ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

««Алгоритмы сортировки»

Выполнила работу

Шакина Анна

Академическая группа № J3110

Принято

Вершинин Владислав Константинович

Санкт-Петербург

2024

1. Введение

Цель лабораторной работы: сравнить три алгоритма сортировки: Pancake Sort, Comb Sort и Pigeonhole Sort. Для этого необходимо протестировать каждый алгоритм на массиве данных и измерить время выполнения сортировки, проанализировать его.

Задачи:

1. Изучить принцип работы каждого из алгоритмов сортировки.
2. Реализовать функции сортировки для каждого из алгоритмов на языке C++.
3. Протестировать алгоритмы на массивах размером от 1е4 до 1е6 значений, измерить время их работы.
4. Построить графики зависимости времени работы алгоритмов от длины массива, проанализировать их.
5. Построить графики box plot для каждого алгоритма, сравнить их и проанализировать.
6. Определить, в каких случаях можно использовать каждый из алгоритмов, проанализировать соответствие теоретической асимптотики с практическими результатами.
7. Теоретическая подготовка
   1. Pancake Sort (Блинная сортировка)

Алгоритм получил свое название благодаря аналогии с переворачиванием блинов на сковороде. Основная идея заключается в том, чтобы последовательно перевернуть элементы массива для достижения правильной сортировки.

1. Находим максимальный элемент в текущей части массива.
2. Переворачиваем часть массива, чтобы максимальный элемент оказался первым.
3. Переворачиваем весь массив, чтобы поставить максимальный элемент на правильное место.
4. Повторяем процесс для оставшихся элементов.

Преимущества:

* Работает хорошо для небольших наборов данных.

Недостатки:

* Очень медленный алгоритм, особенно для больших массивов.
* Неэффективен по сравнению с другими алгоритмами сортировки.
  1. Comb Sort (Сортировка расчёской)

Comb Sort — это улучшенная версия пузырьковой сортировки (Bubble Sort). Она использует стратегию уменьшения расстояния между сравниваемыми элементами, что делает ее более эффективной, чем Bubble Sort.

1. Устанавливаем начальный «шаг» (обычно равный длине массива).
2. Сравниваем элементы с определенным интервалом и меняем местами, если они не упорядочены.
3. Уменьшаем шаг и повторяем процесс до тех пор, пока шаг не станет равным 1.

Преимущества:

* Эффективнее Bubble Sort.

Недостатки:

* Может работать нестабильно для некоторых типов данных.
* Медленнее, чем многие другие алгоритмы сортировки.
  1. Pigeonhole Sort (Сортировка голубиных отверстий)

Pigeonhole Sort — это сортировка, основанная на распределении элементов по «отверстиям», представляющим возможные значения. Это эффективный алгоритм для ограниченных диапазонов значений.

1. Находим максимальный и максимальный элементы массива.
2. Создаем массив «отверстий» (векторов) для каждого возможного значения.
3. Распределяем элементы по соответствующим «отверстиям».
4. Собираем элементы из «отверстий» в правильном порядке.

Преимущества:

* Высокая эффективность для массивов с небольшим диапазоном значений.
* Линейная сложность в случае, если диапазон значений небольшой.

Недостатки:

* Требуется много дополнительной памяти для хранения «отверстий».
* Эффективен только для массивов с ограниченными значениями.

1. Реализация
   1. Используемые библиотеки:

* <iostream> — для работы с вводом и выводом данных.
* <cassert> — для проверки утверждений (aseert) во время выполнения программы.
* <vector> — для хранения массивов данных и результатов сортировки.
  1. Разработка алгоритма

Для хранения промежуточных данных были выбраны векторы (std::vector), которые позволяют динамически изменять размер и обеспечивают эффективное управление памятью.

* + 1. Pancake Sort

Реализована функция, которая переворачивает часть массива с 0 по (i-1):

void **flip**(std::**vector**<int> &arr, int i)

{

    int temp;

    int start = 0;

    while (start < i)

    {

        temp = arr**[**start**]**;

        arr**[**start**]** = arr**[**i**]**;

        arr**[**i**]** = temp;

        start++;

        i--;

    }

}

Реализована функция, которая находит индекс максимального элемента в части массива с 0 по (i-1):

int **find\_max\_index**(const std::**vector** <int> &arr, int i)

{

    int max\_i = 0;

    for (int index = 0; index < i; index++)

    {

        if (arr**[**index**]** > arr**[**max\_i**]**)

        {

            max\_i = index;

        }

    }

    return max\_i;

}

Реализована основная функция, которая сортирует массив:

void **pancake\_sort**(std::**vector**<int> &arr)

{

    for (int curr\_size = arr.**size**(); curr\_size > 1; curr\_size--)

    {

        int max\_index = **find\_max\_index**(arr, curr\_size);

        if (max\_index != curr\_size - 1)

        {

**flip**(arr, max\_index);

**flip**(arr, curr\_size - 1);

        }

    }

}

1. Вызывается функция, которая находит индекс максимального элемента
2. С помощью функции flip этот элемент перемещается в начало массива
3. С помощью функции flip этот элемент перемещается в конец неотсортированной части массива
4. Предыдущие шаги повторяются для всех элементов массива

Написаны тесты, которые проверяют, работает ли функция в лучшем, среднем и худшем случах. Тесты реализованы с использованием библиотеки <cassert>. Лучшим случаем для алгоритма является полностью отсортированный массив, средним и худшим – массив со случайными значениями.

void **test\_best\_case**()

{

    std::**vector**<int> arr = {1, 2, 3, 4, 5};

**pancake\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_average\_case**()

{

    std::**vector**<int> arr = {3, 1, 4, 2, 5};

**pancake\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_worst\_case**()

{

    std::**vector**<int> arr = {5, 4, 3, 2, 1};

**pancake\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

Написаны тесты, которые проверяют, работает ли функция в крайних случах: на пустом массиве, на массиве с одним элементом, на массиве с одинаковыми элементами. Тесты реализованы с использованием библиотеки <cassert>.

void **test\_empty\_array**()

{

    std::**vector** <int> arr;

**pancake\_sort**(arr);

}

void **test\_single\_element**()

{

    std::**vector**<int> arr = {1};

**pancake\_sort**(arr);

**assert**(arr**[**0**]** == 1);

}

void **test\_all\_elements\_same**()

{

    std::**vector**<int> arr = {7, 7, 7, 7, 7};

**pancake\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** == arr**[**i + 1**]**);

    }

}

* + 1. Comb Sort

Реализована функция для вычисления шага, с которым основная функция сортировки будет сравнивать элементы массива. По умолчанию коэффициент сжатия берётся 1.3, чтобы обеспечить баланс между скоростью и точностью.

int **get\_next\_gap**(int gap)

{

    gap = (gap \* 10) / 13;

    if (gap < 1)

    {

        return 1;

    }

    return gap;

}

Реализована основная функция, которая сортирует массив:

void **comb\_sort**(std::**vector**<int> &arr)

{

    if (arr.**empty**())

    {

        return;

    }

    int gap = arr.**size**();

    bool swapped = true;

    while (gap != 1 || swapped == true)

    {

        gap = **get\_next\_gap**(gap);

        swapped = false;

        for (int i = 0; i < arr.**size**() - gap; i++)

        {

            if (arr**[**i**]** > arr**[**i + gap**]**)

            {

                int temp = arr**[**i**]**;

                arr**[**i**]** = arr**[**i + gap**]**;

                arr**[**i + gap**]** = temp;

                swapped = true;

            }

        }

    }

}

1. Сначала шаг становится равным длине массива, а флаг swapped – true, чтобы алгоритм выполнился хотя бы 1 раз.
2. Вычисляется новый шаг, сбрасывается флаг swapped, чтобы проверить, произошли ли перестановки в этом проходе.
3. С помощью цикла for сравниваются все элементы с текущим шагом, сортируются. Если выполнена перестановка, флаг swapped становится равным true.
4. Продолжается выполнение цикла до тех пор, пока шаг не станет равным 1 или не произойдёт ни одной перестановки.

Написаны тесты, которые проверяют, работает ли функция в лучшем, среднем и худшем случах. Тесты реализованы с использованием библиотеки <cassert>. Лучшим случаем для алгоритма является полностью отсортированный массив, средним и худшим – массив, отсортированный в обратном порядке.

void **test\_best\_case**()

{

    std::**vector**<int> arr = {1, 2, 3, 4, 5};

**comb\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_average\_case**()

{

    std::**vector**<int> arr = {3, 1, 4, 2, 5};

**comb\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_worst\_case**()

{

    std::**vector**<int> arr = {5, 4, 3, 2, 1};

**comb\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

Написаны тесты, которые проверяют, работает ли функция в крайних случах: на пустом массиве, на массиве с одним элементом, на массиве с повторяющимися элементами. Тесты реализованы с использованием библиотеки <cassert>.

void **test\_empty\_vector**()

{

    std::**vector**<int> arr;

**comb\_sort**(arr);

**assert**(arr.**empty**());

}

void **test\_single\_element**()

{

    std::**vector**<int> arr = {97};

**comb\_sort**(arr);

**assert**(arr**[**0**]** == 97);

}

void **test\_with\_duplicates**()

{

    std::**vector**<int> arr = {4, 5, 4, 3, 2, 2};

**comb\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

* + 1. Pigeonhole Sort

1. Реализован поиск минимального и максимального элемента в массиве:

int min\_value = arr**[**0**]**, max\_value = arr**[**0**]**;

    for (int i = 1; i < n; i++)

    {

        if (arr**[**i**]** < min\_value)

            min\_value = arr**[**i**]**;

        if (arr**[**i**]** > max\_value)

            max\_value = arr**[**i**]**;

    }

1. Вычисление диапазона значений в массиве:

int range = max\_value - min\_value + 1;

1. Создание вектора векторов, размер которого равен диапазону. Каждый вектор представляет собой «отверстие», содержащее соответствующие элементы.

std::**vector**<std::**vector**<int>> **holes**(range);

1. Проход по входному массиву и размещение каждого элемента в соответствующее ему отверстие

for (int i = 0; i < n;

        holes**[**arr**[**i**]** - min\_value**]**.**push\_back**(arr**[**i**]**);

1. Проход по всем отверстиям одно за другим. для каждого отверстия надо взять его элементы и поместить в массив, который становится отсортированным.

    int index = 0;

    for (int i = 0; i < range; i++)

    {

        for (int value : holes**[**i**]**)

            arr**[**index++**]** = value;

    }

Написаны тесты, которые проверяют, работает ли функция в лучшем, среднем и худшем случах. Тесты реализованы с использованием библиотеки <cassert>. Лучшим случаем для алгоритма является полностью отсортированный массив, средним и худшими – массив, отсортированный в обратном порядке, массив с большим диапазоном значений.

void **test\_best\_case**()

{

    std::**vector**<int> arr = {1, 2, 3, 4, 5};

**pigeonhole\_sort**(arr);

    for (**size\_t** i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_average\_case**()

{

    std::**vector**<int> arr = {3, 1, 4, 2, 5};

**pigeonhole\_sort**(arr);

    for (**size\_t** i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_worst\_case**()

{

    std::**vector**<int> arr = {5, 4, 3, 2, 1};

**pigeonhole\_sort**(arr);

    for (**size\_t** i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_large\_range**()

{

    std::**vector**<int> arr = {1000, -1000, 500, -500, 0};

**pigeonhole\_sort**(arr);

**assert**(arr**[**0**]** == -1000 && arr**[**1**]** == -500 && arr**[**2**]** == 0 && arr**[**3**]** == 500 && arr**[**4**]** == 1000);

}

Написаны тесты, которые проверяют, работает ли функция в крайних случах: на пустом массиве, на массиве с одним элементом, на массиве с двумя отсортированными и неотсортированными элементами, на массиве с отрицательными числами. Тесты реализованы с использованием библиотеки <cassert>.

void **test\_empty\_array**()

{

    std::**vector**<int> arr; *// Пустой вектор*

**pigeonhole\_sort**(arr);

**assert**(arr.**empty**()); *// Проверка, что вектор пустой*

}

void **test\_single\_element**()

{

    std::**vector**<int> arr = {42};

**pigeonhole\_sort**(arr);

**assert**(arr**[**0**]** == 42);

}

void **test\_two\_elements\_sorted**()

{

    std::**vector**<int> arr = {1, 2};

**pigeonhole\_sort**(arr);

**assert**(arr**[**0**]** == 1 && arr**[**1**]** == 2);

}

void **test\_two\_elements\_unsorted**()

{

    std::**vector**<int> arr = {2, 1};

**pigeonhole\_sort**(arr);

**assert**(arr**[**0**]** == 1 && arr**[**1**]** == 2);

}

void **test\_negative\_numbers**()

{

    std::**vector**<int> arr = {-3, -1, -4, -2, -5};

**pigeonhole\_sort**(arr);

    for (**size\_t** i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

1. Экспериментальная часть
   1. Подсчёт по памяти

* Pancake Sort: дополнительной памяти, кроме фиксированного набора переменных, не используется, поэтому пространственная сложность – O(1).
* Comb Sort: дополнительной памяти, кроме фиксированного набора переменных, не используется, поэтому пространственная сложность – O(1).
* Pigeonhole Sort: фиксированный набор переменных занимает O(1) памяти, вектор векторов holes занимает O(range), где range – диапазон входных значений, так как мы выделяем память для range отдельных векторов. Каждый вектор внутри holes занимает O(N) памяти, где N – количество элементов во входном массиве. Таким образом, всего используется O(N + range) памяти.
  1. Подсчёт асимптотики
* Pancake Sort:
  + Лучший случай: O(N) — если массив уже отсортирован.
  + Средний и худший случаи: O(N^2) — массив отсортирован в случайном порядке.
* Comb Sort:
  + Лучший случай: O(N\*log N) — если массив уже отсортирован.
  + Средний случай: O(N^2) — средняя производительность.
  + Худший случай: O(N^2) — массив отсортирован в обратном порядке.
* Pigeonhole Sort:
  + Лучший случай: O(N) — если диапазон значений небольшой.
  + Средний и худший случаи: O(N + range), где range — диапазон значений.
  1. Графики работы алгоритмов, для массивов от 1000 до 1e6 элементов с шагом в 1000

Для Pancake Sort построен график работы на массивах от 1000 до 129000 элементов с шагом в 1000, на массивах с 230\_000, 330\_000, 430\_000, 500000, 750\_000, 1\_000\_000 элементами.

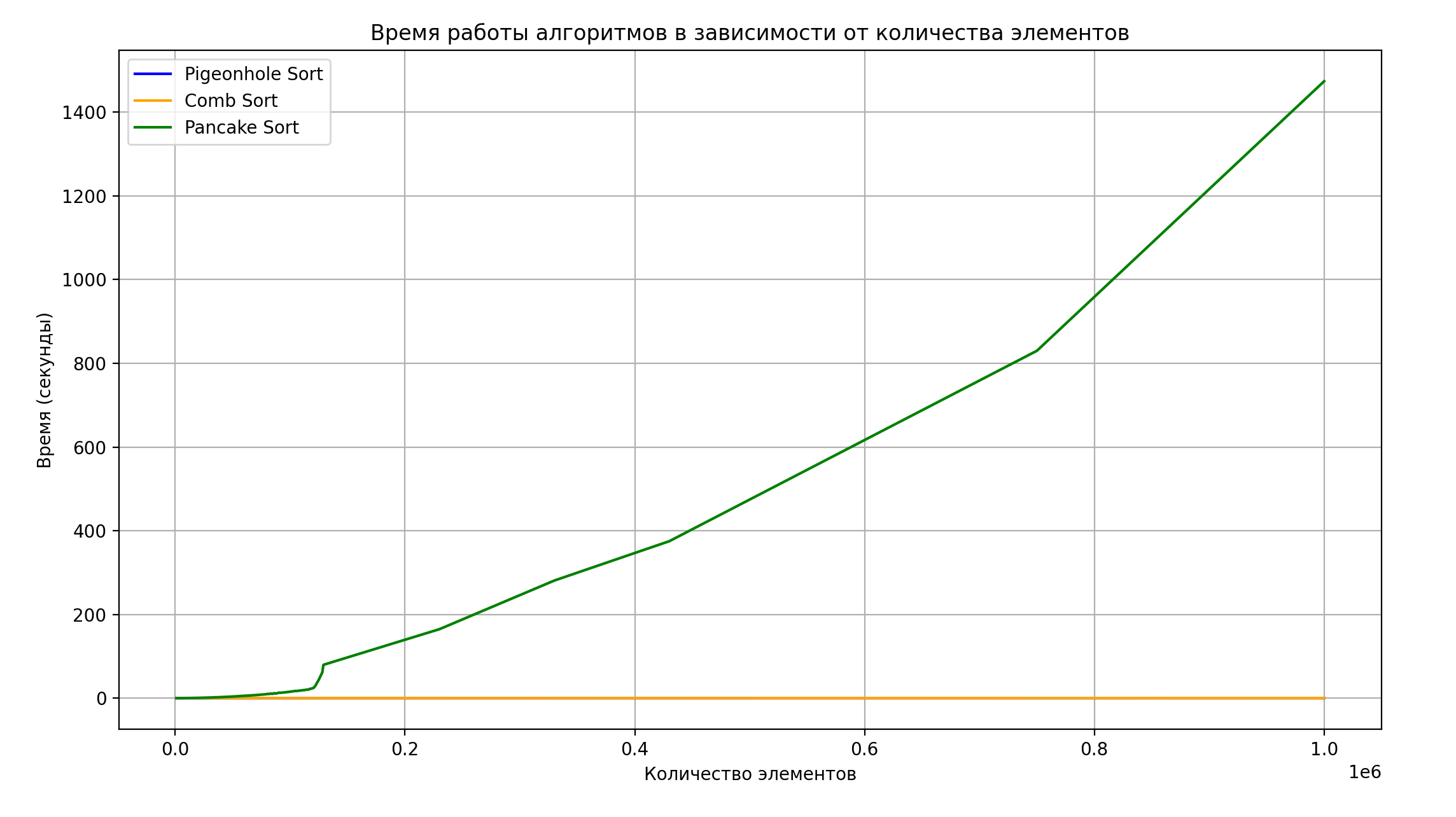


Рис. 1. Линейный график работы алгоритмов в зависимости от количества элементов для трёх алгоритмов

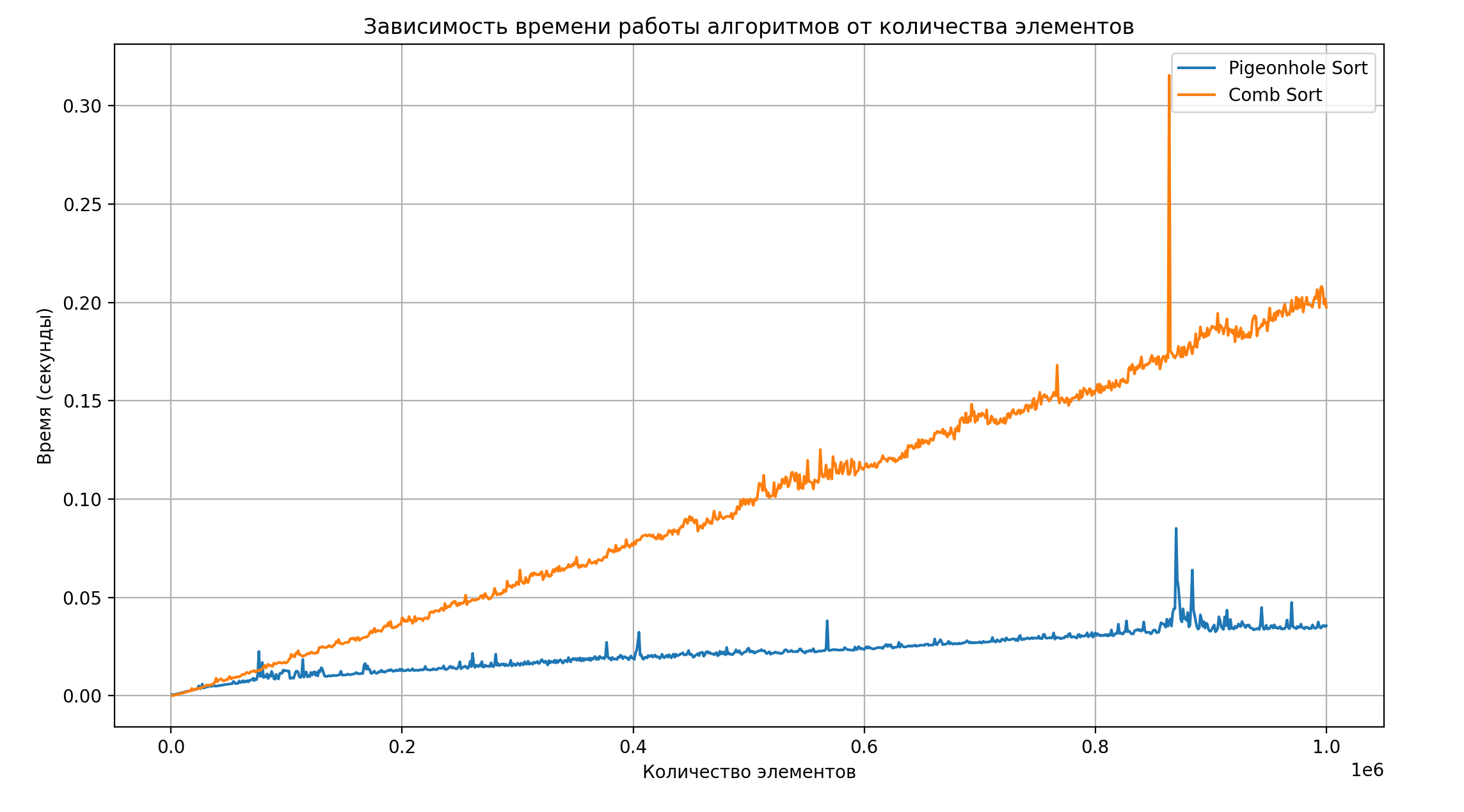


Рис. 2. Линейный график работы алгоритмов в зависимости от количества элементов для двух алгоритмов

*Анализ графиков.*

На рис. 1 заметно, что у Pancake Sort резко возрастает время выполнения, что связано с худшей сложностью O(N^2), так как требуется больше времени для сортировки массива. По сравнению с Comb Sort и Pigeonhole Sort, Pancake Sort требует огромное количество времени для сортировки на больших массивах, в связи с чем для анализа графиков Comb Sort и Pigeonhole Sort был сделан отдельный график.

На рис. 2 можно увидеть, что Pigeonhole Sort является более быстрым и стабильным алгоритмом, чем Comb Sort. Выбросы есть у обоих графиков, но у Pigeonhole Sort они связаны с большим диапазонов входных значений, а у Comb Sort могут быть связаны с неравномерным распределением данным во входном массиве. Построенные графики соответствуют теоретической сложности – O(N) и O(N\*log N).

* 1. Box plot графики для времени работы алгоритмов с числом элементов 1e4 и 1e5

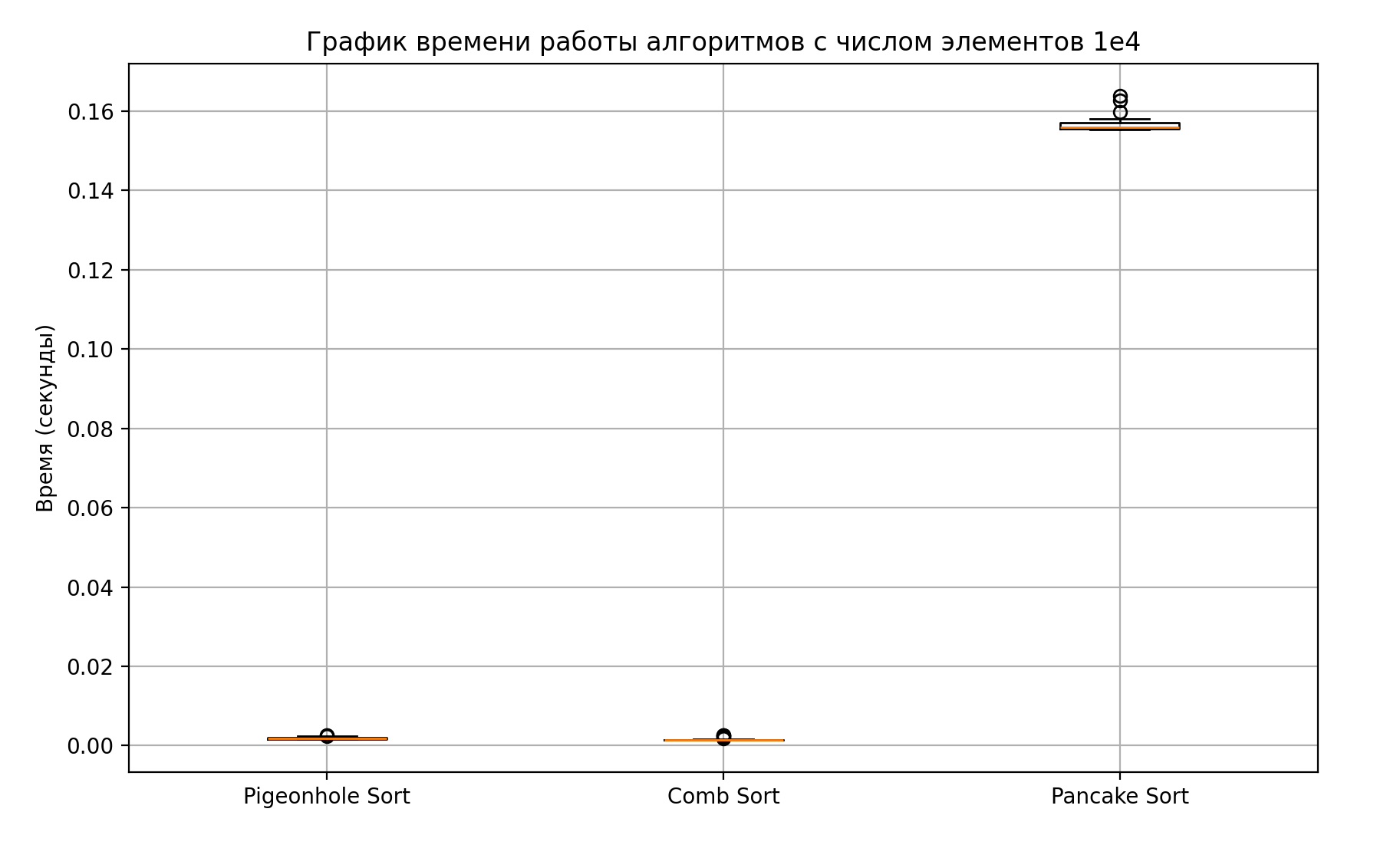


Рис. 3. График времени работы алгоритмов с числом элементов 1e4

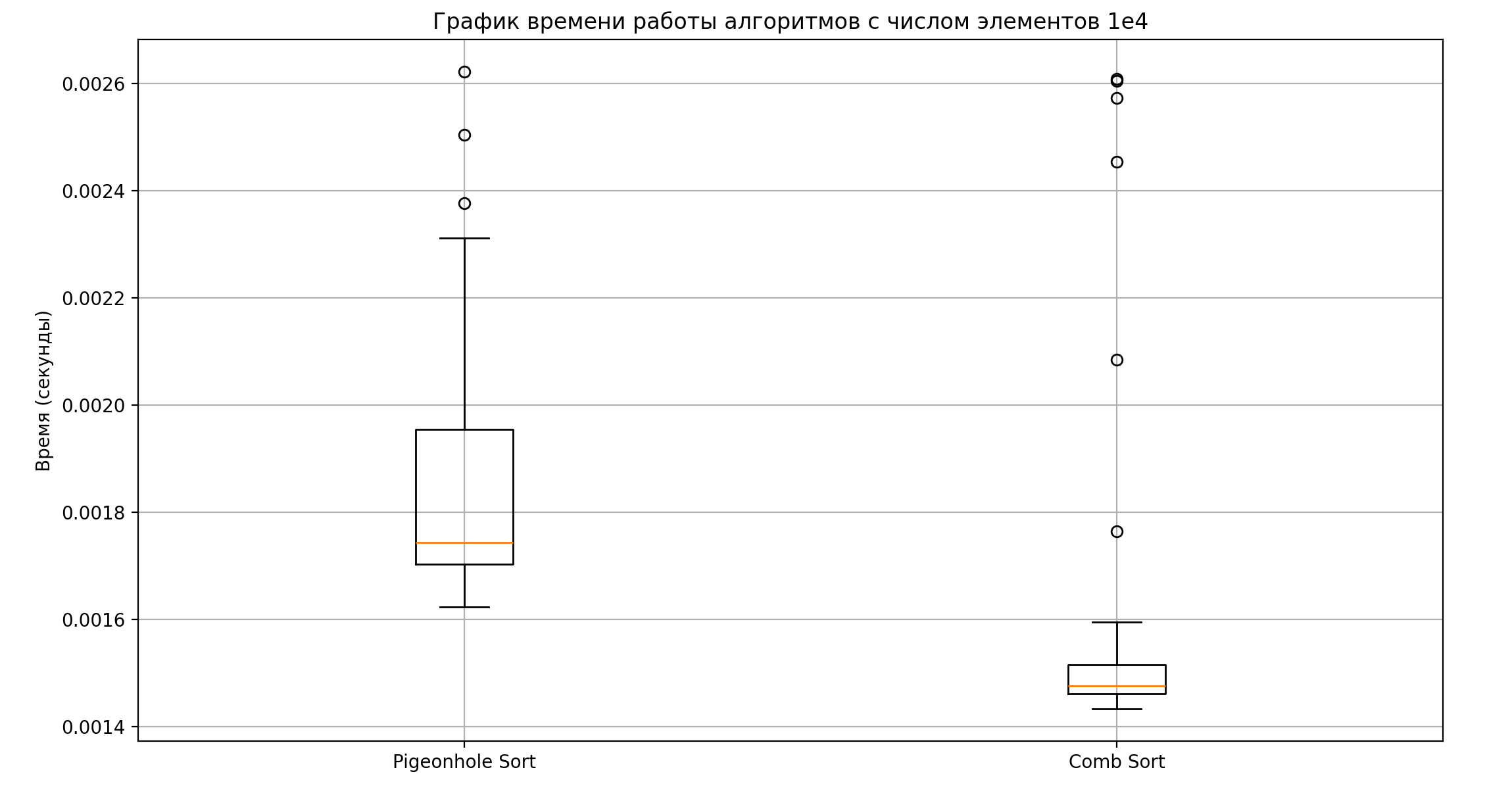


Рис. 4. График времени работы алгоритмов с числом элементов 1e4

*Анализ графиков*

Pancake Sort: на Рис. 3 видно, что время выполнения значительно выше по сравнению с другими алгоритмами из-за своей неэффективности. У алгоритма возникает много выбросов, поскольку он может требовать много переворотов для массивов. Разброс данных большой, что указывает на нестабильность алгоритма.

С помощью графика на Рис. 3 можно убедиться в том, что Pigeonhole Sort и Comb Sort – более стабильные алгоритмы, чем Pancake Sort, но из-за того, что они работают на массиве с числом элементов 1e4 в несколько раз быстрее, чем Pancake Sort, следует рассмотреть Рис. 4.

Pigeonhole Sort: если данные имеют широкий диапазон значений или много уникальных элементов, могут возникнуть выбросы, так как алгоритм будет создавать много пустых «отверстий». В этом можно убедиться на графике: все выбросы располагаются выше «ящика».

Comb Sort: время выполнения ниже, чем у Pigeonhole Sort при широком диапазоне значений. Если входные данные имеют неравномерное распределение (например, много одинаковых значений или значения, расположенные в определённом порядке), это может привести к неожиданно высокому или низкому времени выполнения, следовательно, на графиках появляются выбросы.

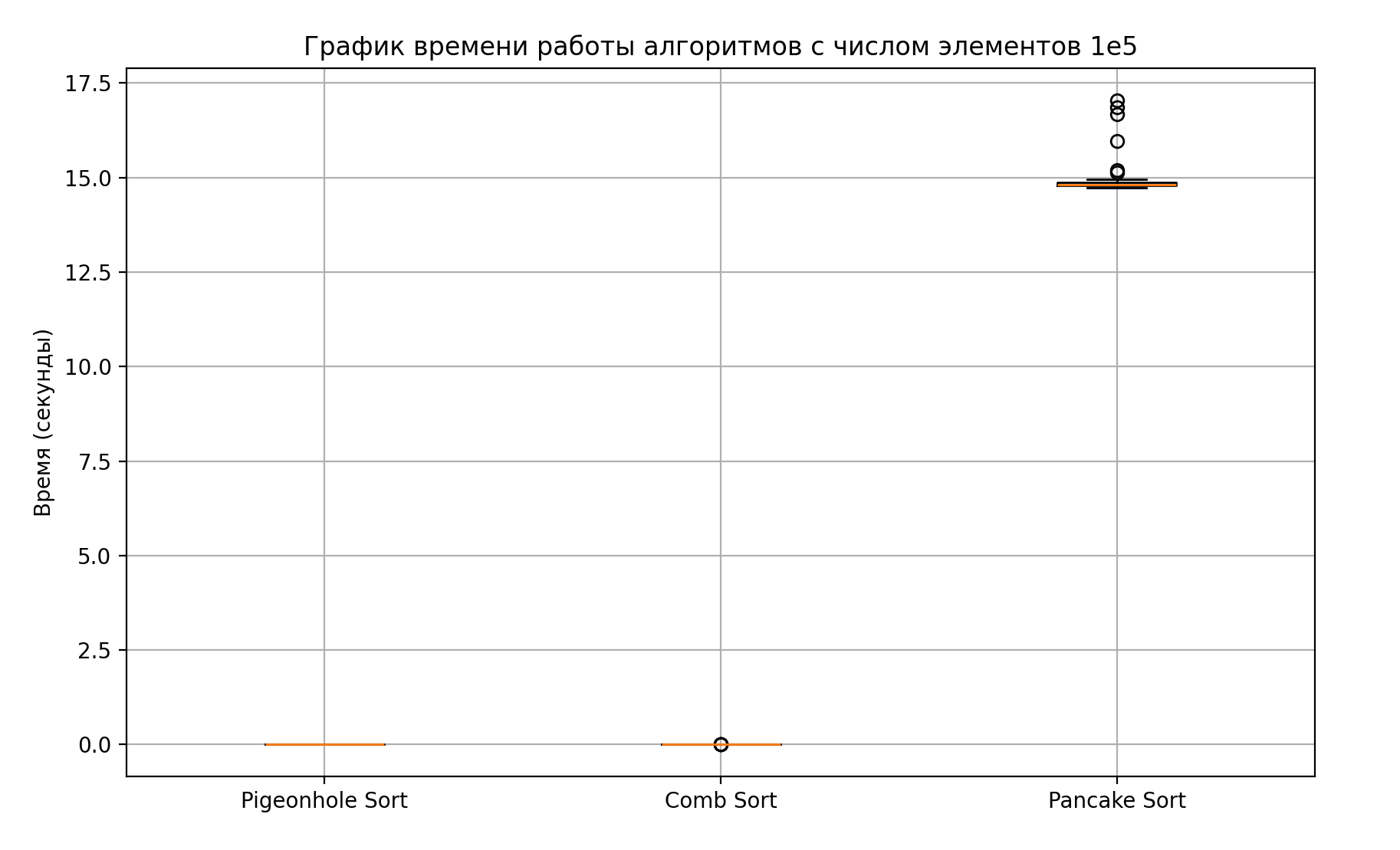


Рис. 5. График времени работы алгоритмов с числом элементов 1e5

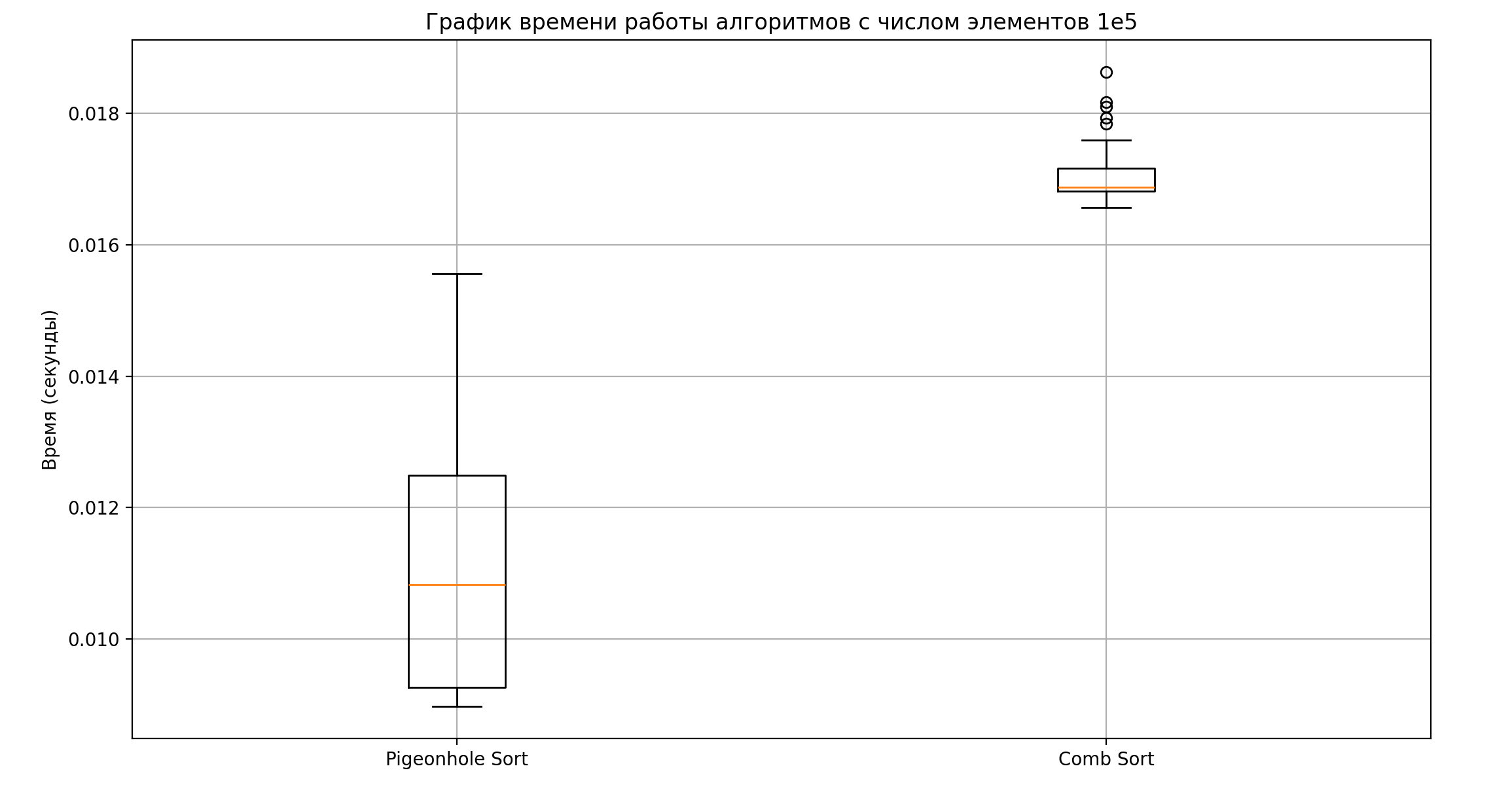


Рис. 6. График времени работы алгоритмов с числом элементов 1e5

*Анализ графиков*

Pancake Sort: на Рис. 5 видно, что время выполнения продолжает увеличиваться, как и количество выбросов, но график похож на Рис. 3, что говорит о том, что алгоритм медленно работает как на массивах с числом элементов 1e4, так и на массивах с числом элементов 1e5.

Pigeonhole Sort: на Рис. 6 заметно, что время выполнения алгоритма становится меньше, чем у Comb Sort. Это происходит из-за того, что время выполнения алгоритма начинает зависеть больше от количества входных данных, чем от диапазона их значений. Однако сохраняется нестабильность: на массивах с большим количеством одинаковых значений алгоритм работает долго.

Comb Sort: видно, что алгоритм является более стабильным, чем Pigeonhole Sort, но время выполнения становится больше.

1. Заключение

В ходе выполнения работы мною были реализованы 3 алгоритма сортировки: Pancake Sort (Блинная сортировка), Comb Sort (Сортировка расчёской), Pigeonhole Sort (Сортировка голубиных отверстий). Цель работы была достигнута путем тестирования каждой из сортировок на массивах различной длины, с анализом временных характеристик и производительности. Полученные результаты также совпадают с теоретическими оценками сложности алгоритма.

В измерениях времени выполнения по каждому алгоритму было отмечено, что сортировка Pancake Sort показала наихудшие результаты на больших объемах данных, в то время как Comb Sort и Pigeonhole Sort продемонстрировали более стабильную производительность. Это подтверждает, что выбор алгоритма сортировки должен основываться не только на теоретических характеристиках, но и на практическом применении в зависимости от конкретных условий.

Pancake Sort не является эффективным для практического использования на больших объемах данных, но может быть полезен в задачах, где требуется демонстрация принципов сортировки или для небольших наборов данных.

Comb Sort является улучшенной версией Bubble Sort и может быть использован для сортировки массивов средней величины. Его простота реализации и более высокая эффективность по сравнению с простыми алгоритмами сортировки делают его подходящим выбором для приложений с умеренными требованиями к производительности.

Pigeonhole Sort эффективен для сортировки целых чисел с ограниченным диапазоном значений. Он может быть применен в ситуациях, когда известны минимальные и максимальные значения элементов массива, что позволяет быстро распределить элементы по "отверстиям".

1. Приложения

Листинг кода файла pancake\_sort.cpp

#include <iostream>

#include <cassert>

#include <vector>

*// переворачивает arr[0..i]*

void **flip**(std::**vector**<int> &arr, int i)

{

    int temp; *// O(1): создание временной переменной*

    int start = 0; *// O(1): создание счетчика*

    while (start < i) *// O(i): цикл выполняется i/2 раз*

    {

        temp = arr**[**start**]**; *// O(1): присваивание значения*

        arr**[**start**]** = arr**[**i**]**; *// O(1): присваивание значения*

        arr**[**i**]** = temp; *// O(1): присваивание значения*

        start++; *// O(1): увеличиваем счетчик*

        i--; *// O(1): уменьшаем индекс*

    }

}

*// находит индекс максимального элемента в arr[0..i-1]*

int **find\_max\_index**(const std::**vector** <int> &arr, int i)

{

    int max\_i = 0; *// O(1): инициализация переменной для хранения индекса максимального элемента*

    for (int index = 0; index < i; index++) *// O(i): цикл выполняется i раз*

    {

        if (arr**[**index**]** > arr**[**max\_i**]**) *// O(1): сравнение двух значений*

        {

            max\_i = index; *// O(1): обновление индекса максимального элемента*

        }

    }

    return max\_i; *// O(1)*

}

*// основная функция, которая сортирует данный массив с помощью операций переворота*

void **pancake\_sort**(std::**vector**<int> &arr)

{

    for (int curr\_size = arr.**size**(); curr\_size > 1; curr\_size--) *// O(n^2): внешний цикл выполняется n раз*

    {

        int max\_index = **find\_max\_index**(arr, curr\_size); *// O(curr\_size): вызов функции поиска максимального элемента*

        if (max\_index != curr\_size - 1) *// O(1): проверка необходимости переворота*

        {

**flip**(arr, max\_index); *// O(max\_index): первый переворот*

**flip**(arr, curr\_size - 1); *// O(curr\_size): второй переворот*

        }

    }

}

*// тесты*

void **test\_best\_case**()

{

    std::**vector**<int> arr = {1, 2, 3, 4, 5};

**pancake\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_average\_case**()

{

    std::**vector**<int> arr = {3, 1, 4, 2, 5};

**pancake\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

*// крайние случаи*

void **test\_empty\_array**()

{

    std::**vector** <int> arr;

**pancake\_sort**(arr);

*// пустой массив остается пустым, программа работает без ошибок*

}

void **test\_single\_element**()

{

    std::**vector**<int> arr = {1};

**pancake\_sort**(arr);

**assert**(arr**[**0**]** == 1);

}

void **test\_all\_elements\_same**()

{

    std::**vector**<int> arr = {7, 7, 7, 7, 7};

**pancake\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** == arr**[**i + 1**]**);

    }

}

int **main**()

{

**test\_best\_case**();

    std::cout **<<** "Тест для лучшего случая пройден" **<<** std::**endl**;

**test\_average\_case**();

    std::cout **<<** "Тест для среднего случая пройден" **<<** std::**endl**;

**test\_empty\_array**();

    std::cout **<<** "Тест для пустого массива пройден" **<<** std::**endl**;

**test\_single\_element**();

    std::cout **<<** "Тест для массива с одним элементом пройден" **<<** std::**endl**;

**test\_all\_elements\_same**();

    std::cout **<<** "Тест для массива с одинаковыми элементами пройден" **<<** std::**endl**;

    return 0;

}

Листинг кода файла comb\_sort.cpp

#include <iostream>

#include <cassert>

#include <vector>

*// функция для вычисления следующего шага*

int **get\_next\_gap**(int gap)

{

*// уменьшаем шаг на коэффициент сжатия*

    gap = (gap \* 10) / 13; *// O(1)*

    if (gap < 1) *// O(1)*

    {

        return 1; *// O(1)*

    }

    return gap; *// O(1)*

}

*// функция для сортировки вектора методом comb sort*

void **comb\_sort**(std::**vector**<int> &arr)

{

    if (arr.**empty**())

    {

        return;

    }

    int gap = arr.**size**(); *// O(1)*

    bool swapped = true; *// O(1): устанавливаем флаг swapped как true, чтобы цикл гарантированно запустился хотя бы один раз*

*// продолжаем выполнение цикла до тех пор, пока шаг не станет равным 1 или не произойдёт ни одной перестановки*

    while (gap != 1 || swapped == true) *// O(N^2)*

    {

        gap = **get\_next\_gap**(gap); *// O(1): вычисляем следующий шаг*

        swapped = false; *// O(1): сбрасываем флаг swapped, чтобы проверить, произошли ли перестановки в этом проходе*

        for (int i = 0; i < arr.**size**() - gap; i++) *// O(N): сравниваем все элементы с текущим шагом*

        {

            if (arr**[**i**]** > arr**[**i + gap**]**) *// O(1)*

            {

                int temp = arr**[**i**]**; *// O(1)*

                arr**[**i**]** = arr**[**i + gap**]**; *// O(1)*

                arr**[**i + gap**]** = temp; *// O(1)*

                swapped = true; *// O(1): установка флага, если была выполнена перестановка*

            }

        }

    }

}

void **test\_best\_case**()

{

    std::**vector**<int> arr = {1, 2, 3, 4, 5};

**comb\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_average\_case**()

{

    std::**vector**<int> arr = {3, 1, 4, 2, 5};

**comb\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_worst\_case**()

{

    std::**vector**<int> arr = {5, 4, 3, 2, 1};

**comb\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_empty\_vector**()

{

    std::**vector**<int> arr;

**comb\_sort**(arr);

**assert**(arr.**empty**());

}

void **test\_single\_element**()

{

    std::**vector**<int> arr = {97};

**comb\_sort**(arr);

**assert**(arr**[**0**]** == 97);

}

void **test\_with\_duplicates**()

{

    std::**vector**<int> arr = {4, 5, 4, 3, 2, 2};

**comb\_sort**(arr);

    for (int i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

int **main**()

{

**test\_best\_case**();

    std::cout **<<** "Тест для лучшего случая пройден" **<<** std::**endl**;

**test\_average\_case**();

    std::cout **<<** "Тест для среднего случая пройден" **<<** std::**endl**;

**test\_worst\_case**();

    std::cout **<<** "Тест для худшего случая пройден" **<<** std::**endl**;

**test\_empty\_vector**();

    std::cout **<<** "Тест для пустого вектора пройден" **<<** std::**endl**;

**test\_single\_element**();

    std::cout **<<** "Тест для вектора с одним элементом пройден" **<<** std::**endl**;

**test\_with\_duplicates**();

    std::cout **<<** "Тест для вектора с дубликатами пройден" **<<** std::**endl**;

    return 0;

}

Листинг кода файла pigeonhole\_sort.cpp

#include <iostream>

#include <vector>

#include <cassert>

*// сортирует массив с использованием алгоритма сортировки голубиных отверстий*

void **pigeonhole\_sort**(std::**vector**<int> &arr)

{

    if (arr.**empty**())

    {

        return;

    }

    int n = arr.**size**(); *// О(1): определение размера вектора*

*// нахождение минимального и максимального значений в массиве*

    int min\_value = arr**[**0**]**, max\_value = arr**[**0**]**; *// O(1): инициализация переменных*

    for (int i = 1; i < n; i++) *// O(N): поиск минимума и максимума*

    {

        if (arr**[**i**]** < min\_value) *// O(1): сравнение*

            min\_value = arr**[**i**]**; *// O(1): обновление минимума*

        if (arr**[**i**]** > max\_value) *// O(1): сравнение*

            max\_value = arr**[**i**]**; *// O(1): обновление максимума*

    }

    int range = max\_value - min\_value + 1; *// O(1): вычисление диапазона*

*// создание вектора векторов. размер массива равен диапазону. каждый вектор представляет собой отверстие, содержащее соответствующие элементы.*

    std::**vector**<std::**vector**<int>> **holes**(range); *// O(range): выделение памяти для векторов*

*// проход по входному массиву и размещение каждого элемента в соответствующее ему отверстие*

    for (int i = 0; i < n; i++) *// O(N): заполнение векторов*

        holes**[**arr**[**i**]** - min\_value**]**.**push\_back**(arr**[**i**]**); *// O(1): вставка элемента в вектор*

*// проход по всем отверстиям одно за другим. для каждого отверстия надо взять его элементы и поместить в массив.*

    int index = 0; *// O(1)*

    for (int i = 0; i < range; i++) *// O(range): проход по каждому вектору*

    {

        for (int value : holes**[**i**]**) *// O(m): m - количество элементов в текущем векторе*

            arr**[**index++**]** = value; *// O(1): копирование значения из вектора обратно в исходный массив*

    }

}

void **test\_best\_case**()

{

    std::**vector**<int> arr = {1, 2, 3, 4, 5};

**pigeonhole\_sort**(arr);

    for (**size\_t** i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_average\_case**()

{

    std::**vector**<int> arr = {3, 1, 4, 2, 5};

**pigeonhole\_sort**(arr);

    for (**size\_t** i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_worst\_case**()

{

    std::**vector**<int> arr = {5, 4, 3, 2, 1};

**pigeonhole\_sort**(arr);

    for (**size\_t** i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

void **test\_large\_range**()

{

    std::**vector**<int> arr = {1000, -1000, 500, -500, 0};

**pigeonhole\_sort**(arr);

**assert**(arr**[**0**]** == -1000 && arr**[**1**]** == -500 && arr**[**2**]** == 0 && arr**[**3**]** == 500 && arr**[**4**]** == 1000);

}

*// тесты для крайних случаев*

void **test\_empty\_array**()

{

    std::**vector**<int> arr; *// Пустой вектор*

**pigeonhole\_sort**(arr);

**assert**(arr.**empty**()); *// Проверка, что вектор пустой*

}

void **test\_single\_element**()

{

    std::**vector**<int> arr = {42};

**pigeonhole\_sort**(arr);

**assert**(arr**[**0**]** == 42);

}

void **test\_two\_elements\_sorted**()

{

    std::**vector**<int> arr = {1, 2};

**pigeonhole\_sort**(arr);

**assert**(arr**[**0**]** == 1 && arr**[**1**]** == 2);

}

void **test\_two\_elements\_unsorted**()

{

    std::**vector**<int> arr = {2, 1};

**pigeonhole\_sort**(arr);

**assert**(arr**[**0**]** == 1 && arr**[**1**]** == 2);

}

void **test\_negative\_numbers**()

{

    std::**vector**<int> arr = {-3, -1, -4, -2, -5};

**pigeonhole\_sort**(arr);

    for (**size\_t** i = 0; i < arr.**size**() - 1; i++)

    {

**assert**(arr**[**i**]** <= arr**[**i + 1**]**);

    }

}

int **main**()

{

**test\_best\_case**();

    std::cout **<<** "Тест для лучшего случая пройден" **<<** std::**endl**;

**test\_average\_case**();

    std::cout **<<** "Тест для среднего случая пройден" **<<** std::**endl**;

**test\_worst\_case**();

    std::cout **<<** "Тест для худшего случая пройден" **<<** std::**endl**;

*// тесты для крайних случаев*

**test\_empty\_array**();

    std::cout **<<** "Тест для пустого массива пройден" **<<** std::**endl**;

**test\_single\_element**();

    std::cout **<<** "Тест для массива с одним элементом пройден" **<<** std::**endl**;

**test\_two\_elements\_sorted**();

    std::cout **<<** "Тест для отсортированного массива из двух элементов пройден" **<<** std::**endl**;

**test\_two\_elements\_unsorted**();

    std::cout **<<** "Тест для неотсортированного массива из двух элементов пройден" **<<** std::**endl**;

**test\_negative\_numbers**();

    std::cout **<<** "Тест для массива с отрицательными числами пройден" **<<** std::**endl**;

**test\_large\_range**();

    std::cout **<<** "Тест для массива с большим диапазоном значений пройден" **<<** std::**endl**;

    return 0;

}