Алгоритми та складність

І семестр

Лекція 1

Структура курсу в першому семестрі

Лекції - щотижня
Лабораторні заняття – раз на 2 тижні

- На лекціях – 2 підсумкових модулі.
- В кінці – залік (сумуються отримані бали).
- Бали: 50 (лекції) + 50 (практика).
- Спроба отримати залік при недостатній сумі балів (<60): 45 балів (з них $\geq$15 на лекціях та $\geq$20 на практиці), інакше одразу на перескладання.

Папка з прочитаними лекціями
https://drive.google.com/drive/folders/11e1zLVa_22FU9scCfevp3YZPAAuh-wP2

Основні джерела

1. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы: построение и анализ. 3-е издание. – М.: ИД "Вильямс", 2013.

Introduction to Algorithms, "CLRS" (Cormen, Leiserson, Rivest, Stein).

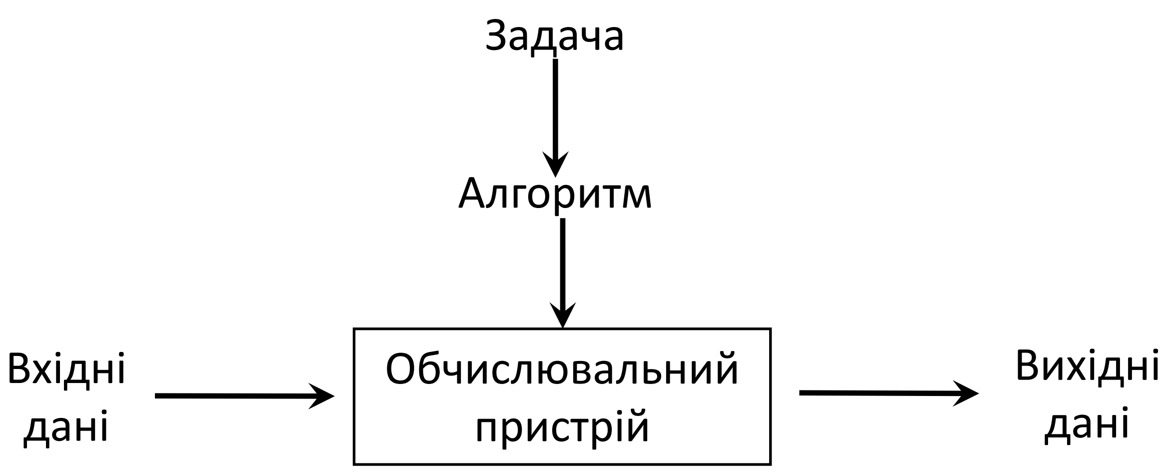
2. А. Левитин. Алгоритмы: введение в разработку и анализ. – М.: ИД "Вильямс", 2006.

Introduction to the Design and Analysis of Algorithms, Anany Levitin.

- **Алгоритм** – послідовність чітко визначених інструкцій, призначених для вирішення деякої задачі.

Іншими словами, це послідовність команд, що дозволяє з коректних вхідних даних отримати вихідний результат за обмежений проміжок часу.

Така сукупність правил задає обчислювальний (алгоритмічний) процес.



Алгоритм:

- Кожен крок алгоритму має бути чітко й однозначно визначений. Ця вимога є обов'язковою і не повинна порушуватися за жодних обставин.
- Повинні бути точно вказані діапазони допустимих значень вхідних даних, що обробляються за допомогою алгоритму; коректним вважається алгоритм, який видає правильний результат для абсолютно всіх вхідних даних з вказаного допустимого діапазону, включно з можливими граничними випадками.
- Один і той же алгоритм можна представити кількома різними способами.
- Для вирішення однієї і тієї ж задачі може існувати декілька різних алгоритмів.
- В основу алгоритмів для вирішення однієї і тієї ж задачі можуть бути покладені абсолютно різні принципи, що може істотно вплинути на швидкість вирішення цієї задачі.

Приклад:
обчислення НСД двох невід’ємних цілих чисел

<u>Алгоритм Евкліда</u>

$$gcd(m, n) = gcd(n, m \;\; mod \; n)$$

Тоді $gcd(m, 0) = m$ та значення m шукане.

- Крок 1. Якщо $n = 0$, повернути m як відповідь і закінчити роботу; інакше перейти до кроку 2.
- Крок 2. Знайти остачу від ділення m на n і присвоїти її змінній r.
- Крок 3. Присвоїти значення n змінній m, а значення r - змінній n. Перейти до кроку 1.

Чому робота алгоритму завершиться?

Приклад:
обчислення НСД двох невід’ємних цілих чисел

Метод послідовного перебору

- Крок 1. Присвоїти значення *min{m,n}* змінній *t*.
- Крок 2. Поділити *m* на *t*. Якщо остача дорівнює нулю, перейти до кроку 3; інакше перейти до кроку 4.
- Крок 3. Розділити *n* на *t*. Якщо остача дорівнює нулю, повернути *t* як відповідь і завершити роботу; інакше перейти до кроку 4.
- Крок 4. Зменшити *t* на одиницю. Перейти до кроку 2.

Чи це алгоритм? Чи він коректно працюватиме?

Приклад:
обчислення НСД двох невід’ємних цілих чисел

Метод «як у школі»

- Крок 1. Розкласти на прості множники *m*.
- Крок 2. Розкласти на прості множники *n*.
- Крок 3. Серед знайдених на кроці 1 і 2 множників виділити спільні дільники *m* та *n*. (Якщо *p* є спільним дільником *m* та *n* і зустрічається в їх розкладі на прості множники відповідно p_m та p_n раз, то треба його взяти $min\{p_m, p_n\}$ разів.)
- Крок 4. Обчислити добуток всіх виділених спільних дільників і повернути його як результат.

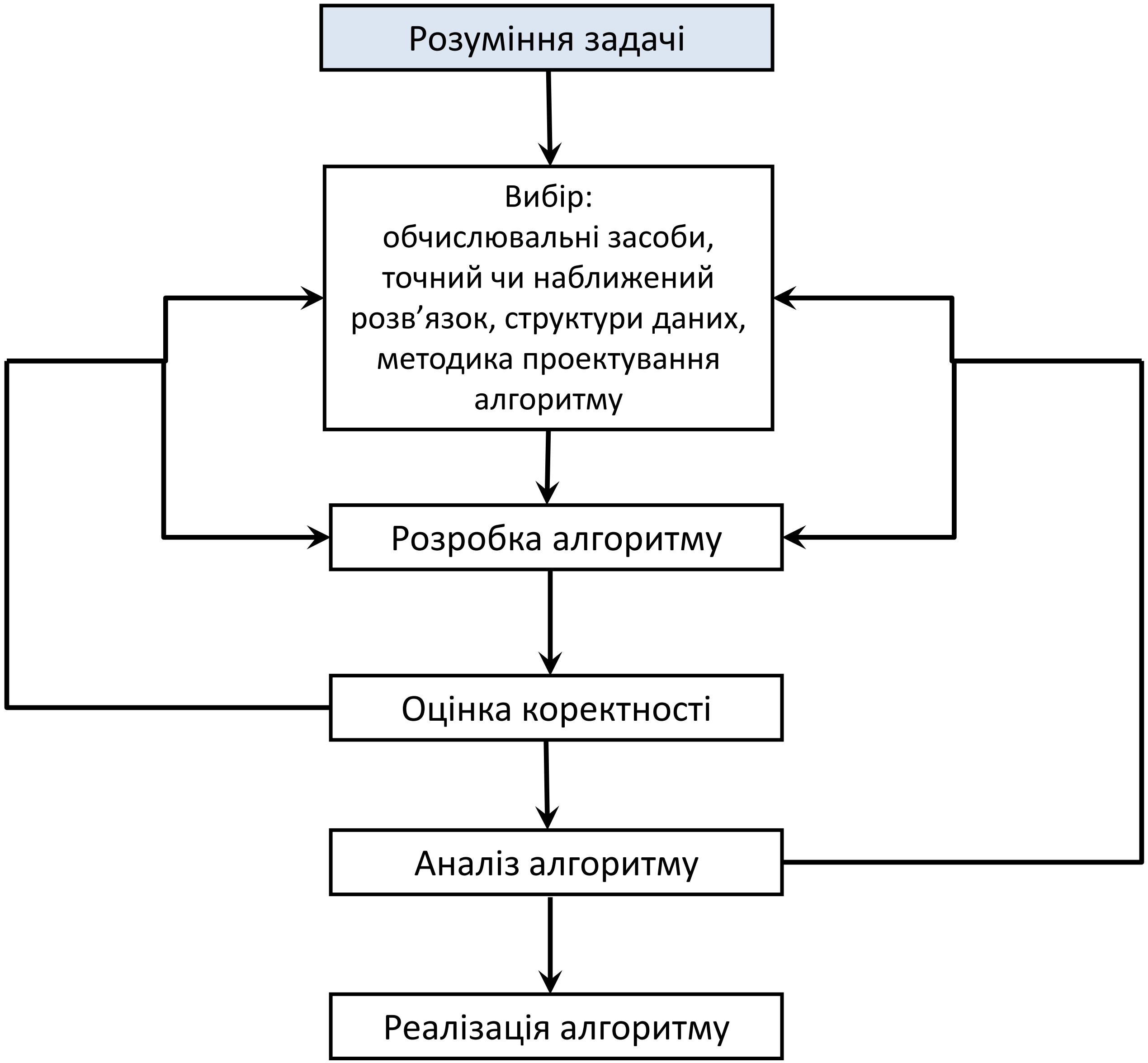
Чи це алгоритм? Чи він коректно працюватиме?

Класифікація алгоритмів

Серед багатьох підходів до класифікації алгоритмів, можна виділити зокрема такі:

- за типом задач, які вони розв’язують (сортування, пошук, обробка рядків, задачі теорії графів, комбінаторні, геометричні, чисельні задачі…);
- за методами проектування (груба сила, декомпозиція, зменшення розміру задачі, жадібні, динамічне програмування…).

Процес проектування і аналізу алгоритму

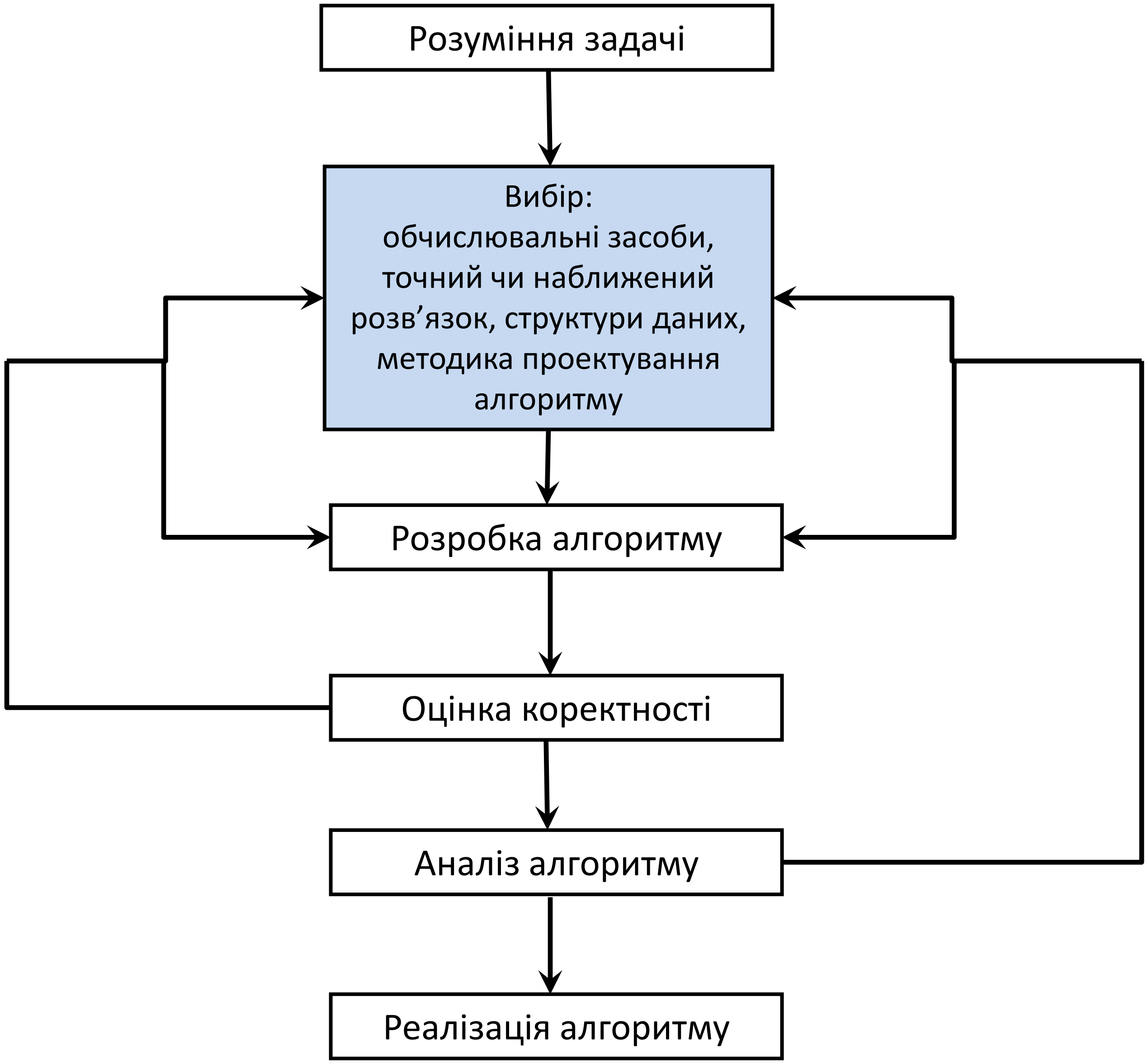


Розуміння задачі

- Уважне вникнення в суть задачі.
- Чітке формулювання умов роботи алгоритму.
- Перевірка існування вже готових методів розв’язання схожих задач та їх оцінка.

- І також уявлення про те, чи можна розв’язати цю задачу взагалі ☺

Процес проектування і аналізу алгоритму



Визначення можливостей обчислювальних засобів

- Оцінка архітектури системи, необхідності задіювання паралелізму.
- Додаткові умови на швидкодію алгоритму, доступну пам’ять, обробка дуже великих об’ємів інформації…

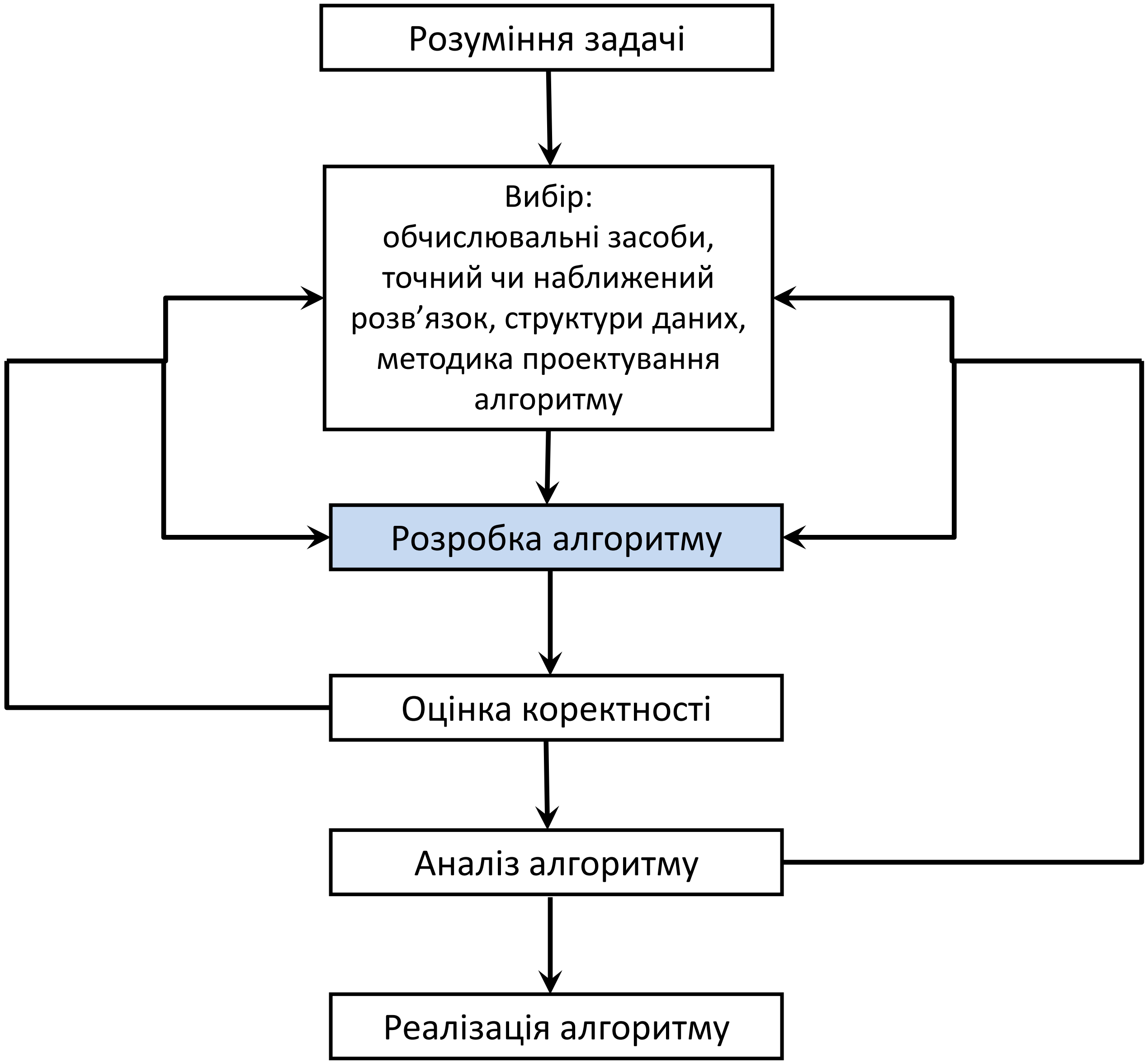
Точний чи наближений метод розв’язання

- Деякі задачі не мають точного розв’язку.
- Точний метод може бути недопустимо повільним.
- Потрібна лише більш груба прикидка розв’язку.

Вибір структур даних та методу проектування

- Навіть якщо не накладено умов на формат представлення вхідних даних, вибір певного алгоритму чи підходу до його розробки може їх зумовити.
- Багато алгоритмів працюють над конкретними структурами даних.
- Деякі методи проектування тісно пов’язані зі структуризацією та реструктуризацією даних, які обробляються.

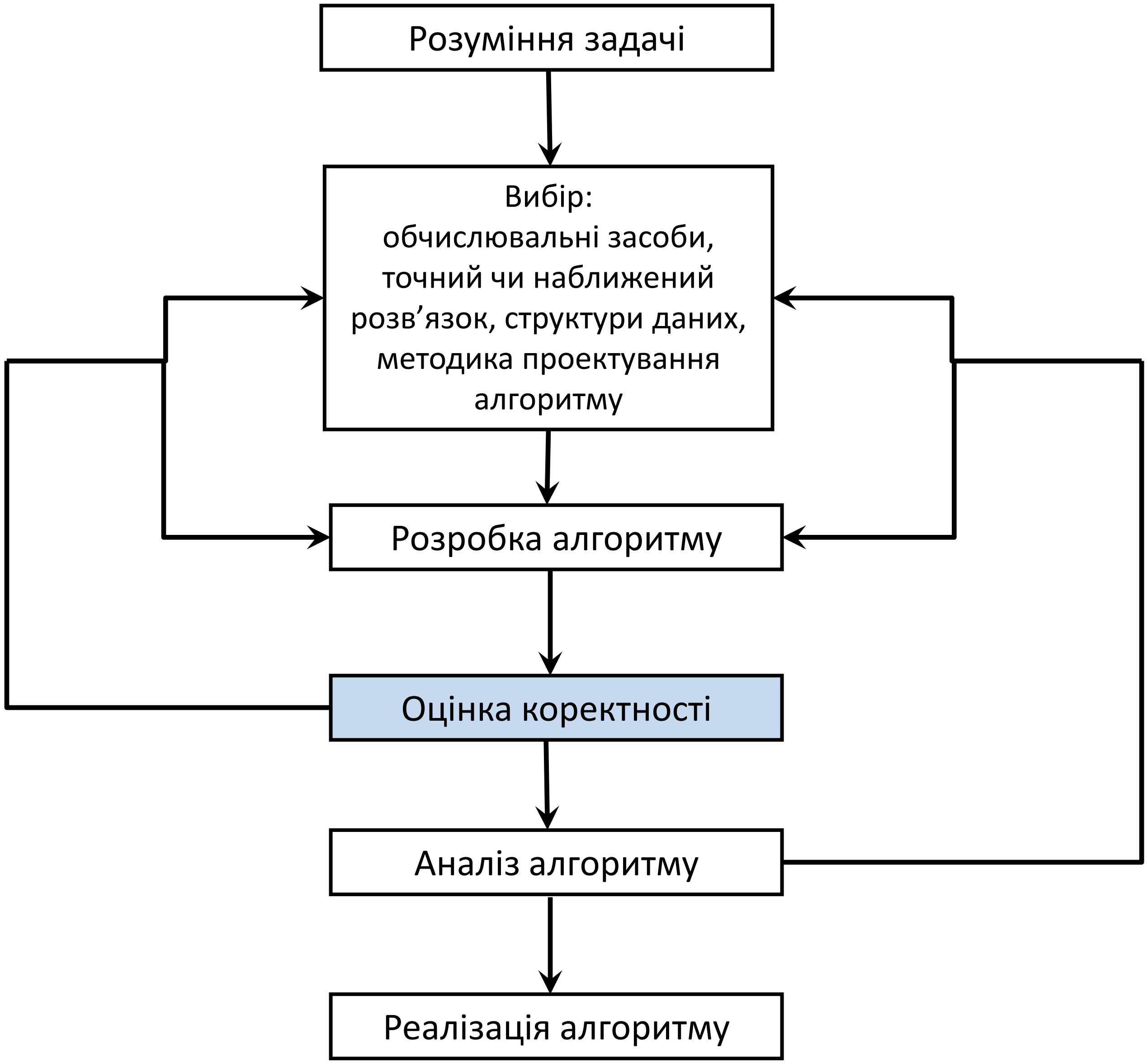
Процес проектування і аналізу алгоритму



Способи представлення алгоритмів

- Найпопулярніша форма – псевдокод.
- Потрібно також вміти описати алгоритм словесно.
- Блок-схеми для великих і складних алгоритмів використовувати незручно.

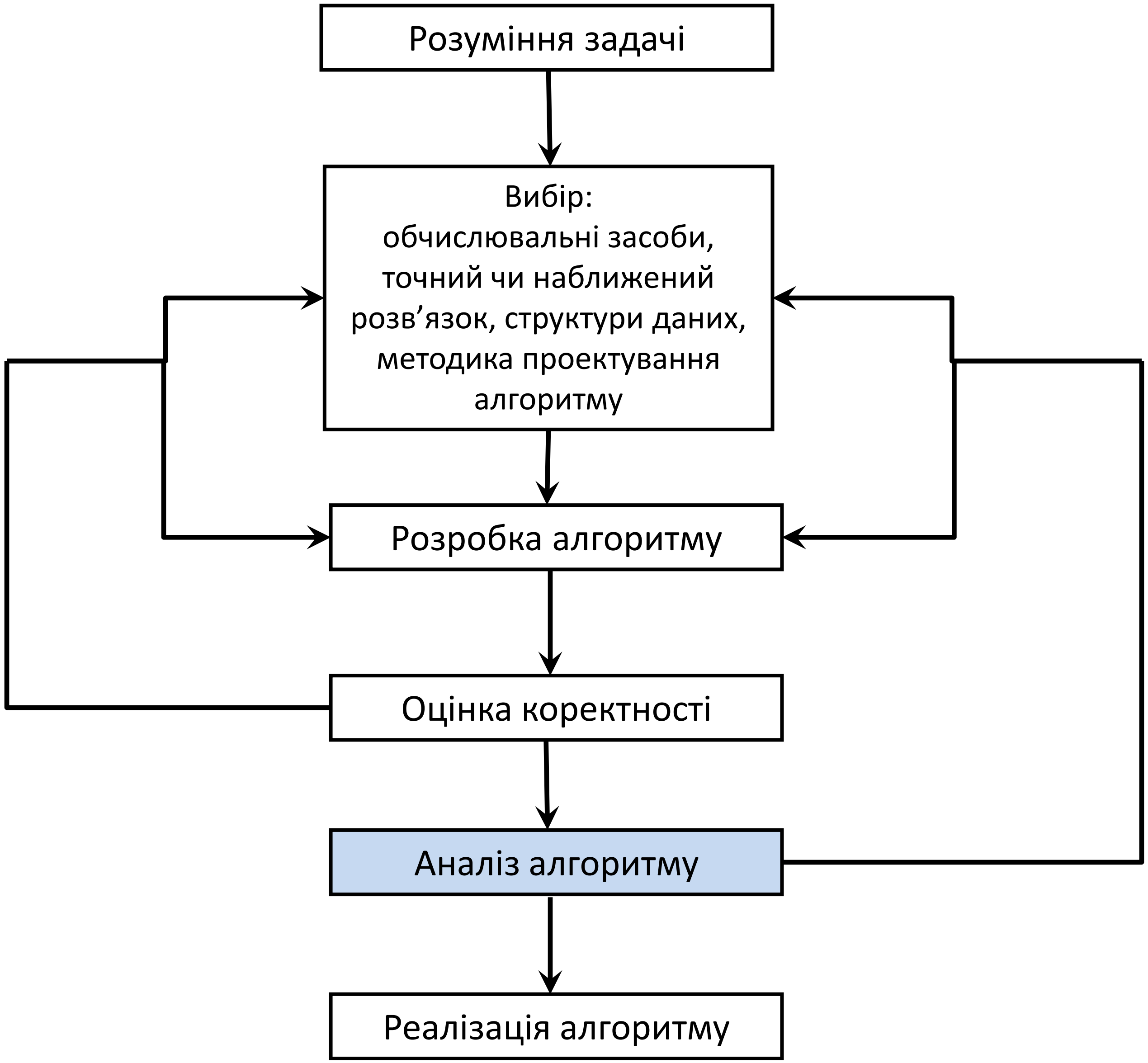
Процес проектування і аналізу алгоритму



Оцінка коректності алгоритму

- Показати, що за скінченний час алгоритм видає потрібний результат для всіх коректних вхідних даних.
- Математична індукція – універсальний метод доведення.
- Для доведення *некоректності* достатньо одного набору вхідних даних, на якому отримується помилковий результат.
- При оцінці коректності наближених алгоритмів показують, що отримана похибка не виходить за встановлені межі.

Процес проектування і аналізу алгоритму



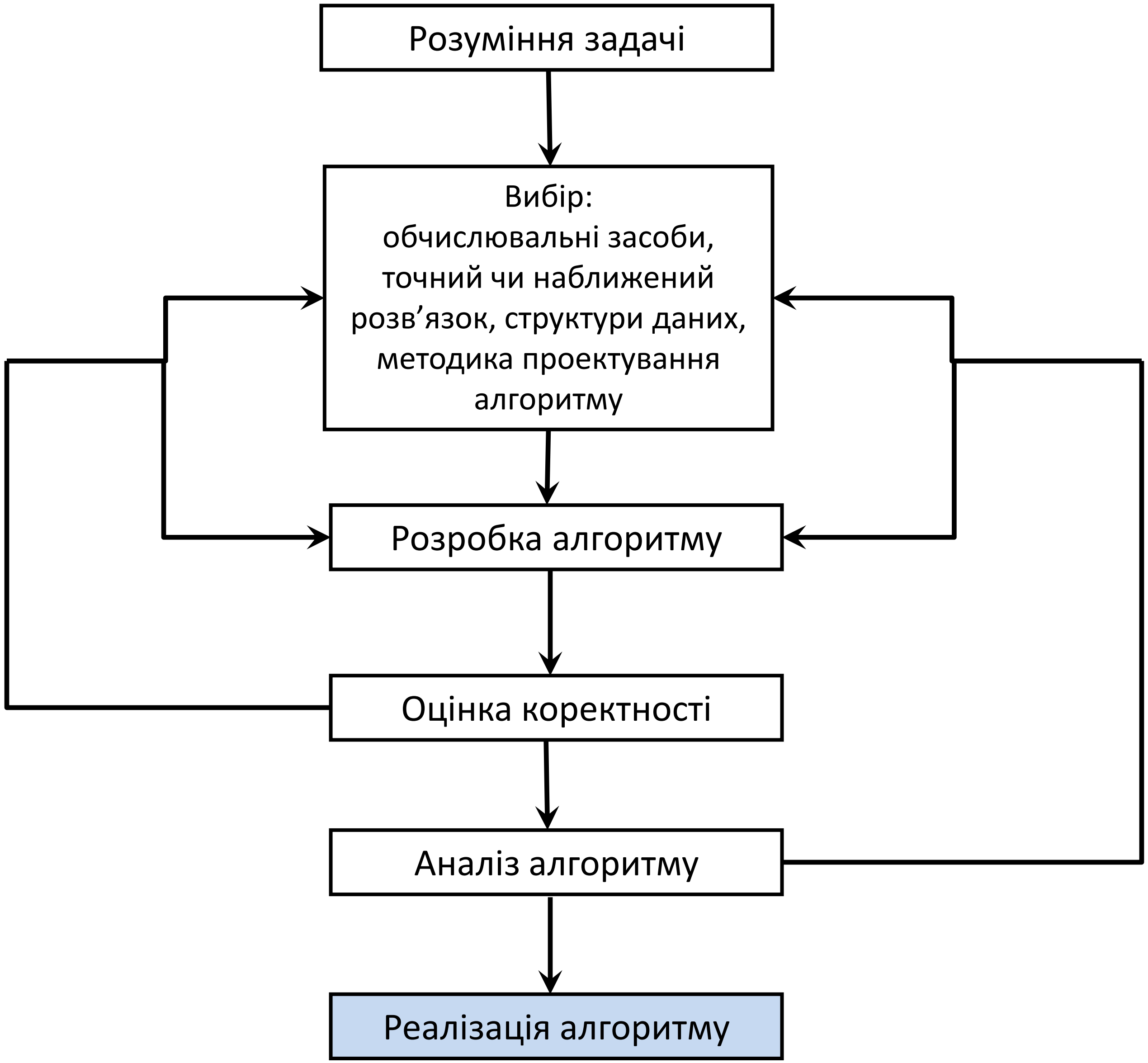
Аналіз алгоритму

- Оцінка ефективності – часової та/або просторової.
- Оцінка оптимальності: наскільки ефективність алгоритму відповідає складності самої задачі, яка розв'язується. (Однак не для всіх, навіть широковідомих, задач відомі оптимальні оцінки.)
- Оцінка простоти. Строгих критеріїв немає, але очевидно, що прості алгоритмі легше зрозуміти і закодувати. Часто такі алгоритми є більш ефективними, хоча не завжди.

Аналіз алгоритму

- Оцінка універсальності. В одних випадках зручніше розробити алгоритм для вирішення загальнішої задачі (взаємна простота vs НСД), а іноді ефективніше розв’язати даний частковий випадок (пошук медіани vs сортування). Діапазон допустимих вхідних значень має бути адекватним поставленій задачі.

Процес проектування і аналізу алгоритму



Кодування алгоритму

- Ефективна реалізація включаючи оптимізацію коду.
- Перевірка роботи через тестування.
- Не забувати про додаткову перевірку отримуваних вхідних даних на їх допустимість.

Базові структури даних

Масив

Зв’язаний список

матриці

списки

рядки

дерева

хеш-таблиці

купи

інші складні структури даних

Абстрактні типи даних

(множини абстрактних об’єктів, що представляють елементи даних і набір операцій, які можуть виконуватися над елементами цієї множини)

- Список
- Дерево
- Граф
- Стек
- Черга
- Черга з пріоритетами
- Дек
- Множина
- Асоціативний масив (словник)
- Мультимножина

Аналіз алгоритмів

- *Часова ефективність (time efficiency)*

  Індикатор швидкості виконання алгоритму

- *Просторова ефективність (space efficiency)*

  Кількість додаткової оперативної пам'яті, необхідної для виконання алгоритму

Головним чином зосереджуються на питаннях часової ефективності.

Надалі в якості моделі обчислень будемо використовувати однопроцесорну машину з довільним доступом (random-access machine, RAM); вона допускає лише послідовне виконання операцій.

Оцінка розміру вхідних даних

Час виконання більшості алгоритмів напряму залежить від розміру вхідних даних – чим більше вхідне дане, тим довше виконуються обчислення.

Можна рахувати

- кількість елементів на вході (сортування, пошук, многочлени, …)
- кількість бітів, необхідних для представлення входу (великі числа)
- параметрів може бути декілька (кількість вершин і ребер графа)

Оцінка часу роботи

Чому не підходять часові одиниці виміру?

Оцінка часу роботи

Чому не підходять часові одиниці виміру?

Тоді отриманий результат залежатиме від

- швидкодії конкретного комп'ютера,
- точності реалізації алгоритму у вигляді програми,
- типу компілятора, використаного для генерації машинного коду,
- точності хронометрування реального часу виконання програми.

Оскільки оцінюємо ефективність алгоритма, а не його програмної реалізації, для вимірювання слід брати одиниці, що не залежать від наведених факторів.

Оцінка часу роботи

Виберемо набір базових операцій, кожна з яких матиме певну ціну:

- присвоєння
- порівняння
- додавання
- множення
- ділення
- взяття елемента за індексом

і порахуємо час виконання алгоритму, обчисливши суму ваг кожної операції, помножених на її частоту.

Але насправді достатньо враховувати лише час виконання тих операцій, які вносять найбільший вклад в роботу алгоритма.

Що краще:
швидкий алгоритм на повільній машині чи повільний алгоритм на швидкій машині?

Нехай перший алгоритм потребує $2n^2$ команд і запущений на машині A, яка виконує мільярд інструкцій в секунду, а другий – $50n\lg n$ команд і запущений на машині B, що виконує 10 мільйонів інструкцій в секунду (тут n – розмір входу).

При $n = 10^6$ маємо:

$$T_1(n) = \frac{2 \cdot (10^6)^2 \text{ команд}}{10^9 \text{ команд/с}} = 2000 \text{ с}$$

$$T_2(n) = \frac{50 \cdot 10^6 \cdot lg10^6 \text{ команд}}{10^7 \text{ команд/с}} \approx 100 \text{ с}$$

При більших значеннях n перевага швидкого алгоритму надалі зростатиме.

Завдання: підрахуйте, скільки часу займе робота обох алгоритмів при $n = 10^7$.

Нехай c_{op} – час виконання основної операції алгоритму на конкретному комп'ютері,

$C(n)$ – кількість разів її виконання при роботі даного алгоритму.

Тоді час виконання програмної реалізації алгоритму на цьому комп'ютері приблизно дорівнюватиме

$$T(n) \approx c_{op} C(n)$$

Формула дозволить оцінити з задовільною точністю час виконання алгоритму для не нескінченно великих та не нескінченно малих n.

Нехай $C(n) = n(n-1)/2$.

Наскільки довше працюватиме програма при подвоєнні вхідних даних?

Нехай $C(n) = n(n-1)/2$. Наскільки довше працюватиме програма при подвоєнні вхідних даних?

Для достатньо великих n буде справедлива формула

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

Тому

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4$$

Як видно, при цьому реальне значення c_{op} можна не знати, воно скорочується, як і константа $\frac{1}{2}$.

З цих причин при аналізі ефективності при достатньо великих вхідних даних не враховують константні множники, а шукають оцінку порядку зростання кількості основних операцій з точністю до константного множника.

Ефективність алгоритму у різних випадках

Час виконання багатьох алгоритмів залежить не лише від розміру вхідних даних, а й від їх особливостей в конкретному випадку.

<u>Задача послідовного пошуку</u>

Алгоритм виконує пошук заданого елемента (ключа пошуку К) в списку, що складається з n елементів, шляхом послідовного порівняння ключа К з кожним з елементів списку. Робота алгоритму завершується або коли заданий ключ знайдений, або коли весь список вичерпаний.

АЛГОРИТМ *SequentialSearch* (A[0 .. $n-1$], K)
// **Вхідні дані:** масив чисел A[0 .. $n-1$] та ключ пошуку K
// **Вихідні дані:** повертається індекс першого елемента
// масиву A, що дорівнює K, або -1,
// якщо заданий елемент не знайдено
i <= 0
while $i < n$ **and** A[i] $\neq K$ **do**
  i <= i + 1
if $i < n$
  return i
else
  return -1

Ефективність алгоритму у різних випадках

Час роботи наведеного алгоритму може варіювати в залежності від того, де розташований шуканий елемент.

- Ідеальний випадок – він перший і знаходиться при першій перевірці.
- Найгірший випадок – шуканий елемент останній в списку або взагалі відсутній; обидва варіанти передбачають перегляд всіх елементів.

Ефективність алгоритму у різних випадках

- *Ефективність алгоритму в найкращому випадку (best-case efficiency)*: ефективність для найкращих вхідних даних. Найкращі дані – такі, при обробці яких алгоритм виконує найменше операцій. В нашому прикладі $C_{best}(n) = 1$.
- *Ефективність алгоритму в найгіршому випадку (worst-case efficiency)*: ефективність для найгірших вхідних даних. Найгірші дані – такі, при обробці яких алгоритм виконує максимальну кількість операцій. В нашому прикладі $C_{worst}(n) = n$.

Ефективність алгоритму у різних випадках

- Аналіз ефективності в найгіршому випадку дає верхню оцінку швидкодії алгоритму – один з найважливіших показників ефективності.

- Аналіз ефективності в найкращому випадку не є настільки важливим, але для деяких алгоритмів хороші показники швидкодії в найкращому випадку будуть зберігатися і для випадків, близьких до ідеальних.

Ефективність алгоритму у різних випадках

На основі аналізу алгоритму для найкращого та найгіршого випадків неможливо зробити висновок про його поведінку на типових або довільних даних.

- *Ефективність алгоритму в середньому випадку (average-case efficiency)*: ефективність для довільних вхідних даних.

Така оцінка також дуже важлива, оскільки алгоритм може поєднувати хорошу середню ефективність з не найкращими показниками для найгіршого випадку.

Аналіз ефективності в середньому випадку

- Перший крок – визначення різних груп, на які слід розбити можливі набори вхідних даних.
- На другому кроці визначається ймовірність, з якою набір вхідних даних належить кожній групі.
- На третьому кроці підраховується час роботи алгоритму на даних з кожної групи. Час роботи алгоритму на всіх вхідних даних однієї групи має бути однаковим, в іншому випадку групу слід ще раз поділити.

Аналіз ефективності в середньому випадку

Середній час роботи обчислюється за формулою

$$C_{avg}(n) = \sum_{i=1}^{m} p_i t_i, \text{ де}$$

n – розмір вхідних даних,

m – кількість груп,

p_i – ймовірність, що вхідні дані належать до групи i,

t_i – час обробки алгоритмом групи i.

Якщо ймовірність потрапляння вхідних даних до кожної з груп однакова, можна використати формулу

$$C_{avg(n)} = \frac{1}{m}\sum_{i=1}^{m} t_i \,.$$

Аналіз ефективності в середньому випадку

Підрахуємо для нашого прикладу.

Зробимо два стандартних припущення:

1. ймовірність успішного пошуку дорівнює p, де $0 \leq p \leq 1$;
2. ймовірність першого збігу ключа з i-м елементом списку однакова для довільного p.

Якщо збіг з ключем знайдено, ймовірність того, що це сталося саме на i-му елементі списку, дорівнює p/n для будь-якого i. При цьому кількість операцій порівняння буде дорівнювати i. Якщо збігу з ключем не знайдено, кількість операцій порівняння все одно буде n, а ймовірність цієї події дорівнює $(1 - p)$.

Аналіз ефективності в середньому випадку

$$C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n(1-p) =$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p) =$$

$$= \frac{p}{n} \cdot \frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p)$$

Звідси можна побачити, що при $p = 1$ (елемент наявний у списку) середньою кількістю операцій порівняння буде $(n+1)/2$, тобто перевірятиметься в середньому половина елементів.

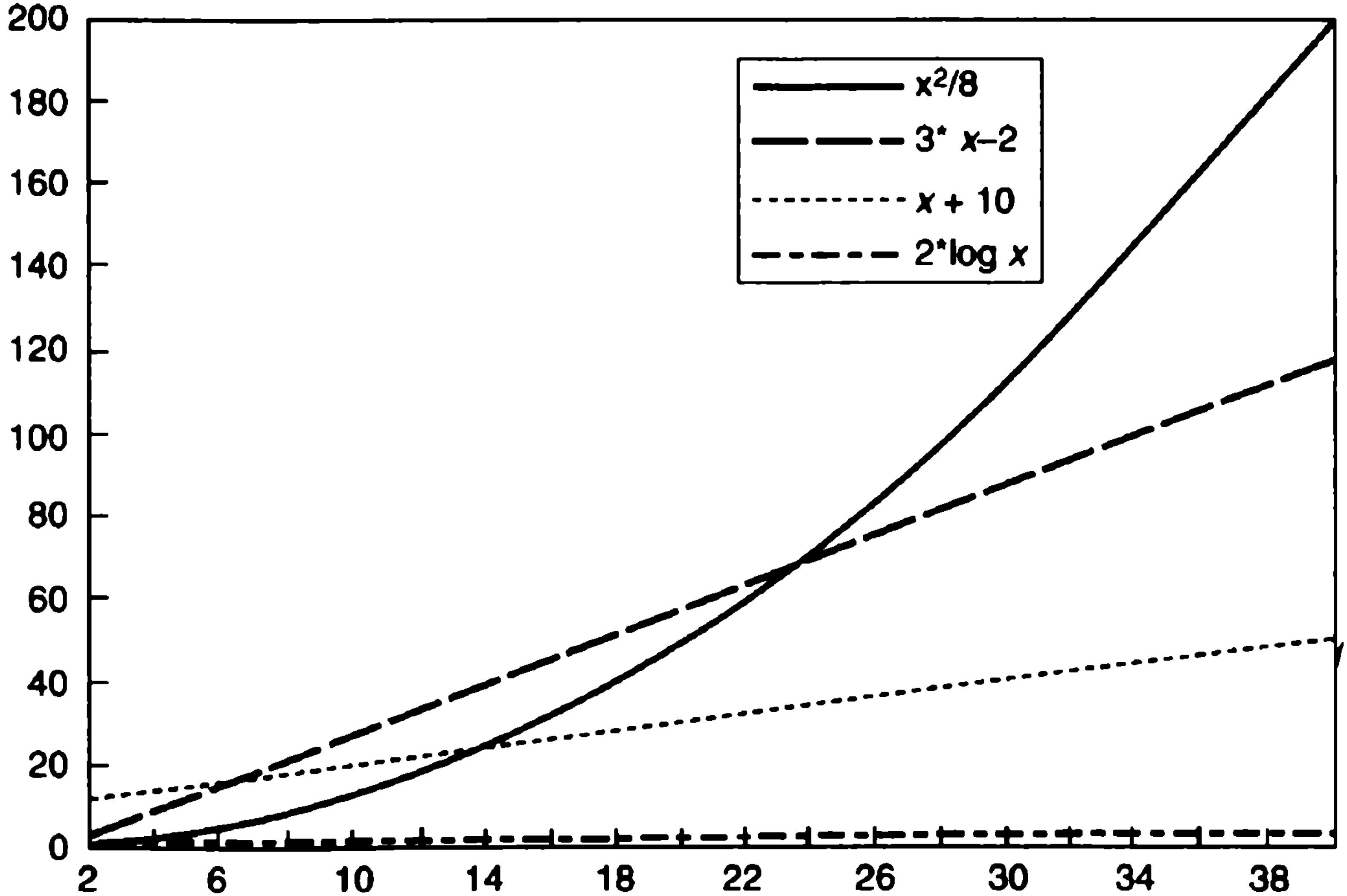
При $p = 0$ (шуканий елемент відсутній у списку) середня кількість операцій буде все одно n, бо перевірятимуться всі елементи.

Порядок зростання (основи)

- Точне знання кількості операцій, виконаних алгоритмом, не грає істотної ролі в аналізі алгоритмів.
- Набагато важливішим виявляється швидкість росту цього числа при зростанні обсягу вхідних даних.
- При малих за обсягом вхідних даних неможливо отримати уявлення про різницю в ефективності алгоритмів.
- Однак при великих даних ефективність виходить на перший план.
- Нас цікавитиме лише загальний характер поведінки алгоритмів, а не подробиці цієї поведінки.

Порядок зростання (основи)

Розглянемо графіки чотирьох різних функцій:



Порядок зростання (основи)

Розглянемо наближені значення важливих для аналізу алгоритмів функцій:

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

$$log_a n = log_a b \cdot log_b n$$

За допомогою алгоритмів, в яких кількість виконуваних операцій зростає за експоненціальним законом, можна вирішити лише задачі дуже малих розмірів.

- Можна також розглядати реакцію функцій на (наприклад) подвоєння параметра *n*.

Запитання і завдання

- Наведіть приклад задачі, відмінної від пошуку НСД, для розв'язку якої існує декілька алгоритмів. Який з цих алгоритмів простіший? Який ефективніший?
- Проаналізуйте наведений нижче алгоритм пошуку мінімальної різниці між двома елементами числового масиву. Чи можна його удосконалити? Або покращити реалізацію? Як варіант можна навести свій алгоритм.

АЛГОРИТМ *MinDistance (* A[0 .. $n-1$])
// **Вхідні дані:** масив чисел A[0 .. $n-1$]
// **Вихідні дані:** мінімальна різниця між двома елементами масиву A
dmin <= ∞
for i <= 0 **to** $n-1$ **do**
  for j <= 0 **to** $n-1$ **do**
    if $i \neq j$ **and** $|A[i] - A[j]| < dmin$
      dmin <= $|A[i] - A[j]|$
return *dmin*

Запитання і завдання

- За якого мінімального значення n алгоритм, що працює за час $100n^2$, буде виконуватись швидше за алгоритм, що працює за час $2n$? (Обидва виконуються на одній машині)
- В таблиці рядки відповідають різним функціям $f(n)$, а стовпці – значенням часу t. Обчисліть максимальні значення n, для яких задача може бути розв'язана за час t, якщо алгоритм вирішує задачу за $f(n)$ мікросекунд.

	1 сек	1 хв	1 год	1 день	1 місяць	1 рік	100 років
$\log n$							
$\sqrt{n}$							
n							
$n \log n$							
n^2							
n^3							
2^n							
$n!$							

Запитання і завдання

- Розгляньте варіант алгоритму пошуку методом послідовного перебору, який повертатиме кількість елементів у списку, що збігаються з заданим ключем пошуку. Чи буде його ефективність відрізнятися від ефективності класичного алгоритму пошуку методом послідовного перебору?
- Розгляньте алгоритм додавання двох матриць розміру n x n відповідно до визначення операції додавання матриць. Назвіть його основну операцію. Визначте залежність кількості основних операцій як функцію від порядку матриці n. Знайдіть залежність кількості основних операцій як функцію від числа елементів в матрицях.
- Виконайте вправу аналогічну попередній для алгоритму множення двох матриць розміру n x n відповідно до визначення операції.