

## Л2. Бінарний пошук

### 1. Бінарний пошук

Припустимо, що ви шукаєте прізвище людини в телефонній книзі. Вона починається з літери "К". Звичайно, ви можете почати з самого початку і гортати сторінки, поки не потрапите до букви "К". Але, швидше за все, для прискорення пошуку краще розкрити книгу посередині: адже буква «К» повинна бути десь ближче до середини телефонної книги.

Або припустимо, що ви шукаєте слово в словнику, і він починається з букви "О". І знову краще почати з середини.

Тепер припустимо, що ви вводите свої дані, коли ви входите в Facebook. Вам потрібно перевірити, чи є у вас обліковий запис на сайті. Для цього потрібно знайти ім'я користувача в базі даних. Скажімо, ви вибрали ім'я користувача «karlking». Facebook може почати з букви А і перевірити все, але краще почати з середини.

У нас є типове завдання пошуку. І в усіх цих випадках можна використовувати один алгоритм для вирішення проблеми: **бінарний пошук**.

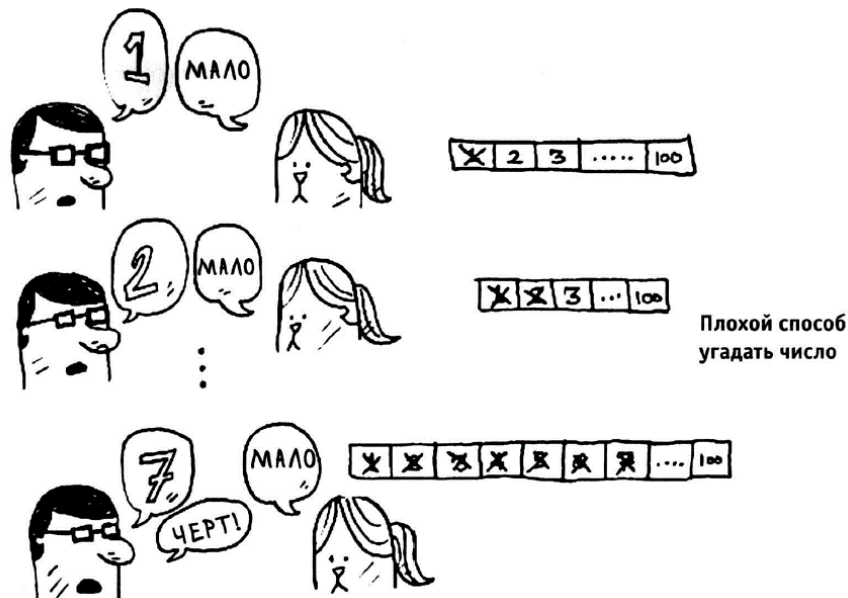
Бінарний пошук — це алгоритм, який отримує відсортований список об'єктів. Якщо потрібний елемент міститься у списку, двійковий пошук повертає позицію, в якій його було знайдено. В іншому випадку двійковий пошук повертає значення NULL .

Давайте розглянемо приклад того, як працює двійковий пошук. Давайте розглянемо просту гру: я загадав номер від 1 до 100.



Ви повинні вгадати мій номер, використовуючи мінімальну кількість спроб. Кожного разу дається одна трьох відповідей: "мало", "багато" або "здогадалися".

Припустимо, ви починаєте перебирати усі варіанти: 1, 2, 3, 4....



Це приклад **простого** пошуку. З кожним припущенням виключається лише одне число. Якщо я загадав, номер 99, це займе 99 спроб дістатися до нього!

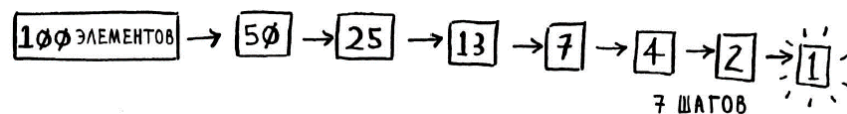
Існує ще один, більш ефективний спосіб. Почнемо з 50.



Занадто мало ... Але ви тільки що усунули половину чисел! Тепер ви знаєте, що всі цифри 1-50 менше, ніж загадка. Наступна спроба: 75.

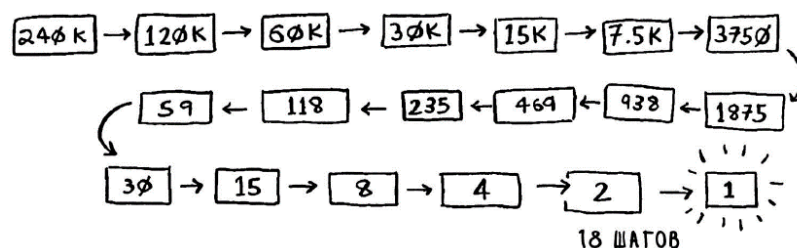
На цей раз більше ... Але ви виключили половину решти номерів знову! З двійковим пошуком ви робите число в середині діапазону кожного разу і усуваєте половину решти чисел. Далі буде номер 63 (від середини між 50 і 75).

Ось як працює двійковий пошук. Давайте спробуємо визначити точніше, скільки чисел буде виключено кожного разу.



Припустимо, що ви шукаєте слово в словнику з 240000 слів. Скільки спроб вам знадобиться в гіршому випадку (для простого та бінарного пошуку)?

Якщо ви шукаєте простим способом, вам може знадобитися 240000 спроб, якщо слово, яке ви шукаєте, знаходиться в останній точці книги. З кожним кроком бінарного пошуку кількість слів вдвічі зменшується, поки не залишилося лише одне слово.



Таким чином, двійковий пошук потребує 18 кроків. Загалом й для списку  $n$  елементів, двійковий пошук буде виконано  $\log_2 n$  кроків, а простий пошук буде виконано за  $n$  кроків.

## 2. Реалізація алгоритму бінарного пошуку.

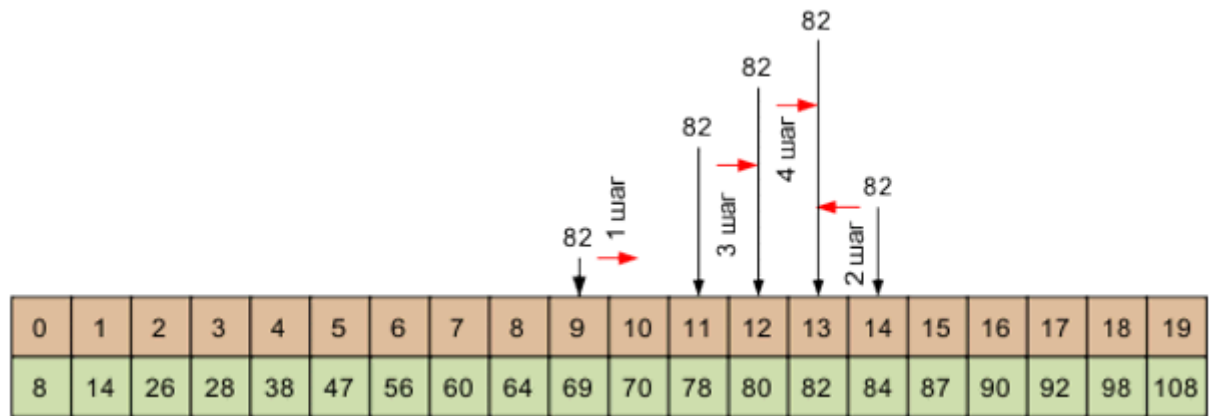
Бінарний пошук проводиться в упорядкованому масиві. Алгоритм може бути визначений в рекурсивній і нерекурсивній формах.

На кожному кроці здійснюється пошук середини відрізка за формулою

$$mid = (left + right) / 2$$

Якщо елемент, що шукають, дорівнює елементу з індексом *mid*, пошук завершується.

У разі якщо елемент, що шукають, менше елемента з індексом *mid*, на місце *mid* переміщається права межа розглянутого відрізка, в іншому випадку - ліва межа.



Підготовка. Перед початком пошуку встановлюємо ліву і праву межі масиву:

$$left = 0, right = 19$$

**Крок 1.** Шукаємо індекс середини масиву (округляємо в меншу сторону):

$$mid = (19 + 0) / 2 = 9$$

Порівнюємо значення за цим індексом з шуканим:

$$69 < 82$$

Зрушуємо ліву межу:

$$left = mid = 9$$

**Крок 2.** Шукаємо індекс середини масиву (округляємо в меншу сторону):

$$mid = (9 + 19) / 2 = 14$$

Порівнюємо значення за цим індексом з шуканим:

$$84 > 82$$

Зрушуємо праву межу:

$$right = mid = 14$$

**Крок 3.** Шукаємо індекс середини масиву (округляємо в меншу сторону):

$$mid = (9 + 14) / 2 = 11$$

Порівнюємо значення за цим індексом з шуканим:

$$78 < 82$$

Зрушуємо ліву межу:

$$left = mid = 11$$

**Крок 4.** Шукаємо індекс середини масиву (округляємо в меншу сторону):

$$mid = (11 + 14) / 2 = 12$$

Порівнюємо значення за цим індексом з шуканим:

$$80 < 82$$

Зрушуємо ліву межу

$$left = mid = 12$$

**Крок 5.** Шукаємо індекс середини масиву (округляємо в меншу сторону):

$$mid = (12 + 14) / 2 = 13$$

Порівнюємо значення за цим індексом з шуканим:

$$82 = 82$$

Рішення знайдено!

Щоб зменшити кількість кроків пошуку можна відразу зміщувати межі пошуку на елемент, наступний за серединою відрізка:

$$left = mid + 1$$

$$right = mid - 1$$

Реалізація бінарного пошуку на C++

```
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>
using namespace std;

int main() {
    setlocale (LC_ALL, "Rus");
    int k[20]; // масив значень
    int key, i;
    // Ініціалізація масиву впорядкованими значеннями
    k[0] = 8; k[1] = 14; k[2] = 26; k[3] = 28; k[4] = 38; k[5] = 47;
    k[6] = 56; k[7] = 60; k[8] = 64; k[9] = 69; k[10] = 70; k[11] = 78;
    k[12] = 80; k[13] = 82; k[14] = 84; k[15] = 87; k[16] = 90; k[17] = 92;
    k[18] = 98; k[19] = 108;

    printf("Введіть key: "); // вводим значення, що будемо шукати
    scanf("%d", &key);
    int left = 0; // задаємо ліву та праву межу пошуку
    int right = 19;
    int search = -1; // індекс елементу дорівнює -1 (елемент не знайдений)
    while (left <= right) // доки ліва межа не "перескочить" праву
    {
        int mid = (left + right) / 2; // пошук середину відрізка
        if (key == k[mid]) { // якщо ключове поле дорівнює потрібному елементу
            search = mid; // знайшли потрібний елемент,
            break; // виходимо з циклу
        }
        if (key < k[mid]) // якщо ключове поле менше знайденої середини
            right = mid - 1; // зсуваємо праву межу, продовжуємо пошук у лівій частині
        else // інакше
```

```

    left = mid + 1;    // зсуваємо ліву межу, продовжуємо пошук у правій частині
}
if (search == -1)      // якщо індекс -1, то елемент не знайдений
    printf("Елемент не знайдений!\n");
else
    printf("key = %d, index = %d ", k[search], search);
return 0;
}

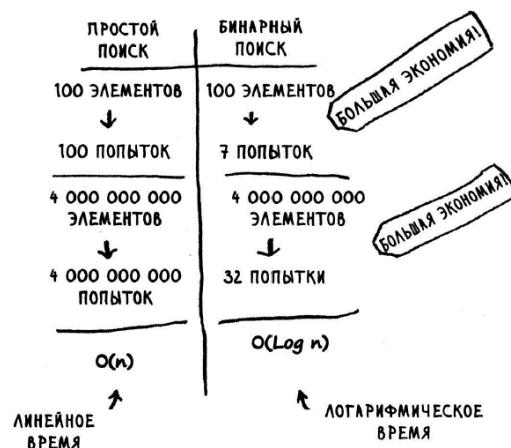
```

<https://prog-cpp.ru/search-binary/>

### 3. Час роботи

Повернемося до *к*. У першій версії ми послідовно перевіряли кожне число по одному. Якщо список складається з 100 чисел, може знадобитися до 100 спроб. Список з 4 мільярдів чисел потребує до 4 млрд спроб. Таким чином, максимальна кількість спроб така ж, як і розмір списку. Цей час виконання називається лінійним.

Якщо список складається з 100 елементів, то для бінарного пошуку потрібно не більше 7 спроб. Список з 4 мільярдів пунктів потребує не більше 32 спроб. Бінарний пошук виконується за логарифмічний час.



Спеціальна нотація «О-велике» описує швидкість алгоритму. Час від часу вам доведеться використовувати алгоритми інших людей, то було б непогано зрозуміти, як швидко або повільно вони працюють.

Боб пише алгоритм пошуку для NASA. Його алгоритм буде працювати, коли ракета буде підлітати до місяці, і має розрахувати точку посадки.

Боб намагається вибрати між простим і бінарним пошуком. Його алгоритм повинен працювати швидко і правильно. З одного боку, бінарний пошук працює швидше. Боб має лише 10 секунд, щоб вибрати посадковий майданчик; якщо він

не впорається за цей час, момент для посадки буде пропущений. З іншого боку, простий пошук легше писати і ймовірність помилок в ньому нижче. Звичайно, Боб абсолютно не хоче помилитися в коді. А потім, заради впевненості, Боб вирішує виміряти час виконання обох алгоритмів для списку з 100 пунктів.

Припустимо, що перевірка одного пункту займає 1 (мс). Якщо використовується простий пошук, Бобу доведеться перевірити 100 пунктів, так що пошук займе 100 мс. З іншого боку, при бінарному пошуку достатньо перевірити лише 7 елементів ( $\log_2 100$  приблизно 7), то пошук займе 7 мс. Але реальний список може містити більше мільярда пунктів.

Скільки часу потрібно для завершення простого пошуку? А як щодо бінарного пошуку?

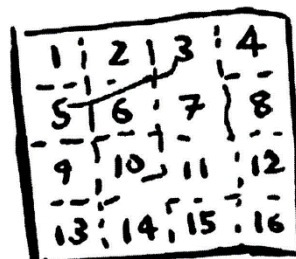
Боб проводить бінарний пошук з 1 млрд пунктів, і це займає 30 мс ( $\log_2 1\,000\,000\,000 = 30$ ). 32мс. Боб думає. Двійковий пошук в 15 разів швидше, ніж простий, тому що простий пошук за 100 пунктів зайняв 100 мс, а бінарний пошук зайняв 7мс. Таким чином, простий пошук займе  $30 \times 15 = 450$ мс, чи не так? Набагато менше, ніж виділені 10 секунд. І Боб вибирає простий пошук. Чи правда його вибір?

	ПРОСТОЙ ПОИСК	БИНАРНЫЙ ПОИСК
100 ЭЛЕМЕНТОВ	100 мс	7 мс
10 000 ЭЛЕМЕНТОВ	10 секунд	14 мс
1 000 000 ЭЛЕМЕНТОВ	11 дней	32 мс

Ні, Боб не правий. Глибоко помиляється. Час роботи для простого пошуку з 1 млрд пунктів буде 1 млрд мс, і це 11 днів! Проблема в тому, що час виконання для бінарного і простого пошуку збільшується на різних швидкостях.

«О-велике» не повідомляє швидкість за секунди, але дозволяє порівняти кількість операцій. Це показує, наскільки швидко збільшується час алгоритму.

Необхідно побудувати сітку з 16 квадратів.



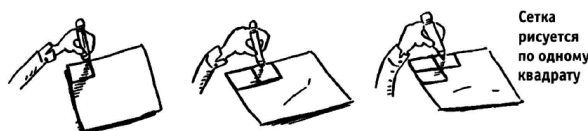
Как должен выглядеть хороший алгоритм для построения этой сетки?

### Алгоритм 1

Як виглядатимуть гарний алгоритм для побудови цієї сітки?

Можно намалювати 16 квадратів, по одному за раз. Нам потрібно намалювати 16 квадратів. Скільки операцій малювання квадрату доведеться виконати?





## Алгоритм 2

Зігніть лист навпіл. На цей раз операція вважається складання листа. Виявляється, одна операція створює два прямокутники відразу! Зігніть папір знову, а потім знову і знову. Розгорніть листок після чотирьох складок виявився чудовою сіткою! Кожне доповнення подвоює кількість прямокутників. За 4 операції створено 16 прямокутників!



Як записати час, потрібен для запуску цього алгоритму?

Відповіді: Алгоритм 1 виконується за час  $O(n)$  і алгоритм 2 - за час  $O(\log_2 n)$ .

## 4. Асимптотичний аналіз

Порядок зростання описує те, як складність алгоритму росте зі збільшенням розміру вхідних даних. Найчастіше він представлений у вигляді  $O(f(x))$ , де  $f(x)$  – формула, що виражає складність алгоритму. У формулі може бути присутньою змінна  $n$ , що представляє розмір вхідних даних. Нижче наводиться список порядків зростання, що найчастіше зустрічаються.

**Константний** –  $O(1)$ . Порядок зростання  $O(1)$  означає, що обчислювальна складність алгоритму не залежить від розміру вхідних даних. Слід пам'ятати, однак, що одиниця у формулі не означає, що алгоритм виконується за одну операцію або потребує дуже мало часу. Він може тривати і мікросекунду, і рік. Важливо те, що цей час не залежить від вхідних даних.

**Лінійний** –  $O(n)$ . Порядок зростання  $O(n)$  означає, що складність алгоритму лінійно росте зі збільшенням вхідного масиву. Якщо лінійний алгоритм обробляє один елемент п'ять мілісекунд, то ми можемо чекати, що тисячу елементів він обробить за п'ять секунд.

Такі алгоритми легко впізнати за наявністю циклу з кожного елемента вхідного масиву.

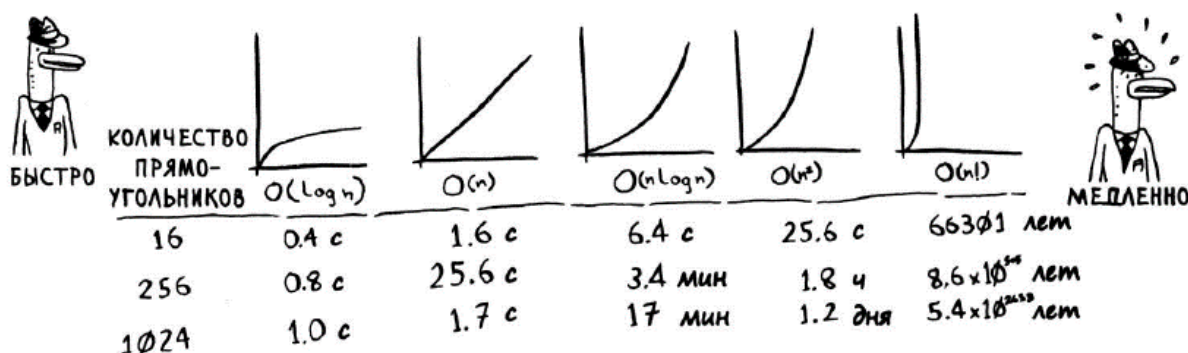
**Логарифмічний** –  $O(\log n)$ . Порядок зростання  $O(\log n)$  означає, що час виконання алгоритму росте логарифмічно зі збільшенням розміру вхідного масиву. (в аналізі алгоритмів за умовчанням використовується логарифм з

основою 2). Більшість алгоритмів, які працюють за принципом “ділення навпіл”, мають логарифмічну складність).

*Лінеарифметичний* –  $O(n \cdot \log n)$ . Лінеарифметичний (чи лінійно-логарифмічний) алгоритм має порядок зростання  $O(n \cdot \log n)$ . Деякі алгоритми типу “розділай і володарюй” потрапляють до цієї категорії.

*Квадратичний* –  $O(n^2)$ . Час роботи алгоритму з порядком зростання  $O(n^2)$  залежить від квадрата розміру вхідного масиву. Попри те, що такої ситуації іноді не уникнути, квадратична складність – привід переглянути використовувані алгоритми або структури даних. Проблема полягає в тому, що вони погано масштабуються. Наприклад, якщо масив з тисячі елементів потребує 1 000 000 операцій, масив з мільйона елементів – 1 000 000 000 000 операцій. Якщо одна операція вимагає мілісекунду для виконання, квадратичний алгоритм оброблятиме мільйон елементів 32 роки. Навіть якщо він буде в сто разів швидше, робота триватиме 84 дні.

- $O(\log n)$  – логарифмічний час (бінарний пошук);
- $O(n)$  – лінійний час (пошук);
- $O(n \log n)$  – ефективні алгоритми сортування (швидке сортування);
- $O(n^2)$  – алгоритми повільного сортування (сортування вибору);
- $O(n!)$  – дуже повільні алгоритми (задача комівояжера).



Припустімо, що ви будете сітку з 16 квадратів знову, і можете вибрати один з 5 алгоритмів, щоб вирішити цю проблему. За допомогою першого алгоритму сітка буде побудована за  $O(\log n)$ . За секунду виконуються 10 операцій. З  $O(\log n)$  часу, потрібно 4 операції, щоб побудувати сітку 16 квадратів. Таким чином, сітка буде побудована за 0,4 секунди. Що робити, якщо необхідно побудувати 1024 квадратів? Це займе 10 операцій, або 1 секунду.

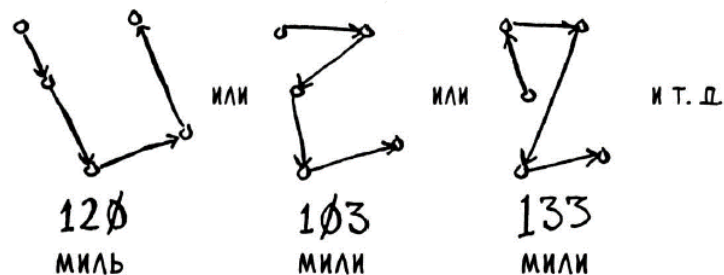
## 5. Задача комівояжера

Розглянемо алгоритм з дуже, дуже повільним часом виконання. Комівояжер повинен подорожувати по 5 містах.





Комівояжер хоче відвідати кожне з 5 міст за мінімальну загальну відстань. Одне з можливих рішень: потрібно пройти всі можливі комбінації порядку обходу міст.



Всі відстані додаються, а потім шлях вибирається з найкоротшою відстанню. Для 5 міст можна створити 120 перестановок, тому вирішення проблеми для 5 міст потрібно 120 операцій. Для 6 міст кількість операцій збільшується до 720, а для 7 міст знадобиться 5040 операцій!

ГОРОДА	ОПЕРАЦИИ
6	720
7	5040
8	40320
...	...
15	1307674368000
...	...
30	265252859812171058526308480000000

Загалом для обрахування результату при  $n$  елементів необхідно  $n!$  операцій. При будь-якому серйозному розмірі списку кількість операцій буде величезною. Це одна з відомих невирішених проблем в області обчислювальної теорії.

## Висновки

- Двійковий пошук працює набагато швидше, ніж простий.
- Час виконання  $O(\log n)$  швидше, ніж  $O(n)$ , і зі зростанням кількості елементів, він стає набагато швидше.
- Швидкість алгоритмів не вимірюється в секундах.
- Час алгоритму описується збільшенням кількості операцій.
- Час виконання алгоритмів позначається як "О-велике".

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. СПб.: Питер, 2017. 288 с.
2. <https://prog-cpp.ru/search-binary/>