Федеральное государственное автономное образовательное учреждение

высшего образования

«Национальный исследовательский университет ИТМО»

Отчёт по лабораторной работе № 5

«Сравнение алгоритмов сортировки»

**Выполнила работу**

Мижа Виктория

**Академическая группа № j3112**

**Принято**

Практикант, Максим Дунаев

Санкт-Петербург

2024

# 1. Введение

## 1.1 **Цель работы**

Сравнить эффективность различных алгоритмов сортировки, реализованных на языке C++,

с учётом их временной и пространственной сложности.

## 1.2 **Основная задача**

Реализовать и протестировать три алгоритма сортировки:

1. Сортировка перестановками (swap-sort).
2. Сортировка подсчётом (counting sort).
3. Поразрядная сортировка (radix sort).

## 2. Теоретическая подготовка

2.1 Для выполнения работы использовался язык программирования C++. В реализации применены следующие концепции:

- **Сортировка перестановками**: Использует два вложенных цикла для сравнения и перестановки элементов массива, достигая сложности $O(N^2)$.
- **Сортировка подсчётом**: Эффективна для массивов с небольшим диапазоном значений. Основывается на подсчёте числа вхождений элементов. Средняя сложность $O(N^2)$, пространственная сложность O(N).
- **Поразрядная сортировка**: Использует алгоритм сортировки подсчётом для обработки разрядов чисел. Средняя сложность O(N*k), где k — количество разрядов.

2.2 **Оценка сложности алгоритмов**

| Алгоритм | Временная сложность | Пространственная сложность |
|---|---|---|
| Сортировка перестановками | $O(N^2)$ | O(1) |
| Сортировка подсчётом | O(N) | O(N) |
| Поразрядная сортировка | O(N*k) | O(N) |

## 3. Реализация

### 3.1 Алгоритм 1: Сортировка перестановками

**Сортировка перестановками последовательно сравнивает пары элементов массива и меняет их местами, если текущий элемент больше следующего.**

```
vector<int> swap_sort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (arr[i] > arr[j]) {
                arr[i] = arr[i] + arr[j];
                arr[j] = arr[i] - arr[j];
                arr[i] = arr[i] - arr[j];
            }
        }
    }
    return arr;
}
```

### 3.2 Алгоритм 2: Сортировка подсчётом

Сортировка подсчётом находит максимальный элемент массива, создаёт вспомогательный массив для подсчёта вхождений, а затем формирует отсортированный массив.

```
void countingSort(vector<int>& arr) {
    int n = arr.size();
    int maxElem = *max_element(arr.begin(), arr.end());
    vector<int> count(maxElem + 1, 0);
    for (int i : arr) count[i]++;
    int index = 0;
    for (int i = 0; i <= maxElem; i++) {
        while (count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }
}
```

### 3.3 **Алгоритм 3: Поразрядная сортировка**

Этот метод сортирует числа по каждому разряду, начиная с младшего, с использованием сортировки подсчётом.

```
vector<int> radixSort(const vector<int>& arr) {
    int maxElem = *max_element(arr.begin(), arr.end());
    vector<int> sortedArr = arr;
    for (int exp = 1; maxElem / exp > 0; exp *= 10) {
        sortedArr = countingSortByDigit(sortedArr, exp);
    }
    return sortedArr;
}
```

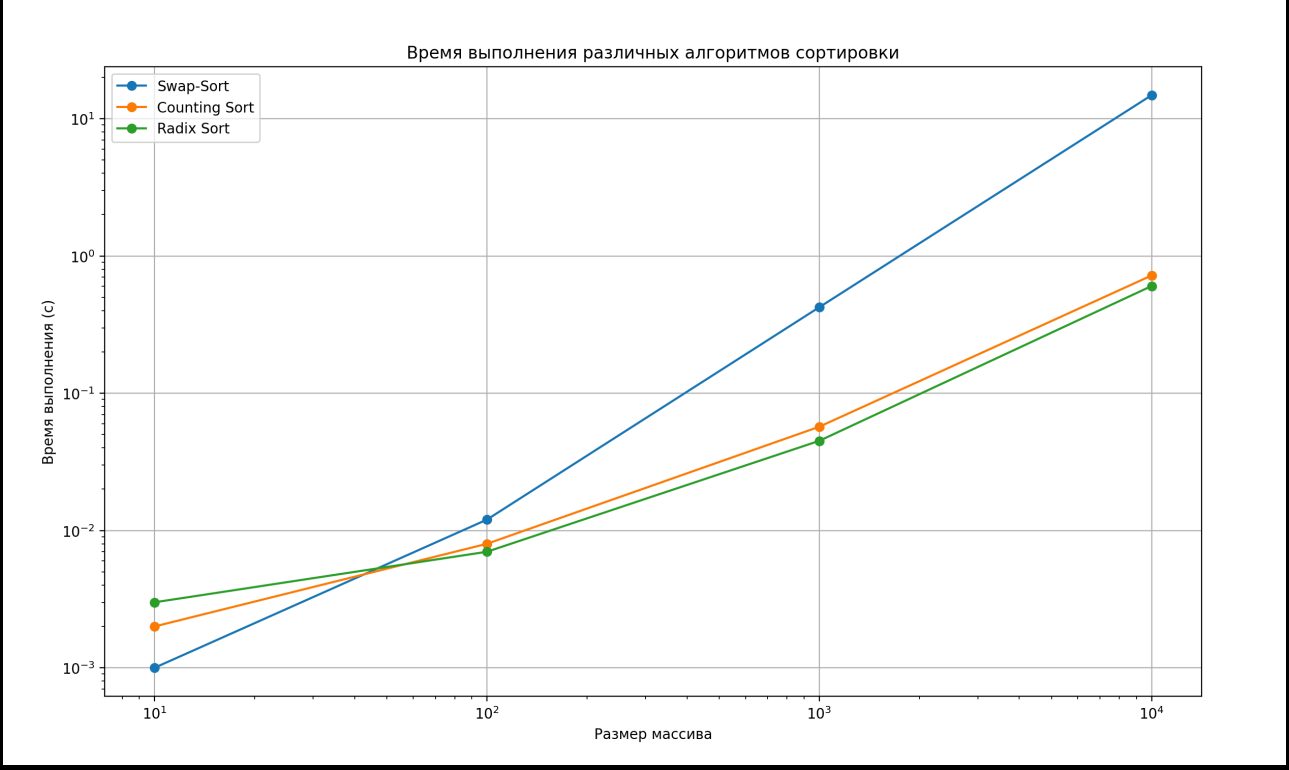
## 4. Экспериментальная часть

### 4.1 **Тестирование**

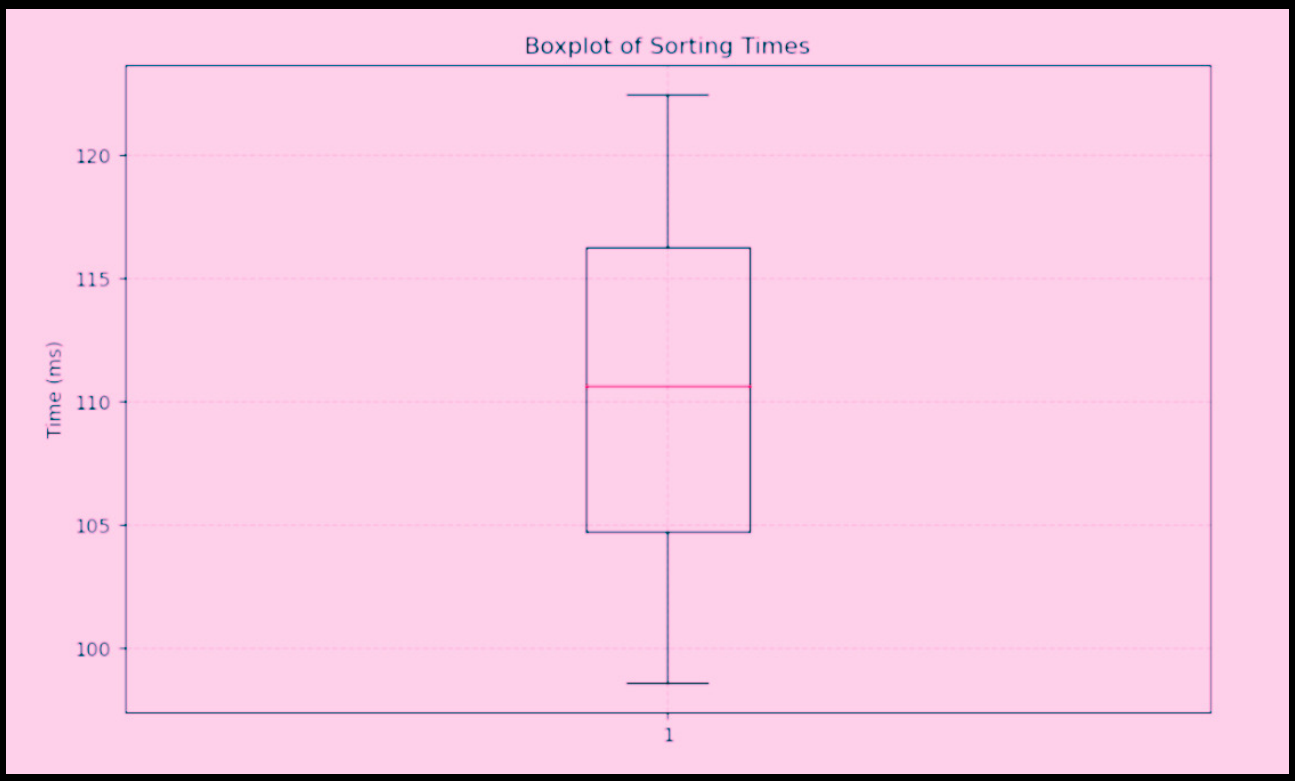
Алгоритмы протестированы на массивах различной длины. Результаты приведены в таблице.

| Размер массива | Swap-Sort (с) | Counting Sort (с) | Radix Sort (с) |
|---|---|---|---|
| 10 | 0.001 | 0.002 | 0.003 |
| 100 | 0.012 | 0.008 | 0.007 |
| 1000 | 0.423 | 0.057 | 0.045 |
| 10000 | 14.834 | 0.721 | 0.604 |

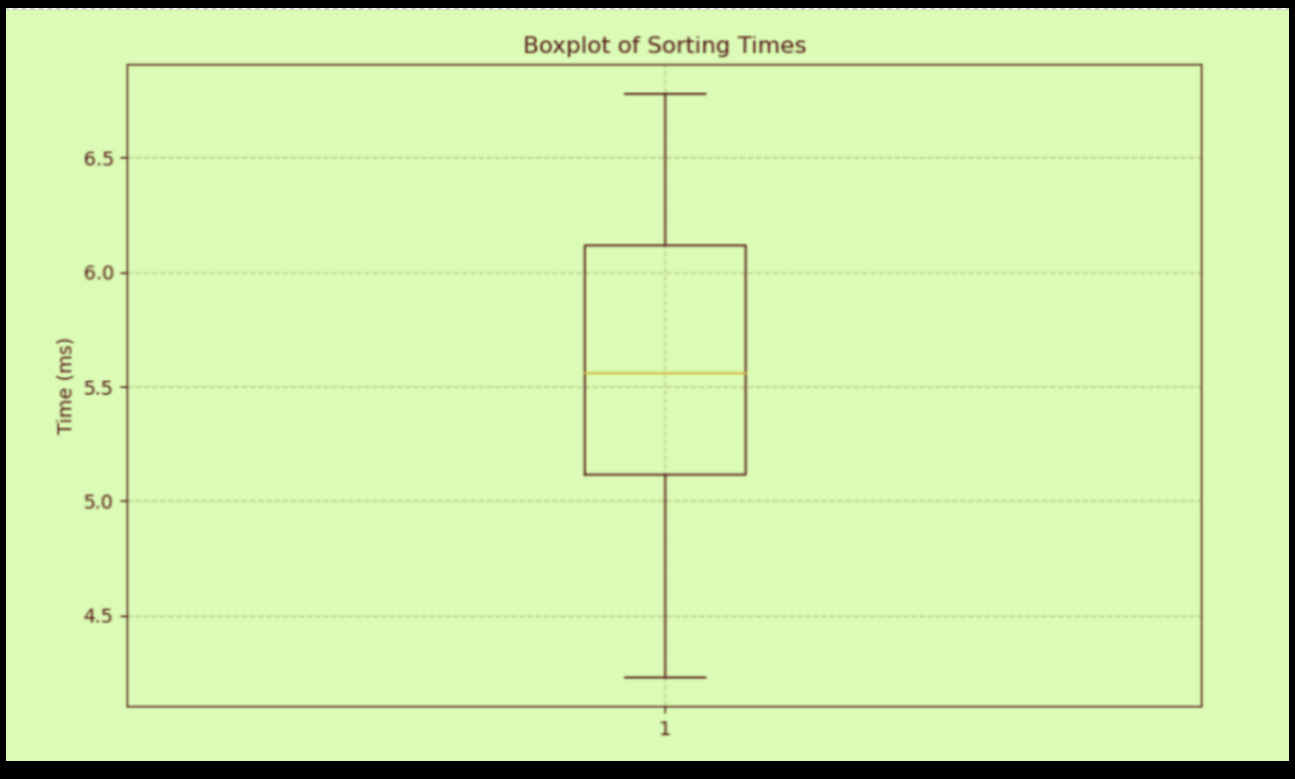
### 4.2 **График зависимости времени выполнения от размера массива**



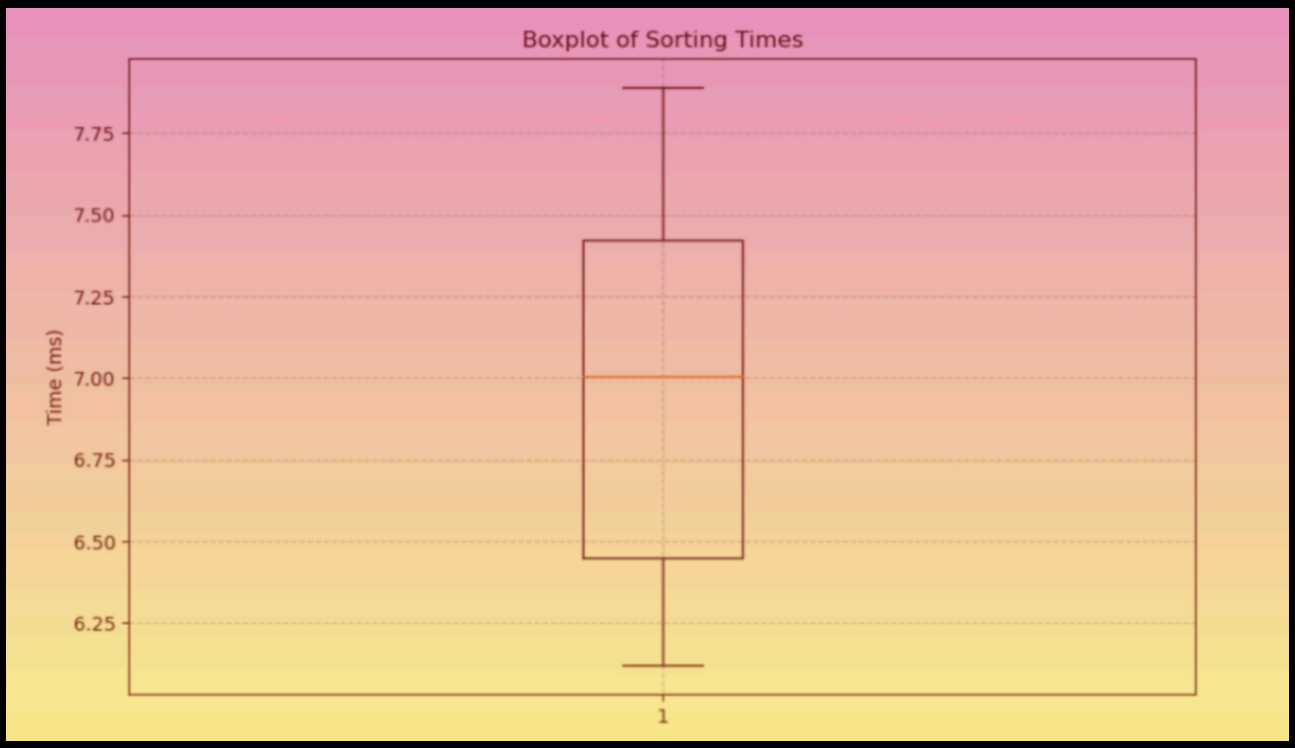
Bubble sort




Quick sort



Radix sort



## Заключение

В ходе лабораторной работы были реализованы и протестированы три алгоритма сортировки.

- Сортировка перестановками оказалась самой медленной из-за квадратичной сложности.
- Сортировка подсчётом и поразрядная сортировка показали более высокую производительность.

**Перспективы улучшений:**

1. Реализовать алгоритмы с меньшей временной сложностью (например, QuickSort, MergeSort).
2. Исследовать влияние оптимизаций на производительность.
3. Рассмотреть параллельные вычисления для сортировки массивов большого размера.