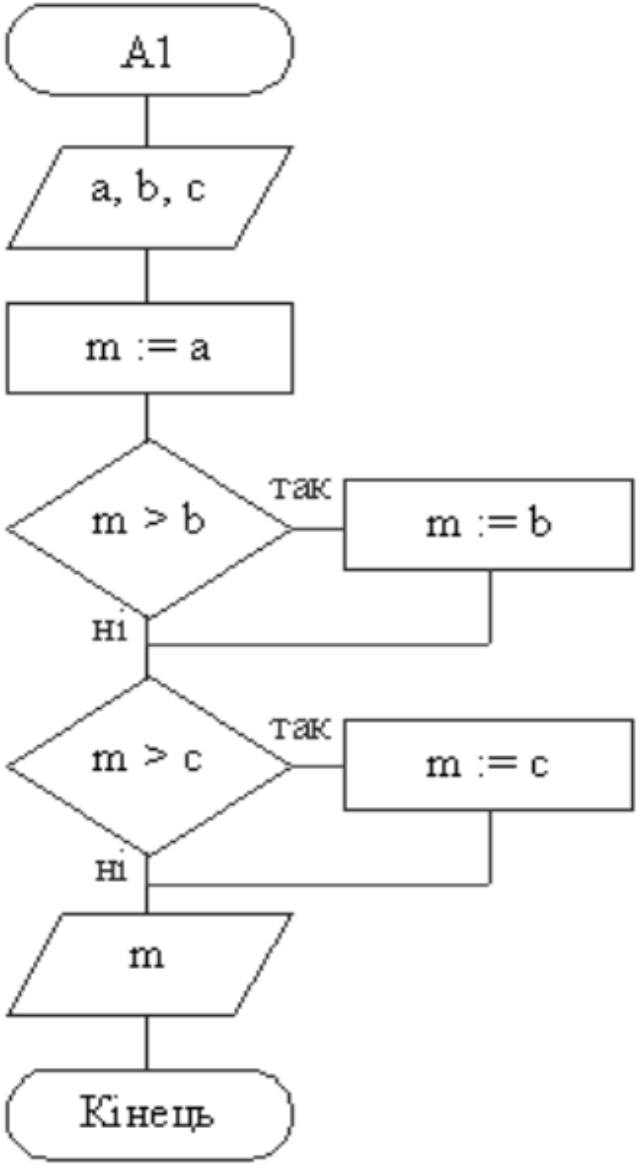
Приклад псевдокоду - шкільна алгоритмічна мова

Загальний вид алгоритму:

поч назва алгоритму (аргументи і результати)?
дано умови застосовності алгоритму?
ціль мета виконання алгоритму
поч
опис проміжних величин
послідовність команд (тіло алгоритму)
кін

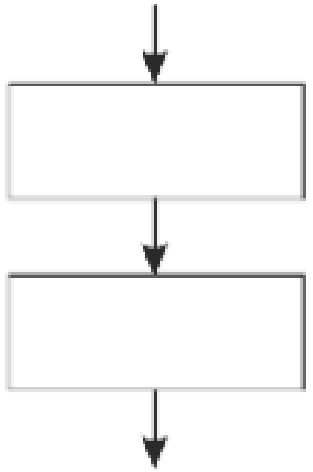
Методи запису алгоритмів

Графічний опис (блок-схема) – більш наочна, хоча й менш компактна форма подання алгоритму. Для позначення дій використовуються спеціальні графічні фігури, вигляд і призначення яких регламентуються стандартами

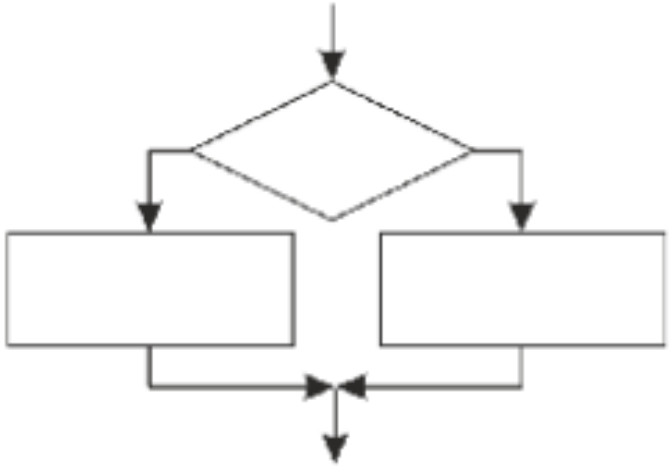


Види алгоритмів

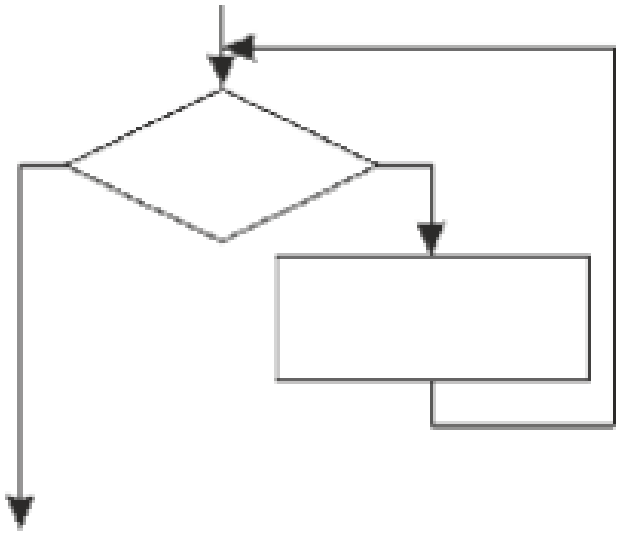
Лінійний – всі його дії виконуються послідовно, одна за одною, від початку до кінця



Розгалужений – алгоритм, в якому перевіряється певна умова, від виконання якої залежать усі подальші дії



Циклічний – алгоритм, у якому передбачено повторення деякої серії команд; однотипні дії, що повторюються декілька разів



Мова програмування

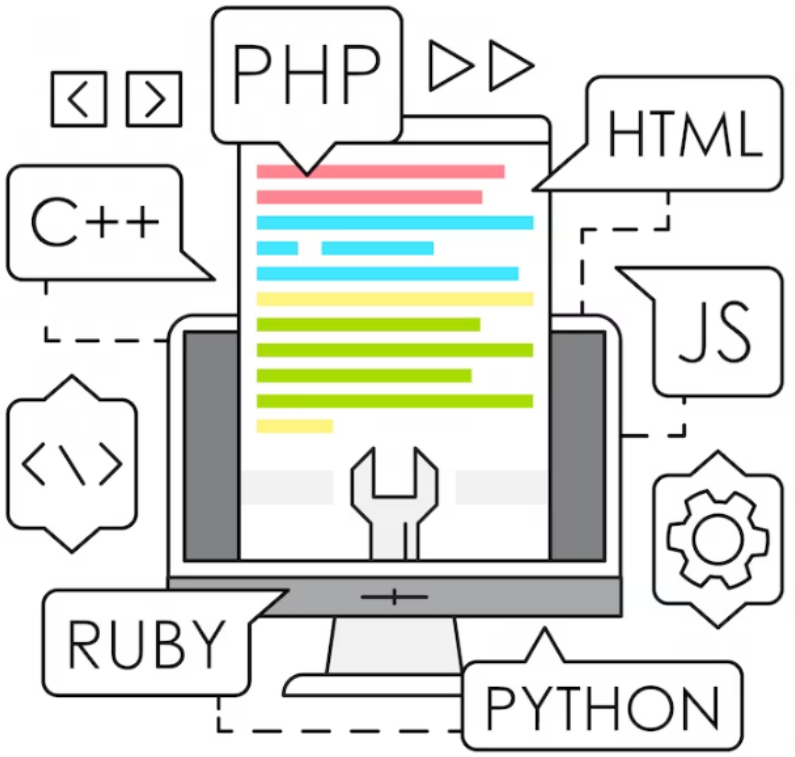
це штучна мова, що являє собою систему позначень і правил для запису алгоритмів у формі, придатній для їх виконання комп'ютером

Складові мови програмування:

Алфавіт мови – набір символів, із яких утворюються команди та інші мовні конструкції

Синтаксис мови – правила побудови команд мови програмування

Семантика мови – правила виконання комп'ютером команд, записаних мовою програмування



Аналіз алгоритму

Аналіз полягає у визначенні та аналізі таких критеріїв:

1) Час роботи програми, як функцію від вхідних даних;

2) Загальну кількість пам’яті, що необхідна для даних програми;

3) Загальний об’єм програмного коду;

4) Чи програма розв’язує коректно (правильно) поставлену задачу;

5) Чи стресостійка програма, тобто як добре буде поводити себе програма з некоректними вхідними даними;

6) Комплексність програми, тобто чи легко програму читати, розуміти та модифікувати

Час роботи (running time)

Потрібно оцінити на скільки оптимальним (швидким) є алгоритм

Виникає низка питань:

- Як оцінити швидкодію алгоритму?
- У яких одиницях швидкодію вимірювати?
- Яка врешті буде складність алгоритму?

Для оцінки складності алгоритму необхідно використовувати критерій, що не залежить від потужності комп’ютера або мови програмування на якій реалізовано алгоритм

Час роботи (running time)

Робота будь-якої програми складається з елементарних операцій, які об’єднуються у блоки для утворення інструкцій мови програмування:

- звернення до об’єкту в пам’яті;
- присвоєння;
- елементарні арифметичні операції (додавання, віднімання, множення, ділення, ділення без остачі та остача від ділення) ;
- операції порівняння;
- булеві оператори;
- виклик функції/методу;
- повернення результату функцією;
- звернення за індексом до елементу списку (масиву)
- ……..

Час роботи (running time) - це конкретна кількість операцій або навіть секунд, які алгоритм витрачає при даному розмірі вхідних даних n: $\boldsymbol{T(n)}$

Приклади

1) Дано вектор заданої величини $\boldsymbol{n}$. Оцінити час виконання програми знаходження розміру цього вектору

Наприклад, vector_length = len(my_vector)

Розмір вектору наперед заданий (і зберігається разом з вектором), тому для його визначення достатньо однієї операції – звернення до пам’яті. Таким чином:

$$\boldsymbol{T(n) = 1}$$

2) Знайти кількість операцій алгоритму, що обчислює суму компонент заданого вектора розмірності n: $a = (a1, \dots, an)$

Функція, що розв'язує цю задачу буде мати вигляд:

```
1  def sumV(a, n):
2  result = a[0]
3       i = 1
4          while i < n:
5          result += a[i]
6        i+=1
7  return result
```

Рядок	Час
2	2
3	3
4	$3 \times n$
5	$5 \times (n - 1)$
6	$4 \times (n - 1)$
7	2

Постійні витрати (не залежать від n)

Рядок 2: вартість первинної ініціалізації, наприклад, оголошення змінної для зберігання загальної суми.

Рядок 3: вартість ініціалізації лічильника циклу

Рядок 7: вартість завершальних операцій, наприклад, повернення результату.

Сума цих постійних витрат: **2+3+2=7**

Витрати, що залежать від n, пов'язані з циклом, який проходить по всіх n елементах вектору

Рядок 4: вартість операції, яка виконується n разів, перевірка умови в циклі (наприклад, i < n), яка виконується на кожній ітерації

Рядок 5: вартість основної операції додавання; щоб додати n елементів, потрібно виконати n-1 операцію додавання. Кожній такій операції присвоєно вартість 5 (зчитування двох чисел, їх додавання та запис результату)

Рядок 6: вартість інших операцій всередині циклу, що також виконуються n-1 разів (доступ до елемента масиву (a[i]) та збільшення лічильника i на 1)

Висновок для прикладу 2

Якщо скласти всі компоненти, отримаємо:

$$T(n)=(2+3+2)+3n+(5+4)(n-1)=7+3n+9(n-1)=7+3n+9n-9=12n-2$$

Ця формула є прикладом детального аналізу складності алгоритму, де кожній елементарній дії присвоєно певну "вартість".

Час виконання алгоритму залежить від **n** лінійно. В теорії алгоритмів така залежність позначається як **O(n)**, оскільки коефіцієнти та постійні величини відкидаються, і залишається лише основна складова, що описує зростання.

Часова складність алгоритму – асимптотична оцінка

Час роботи (running time) - це конкретна кількість операцій або навіть секунд, які алгоритм витрачає при даному розмірі вхідних даних **n**. Наприклад: «цей алгоритм робить $5n^2 + 3n + 7$ операцій»

Часова складність (time complexity) - це асимптотична оцінка зростання часу роботи. Відкидаються дрібні доданки й константи і залишається лише головний порядок зростання. Наприклад: «часова складність цього алгоритму - $O(n^2)$»

Часова складність алгоритму - це міра того, скільки часу займає виконання алгоритму як функція від розміру вхідних даних. Вона дозволяє оцінити, наскільки ефективним є алгоритм, не враховуючи такі фактори, як швидкість процесора чи мова програмування

Асимптотична нотація

Дональд Кнут (Donald Knuth) популяризував так звану **асимптотичну нотацію**, яка використовується для опису ефективності алгоритмів. Ця система дозволяє оцінити швидкість зростання часу виконання алгоритму (або використаної пам'яті) залежно від розміру вхідних даних (n), ігноруючи постійні коефіцієнти та незначні складові. Вона допомагає зрозуміти, як алгоритм буде поводитися при роботі з дуже великими обсягами даних

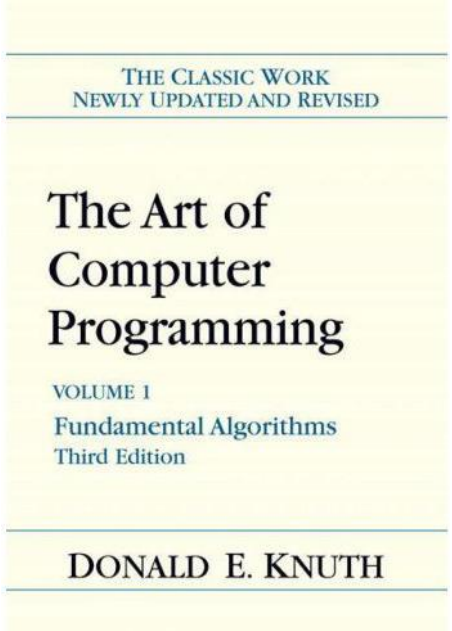
Фото з habr
Дональд Кнут - автор «Мистецтво програмування» та великий майстер ордену програмістів Землі

Відомі цитати Д.Кнута:

Остерігайтеся помилок у наведеному вище коді; я лише довів його правильність, але не перевіряв його.

Випадкові числа не повинні генеруватися випадковим чином вибраним методом.

Давайте змінимо наше традиційне ставлення до побудови програм: замість того, щоб уявляти, що наше основне завдання – вказувати комп'ютеру, що йому робити, зосередимося на поясненні людям того, що ми хочемо, щоб комп'ютер робив.



Верхні оцінки: Нотація Big O

Описує **найгірший випадок** поведінки функції. Якщо час виконання алгоритму **T(n)** належить до **O(f(n))**, це означає, що для достатньо великих значень **n**, функція **T(n)** зростає не швидше, ніж **f(n)**, помножена на деяку константу.

f(n) є асимптотичною верхньою межею для T(n)

Математичний вираз: T(n) ∈ O(f(n)) означає, що існують позитивні константи **c** і n_0 такі, що: **0 ≤ T(n) ≤ c·f(n) для всіх** $n \geq n_0$

Наприклад, якщо час виконання алгоритму T(n)=$3n^2$+5n+10, то T(n) ∈ O(n^2), оскільки для великих n домінуючим є складова n^2.

верхня оцінка - O(n^2)

Це означає, що для великих n час виконання зростає як n^2 , а менш значущі складові (2n+5) та константа (3) ігноруються.

Big O - це міра того, як швидко буде сповільнюватись робота коду з ростом кількості вхідних даних (не швидкість виконання програм!)

Нижні оцінки: Нотація **Big Ω**

Описує **найкращий випадок** поведінки функції. Якщо **T(n) ∈ Ω(f(n))**, це означає, що для достатньо великих **n**, функція **T(n)** зростає не повільніше, ніж **f(n)**, помножена на деяку константу. Таким чином, **f(n)** є **асимптотичною нижньою межею** для **T(n)**

Математичний вираз: T(n) ∈ Ω(f(n)) означає, що існують позитивні константи **c** і n_0 такі, що:

$$0 \leq c \cdot f(n) \leq T(n) \text{ для всіх } n \geq n_0$$

Наприклад, будь-який алгоритм сортування масиву за допомогою порівнянь у найкращому випадку має складність не менше, ніж **O(n)**, тобто T(n) ∈ Ω(n).

Ефективні оцінки: Нотація Θ

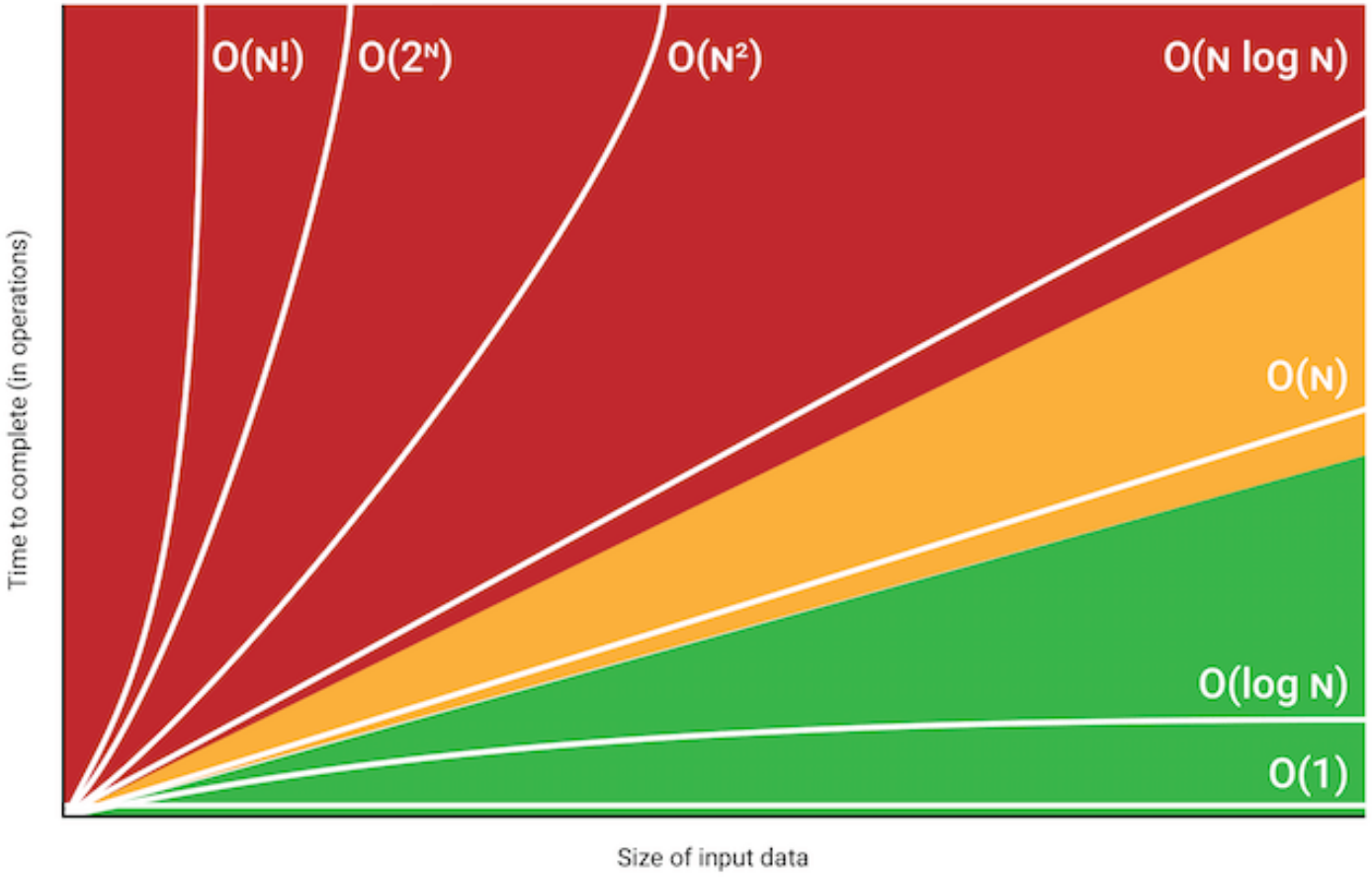
Описує **точну асимптотичну межу**. Вона показує, що час роботи алгоритму **f(n)** зростає так само, як і **g(n)**, з точністю до постійного множника. *Це об'єднання верхньої та нижньої оцінок*

Математичний вираз: f(n) ∈ Θ(g(n)) тоді і тільки тоді, коли існують такі додатні константи $\boldsymbol{c_1}$, $\boldsymbol{c_2}$ і $\boldsymbol{n_0}$, що для всіх n> $\boldsymbol{n_0}$ виконується нерівність:

$$0 \leq \boldsymbol{c_1} \cdot g(n) \leq f(n) \leq \boldsymbol{c_2} \cdot g(n)$$

Приклад: Алгоритм, який обчислює суму всіх елементів масиву з $\boldsymbol{n}$ елементів, завжди повинен пройти через кожен елемент. Тому його нижня межа Ω(n) і верхня межа O(n), а отже, точна оцінка - **Θ(n)**. Це означає, що час виконання завжди зростає лінійно, незалежно від конкретних вхідних даних

Складність за Big O-нотацією
Big O Notation Complexity



O(1) (Константна): Час виконання залишається незмінним, незалежно від обсягу даних. Це найефективніший вид складності.

O(logn) (Логарифмічна): Час виконання збільшується дуже повільно. Збільшення n у 2 рази, збільшує час виконання лише на 1 крок. Це дуже ефективно для великих обсягів даних.

O(n) (Лінійна): Час виконання прямо пропорційний розміру даних. Подвоєння n приблизно подвоює час виконання.

O(n^2) (Квадратична): Час виконання зростає значно швидше за n. Подвоєння n збільшує час виконання приблизно у 4 рази. Такі алгоритми стають неефективними для великих n.

O(2^n) (Експоненціальна): Час виконання зростає дуже швидко. Такі алгоритми практично непридатні для використання на великих обсягах даних.

Час роботи алгоритму та часова складність

Алгоритм	Час роботи (T(n)) – точна кількість кроків	Часова складність – асимптотична оцінка
Лінійний пошук (шукаємо елемент у масиві з n елементів)	У гіршому випадку: $T(n) = n$ порівнянь	**O(n)**
Пошук максимуму в масиві	$T(n) = n - 1$ порівнянь	**O(n)**
Бінарний пошук (у відсортованому масиві)	$T(n) = log_2 n + 1$ крок	**O(log n)**
Бульбашкове сортування	У гіршому випадку: $T(n) = n(n-1)/2 \approx n^2/2$ порівнянь	**$O(n^2)$**
Швидке сортування (QuickSort)	У середньому: близько $T(n) \approx 1.39 n log_2 n$ кроків	**O(n log n)**
Перемноження двох n×n матриць (звичайний метод)	$T(n) = 2n^3 - n^2$ операцій множення/додавання	**$O(n^3)$**