Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №3 по курсу "Анализ алгоритмов"

Тема Трудоемкость сортировок

Студент Золотухин А.В.

Группа ИУ7-54Б

Оценка (баллы)

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2022 г.

Оглавление

Введение

Существует огромное количество вариаций сортировок. Эти алгоритмы необходимо уметь сравнивать, чтобы выбирать наилучшим образом подходящие для конкретного случая. Алгоритм сортировки – это алгоритм для упорядочивания элементов в массиве или списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки.

Цель данной лабораторной: изучение и исследование трудоемкости алгоритмов сортировки.

Задачи данной лабораторной:

- проанализировать и реализовать три различных алгоритма сортировки;
- рассчитать их трудоемкость;
- сравнить их временные характеристики экспериментально;
- на основании проделанной работы сделать выводы об эффективности реализаций рассмотренных алгоритмов сортировки.

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов быстрой сортировки, сортировки пузырьком и сортировки расческой[1].

1.1 Сортировка пузырьком

В начале сравниваются первые два элемента списка. Если первый больше второго, они меняются местами. Если они уже в нужном порядке, то остаются как есть. Затем осуществляется переход к следующей паре элементов, сравниваются их значения и меняются местами при необходимости. Этот процесс продолжается до последней пары элементов в списке.

По достижении конца списка, процесс повторяется для каждого элемента снова. В этом случае необходимо обойти список n раз.

Очевидно, что для оптимизации алгоритма нужно остановить его, когда закончится сортировка. Если бы элементы были отсортированы, то не пришлось бы их менять местами. Таким образом, всякий раз, когда меняются элементы, флаг устанавливается в $True$, чтобы повторить процесс сортировки. На каждой итерации флаг сбрасывается. Если перестановок не произошло, флаг останется $False$, и алгоритм остановится.

1.2 Быстрая сортировка

Общая идея алгоритма состоит в следующем.

1. Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность.

2. Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: элементы, меньшие опорного, равные опорно-

му и большие опорного (по значению ключа).

3. Для отрезков, содержащих "меньшие" и "большие" значения, выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, "меньшие опорного" и "равные и большие"; такой подход упрощает алгоритм разделения.

1.3 Сортировка расческой

Сортировка расческой — разновидность пузырьковой сортировки.

При пузырьковом алгоритме сравниваются постоянно два элемента. В сортировке расческой эти элементы берутся не соседними, а как бы по краям "расчески" — первый и последний. Расстояние между сравниваемыми элементами наибольшее из возможных, то есть, это максимальный размер расчески. Теперь уменьшая "расческу" на единицу и начинается сравнение элементов находящихся ближе: первого и предпоследнего, второго и последнего.

Затем расстояние между сравниваемыми элементами снова уменьшается и сравниваются первый с перед предпоследним, второй с предпоследним, третий с последним. Дальнейшие итерации проводятся постепенно уменьшая размер "расчески", то есть уменьшая расстояние между сравниваемыми элементами.

Первая длина "расчески", то есть расстояние между сравниваемыми элементами, выбирается с учетом специального коэффициента — 1,247. В начале сортировки расстояние между числами равно отношению размера массива и указанного числа. Затем, отсортировав массив с этим шагом, шаг делится на это число и выполняется новая итерация. Продолжаем выполнять действия до тех пор, пока разность индексов не достигнет единицы. В этом случае массив доупорядочивается обычным пузырьковым методом.

Вывод

В данном разделе были описаны три алгоритма сортировки, а именно: быстрой, пузырьком и расческой.

2 Конструкторская часть

В этом разделе будут приведены схемы алгоритмов и вычисления трудоемкости данных алгоритмов.

2.1 Разработка алгоритмов

На рисунках 2.1, 2.2 и 2.3 представлены схемы алгоритмов сортировки пузырьком, быстрой и расческой соответственно.

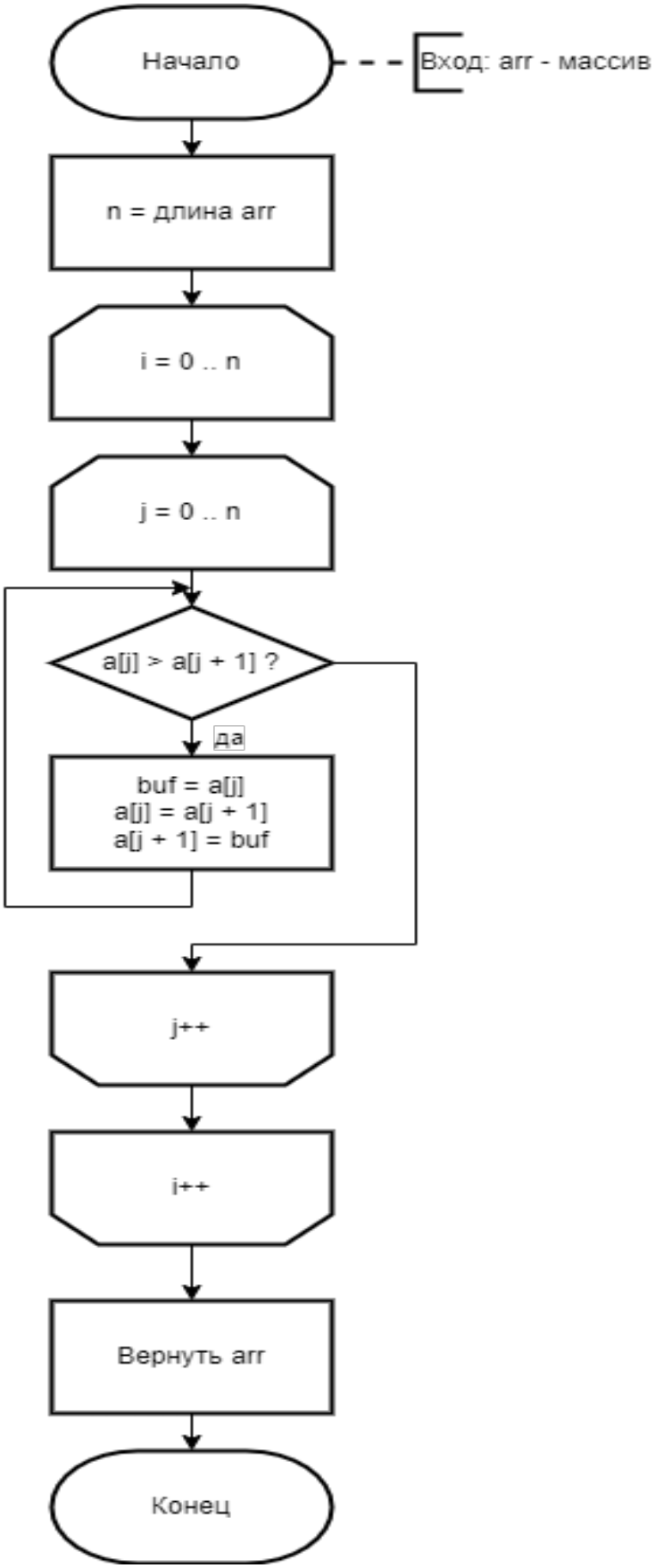

Рисунок 2.1 – Схема алгоритма сортировки пузырьком

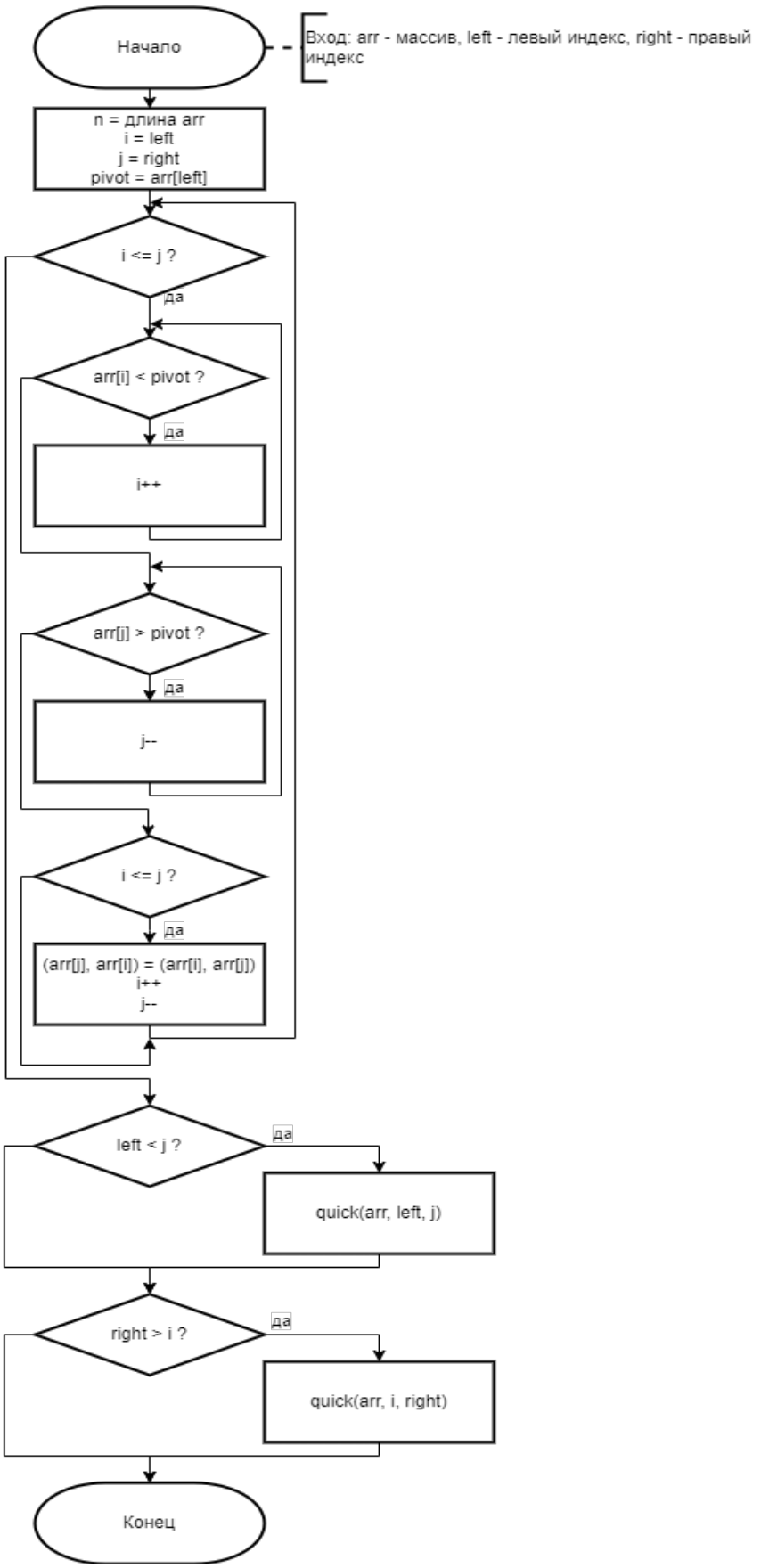

Рисунок 2.2 – Схема алгоритма быстрой сортировки

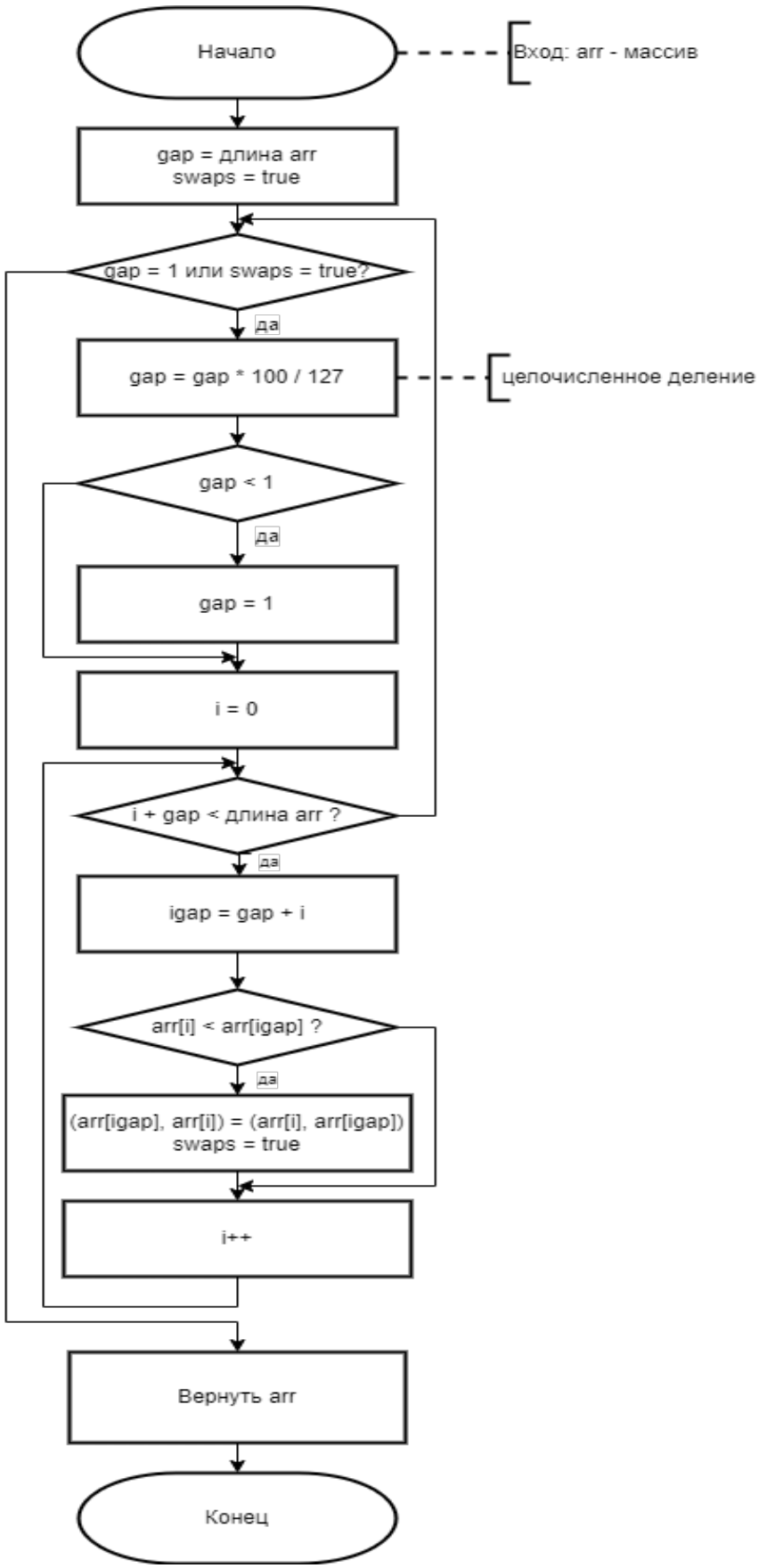

Рисунок 2.3 – Схема алгоритма сортировки расческой

2.2 Модель вычислений

Для последующего вычисления трудоемкости необходимо ввести модель вычислений, или оценки трудоемкости.

1. Операции из следующего списка имеют трудоемкость 1:

$$\begin{gathered} +, -, /, *, \%, =, + =, - =, * =, / =, \% =, \\ ==, ! =, <, >, <=, >=, [], ++, -- \end{gathered} \tag{2.1}$$

2. Трудоемкость оператора выбора `if условие then A else B` рассчитывается как

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.2}$$

3. Трудоемкость цикла рассчитывается как

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремент}} + f_{\text{сравнения}}) \tag{2.3}$$

4. Трудоемкость вызова функции равна 0.

2.3 Трудоёмкость алгоритмов

Обозначим во всех последующих вычислениях размер массивов как N.

2.3.1 Алгоритм сортировки пузырьком

Трудоёмкость алгоритма сортировки пузырьком при прямом расчёте включает переменный член:

$$f = 1 + 2 + (N-1)(1 + 2 + 1 + 2 + \cancel{(N-i)} \cdot f1, \tag{2.4}$$

где $f1$ — трудоёмкость сравнения элементов массива и тела условного оператора.

Количество выполненных сравнений известно:

$$Q_{\text{ср}} = \frac{a_1 + a_n}{2} n = \frac{N^2 - N}{2} \tag{2.5}$$

На одно сравнение приходится следующая работа:

$$f_{\text{ср}} = 4 + \left[\begin{array}{l} 0 \text{ лучший сл.} \\ 9 \text{ худший сл.} \end{array} \right. + 1 + 2 \tag{2.6}$$

и

$$f_{\text{служебные}} = 1 + 2 + (N-1)(1 + 2 + 1 + 2) \tag{2.7}$$

Итоговая трудоёмкость алгоритма сортировки пузырьком:

$$f_{\text{п}} = N^2 \left[\begin{array}{l} 7/2 \text{ лучший сл.} \\ 16/2 \text{ худший сл.} \end{array} \right. + N \left[\begin{array}{l} 6 - 7/2 \text{ лучший сл.} \\ 6 - 16/2 \text{ худший сл.} \end{array} \right. - 3 \tag{2.8}$$

При этом лучший случай достигается когда элементы в массиве расположены по возрастанию, а худший случай — по убыванию.

2.3.2 Алгоритм быстрой сортировки

Худший случай при несбалансированном дереве вызовов, т.е. количество вызовов равно количеству элементов массива. Таким образом временная сложность лучшего случая будет равна $O(N^2)$.

Лучший случай при сбалансированном дереве вызовов, т.е. количество вызовов равно высоте дерева $\log(N)$. Таким образом временная слож-

ность лучшего случая будет равна $O(N \log(N)$.

2.3.3 Алгоритм сортировки расческой

Худший случай

Худший случай для этой сортировки наступает тогда, когда все элементы уже отсортированы, но в обратном порядке.

$$a_i = N - floor(100 * gap_i/127) \tag{2.9}$$

$$gap_i = gap_i - 1 - floor(100 * (gap_i - 1)/127) \tag{2.10}$$

Если выразить верхнюю границу a_i из уравнения (2.9) и рекуррентного соотношения (2.10), будет получено выражение, содержащее члены второй и первой степени N с константой, дающей верхнюю границу N^2. Таким образом, временная сложность наихудшего случая равна $O(N^2)$.

Лучший случай

Для лучшего случая все массива должен быть отсортирован. В этом случае цикл с $gap = 1$ пройдет всего один раз.

$$\begin{gathered} a_N = n \\ a_{N-1} = \frac{N}{1.27} \\ a_{N-2} = \frac{N}{1.27^2} \\ ... \\ a_{N-i} = \frac{N}{1.27^i} \end{gathered} \tag{2.11}$$

$$S_N = a_1 + a_2 + ... + a_N \tag{2.12}$$

$$S_N = \sum_{r=1}^{N} (\frac{1}{1.27^r}) \tag{2.13}$$

$$H_{N,r} = \sum_{r=1}^{N} (\frac{1}{kr}) \tag{2.14}$$

Для $k \approx 1$ $H_{N,1} \in O(\log(N))$

Таким образом трудоемкость сортировки расческой получилась $O(N \log(N))$

В худшем случае алгоритм будет работать, как обычный "пузырек".

Асимптотическая трудоёмкость алгоритма сортировки расческой $O(N^2)$ в худшем случае и $O(N \log(N))$ — в лучшем [2].

Вывод

Были разработаны схемы всех трех алгоритмов сортировки, а именно алгоритма сортировки пузырьком, быстрой сортировки и сортировки расческой. Для каждого из них были рассчитаны и оценены лучшие и худшие случаи.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к программному обеспечению

К программе предъявляется ряд требований.

На вход подаётся массив сравнимых элементов (целые числа).

На выходе — тот же массив, но в отсортированном виде.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования C# [3].

Язык C# является полностью объектно-ориентированным. Все необходимые библиотеки для реализации поставленной задачи являются стандартными.

Время работы алгоритмов было замерено с помощью функции clock() [4].

3.3 Сведения о модулях программы

Программа состоит из пяти модулей.

1. Programm.cs — главный файл программы, в котором располагается код меню;
2. BaseSort.cs — файл с базовым классом алгоритмов.
3. BubbleSort.cs, QuickSort.cs, CombSort.cs — файлы с кодами алгоритмов

3.4 Реализация алгоритмов

В листингах 3.1, 3.2, 3.3 представлены реализации алгоритмов сортировок (быстрой, пузырьком и расческой).

Листинг 3.1 – Класс быстрой сортировки

```
internal class QuickSort<Type> : BaseSort<Type>
{
    public QuickSort(Comparator comp) : base(comp) { }
    public override Type[] Sort(Type[] array)
    {
        return Sort(array, 0, array.Length - 1);
    }

    private Type[] Sort(Type[] array, int left, int right)
    {
        int i = left;
        int j = right;
        Type pivot = array[left];
        while (i <= j)
        {
            while (comp(array[i], pivot) < 0)
            i++;
            while (comp(array[j], pivot) > 0)
            j--;
            if (i <= j)
            {
                (array[j], array[i]) = (array[i], array[j]);
                i++;
                j--;
            }
        }

        if (left < j)
        Sort(array, left, j);
        if (i < right)
        Sort(array, i, right);
        return array;
    }
}
```

Листинг 3.2 – Класс сортировки пузырьком

```
internal class BubbleSort<Type> : BaseSort<Type>
{
    public BubbleSort(Comparator comp) : base(comp) { }
    public override Type[] Sort(Type[] array)
    {
        int n = array.Length;
        for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
        if (comp(array[j], array[j + 1]) > 0)
        (array[j + 1], array[j]) = (array[j], array[j + 1]);
        return array;
    }
}
```

Листинг 3.3 – Класс сортировки расческой

```
internal class CombSort<Type> : BaseSort<Type>
{
    readonly int coef = 127;
    public CombSort(Comparator comp) : base(comp) { }

    public override Type[] Sort(Type[] array)
    {
        int gap = array.Length;
        bool swaps = true;

        while (gap > 1 || swaps)
        {
            gap = gap * 100 / coef;

            if (gap < 1)
            gap = 1;

            swaps = false;

            for (int i = 0; i + gap < array.Length; i++)
            {
                int igap = i + gap;

                if (comp(array[i], array[igap]) > 0)
                {
```

```
26                          (array[igap], array[i]) = (array[i],
                                array[igap]);
27                          swaps = true;
28                      }
29                  }
30              }
31              return array;
32          }
33  }
```

3.5 Модульные тесты

В таблице 3.1 приведены тесты для методов, реализующих алгоритмы сортировки. Тесты пройдены успешно.

Таблица 3.1 – Модульные тесты

Входной массив	Ожидаемый результат	Результат
$[1, 2, 3, 4]$	$[1, 2, 3, 4]$	$[1, 2, 3, 4]$
$[5, 4, 3, 2, 1]$	$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$
$[3, 2, -5, 0, 1]$	$[-5, 0, 1, 2, 3]$	$[-5, 0, 1, 2, 3]$
$[4]$	$[4]$	$[4]$
$[]$	$[]$	$[]$

Вывод

Были реализованы все три алгоритма сортировок (быстрой, пузырьком и расческой). Выполнено модульное тестирование реализаций трех алгоритмов сортировки.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программ, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование.

1. Операционная система: Windows 10 Корпоративная, Версия 21H1, Сборка ОС 19043.2006.
2. Память: 8 ГБ.
3. Процессор: AMD Ryzen 5 4600H с видеокартой Radeon Graphics 3.00 ГГц [5].

Исследование проводилось на ноутбуке, включенном в сеть электропитания. Во время исследования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой.

4.2 Демонстрация работы программы

На рисунках 4.1, 4.2, 4.3 представлены результаты работы реализованных алгоритмов.

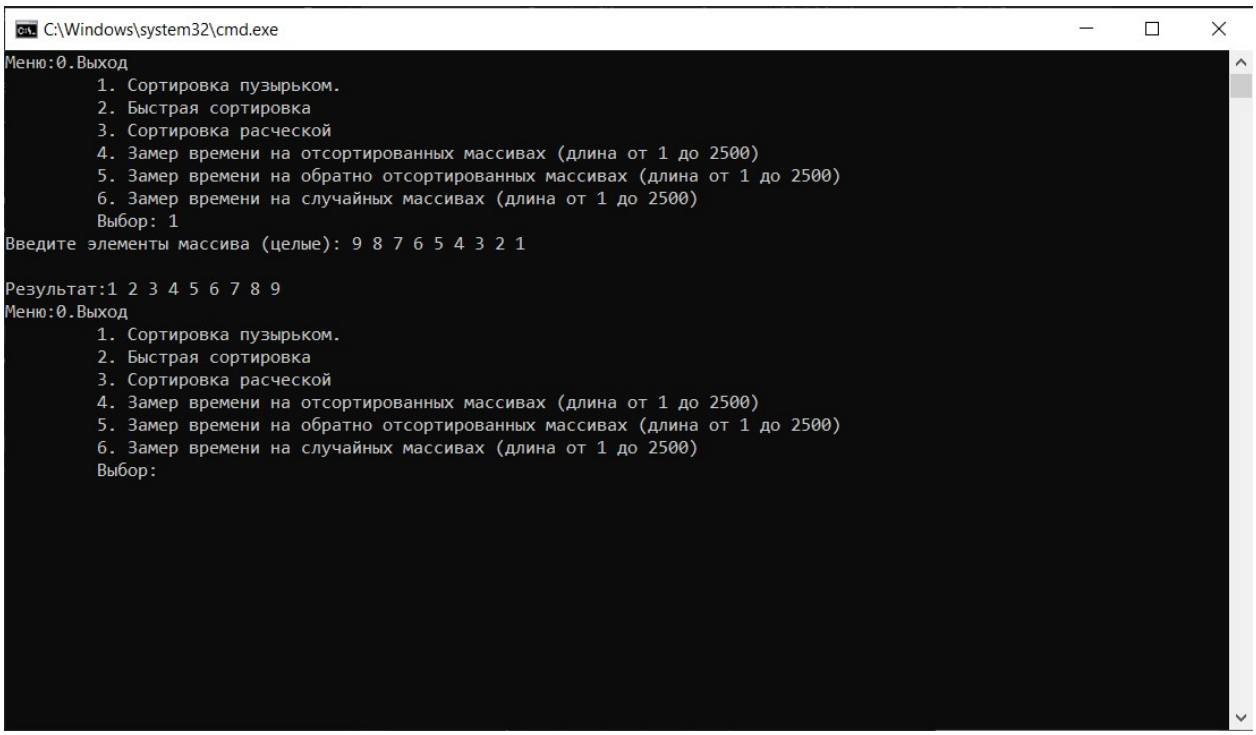

Рисунок 4.1 – Демонстрация работы реализации алгоритма сортировки пузырьком

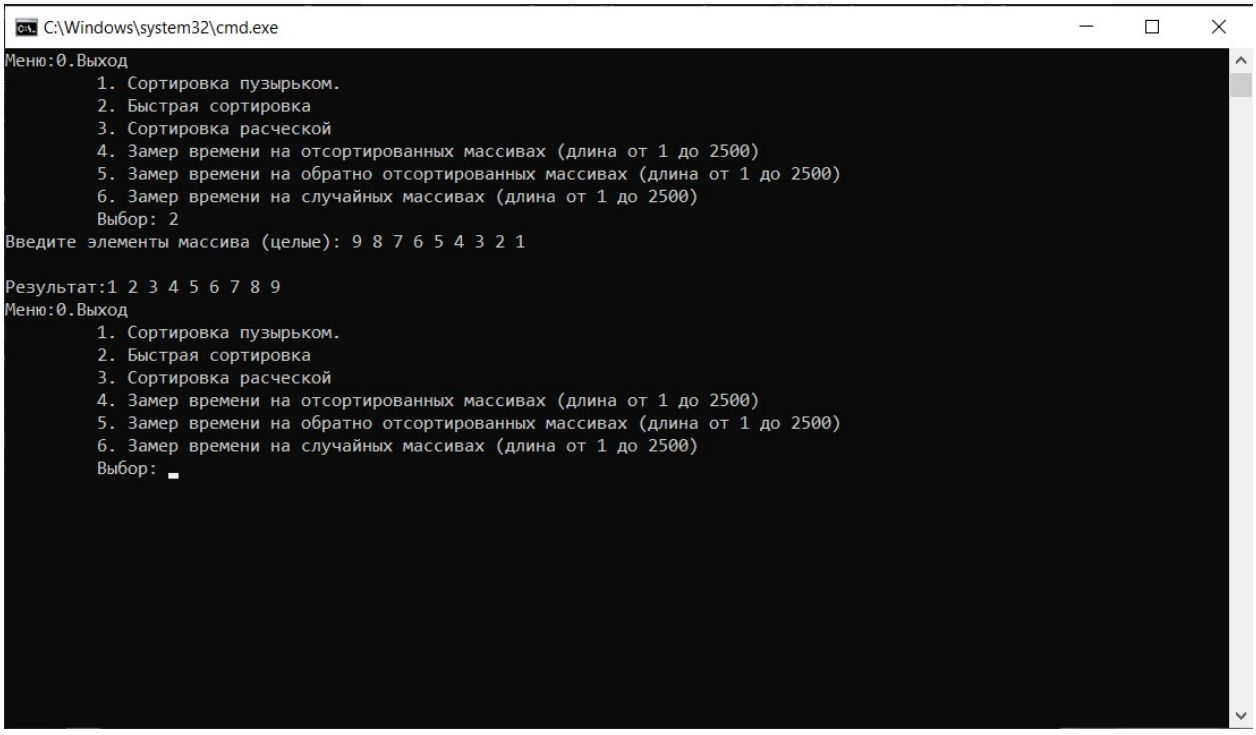

Рисунок 4.2 – Демонстрация работы реализации алгоритма быстрой сортировки

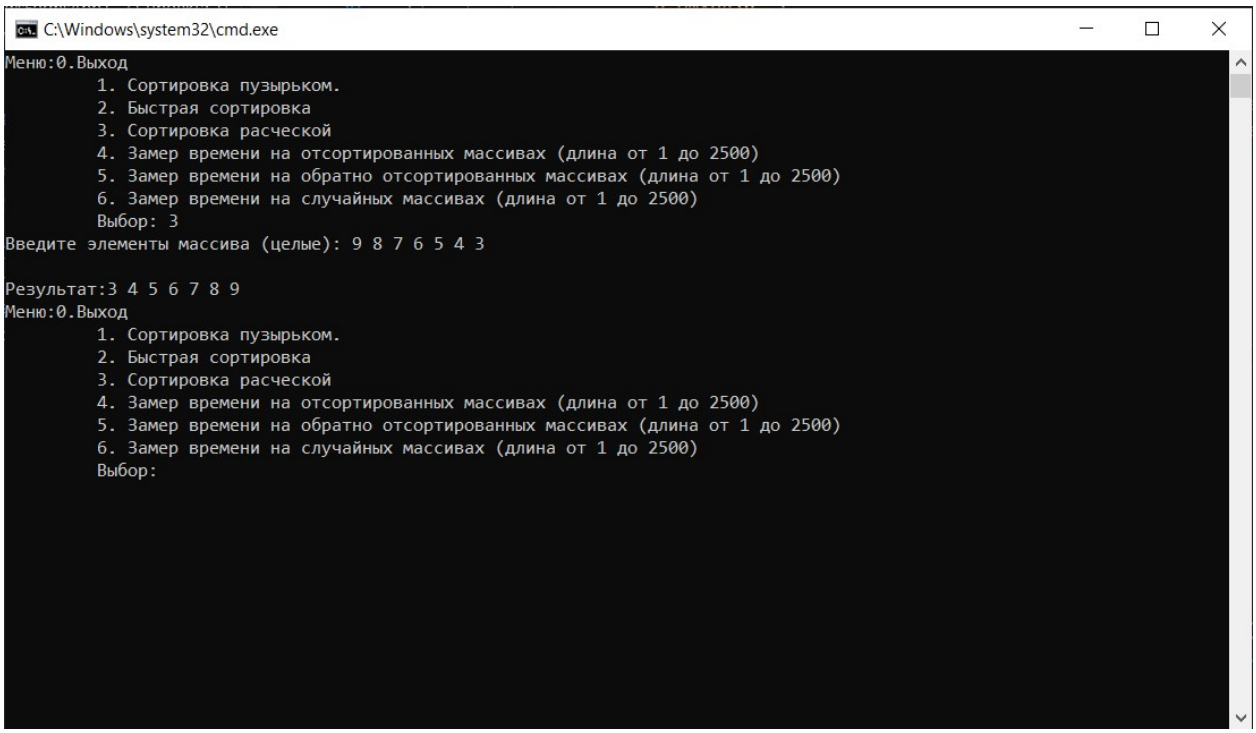

Рисунок 4.3 – Демонстрация работы реализации алгоритма сортировки расческой

4.3 Время выполнения реализаций алгоритмов

Время выполнения реализаций алгоритмов было замерено при помощи функции clock() [4]. Данная функция всегда возвращает значения времени, а именно сумму системного и пользовательского процессорного времени текущего процесса, значения типа float — время в тиках.

Замеры времени для каждой длины массива проводились 100 раз. В качестве результата взято среднее время работы алгоритма на данной длине массива.

Результаты замеров приведены в таблицах 4.1, 4.2 и 4.3 (время в тиках). На рисунках 4.4, 4.5 и 4.6, приведены графики зависимостей времени работы алгоритмов сортировки от размеров массивов на отсортированных, обратно отсортированных и случайных данных.

Таблица 4.1 – Время выполнения реализаций алгоритмов сортировки на отсортированных данных (время в тиках)

Размер	BubbleSort	QuickSort	CombSort
1	0,01	0	0
5	0	0	0
10	0	0	0
50	0,01	0	0
100	0,04	0,02	0,01
500	0,86	0,61	0,07
1000	3,43	2,41	0,15
2000	13,74	9,57	0,35
2500	21,5	14,93	0,46

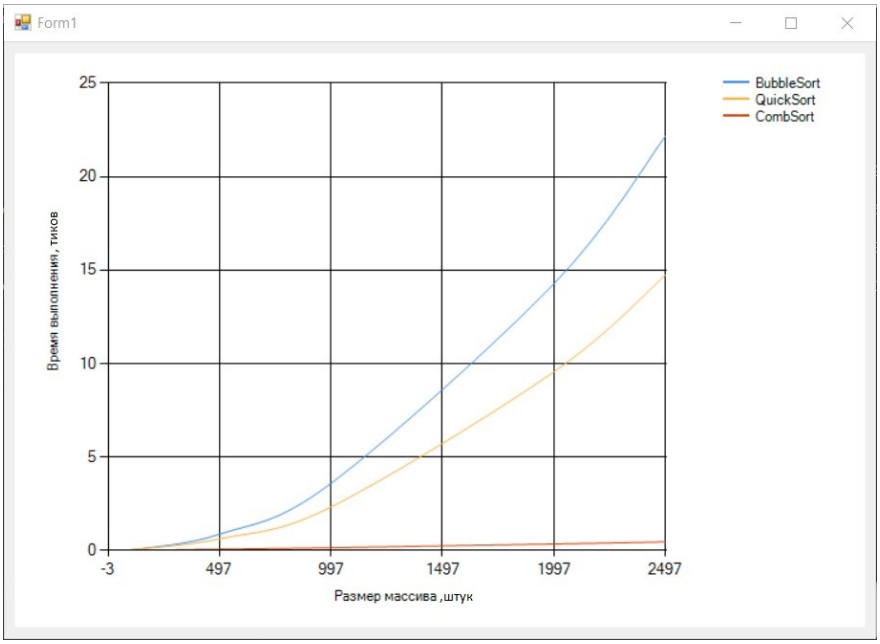

Рисунок 4.4 – Зависимость времени сортировки отсортированного массива от размера массива (время в тиках)

Таблица 4.2 – Время выполнения реализаций алгоритмов сортировки на обратно отсортированных данных (время в тиках)

Размер	BubbleSort	QuickSort	CombSort
1	0	0	0
5	0	0	0
10	0	0	0
50	0,01	0,01	0
100	0,04	0,03	0,01
500	0,86	0,61	0,07
1000	3,44	2,4	0,15
2000	13,77	9,52	0,35
2500	21,52	14,86	0,45

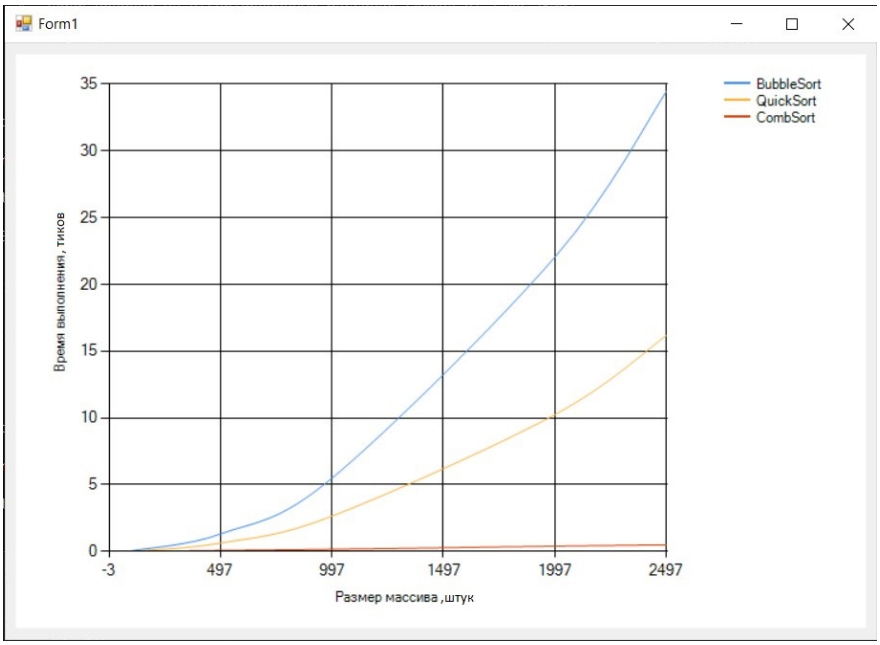

Рисунок 4.5 – Зависимость времени сортировки обратно отсортированного массива от размера массива (время в тиках)

Таблица 4.3 – Время выполнения реализаций алгоритмов сортировки на случайных данных (время в тиках)

Размер	BubbleSort	QuickSort	CombSort
1	0	0,01	0
5	0,01	0	0
10	0	0	0
50	0,01	0,01	0
100	0,05	0,01	0,02
500	1,25	0,06	0,09
1000	5,13	0,14	0,21
2000	20,99	0,33	0,47
2500	33,01	0,4	0,65

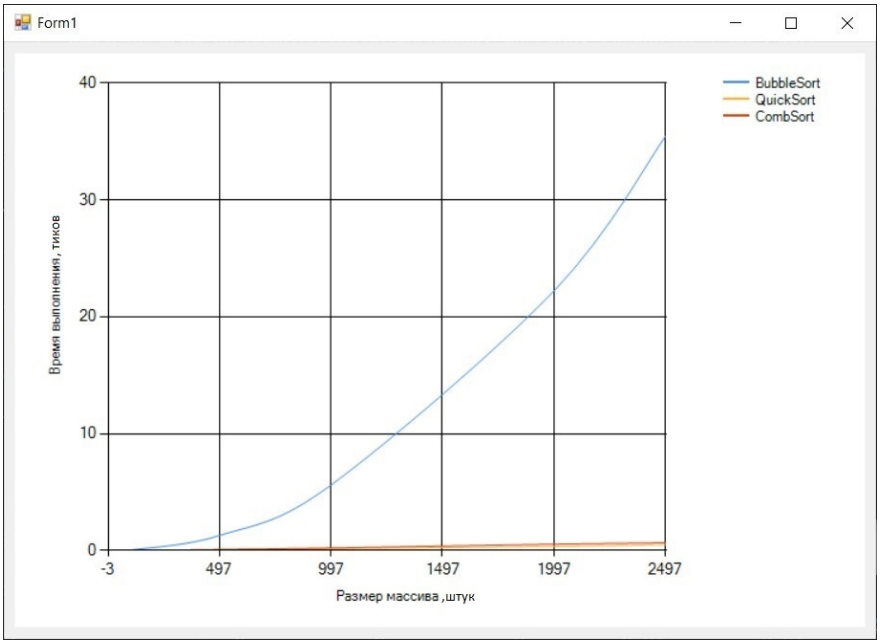

Рисунок 4.6 – Зависимость времени сортировки случайного массива от размера массива (время в тиках)

Вывод

Реализация алгоритма сортировки расческой работает быстрее реализаций остальных алгоритмов при прямо и обратно отсортированных массивах. Реализация алгоритма быстрой сортировки эффективнее по времени чем реализации двух других алгоритмов на массивах заполненных случайными числами. Также реализация алгоритма быстрой сортировки работает значительно дольше, если массив отсортирован по убыванию или возрастанию.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- изучены и реализованы 3 алгоритма сортировки: быстрой, пузырьком, расческой;
- проведен сравнительный анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
- проведен сравнительный анализ алгоритмов на основе экспериментальных данных;
- подготовлен отчет о лабораторной работе.

Были изучены и исследованы трудоемкости алгоритмов сортировки. Поставленная цель достигнута.

Список использованных источников

[1] Основные виды сортировок и примеры их реализации. Режим доступа: https://academy.yandex.ru/journal/osnovnye-vidy-sortirovok-i-primery-ikh-realizatsii(дата обращения 25.09.2022).

[2] Time and Space Complexity of Comb Sort [Электронный ресурс]. Режим доступа: https://iq.opengenus.org/time-and-space-complexity-of-comb-sort/ (дата обращения: 06.10.2021).

[3] Краткий обзор языка C# [Электронный ресурс]. Режим доступа: https://learn.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/ (дата обращения: 25.09.2022).

[4] ISO/IEC 9899:1999 [Электронный ресурс]. Режим доступа: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf Глава 7.23.2.1 The clock function (дата обращения: 25.09.2022).

[5] AMD Ryzen™ 5 4600H [Электронный ресурс]. Режим доступа: https://www.amd.com/ru/products/apu/amd-ryzen-5-4600h (дата обращения: 25.09.2022).