АЛГОРИТМИ І СТРУКТУРИ ДАНИХ

Тема 1.

Вступ до курсу.

Алгоритми, час виконання алгоримів

Олексій Панасенко

17 серпня 2023 р.

День в історії

17 серпня 1943 року — народився Роберт Де Ніро, американський актор. Дворазовий володар премії «Оскар» (1975, 1981 роки).

17 серпня 2008 року — на Олімпійський іграх в Пекіні американський пловець Майкл Фелпс допоміг команді США виграти золоті нагороди в естафеті $4\times$ 100 м і став першим атлетом в історії, який здобув вісім золотих медалей впродовж лише одних Олімпійських ігор.

17 серпня 1968 року — народився Андрій Кузьменко (псевдонім — Кузьма, Кузьма Скрябін), український співак, композитор, поет, письменник. Лідер гурту «Скрябін», Герой України.

Вступ до курсу

Про що цей курс

- Про алгоритми для розв’язування обчислювальних задач
- Про розуміння того, як алгоритми працюють
- Наскільки побудований алгоритм ефективний?
- Як створювати ефективні алгоритми?
- Структури даних, ефективність операцій і як ефективно використовувати структури даних

Структура курсу

1. Вступ до дисципліни. Математичні основи аналізу алгоритмів.
2. Рекурентні співвідношення та рекурсія.
3. Пошукові техніки.
4. Повний перебір. Метод «розділяй і володарюй».
5. Алгоритми сортування – 1.
6. Алгоритми сортування – 2.
7. Елементарні структури даних (стек, черга, зв’язаний список).
8. Дерева і графи як структури даних.
9. Жадібні алгоритми.
10. Динамічне програмування.
11. Алгоритми на графах.
12. Алгоритми на зважених графах.

Основна література

- Кормен Т., Лейзерсон Ч., Рівест Р., Стайн К. Вступ до алгоритмів : переклад з англійської третього видання. — К. : К. І. С., 2019. — 1288 с.

  Прев'ю цієї книги на Google-books: *https://cutt.ly/kwgRpUQF*

  Історію про те, як ця книга перекладалась українською мовою, можна прочитати тут: *https://linux.org.ua/index.php?topic=11997.msg204710*

- Креневич А. Алгоритми і структури даних : підручник. — К. : ВПЦ "Київський Університет", 2021. – 200 с.

  Підручник та інші матеріали можна завантажити тут: *https://github.com/krenevych/algo/tree/master*

Алгоритм.
Коректність алгоритму.
Час виконання алгоритму

Алгоритм

- Алгоритм — це будь-яка коректно визначена обчислювальна процедура, яка бере деяке значення або набір значень як *входові дані* і її результатом є певне значення або набір значень (*виходові дані*).
- Приклад: задача сортування
  Входові дані: послідовність n чисел: $(a_1, a_2, \ldots, a_n)$
  Виходові дані: перестановка $(a'_1, a'_2, \ldots, a'_n)$ входової послідовності така, що $a'_1 \leqslant a'_2 \leqslant \ldots \leqslant a'_n$.
- При цьому конкретна входова послідовність, наприклад (31, 52, 6, 26, 58), називається примірником (**instance**) задачі сортування.

Коректність алгоритму

- Алгоритм називається коректним, якщо для *будь-яких* входових даних результатом його роботи є правильні виходові дані. Тоді кажуть, що коректний алгоритм **розв'язує** обчислювальну задачу.
- Як встановити, чи алгоритм є коректним?
- Ким би Ви хотіли бути?
  - A: «Я розробив алгоритм, і я думаю, що він коректний»
  - B: «Я протестував свій алгоритм на трьох входових примірниках і він працює!»
  - C: «Я можу **довести**, що мій алгоритм продукує *завжди* коректний результат».
- В ідеалі всі алгоритми повинні супроводжуватись підтвердженням його коректності.

Структури даних

- Структура даних — це спосіб зберігання та впорядкування даних таким чином, щоб полегшити до них доступ і їх модифікацію.
- Жодна конкретна структура даних не підходить добре до всіх потреб, тож важливо знати переваги та обмеження деяких з них.

Як оцінити час роботи алгоритму?

- Комп'ютери різні (тактова частота процесора, швидкість запису і читання з пам'яті...)
- Архітектури комп'ютерів можуть бути різними...
- Вибір мови програмування впливає на час виконання алгоритму
- Є потреба в моделі, яка забезпечує **хороший рівень абстракції**:
  - Дає хорошу уявлення про час, який потрібний алгоритму
  - Дозволяє порівнювати різні алгоритми

Аналіз алгоритмів

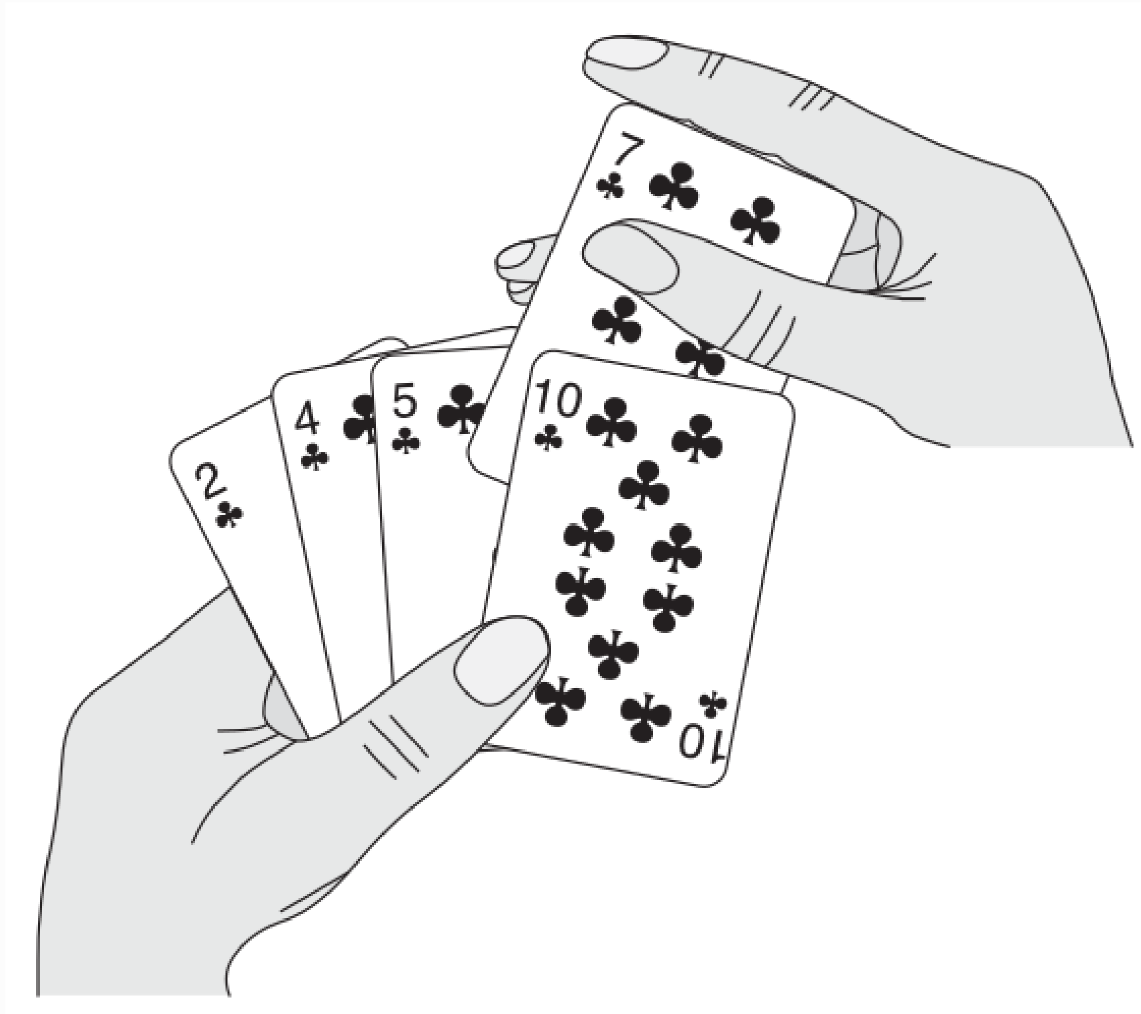
- *Аналіз* алгоритму полягає в передбаченні того, яких ресурсів той потребуватиме.
- Рівнодоступна адресна машина (RAM — Random-Access Machine).
- Інструкції виконуються одна за одною, жодні операції не виконуються одночасно.
- Елементарні операції:
    - Додавання, віднімання, множення, ділення, остача
    - Логічні операції, зсуви, порівняння
    - Присвоєння значень змінним
    - Інструкції контролю: цикли, виклик методів.

Аналіз алгоритмів

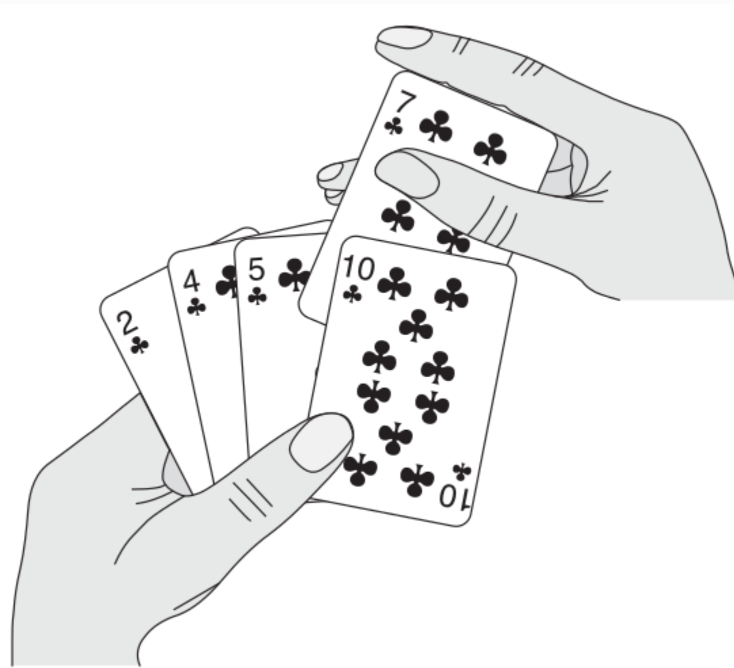
- Модель загальної вартості: підраховують кількість елементарних операцій в моделі RAM
- Припускається, що усі операції потребують однакового часу.
  Час роботи алгоритму A на примірнику I — кількість елементарних операцій у RAM-моделі для алгоритму A на примірнику I
- Але… не треба захоплюватись детальним підрахунком операцій
  - Абстрагуємося від констант.
  - Фокус на **асимптотичному зростанні** часу виконання залежно від розміру задачі.
  - Будемо використовувати спеціальну нотацію.

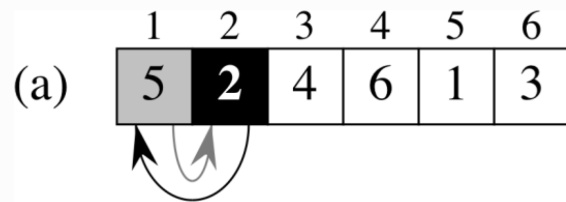
Сортування вставлянням (алгоритм InsertionSort)

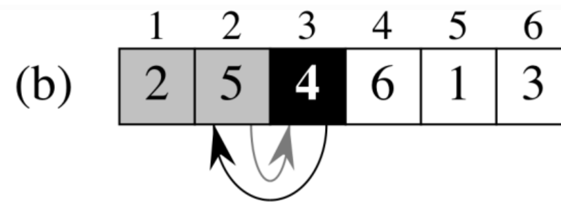
Ідея: побудувати відсортовану послідовність, вставити наступний елемент у потрібне положення.

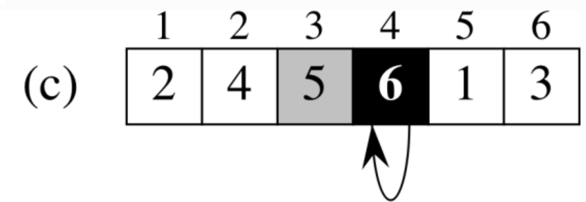


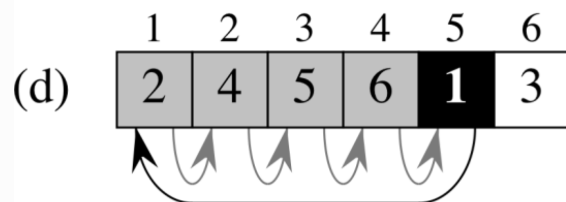
Сортування вставлянням (алгоритм InsertionSort)

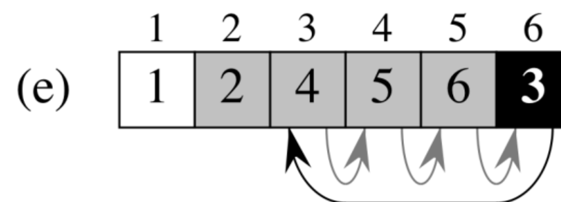


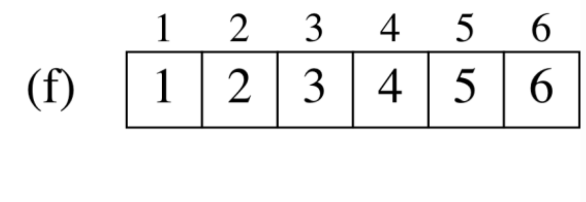












Сортування вставлянням (алгоритм InsertionSort)

```
1 for j = 2 to A.length:
2     key = A[j]
3     // Вставити A[j] у відсортовану послідовність A[1...j-1]
4     i = j - 1
5     while i > 0 and A[i] > key:
6         A[i+1] = A[i]
7         i = i - 1
8     A[i+1] = key
```

Сортування вставлянням (алгоритм InsertionSort)

Хочемо знати відповіді на такі питання:

1. Чи можемо ми стверджувати, що алгоритм InsertionSort є коректним завжди?
    - Доведення із використанням інваріантів циклу
2. Як довго працює InsertionSort? (Відповідь буде далі)

Метод інваріантів циклів

- Популярний спосіб доведення коректності алгоритму з циклами.
  - Приклад: «Після i ітерацій циклу принаймні i об'єктів є хорошими»
  - **Започаткування (ініціалізація)**. Інваріант циклу справедливий перед першою ітерацією циклу.
  - **Збереження**. Якщо інваріант циклу справедливий після i-ої ітерації, то він справедливий і після $i+1$-ої.
  - **Завершення**. По завершенню циклу інваріант циклу дозволяє пересвідчитись в коректності алгоритму.

Коректність алгоритму InsertionSort

- Інваріант циклу: «На початку кожної ітерації циклу **for**, підмасив $A[1..j-1]$ складається із початкових елементів $A[1..j-1]$, але у відсортованому вигляді»
- Започаткування: для $j = 2$ підмасив $A[1]$ є початковим $A[1]$ і, очевидно, є відсортованим.
- Збереження: Цикл **while** зміщує $A[j-1]$, $A[j-2]$, … на одну позицію праворуч і вставляє $A[j]$ в коректне місце $i + 1$. Тоді $A[1..j]$ містить початкові $A[1..j]$, але у відсортованому порядку.
- Завершення: Цикл **for** завершеється при $j = n + 1$. Тоді інваріант циклу для $j = n + 1$ стверджує, що масив містить елементи початкового масиву $A[1..n]$ у відсортованому порядку.

Час виконання (runtime) InsertionSort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	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	$n-1$

(тут t_j — кількість перевірок умов в циклі while)

Час виконання (runtime) InsertionSort

- Як проаналізувати час виконання алгоритму InsertionSort (наївний спосіб)
  1. Припустимо, що рядок i виконується впродовж часу c_i
  2. Порахуємо скільки разів виконається цей рядок
  3. Обчислити суму добутків разів і витрат
- Результат (трошки «сумбурний», потім будуть спрощені позначення):

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

$$+ c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Час виконання (runtime) InsertionSort

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

Найкращий випадок: масив відсортований, $t_j = 1$. Тоді

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

Отримали лінійну функцію від n.

Час виконання (runtime) InsertionSort

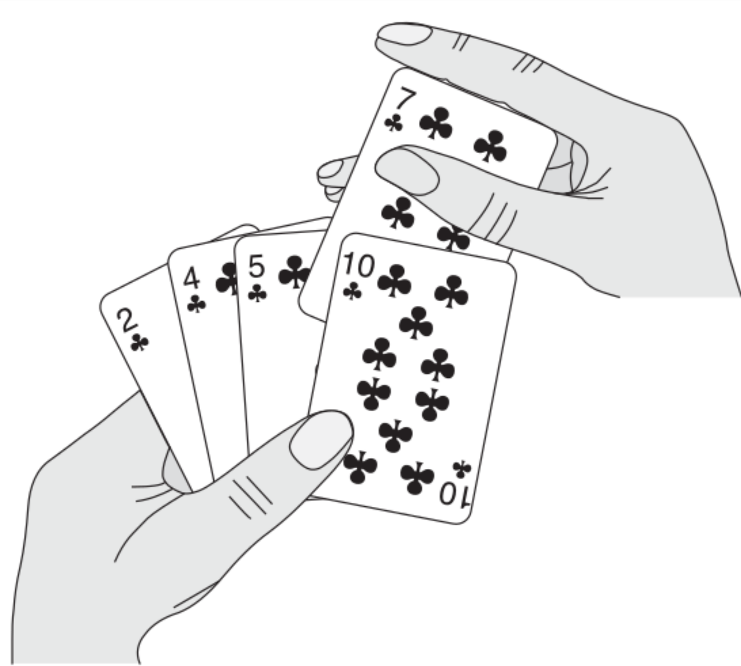
Найгірший випадок: масив відсортований у зворотньому порядку, $t_j = j$.
Тоді, врахувавши, що $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$:

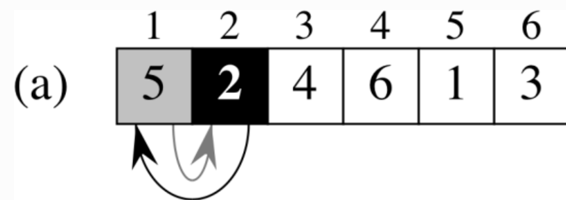
$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) +$$
$$+ c_6\left(\frac{n(n+1)}{2}\right) + c_7\left(\frac{n(n+1)}{2}\right) + c_8(n-1) =$$
$$= \ldots = an^2 + bn + c.$$

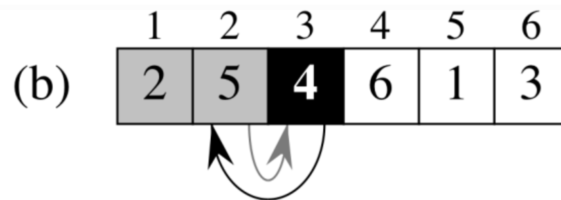
Отримали квадратичну функцію від n.

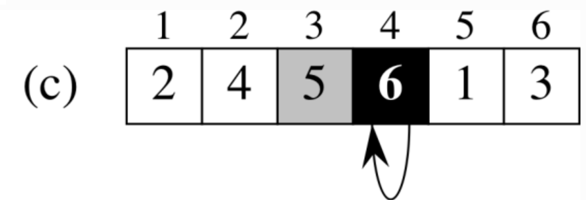
Час виконання алгоритму

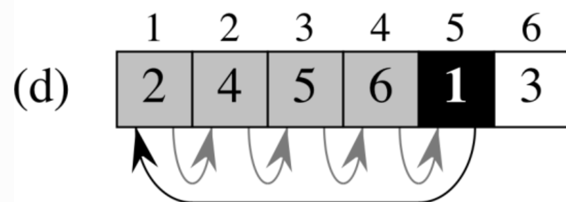
Резюме: InsertionSort

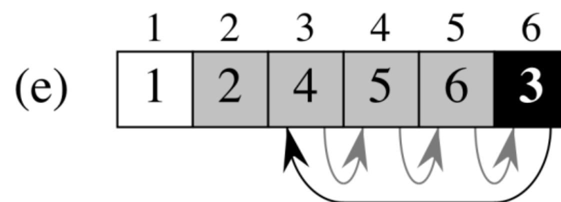


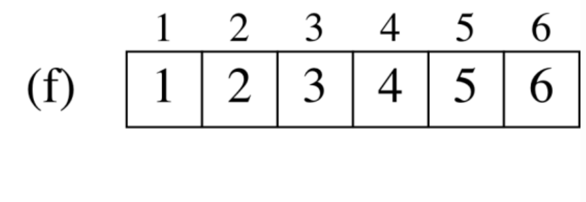












Резюме: Час виконання InsertionSort

- Загальна формула:

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

- Для найкращого випадку формула спрощується до $T(n) = an + b$ для деяких констант $a > 0$, b (які залежать від $c_1, c_2, \ldots$) — **лінійна** функція
- Для найгіршого випадку формула спрощується до $T(n) = an^2 + bn + c$ для деяких констант $a > 0$, b, ,c (які залежать від $c_1, c_2, \ldots$) — **квадратична** функція

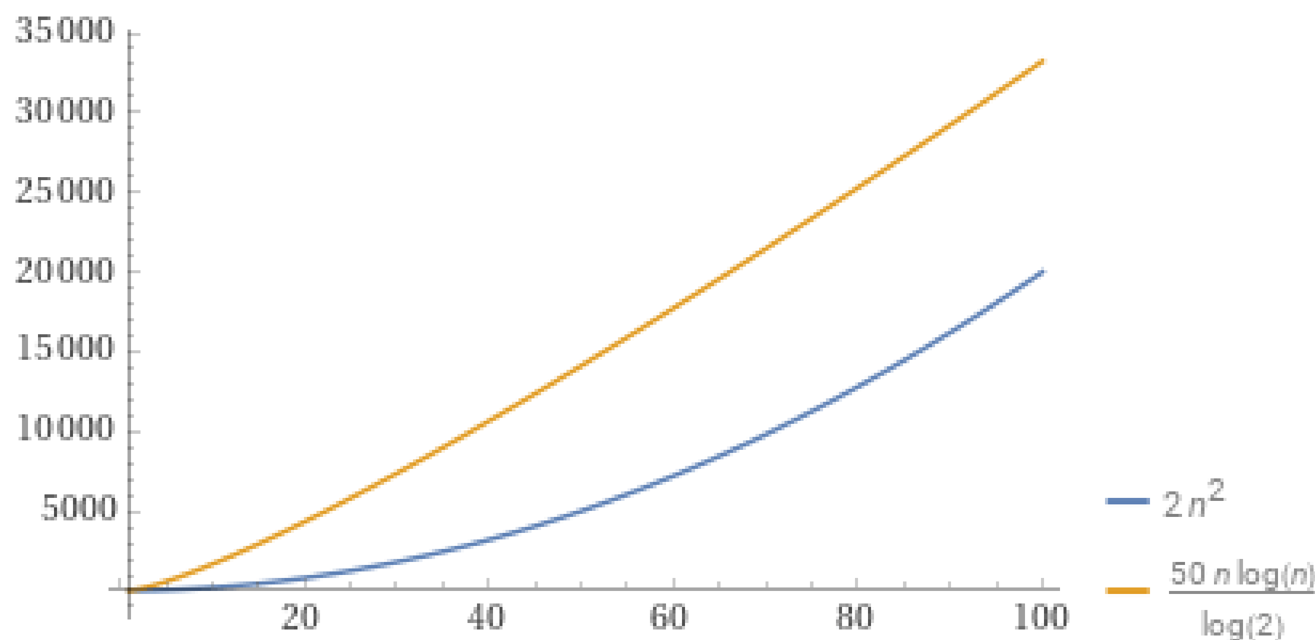
Про найкращий і найгірший випадок

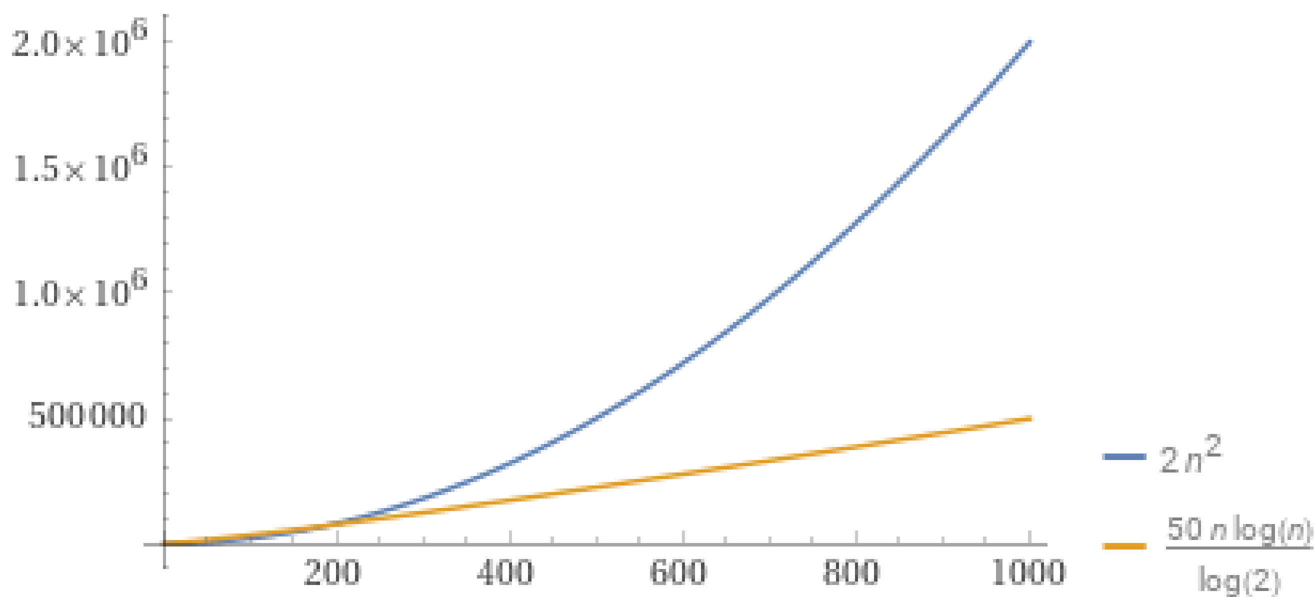
- Час виконання для кожного окремого примірника буде між найкращим і найгіршим випадком.
- Найкращий випадок vs найгірший випадок — що важливіше?
- Проміжний випадок: час виконання на «середніх» вхідних даних. Але що значить «середні» вхідні дані?
- Чому найгірший випадок важливіший:
  - гарантія того, що алгоритм ніколи не працюватиме довше
  - для окремих алгоритмів найгірший випадок є доволі типовим
  - Часто (але не завжди) «середній» випадок настільки ж «поганий», як і найгірший

Порівняємо час виконання двох алгоритмів

- Давайте порівняємо два алгоритми:
  - Час виконання алгоритму A $2n^2$
  - Час виконання алгоритму B $50n \lg n$
- Якому ви віддасте перевагу?
- Побудуємо графіки, з допомогою wolframaplha: https://tinyurl.com/rm2o29o

Порівняємо час виконання двох алгоритмів





Порівняємо час виконання двох алгоритмів

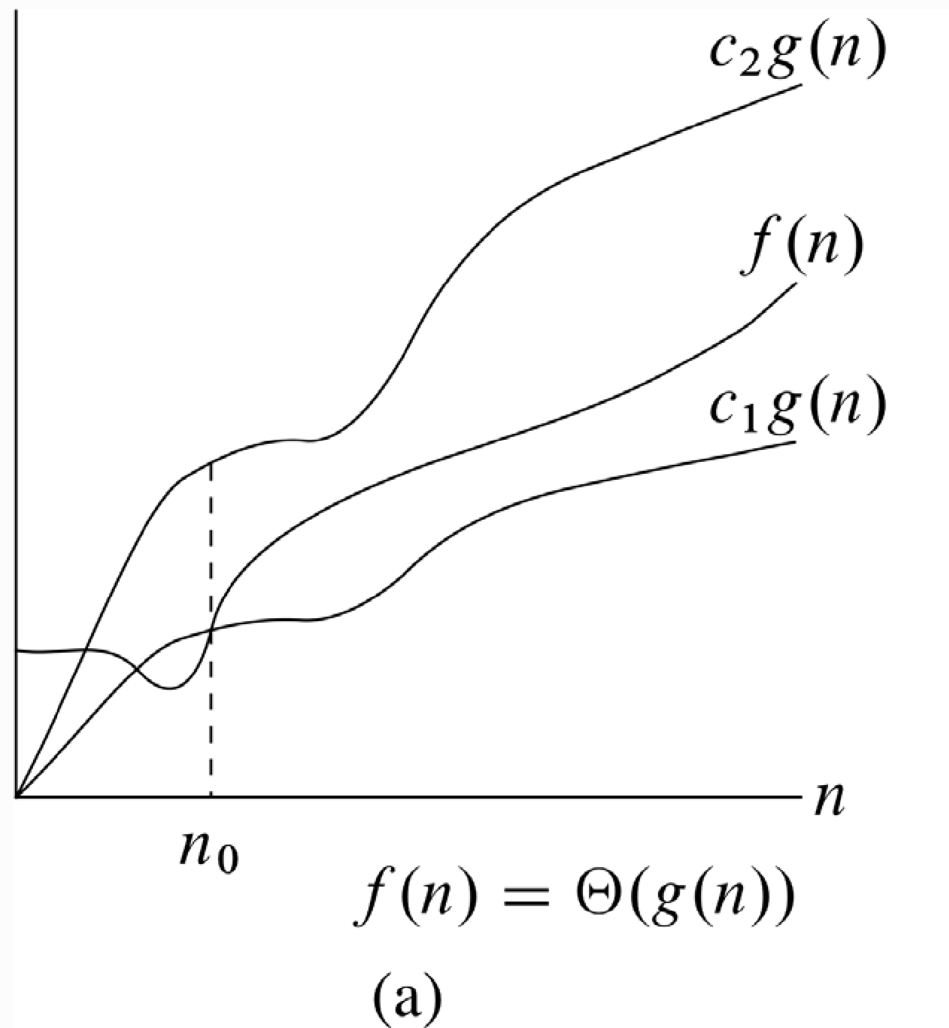
- Із зростанням n n^2 стає набагато більшим ніж $n \log n$.
- Для великих n сталий множник стає не настільки важливим.
- Доданки меншого порядку (типу $+10n$ у $2n^2 + 10n$) стають *несуттєвими* для великих n.
- Варто думати про великі n, оскільки задачі з невеликими значеннями n в будь-якому разі є простими.
- Поради:
    - Якщо задача не допускає великих n — слід використовувати **найпростіший** алгоритм.
    - В іншому разі — слід використовувати **найефективніший** алгоритм (найкраще зростання відносно n)

Асимптотичні позначення

Асимптотичні позначення: Θ

- Ідея: стежити за **асимптотичним зростанням**.
- Нехтуємо сталими множниками
- Нехтуємо доданками меншого порядку
- Нехтуємо «сплесками» для малих n
- Інтуїтивне позначення: Θ вказує на доданок найбільшого зростання: $2n^2 + 3n = \Theta(n^2)$

Асимптотичні позначення: Θ



Асимптотичні позначення: Θ

- Для заданої невід'ємної функції $g(n)$ позначимо через $\Theta(g(n))$ таку множину функцій

$$\Theta(g(n)) = \{f(n)\colon \text{існують такі константи } 0 < c_1 \leqslant c_2 \text{ і } n_0\text{, що } 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n) \text{ для всіх } n \geqslant n_0\}$$

- Функція $f(n)$ належить множині $\Theta(g(n))$ якщо для достатньо великих n її можна обмежити згори і знизу функціями $c_1 g(n)$ і $c_2 g(n)$. Тоді $f(n) \in \Theta(g(n))$.
- Говорять, що функція $g(n)$ є асимптотично точною оцінкою функції $f(n)$.

Асимптотичні позначення: Θ

Приклад: Доведемо, що $\frac{3}{2}n^2 + \frac{7}{2}n - 4 = \Theta(n^2)$.

- Для доведення достатньо знайти константи c_1, c_2 і n_0 такі, що для всіх $n \geqslant n_0$:

$$0 \leqslant c_1 n^2 \leqslant \frac{3}{2}n^2 + \frac{7}{2}n - 4 \leqslant c_2 n^2$$

- Поділимо на n^2:

$$0 \leqslant c_1 \leqslant \frac{3}{2} + \frac{7}{2n} - \frac{4}{n^2} \leqslant c_2$$

- Це справджується, наприклад, для $c_1 = \frac{3}{2}$, $c_2 = 2$, $n_0 = 7$.

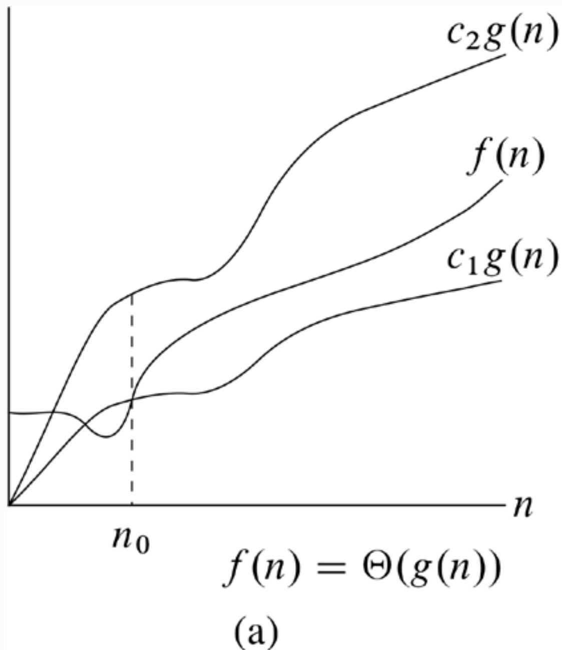
Асимптотичні позначення: Θ

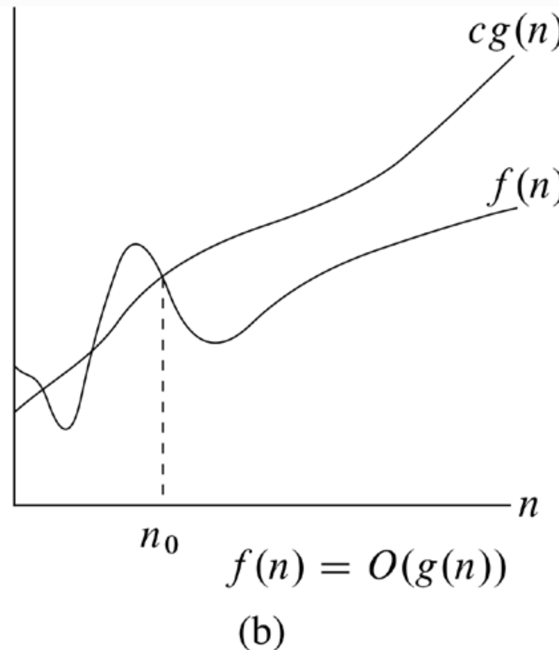
Приклади:

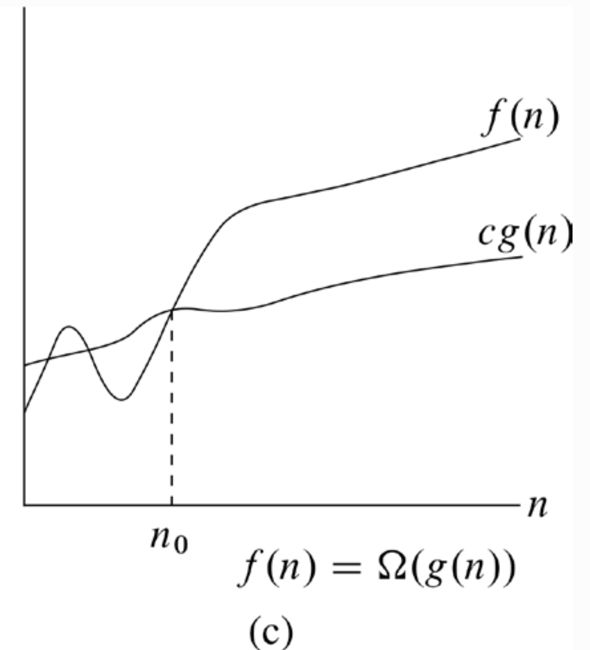
- $2n^2 = \Theta(n^2)$
- $2n^2 - 10n = \Theta(n^2)$
- $50n \log n = \Theta(n \log n)$
- **Але**: $2n^2 \neq \Theta(n)$, оскільки не існує константи c_2 такої, що $2n^2 \leqslant c_2 n$ *для всіх* $n \geqslant n_0$.
- **Але**: $2n^2 \neq \Theta(n^3)$, оскільки не існує константи c_1 такої, що $2n^2 \geqslant c_1 n^3$ *для всіх* $n \geqslant n_0$.

Асимптотичні позначення: Θ, O, Ω

- Θ виражає точну верхню і нижню оцінки для $f(n)$.
- Використовуємо O („O-велике“) якщо ми хочемо оцінити лише верхню межу.
- Використовуємо Ω якщо ми хочемо оцінити лише нижню межу







Асимптотичні позначення: Θ, O, Ω

- Для заданої невід'ємної функції $g(n)$ позначимо через $O(g(n))$ і $\Omega(g(n))$ такі множини функцій:

$$O(g(n)) = \{f(n)\colon \text{існують такі константи } 0 < c \text{ і } n_0\text{, що } 0 \leqslant f(n) \leqslant cg(n) \text{ для всіх } n \geqslant n_0\}$$

$$\Omega(g(n)) = \{f(n)\colon \text{існують такі константи } 0 < c \text{ і } n_0\text{, що } 0 \leqslant cg(n) \leqslant f(n) \text{ для всіх } n \geqslant n_0\}$$

- O і Ω разом дають Θ: для довільних $f(n)$ і $g(n)$: $f(n) = \Theta(g(n))$ тоді і тільки тоді, коли $f(n) = O(g(n))$ і $f(n) = \Omega(g(n))$.

Асимптотичні позначення: o, ω

- Асимптотична верхня межа, позначена через O, може описувати асимптотичну поведінку функції з різною точністю (порівняйте: $2n^2 = O(n^2)$ і $2n = O(n^2)$).
- Для відзначення того, що верхня межа не є асимптотично точною оцінкою функції, використовують o-позначення.
- Для заданої невід'ємної функції $g(n)$ позначимо через $o(g(n))$ таку множину функцій:

$$o(g(n)) = \{f(n)\colon \text{для будь-якого } c > 0 \text{ існує } n_0 > 0\text{, що } \\ 0 \leqslant f(n) \leqslant cg(n) \text{ для всіх } n \geqslant n_0\}$$

- Наприклад, $2n = o(n^2)$, але $2n^2 \neq o(n^2)$.
- $f(n) = o(g(n))$ тоді і тільки тоді, коли $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Асимптотичні позначення: o, ω

- Для заданої невід'ємної функції $g(n)$ позначимо через $\omega(g(n))$ таку множину функцій:

$$\omega(g(n)) = \{f(n)\colon \text{для будь-якого } c > 0 \text{ існує } n_0 > 0\text{, що } 0 \leqslant cg(n) \leqslant f(n) \text{ для всіх } n \geqslant n_0\}$$

- $f(n) \in \omega(g(n))$ тоді і тільки тоді, коли $g(n) \in o(f(n))$.
- $f(n) = \omega(g(n))$ тоді і тільки тоді, коли $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Асимптотичні позначення: огляд

Позначення	Пояснення	Аналогія
$f(n) = O(g(n))$	f зростає щонайбільше так, як g	$f \leqslant g$
$f(n) = \Omega(g(n))$	f зростає принаймні так, як g	$f \geqslant g$
$f(n) = \Theta(g(n))$	f зростає так само швидко, як і g	$f = g$
$f(n) = o(g(n))$	f зростає повільніше, ніж g	$f < g$
$f(n) = \omega(g(n))$	f зростає швидше, ніж g	$f > g$

- Рівності можна читати тільки зліва-направо: $f(n) = O(g(n))$ по суті означає, що $f(n) \in O(g(n))$
  (тобто $n = O(n^2)$ — правильно, але $O(n^2) = n$ — ні)

Стандартні runtimes

$\Theta(1)$	константний час
$\Theta(\log n)$	логарифмічний час
$\Theta(n)$	лінійний час
$\Theta(n^2)$	квадратичний час
$\Theta(n^3)$	кубічний час
n^k, де $k = \Theta(1)$	поліноміальний час
2^n	експоненційний час

- Кожний многочлен від $\log n$ зростає строго повільніше, ніж будь-який многочлен від n. Наприклад: $(\log n)^{100} = o(n^{0,01})$.
- Кожний многочлен від n зростає строго повільніше, ніж кожна показникова функція. Наприклад: $n^{100} = o(2^{n^{0,01}})$.

Приклади

Приклади використання асимптотичних позначень:

- $2n + 1 = O(n)$
- $42 = O(n)$ (але не $\Theta(n)$)
- $n - 9 = \Omega(n)$
- $n^2 + n = \Omega(n)$ (але ні $O(n)$, ні $\Theta(n)$)
- $n^3 = o(n^4) = o(2^n)$
- $\sqrt{n} = \omega(\log n)$

Приклади

- Час виконання алгоритму InsertionSort — це $\Omega(n)$ і $O(n^2)$ (зростає принаймні так, як n і щонайбільше так, як n^2)
- Справді, найкращий runtime — це $\Theta(n)$; найгірший runtime — це $\Theta(n^2)$

Правила для простішої оцінки runtime

Для двох невід'ємних функцій $f(n)$ і $g(n)$:

- Доданки, які ростуть повільніше можна відкинути:

$$f(n) + g(n) = \Theta(\max(f(n), g(n)))$$

- Асимптотичний час можна множити:

$$\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$$

Правила для простішої оцінки runtime

```
1 foo
2 foo
3 for i = 1 to n:
4     foo
5     foo
6     foo
```

- Час на перші два рядки $\Theta(1)$
- Одна ітерація циклу потребує $\Theta(1)$ часу
- Цикл **for** виконується $\Theta(n)$ разів
- Загальний час:

$$\Theta(1) + \Theta(n) \cdot \Theta(1) = \Theta(n)$$

Підсумок

- Ми можемо оцінювати час виконання для найкращого випадку, найгіршого випадку, або ж середнього випадку. Зазвичай фокусуються на часі виконання для найгіршого випадку.
- Найважливішим аспектом ефективності є масштабованість: як час виконання збільшується з розміром входових даних n.
- Асимптотичні позначення (O, Ω, Θ, o, ω) приховують константи і доданки менших порядків, вказуючи асимптотичний час виконання.
- Асимптотичні позначення стосуються наборів функцій, але для зручності їх записують у вигляді рівностей, які читаються зліва направо.