Грунтовний порівняльний аналіз алгоритмів сортування

Сортування вставкою

Швидке сортування

Виконав студент групи ПМі-25с
Ліщинський Владислав

Insertion Sort

(Сортування вставками)

Опис алгоритму

Сортування вставками - це простий алгоритм сортування, який працює за принципом поступового впорядкування елементів у списку. Його суть полягає в тому, що ми беремо елементи один за одним і вставляємо їх у відповідне місце у вже відсортованій частині масиву

Принцип роботи

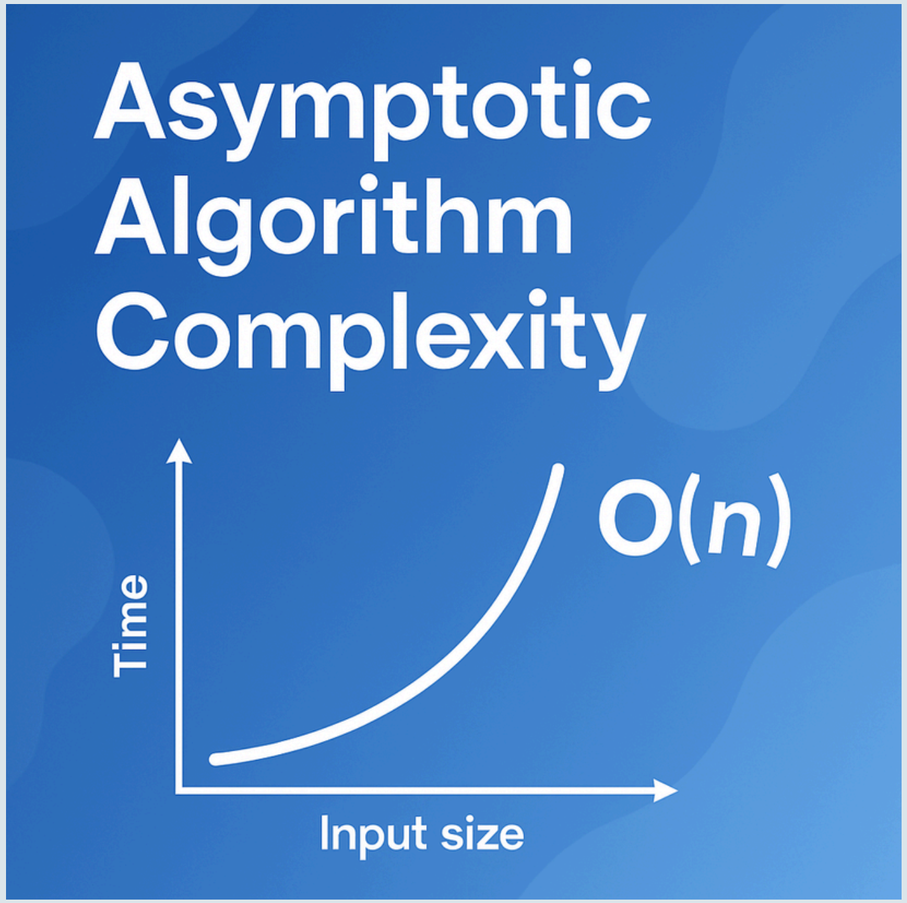
- Починаємо з другого елемента (перший вважається відсортованим).
- Беремо поточний елемент і порівнюємо його з попередніми.
- Переміщуємо більші елементи праворуч, звільняючи місце для поточного.
- Вставляємо поточний елемент у правильну позицію.
- Повторюємо процес для всіх елементів масиву.

📈 Асипмпотична складність

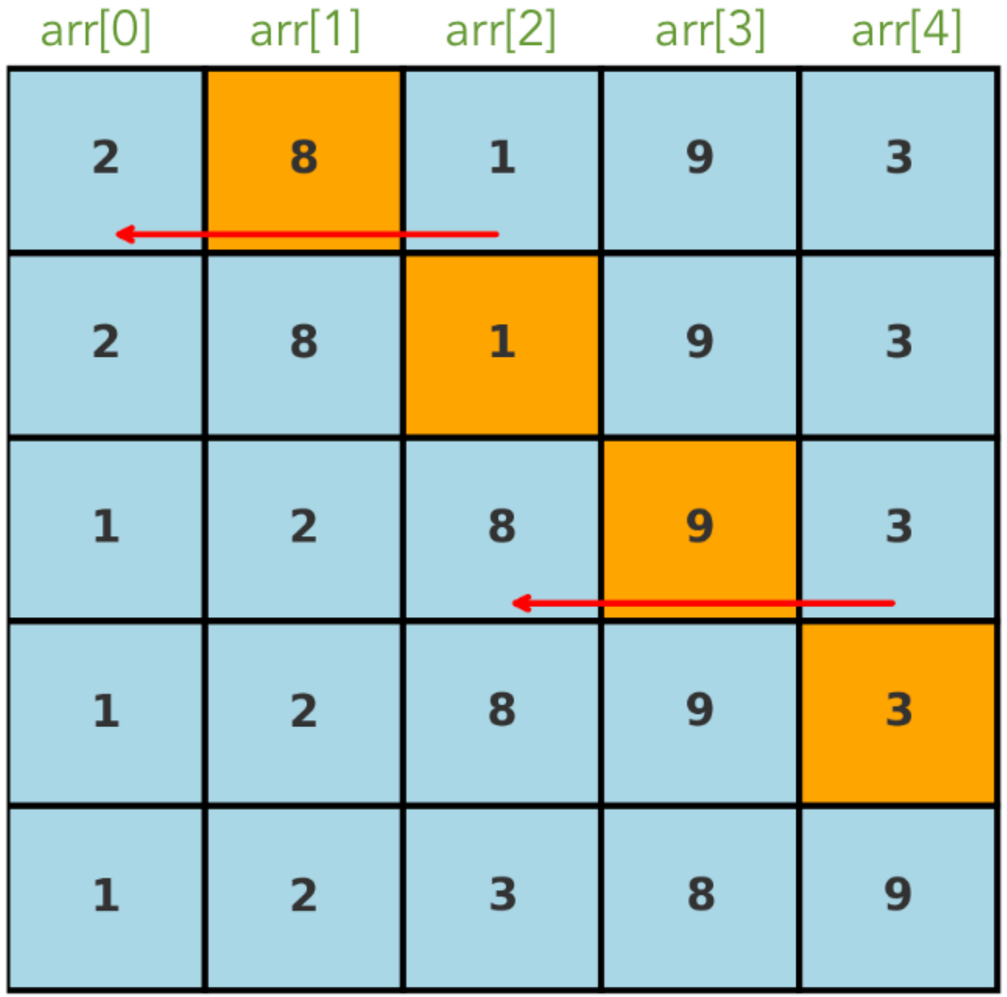
Найгірший випадок: O(n²), коли елементи масиву сортуються у зворотному порядку. Це означає, що кожен елемент потрібно порівняти з усіма попередніми елементами.

Найкращий випадок: O(n), коли масив вже відсортований. Кожен елемент буде порівнюватися тільки з попереднім

Середній випадок: O(n²), оскільки в середньому кожен елемент буде порівнюватися з половиною попередніх елементів.



Візуалізація на прикладі [2, 8, 1, 9, 3]



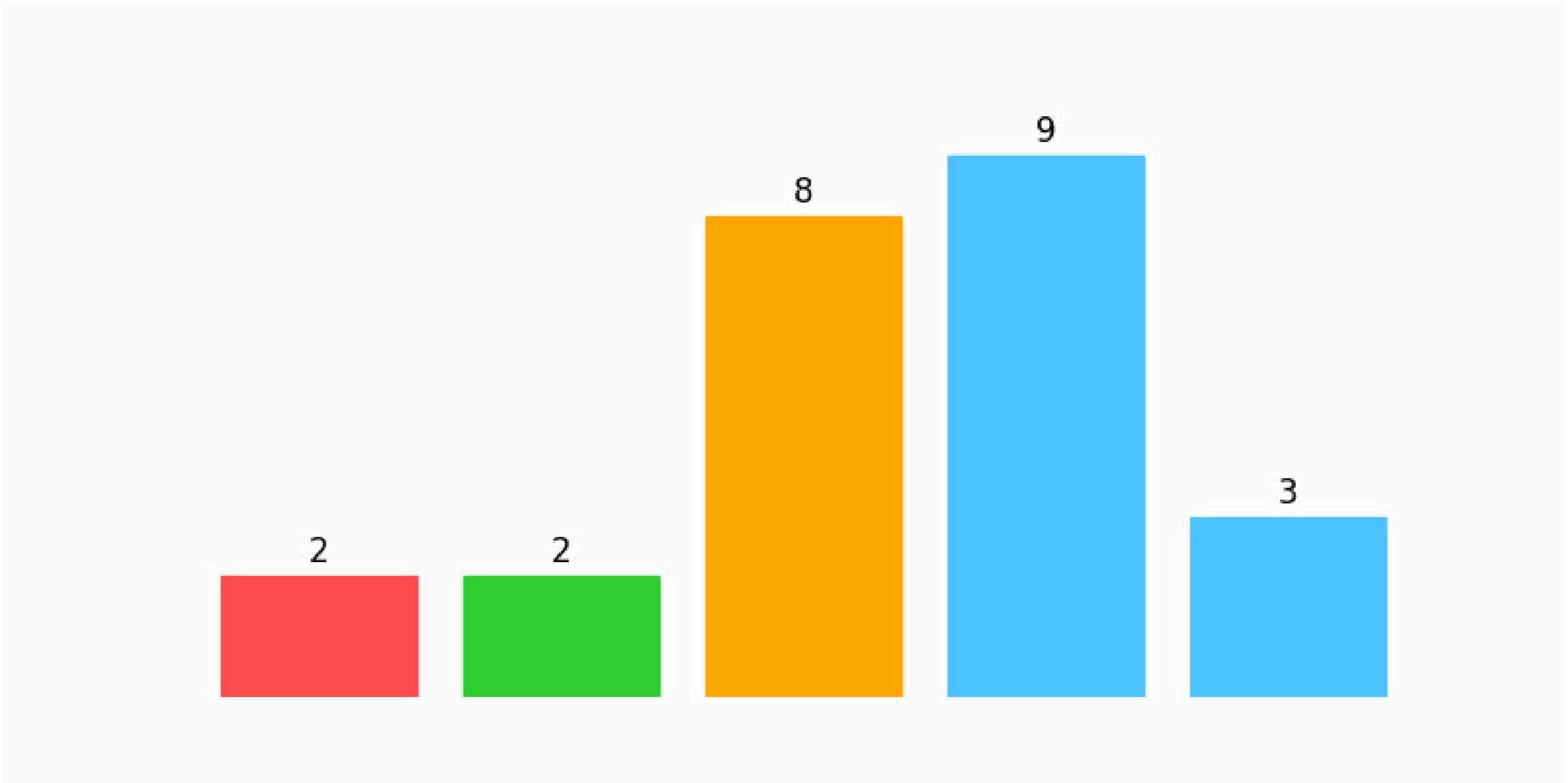
Крок 1: 8 > 2 → нічого не змінюємо

Крок 2: 1 < 8, 1 < 2 → зсуваємо 8 і 2 → вставляємо 1 на початок

Крок 3: 9 > 8 → нічого не змінюємо

Крок 4: 3 < 9, 3 < 8 → зсуваємо 9 і 8 → вставляємо 3 після 2

Кінцевий масив


2
2
8
9
3

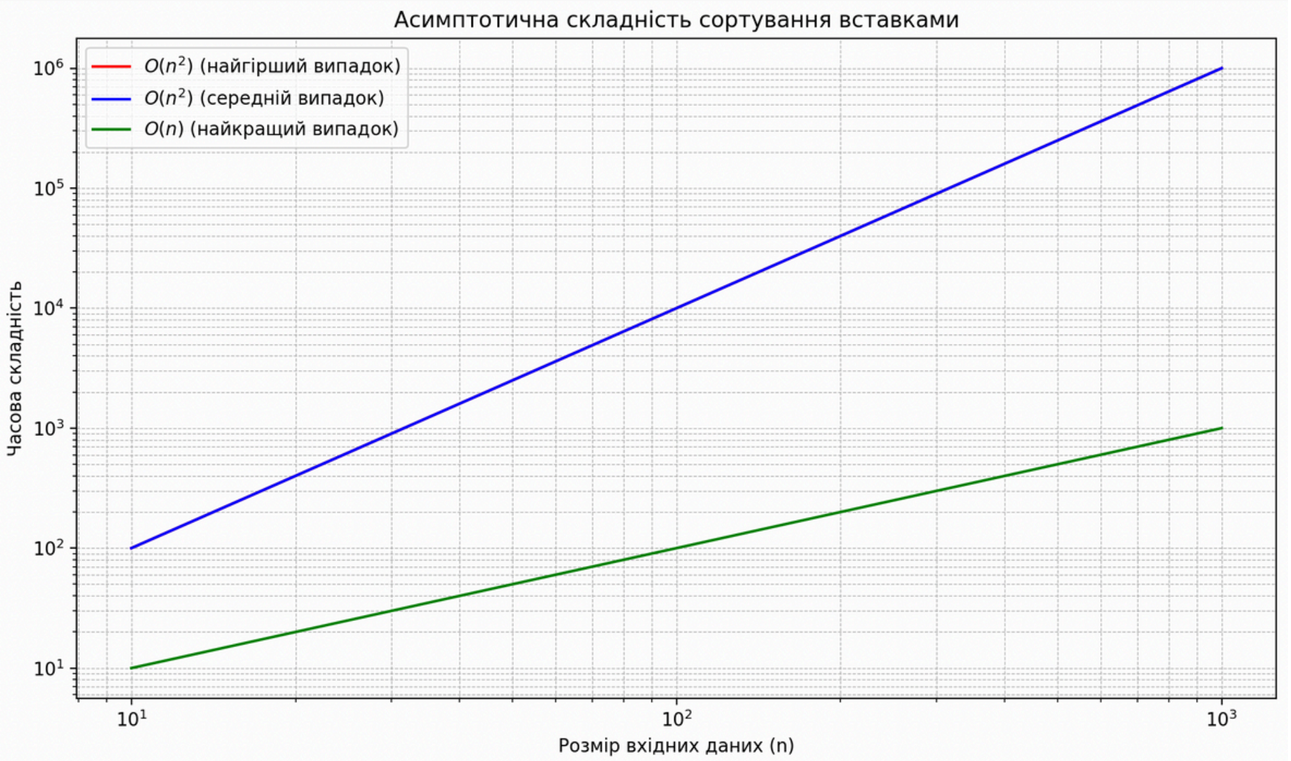
Програмна реалізація

```cpp
template <typename T>
void InsertionSort(T arr[], size_t size)
{
    // Проходимо по елементах масиву, починаючи з другого (індекс 1)
    for (size_t i = 1; i < size; i++)
    {
        // Зберігаємо поточне значення, яке потрібно вставити у відсортовану частину
        T key = arr[i];

        // j — індекс для проходу по відсортованій частині масиву
        int j;

        // Поки не дійшли до початку масиву і поточний елемент arr[j] більший за key,
        // зсуваємо елемент arr[j] на одну позицію вправо
        for (j = i - 1; j >= 0 && arr[j] > key; j--)
        {
            arr[j + 1] = arr[j];
        }

        // Вставляємо елемент key на правильну позицію
        arr[j + 1] = key;
    }
}
```




```cpp
template <typename T>
void InsertionSort(T arr[], size_t size)
{
    // Точна складність:
    // Зовнішній цикл виконується n-1 разів, де n - кількість елементів у масиві.
    // Внутрішній цикл для кожного елемента може виконувати до і порівнянь, де і - індекс елемента.
    // Таким чином, для кожного елемента на і-ій позиції в масиві внутрішній цикл виконується до і
    // В найгіршому випадку (коли масив відсортований у зворотному порядку), кількість порівнянь:
    // 1 + 2 + 3 + ... + (n-1) = (n-1) * n / 2.
    // Отже, точна складність алгоритму в найгіршому випадку O(n^2).

    // Асимптотична складність:
    // У найгіршому випадку, коли кожен елемент потрібно переміщати до початку масиву,
    // кількість порівнянь та операцій переміщення дорівнює O(n^2).
    // Тому асимптотична складність алгоритму - O(n^2) для найгіршого випадку.
    // У кращому випадку, коли масив вже відсортований, складність буде O(n),
    // оскільки внутрішній цикл лише перевіряє умови без переміщення елементів.

    // Проходимо по елементах масиву, починаючи з другого (індекс 1)
    for (size_t i = 1; i < size; i++)
    {
        // Зберігаємо поточне значення, яке потрібно вставити у відсортовану частину
        T key = arr[i];

        // j - індекс для проходу по відсортованій частині масиву
        int j;

        /// Внутрішній цикл переміщає елементи, більші за key
        // в праву сторону, поки не знайде правильну позицію для key
        for (j = i - 1; j >= 0 && arr[j] > key; j--)
        {
            arr[j + 1] = arr[j];
        }

        // Вставляємо елемент key на правильну позицію
        arr[j + 1] = key;
    }
}
```

Quick Sort

(Швидке сортування)

Опис алгоритму

Алгоритм QuickSort працює за принципом "розділяй і володарюй", що означає розбиття великої задачі на менші підзадачі, їх вирішення, а потім об'єднання результатів
Його основна ідея — вибрати опорний елемент (pivot), і розмістити всі елементи менші за нього зліва, а більші — справа. Після цього ті ж дії виконуються рекурсивно для кожної з підчастин

Принцип роботи

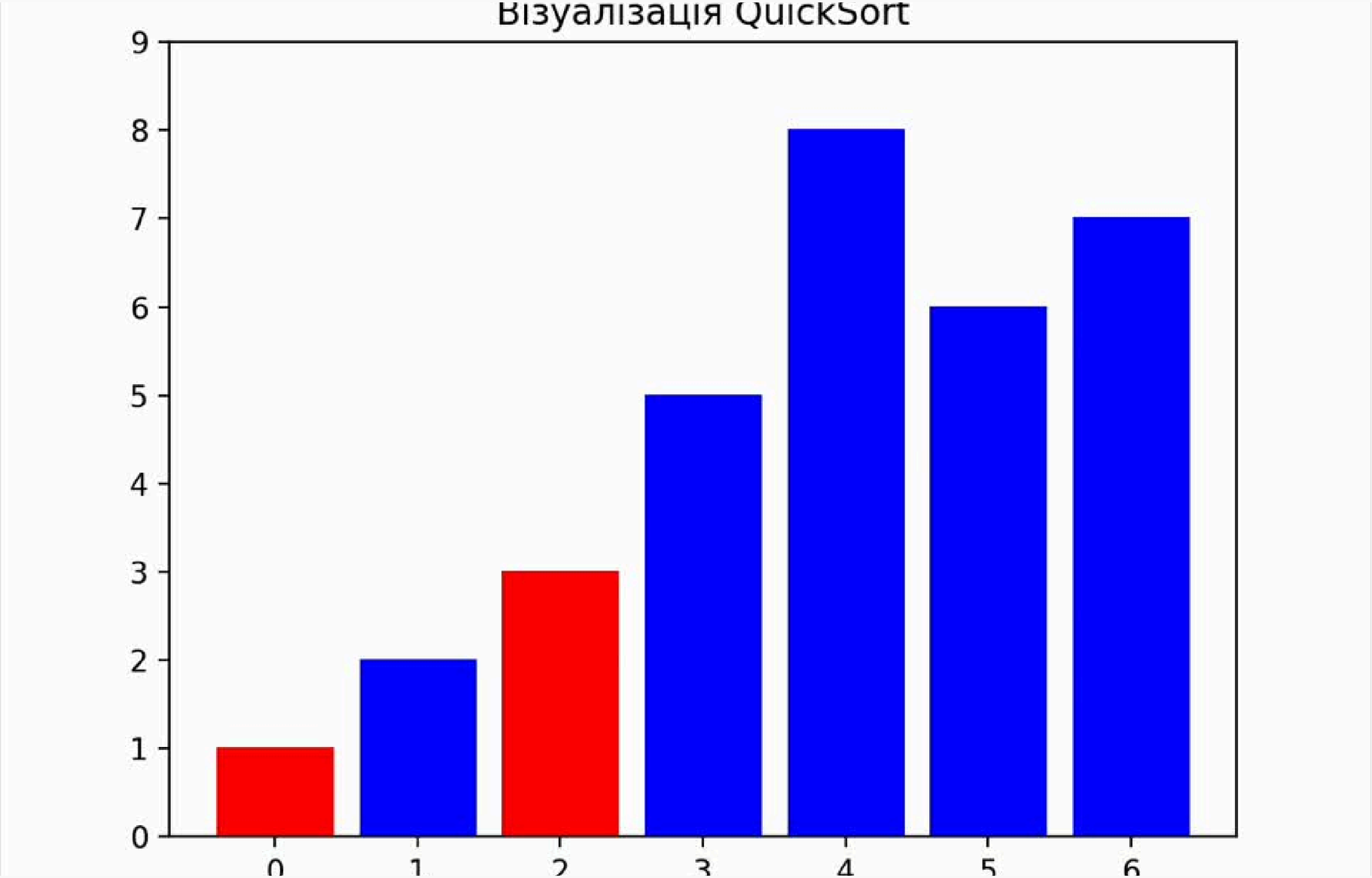
- Вибір елемента (називається опорним - pivot - pivotal (керуючий))
- Розбиття масиву на дві частини: менші за опорний і більші..
- Рекурсивно сортує ці частини..

📈 Складність алгоритму

Випадок	Кількість операцій
Найгірший	$O(n^2)$ – якщо опорний елемент найменший або найбільший
Середній	$O(n \log n)$
Найкращий	$O(n \log n)$ – якщо масив добре поділяється

У середньому, QuickSort є **одним із найшвидших алгоритмів сортування**.

Візуалізація на прикладі



Програмна реалізація та обчислення складності

```cpp
template <typename T>
void QuickSort(T arr[], int first, int last)
{
    int i = first;
    int j = last;
    T middle = arr[(first + last) / 2];

    while (i <= j) // 5
    {
        while (arr[i] < middle) i++; // 3
        while (arr[j] > middle) j--; // 2

        if (i <= j) // 4
        {
            std::swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }

    if (j > first)
        QuickSort(arr, first, j);
    if (i < last)
        QuickSort(arr, i, last);
}
```

```cpp
// QuickSort для масиву з 7 елементів: {7, 2, 1, 6, 8, 5, 3}

// Кількість порівнянь і обмінів:
// Рівень 1: pivot = 6 → порівнянь ≈ 6, обмінів = 3
// Рівень 2 (ліва частина): {2,1,3} → pivot = 1 → порівнянь ≈ 2, обмінів = 1
// Рівень 3 (від {2,3}): pivot = 2 → порівнянь ≈ 1, обмінів = 1
// Рівень 2 (права частина): {7,8,5} → pivot = 8 → порівнянь ≈ 2, обмінів = 1
// Рівень 3 (від {7,5}): pivot = 7 → порівнянь ≈ 1, обмінів = 1

// ПІДСУМОК:
// Загальна кількість порівнянь: ≈ 12
// Загальна кількість обмінів: 7
```

```cpp
// 1. Взяти центральний елемент
// 2. В циклі знайти елемент справа від центрального елемента, який менший за нього
// 3. В циклі знайти елемент зліва від центрального елемента, який більший за нього
// 4. Поміняти місцями найдені значення якщо їх індекси не перетнулись
// 5. Виконати дії 2-4 поки індекси не перетнулись
// 6. Рекурсивно виконати дії 1-5 для лівої частини масиву, якщо правий індекс не дійшов до початку
// 7. Рекурсивно виконати дії 1-5 для правої частини масиву, якщо лівий індекс не дійшов до кінця
```


Асимптотична складність швидкого сортування
$O(n^2)$ (найгірший випадок)
$O(n \log n)$ (середній випадок)
$O(n \log n)$ (найкращий випадок)
Часова складність
10^6
10^5
10^4
10^3
10^2
10^1
10^1
10^2
10^3
Розмір вхідних даних (n)

Порівняння швидкодії алгориту на масивах різних видів заданої довжини

Для порівняння алгоритмів було проведено тестування на трьох типах масивів:

- Найкращий випадок (відсортований масив)
- Середній випадок (випадковий масив)
- Найгірший випадок (обернений масив)

Розмір масиву: **500 і 5000** елементів.

Порівняльна таблиця

Розмір масиву	Випадок	**Швидке сортування (сек)**	**Сортування вставками (сек)**
500 елементів	Найкращий випадок	0.0008	0
	Середній випадок	0.0006	0.0043
	Найгірший випадок	0.0007	0.0081
5000 елементів	Найкращий випадок	0.0069	0.0004
	Середній випадок	0.0072	0.4349
	Найгірший випадок	0.0073	0.8555

Висновок

Швидке сортування є загалом набагато ефективнішим для середніх і великих масивів, незалежно від початкового порядку даних. Воно має середню складність, що забезпечує стабільну швидкодію в більшості випадків.

Сортування вставками, незважаючи на свою простоту та ефективність у найкращому випадку (майже відсортовані масиви), не підходить для великих обсягів даних через експоненційне зростання часу обчислення в середньому та найгіршому випадках.