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

# Отчёт по лабораторной работе № 5

# «Сортировка»

Выполнил работу

Мирасов Константин

Академическая группа J3112

Принято

Ассистент, Дунаев Максим

Санкт-Петербург

2024

## Введение

1. **Цель работы:** изучить и реализовать три алгоритма сортировки (HeapSort, BucketSort и Cocktail Shaker Sort), провести их анализ с точки зрения временной и пространственной сложности, а также провести экспериментальные измерения для оценки их производительности на массивах различного размера.
2. **Задачи работы:**

- Изучить и реализовать алгоритмы HeapSort, BucketSort и Cocktail Shaker Sort.
- Провести анализ временной и пространственной сложности каждого алгоритма.
- Провести экспериментальные замеры времени выполнения алгоритмов для различных наборов данных и построить графики зависимости времени выполнения от количества элементов.
- Сравнить результаты с теоретическими оценками и сделать выводы.

## Теоретическая подготовка

**HeapSort** — алгоритм на основе структуры данных "куча". В его основе лежит бинарная максимальная куча. Алгоритм сначала строит кучу, затем последовательно извлекает максимальные элементы для построения отсортированного массива.
*Теоретическая сложность*:

- Лучшая, средняя и худшая временная сложность: O(n * logn).
- Пространственная сложность: O(1) (алгоритм "in-place").

**BucketSort** — алгоритм, который делит массив на подмассивы-корзины, сортирует элементы в каждой корзине и объединяет их для получения отсортированного массива. Подходит для равномерно распределённых данных.
*Теоретическая сложность*:

- Лучшая и средняя временная сложность: O(n+k) где k — средний размер корзины.
- Худшая временная сложность: O(n^2) при неравномерном распределении данных.
- Пространственная сложность: O(n+m), где m — количество корзин.

**Cocktail Shaker Sort** — разновидность пузырьковой сортировки с попеременными проходами в двух направлениях. Эффективен для почти отсортированных данных.
*Теоретическая сложность*:

- Лучшая временная сложность: O(n) для уже отсортированных данных.
- Средняя и худшая временная сложность: O(n^2).
- Пространственная сложность: O(1).

**Практическая часть**

Используя C++ были написаны алгоритмы данных сортировок:

```cpp
// CocktailShakerSort (task #1)
// Суть алгоритма: улучшенная версия пузырьковой сортировки, в которой проход идет в обе стороны -
// сначала слева направо, потом справа налево. Это уменьшает количество необходимых итераций.
// Временная сложность:
//     - В лучшем случае: O(n) (если массив уже отсортирован).
//     - В среднем и худшем случаях: O(n^2).
// Пространственная сложность: O(1) - алгоритм работает на месте без дополнительной памяти.
void Sort::CocktailShaker(std::vector<int>& data) {
    bool swapped = true;
    int endVector = static_cast<int>(data.size());
    int startVector = 0;
    while (swapped) {
        swapped = false;
        // Проход слева направо
        for (int i = 0; i < endVector - 1; i++) {
            if (data[i] > data[i + 1]) {
                std::swap(data[i], data[i + 1]);
                swapped = true;
            }
        }
        if (swapped == false) break;
        --endVector;
        swapped = false;
        // Проход справа налево
        for (int i = endVector; i > startVector; --i) {
            if (data[i] < data[i - 1]) {
                std::swap(data[i], data[i - 1]);
                swapped = true;
            }
        }
        startVector++;
    }
}
```

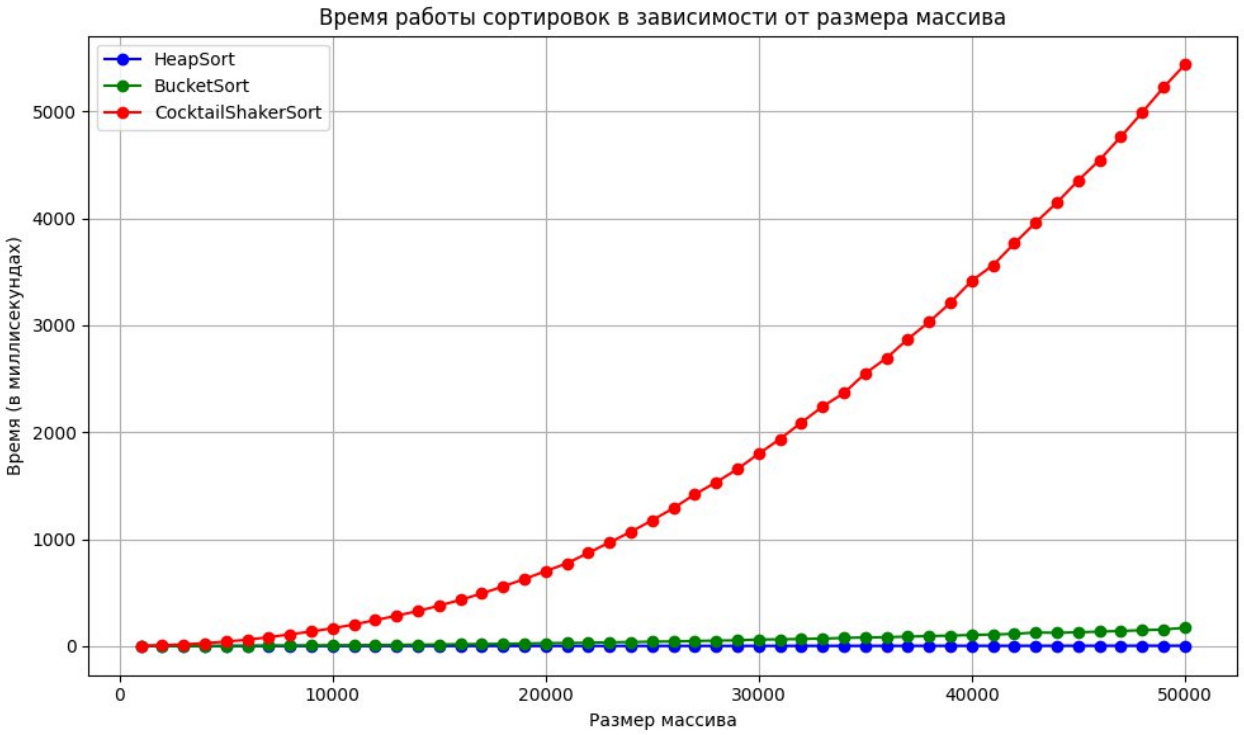
```cpp
// HeapSort (task #2)
// Суть алгоритма: сортировка на основе двоичной кучи (h
eap). Построение max-heap и затем извлечение
// максимального элемента в конец массива с последующей
перестройкой кучи.
// Временная сложность:
//     - Всегда: O(n log n).
// Пространственная сложность: O(1) - алгоритм работает
на месте.
void Sort::Heap(std::vector<int>& data) {
    const int n = static_cast<int>(data.size());
    // Построение max-heap
    for (int i = n / 2 - 1; i ⩾ 0; i--) {
        Heapify(data, n, i);
    }
    // Уменьшаем heap, перемещая максимум в конец
    for (int i = n - 1; i > 0; i--) {
        std::swap(data[0], data[i]);
        Heapify(data, i, 0);
    }
}

// Heapify (вспомогательный метод для HeapSort)
// Суть: поддержание свойства max-heap. Если родительски
й элемент меньше, чем один из потомков, меняем местами и
рекурсивно вызываем для потомка.
void Sort::Heapify(std::vector<int>& data, const int
currentSize, const int currentNode) {
    int largest = currentNode;
    const int left = 2 * currentNode + 1;
    const int right = 2 * currentNode + 2;
    if (left < currentSize && data[left] > data[largest]
) {
        largest = left;
    }
    if (right < currentSize && data[right] > data[
largest]) {
        largest = right;
    }
    if (largest ≠ currentNode) {
        std::swap(data[currentNode], data[largest]);
        Heapify(data, currentSize, largest);
    }
}
```

```cpp
// BucketSort (task #3)
// Суть алгоритма: распределяет элементы по диапазонам
(buckets), затем сортирует каждый bucket
// с помощью более простого алгоритма, например, сортиро
вки вставками.
// Временная сложность:
//     - В лучшем случае: O(n + k), где k - число корзин.
//     - В худшем случае: O(n^2) (если элементы распредел
яются неравномерно по корзинам).
// Пространственная сложность: O(n + k) - память на масс
ив данных и k корзин.
void Sort::Bucket(std::vector<int>& data, const int
numBuckets) {
    if (data.size() <= 1) return;
    const int maxValue = GetMaxValue(data);
    const int minValue = GetMinValue(data);
    std::vector<std::vector<int>> buckets(numBuckets);
    // Распределение элементов по корзинам
    for (const int element : data) {
        const int index = (element - minValue) / (
maxValue - minValue + 1) * numBuckets;
        buckets[index].push_back(element);
    }
    // Сортировка каждой корзины
    for (std::vector<int>& bucket : buckets) {
        Insertion(bucket);
    }
    // Объединение корзин в один массив
    data.clear();
    for (int i = 0; i < numBuckets; i++) {
        data.insert(data.end(), buckets[i].begin(),
buckets[i].end());
    }
}
```

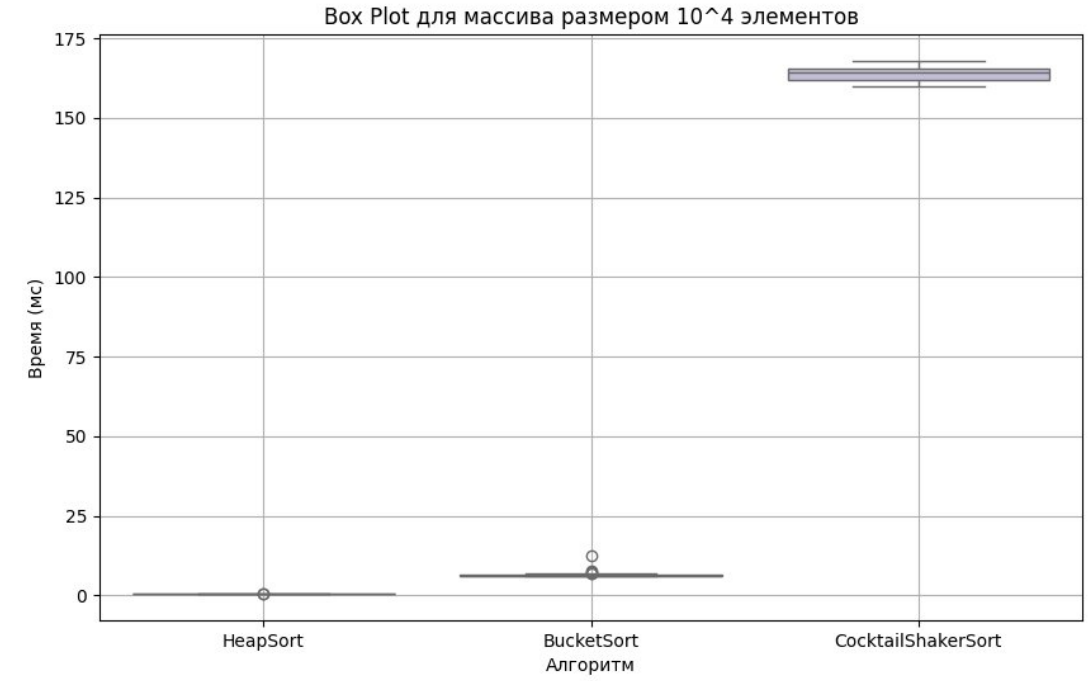
**Вспомогательный код:**

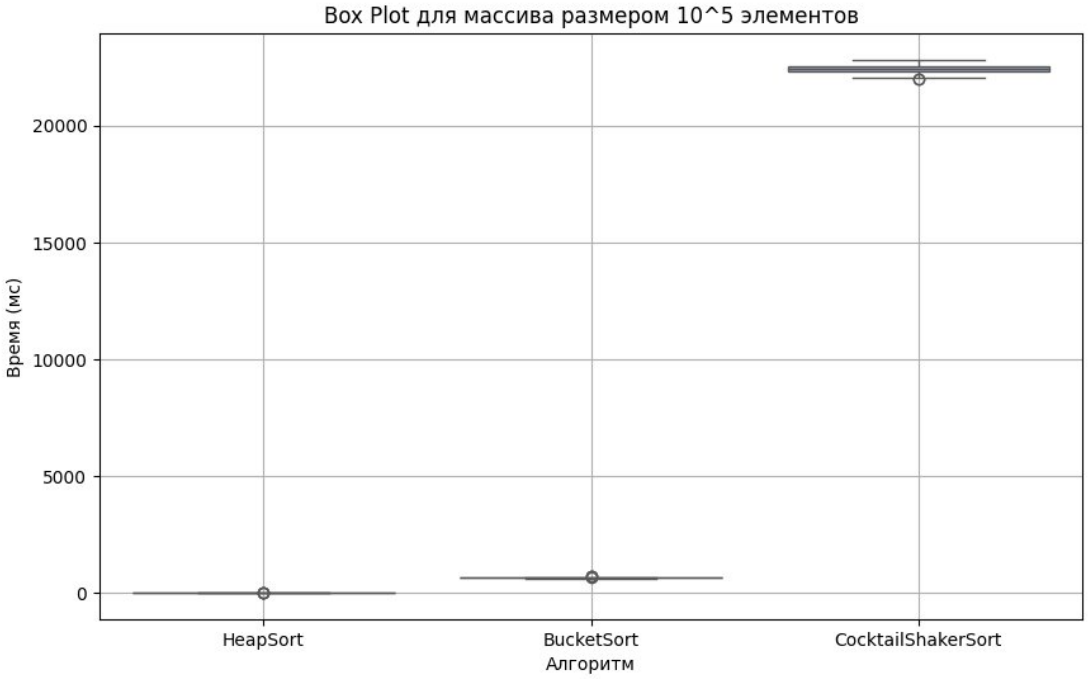
```cpp
// InsertionSort (вспомогательный метод для BucketSort)
// Суть алгоритма: последовательное вставление элементов в правильное место.
// Временная сложность:
//     - В лучшем случае: O(n) (если массив уже отсортирован).
//     - В худшем и среднем случаях: O(n^2).
// Пространственная сложность: O(1) - алгоритм работает на месте.
void Sort::Insertion(std::vector<int>& data) {
    for (int i = 1; i < data.size(); i++) {
        const int key = data[i];
        int j = i - 1;
        while (j ⩾ 0 && data[j] > key) {
            data[j + 1] = data[j];
            j = j - 1;
        }
        data[j + 1] = key;
    }
}
```

## Линейный график работы алгоритмов



# BoxPlot графики



## Unit Tests (написано на CUnit)

```
int main()
{
#ifndef NDEBUG
    if (CUE_SUCCESS != CU_initialize_registry())
        return CU_get_error();

    CU_pSuite pSuite = CU_add_suite("Sorting_Test_Suite", 0, 0);
    if (!pSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    if (!CU_add_test(pSuite, "Test CocktailShaker Sort", TestCocktailShakerSort) ||
        !CU_add_test(pSuite, "Test Heap Sort", TestHeapSort) ||
        !CU_add_test(pSuite, "Test Bucket Sort", TestBucketSort)) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();

    return CU_get_error();
#endif
    std::cout << "Release MOD..." << std::endl;
}
```

```cpp
template<typename Func>
void TestSort(Func sortFunction)
{
    std::vector<int> data;

    // Тест 1: Пустой массив
    data = {};
    sortFunction(data);
    CU_ASSERT(data.empty());

    // Тест 2: Уже отсортированный массив
    data = {1, 2, 3, 4, 5};
    sortFunction(data);
    CU_ASSERT(data == std::vector<int>({1, 2, 3, 4, 5}));

    // Тест 3: Обратно упорядоченный массив
    data = {5, 4, 3, 2, 1};
    sortFunction(data);
    CU_ASSERT(data == std::vector<int>({1, 2, 3, 4, 5}));

    // Тест 4: Массив с дублирующимися элементами
    data = {3, 1, 2, 1, 3};
    sortFunction(data);
    CU_ASSERT(data == std::vector<int>({1, 1, 2, 3, 3}));

    // Тест 5: Произвольный массив
    data = {8, -3, 7, 0, 2};
    sortFunction(data);
    CU_ASSERT(data == std::vector<int>({-3, 0, 2, 7, 8}));
}
```

**Сортировки проходят все тесты**

```
     CUnit - A unit testing framework for C - Version 3.4.4-cunity
     http://cunit.sourceforge.net/


Suite: Sorting_Test_Suite
  Test: Test CocktailShaker Sort ...passed
  Test: Test Heap Sort ...passed
  Test: Test Bucket Sort ...passed

Run Summary       -      Run    Failed  Inactive   Skipped
     Suites       :        1         0         0         0
     Asserts      :       15         0       n/a       n/a
     Tests        :        3         0         0         0

Elapsed Time: 0.000(s)
```

**Вывод:**

В ходе выполнения работы были реализованы три алгоритма сортировки: HeapSort, BucketSort и Cocktail Shaker Sort. Проведённые экспериментальные замеры подтвердили теоретические оценки временной и пространственной сложности. HeapSort продемонстрировал наиболее стабильную производительность, а BucketSort эффективно справился с равномерно распределёнными данными. Cocktail Shaker Sort показал высокую эффективность на почти отсортированных данных, однако его производительность на случайных данных заметно уступает другим алгоритмам.

Возможным направлением для будущих исследований является оптимизация каждого из алгоритмов, в частности рассмотрение многопоточной реализации для обработки больших объёмов данных.