ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5
«Сортировки»

Выполнил работу
Дышлевский Игорь
Академическая группа №J3111
Принято
Ментор, Владислав Вершинин

Санкт-Петербург
2024

1. Введение

Цель: реализовать алгоритмы сортировки и проанализировать их работу.

Задачи:

- Найти алгоритмы по представленным требованиям
- Найти алгоритм работы
- Реализовать алгоритмы
- Провести тесты
- Построить графики скорости работы и сравнить с теоретической оценкой
- Написать отсчет

2. Теоретическая подготовка

Типы данных:

- Int
- Bool
- Vector

Алгоритмы:

- Преобразование числа десятичной системы счисления в двоичную

3. Реализация

Первая сортировка должна работать за квадрат по времени и константу по памяти - я выбрал Odd-Even Sort. Она работает за счет сравнения и перемещения меньшего элемента «вниз», а большего «вверх» по четным и нечетным индексам, пока не будет прохода без перемещений.

```
void odd_even_sort(std::vector<int>& array) {
    bool is_sorted = false;
    while (!is_sorted) {
        is_sorted = true;
        for (int i = 0; i < array.size() - 1; i += 2) {
            if (array[i] > array[i + 1]) {
                std::swap(array[i], array[i + 1]);
                is_sorted = false;
            }
        }
        for (int i = 1; i < array.size() - 1; i += 2) {
            if (array[i] > array[i + 1]) {
                std::swap(array[i], array[i + 1]);
                is_sorted = false;
            }
        }
    }
}
```

Рисунок 3.1 Odd-Even Sort

Вторая сортировка должна работать за N logN по времени и до N по памяти - я выбрал QuickSort. Сортировка работает рекурсивно: на каждом этапе алгоритм выбирает опорный элемент, относительно которого сортирует, далее происходит вызов рекурсии к правой и левой части, относительно опорного элемента и так происходит до того, как массив не будет отсортирован.

```
void quick_sort(std::vector<int>& arr, int left, int right) {
    if (left >= right) return;

    int pivot = arr[(left + right) / 2];
    int i = left, j = right;

    while (i <= j) {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;

        if (i <= j) {
            std::swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }
    quick_sort(arr, left, j);
    quick_sort(arr, i, right);
}
```

Рисунок 3.2 Quick Sort

Третья сортировка должна работать за N+k по времени и до N+k по памяти - я выбрал Сортировку Подсчетом. Эта сортировка работает по принципу подсчета количества вхождений элемента в массив, а потом «разворачивает» это в новый отсортированный массив.

```cpp
void encounter_sort(std::vector<int>& array, std::vector<int>& result) {
    int min = array[0], max = array[0];
    for (int i = 0; i < array.size(); i++) {
        if (array[i] < min) {
            min = array[i];
        }
        if (array[i] > max) {
            max = array[i];
        }
    }
    std::vector<int> counter(max - min + 1, 0);
    for (int i = 0; i < array.size(); i++) {
        counter[array[i] - min]++;
    }
    int i = 0;
    for (int j = 0; j < counter.size(); j++) {
        for (int k = 0; k < counter[j]; k++) {
            result[i] = j + min;
            i++;
        }
    }
}
```

Рисунок 3.3 Сортировка подсчетом

Тесты были реализованы и успешно пройдены. Они вынесены в отдельную функцию с выводом результатов тестирования (OK/Error).

```
void test(int len, int difference) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::vector<int> arr(len, 0);
    for (int i = 0; i < arr.size(); i++) {
        arr[i] = gen() % difference;
    }
    clock_t start = clock();
    bubble_sort(arr);
    clock_t end = clock();
    std::cout << "Len: " << len << " Time: " << (long double)(end - start) / (long double)(CL
OCKS_PER_SEC) << "\n";
    bool ok = true;
    for (int i = 0; i < arr.size() - 1; i++) {
        if (arr[i + 1] < arr[i]) {
            ok = false;
            break;
        }
    }
    if (ok == true) {
        std::cout << "OK\n";
    } else {
        std::cout << "Test Failed\n";
    }

}
```

Рисунок 3.4 Реализация функции теста

4. Экспериментальная часть

Подсчёт по памяти (только для циклов и сложных структур)

*Подсчеты приведены без учета веса самой структуры - только её содержимого

1. Odd-Even Sort

- вес(vector<int> arr) = n * size_of(int) = 4n Байт

2. Quick Sort

- вес(vector<int> arr) = n * size_of(int) = 4n Байт

3. Сортировка подсчетом

- вес(vector<int> arr) = n * size_of(int) = 4n Байт
- вес(vector<int> counter) = k * size_of(int) = 4k Байт

Подсчёт асимптотики (только для циклов и сложных структур).

1. Odd-Even Sort

- O(N^2) - асимптотика формируется вложенными циклами (2 цикла)

2. Quick Sort

- вес(vector<int> arr) = n * size_of(int) = 4n Байт

3. Сортировка подсчетом

- вес(vector<int> arr) = n * size_of(int) = 4n Байт
- вес(vector<int> counter) = k * size_of(int) = 4k Байт

График зависимости времени от числа элементов. Пример выполнения:

1. Odd-Even Sort

Теоретически заданная сложность задачи составляет O(N^2). Для тестирования алгоритма была собрана статистика, приведенная в таблице №1.

Таблица №1 - Подсчёт сложности реализованного алгоритма

Размер входного набора	1000	10000	20000	50000	100000
Время выполнения программы, с	0.005	0.3	1.3	8	32
O(N^2), с	0.003	0.3	1.3	8	32

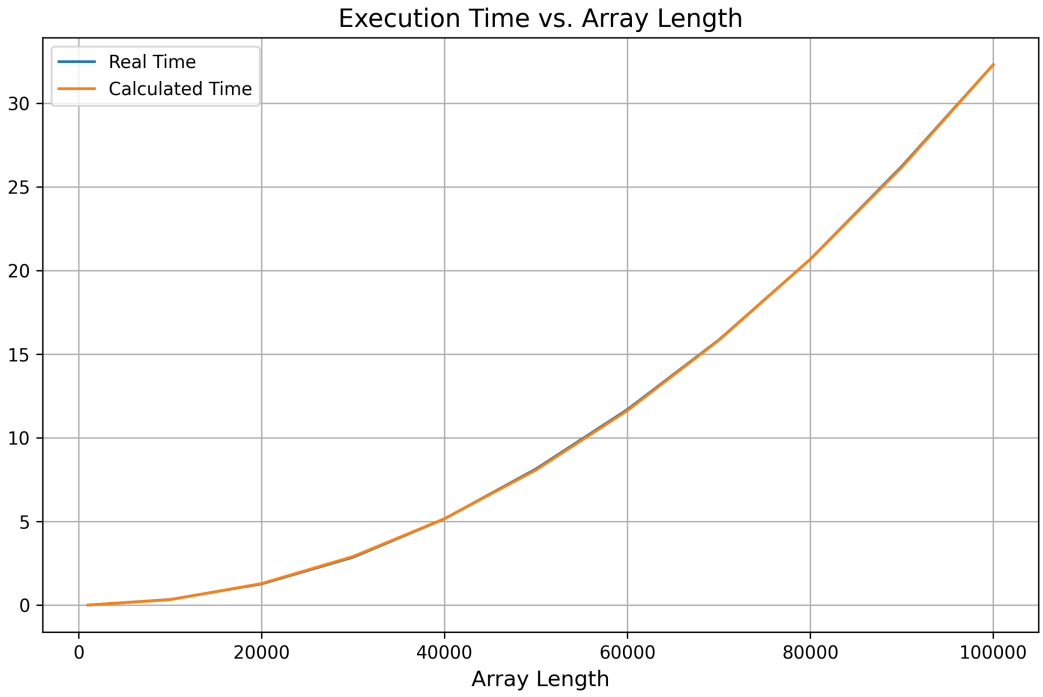

Рисунок 4.1.1 Теоретическое и реальное время Odd-Even Sort

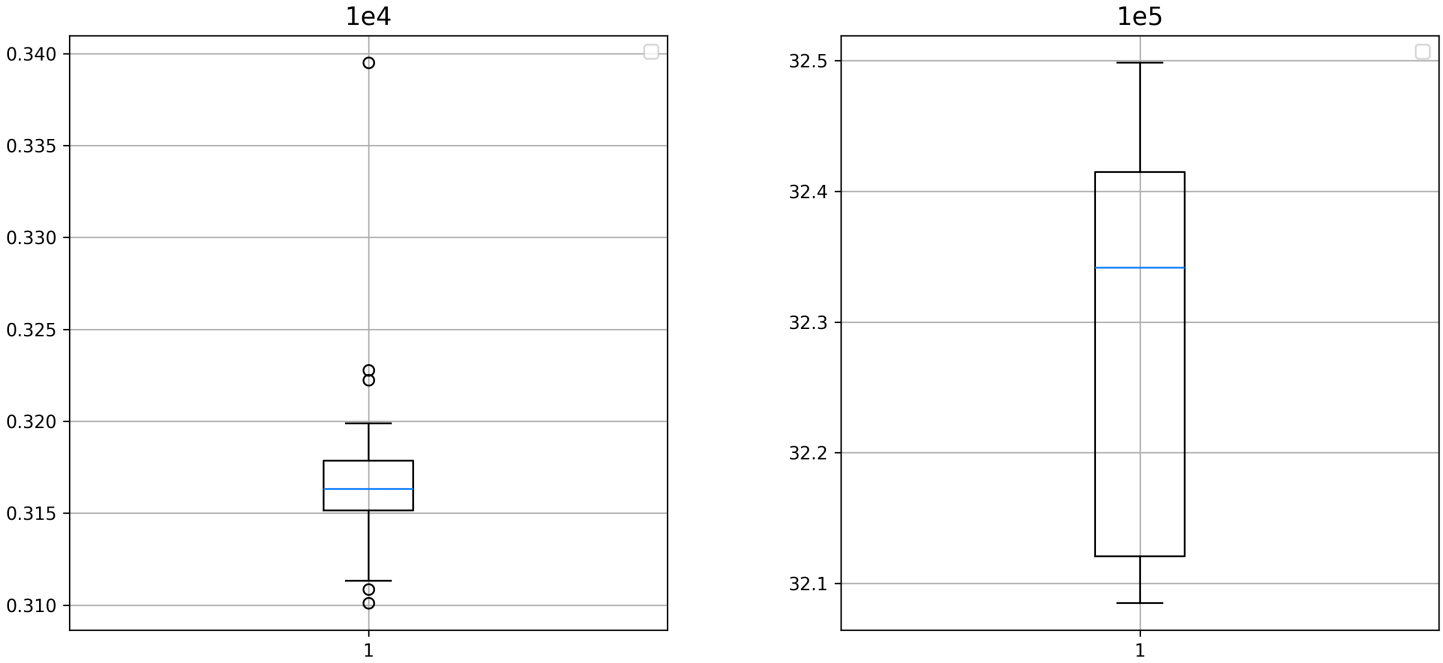

Рисунок 4.1.2 Стабильность Odd-Even Sort

2. Quick Sort

Теоретически заданная сложность задачи составляет O(NlogN). Для тестирования алгоритма была собрана статистика, приведенная в таблице №2.

Таблица №2 - Подсчёт сложности реализованного алгоритма

Размер входного набора	1000	10000	100000	1000000
Время выполнения программы, с	0.0002	0.002	0.013	0.16
O(NlogN), с	0.0001	0.001	0.013	0.16

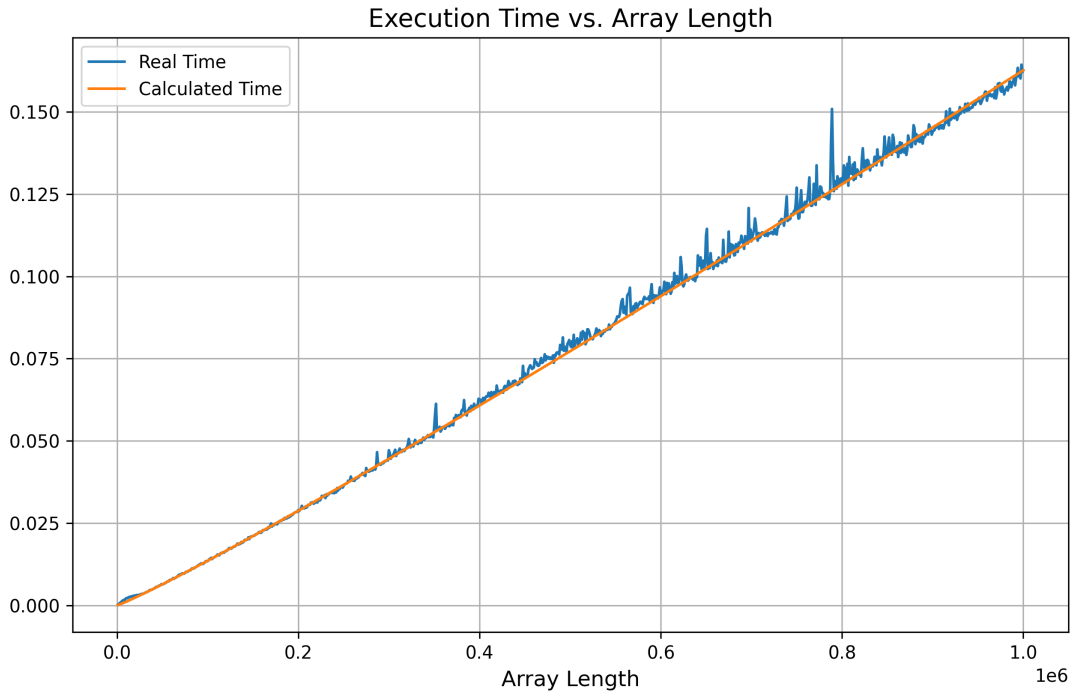

Рисунок 4.2.1 Теоретическое и реальное время Quick Sort

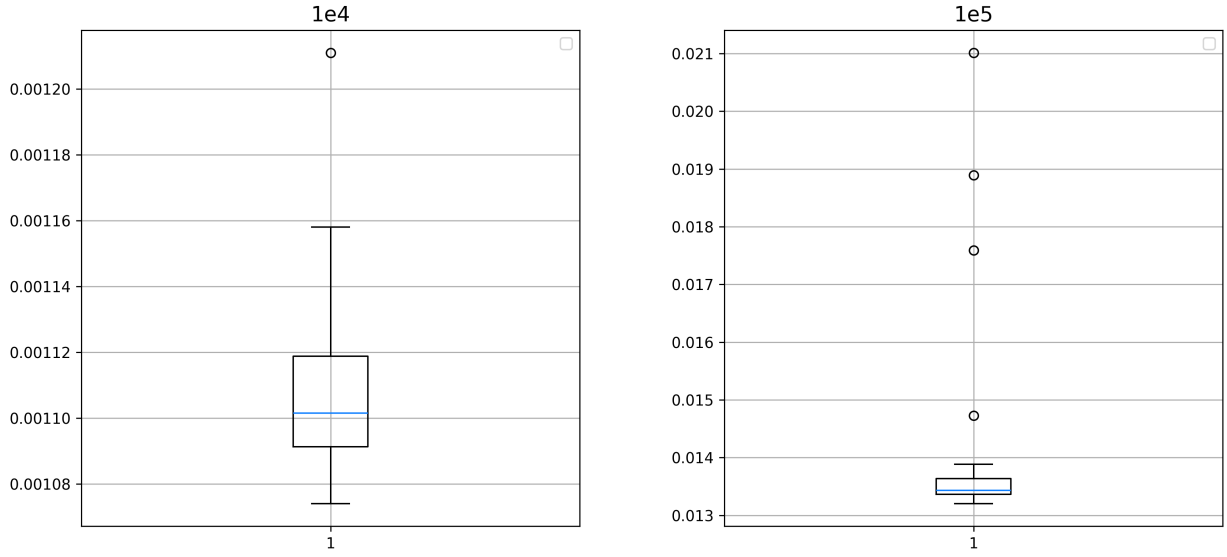

Рисунок 4.2.2 Стабильность Quick Sort

3. Сортировка подсчетом

Теоретически заданная сложность задачи составляет O(N+k). Для тестирования алгоритма была собрана статистика, приведенная в таблице №3.

Таблица №3 - Подсчёт сложности реализованного алгоритма

Размер входного набора	1000	10000	100000	1000000
Время выполнения программы, с	0.0002	0.0004	0.001	0.0097
O(N+k), с	0.0001	0.0002	0.001	0.0097

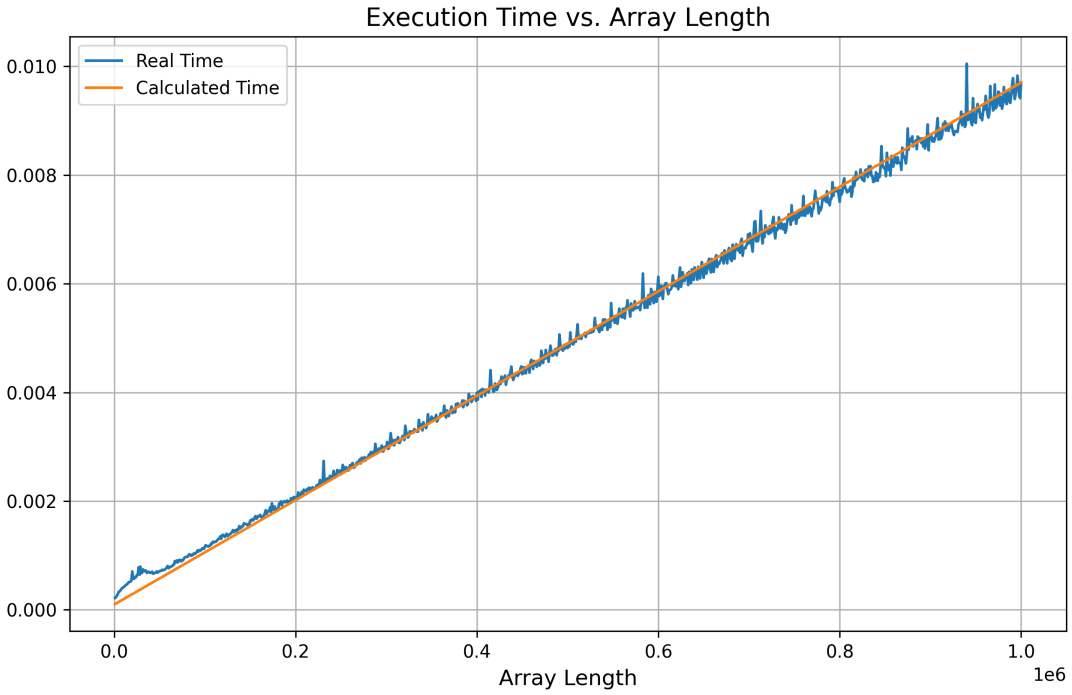

Рисунок 4.3.1 Теоретическое и реальное время Сортировки Подсчетом

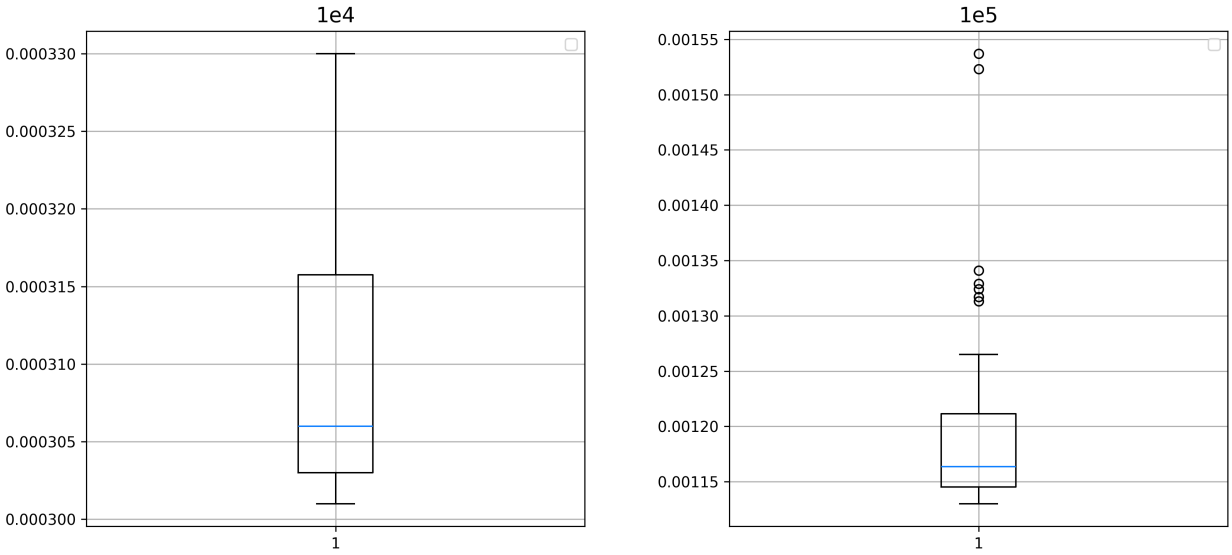

Рисунок 4.3.2 Стабильность Сортировки Подсчетом

5. Заключение

В ходе работы были реализованы три алгоритма сортировки с различной временной и пространственной сложностью. Проведенное тестирование подтвердило теоретические оценки. Сортировка Подсчетом продемонстрировал лучшую производительность в большинстве случаев, в то время как Odd-Even Sort оказался наименее быстрым. Самым оптимальным по времени и памяти оказался Quick Sort, который может работать не только с числами (в отличие от Сортировки Подсчетом), но и с любыми сравнимыми данными, также временная и пространственная сложность является приемлемой для большинства задач, что делает этот алгоритм наиболее универсальным.

6. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла main5.1.2.cpp

```
#include <iostream>
#include <random>
#include <ctime>
#include <vector>

// using namespace std;



void odd_even_sort(std::vector<int>& array) {
    bool is_sorted = false;
    while (!is_sorted) {
        is_sorted = true;
        for (int i = 0; i < array.size() - 1; i += 2) {
            if (array[i] > array[i + 1]) {
                std::swap(array[i], array[i + 1]);
```

```
                is_sorted = false;
            }
        }
        for (int i = 1; i < array.size() - 1; i += 2) {
            if (array[i] > array[i + 1]) {
                std::swap(array[i], array[i + 1]);
                is_sorted = false;
            }
        }
    }
}


void test(int len, int difference) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::vector<int> arr(len, 0);
    for (int i = 0; i < arr.size(); i++) {
        arr[i] = gen() % difference;
    }
    clock_t start = clock();
    odd_even_sort(arr);
    clock_t end = clock();
    std::cout << "Len: " << len << " Time: " << (long double)(end - start) / (long
double)(CLOCKS_PER_SEC) << "\n";
    bool ok = true;
    for (int i = 0; i < arr.size() - 1; i++) {
        if (arr[i + 1] < arr[i]) {
            ok = false;
            break;
```

```
        }
    }
    if (ok == true) {
        std::cout << "OK\n";
    } else {
        std::cout << "Test Failed\n";
    }

}


int main() {
    test(10000, 1000000);
    test(20000, 1000000);
    test(40000, 1000000);
    std::vector<int> arr = {1, 3, 4, 10, 2};
    odd_even_sort(arr);
    for (int i = 0; i < arr.size(); i++) {
        std::cout << arr[i] << " ";
    }
    return 0;
}
```

Листинг кода файла main5.2.cpp

```
#include <iostream>
#include <random>
#include <ctime>
#include <vector>

// using namespace std;
```

```
void quick_sort(std::vector<int>& arr, int left, int right) {
    if (left >= right) return;

    int pivot = arr[(left + right) / 2];
    int i = left, j = right;

    while (i <= j) {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;

        if (i <= j) {
            std::swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }
    quick_sort(arr, left, j);
    quick_sort(arr, i, right);
}

void test(int len, int difference) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::vector<int> arr(len, 0);
    for (int i = 0; i < arr.size(); i++) {
        arr[i] = gen() % difference;
    }
    clock_t start = clock();
    quick_sort(arr, 0, arr.size() - 1);
    clock_t end = clock();
```

```
    std::cout << "Len: " << len << " Time: " << (long double)(end - start) / (long
double)(CLOCKS_PER_SEC) << "\n";
    bool ok = true;
    for (int i = 0; i < arr.size() - 1; i++) {
        if (arr[i + 1] < arr[i]) {
            ok = false;
            break;
        }
    }
    if (ok == true) {
        std::cout << "OK\n";
    } else {
        std::cout << "Test Failed\n";
    }

}

int main() {
    test(1000000, 1000000);
    test(2000000, 1000000);
    test(4000000, 1000000);
    std::vector<int> arr = {1, 3, 4, 10, 2};
    quick_sort(arr, 0, arr.size() - 1);
    for (int i = 0; i < arr.size(); i++) {
        std::cout << arr[i] << " ";
    }
    return 0;
}
```

Листинг кода файла main5.3.cpp

```
#include <iostream>
```

```
#include <random>
#include <vector>
#include <unordered_map>

// using namespace std;



void encounter_sort(std::vector<int>& array, std::vector<int>& result) {
    int min = array[0], max = array[0];
    for (int i = 0; i < array.size(); i++) {
        if (array[i] < min) {
            min = array[i];
        }
        if (array[i] > max) {
            max = array[i];
        }
    }
    std::vector<int> counter(max - min + 1, 0);
    for (int i = 0; i < array.size(); i++) {
        counter[array[i] - min]++;
    }
    int i = 0;
    for (int j = 0; j < counter.size(); j++) {
        for (int k = 0; k < counter[j]; k++) {
            result[i] = j + min;
            i++;
        }
    }
}
```

```
void test(int len, int difference) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::vector<int> arr(len, 0);
    std::vector<int> res(arr.size(), 0);
    for (int i = 0; i < arr.size(); i++) {
        arr[i] = gen() % difference;
    }
    clock_t start = clock();
    encounter_sort(arr, res);
    clock_t end = clock();
    std::cout << "Len: " << len << " Time: " << (long double)(end - start) / (long double)(CLOCKS_PER_SEC) << "\n";
    bool ok = true;
    for (int i = 0; i < res.size() - 1; i++) {
        if (res[i + 1] < res[i]) {
            ok = false;
            break;
        }
    }
    if (ok == true) {
        std::cout << "OK\n";
    } else {
        std::cout << "Test Failed\n";
    }

}

int main() {
    test(10000000, 1000000);
```

```
    test(20000000, 1000000);
    test(40000000, 1000000);
    std::vector<int> arr = {1, 3, 4, 10, 2};
    std::vector<int> res(arr.size(), 0);
    encounter_sort(arr, res);
    for (int i = 0; i < res.size(); i++) {
        std::cout << res[i] << " ";
    }
    return 0;
}
```