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»

Отчёт по лабораторной работе №3
по дисциплине «Анализ алгоритмов»

Тема: Алгоритмы сортировки
Студент: Княжев А. В.
Группа: ИУ7-52Б
Оценка (баллы): _____
Преподаватели: Волкова Л. Л., Строганов Ю. В.

Москва — 2022 г.

Оглавление

Введение **3**

1. Аналитическая часть **4**

1.1. Блочная сортировка 4

1.2. Пирамидальная сортировка 4

1.3. Сортировка подсчетом 5

2. Конструкторская часть **6**

2.1. Разработка алгоритма блочной сортировки 6

2.2. Разработка алгоритма пирамидальной сортировки 8

2.3. Разработка алгоритма сортировки подсчетом 10

2.4. Оценка трудоемкости алгоритмов сортировки 10

2.4.1. Модель вычислений 10

2.4.2. Трудоемкость алгоритма блочной сортировки 12

2.4.3. Трудоемкость алгоритма пирамидальной сортировки 14

2.4.4. Трудоемкость алгоритма сортировки подсчетом 15

3. Технологическая часть **16**

3.1. Требования к ПО 16

3.2. Средства реализации 16

3.3. Реализации алгоритмов 16

3.4. Тестирование 21

4. Экспериментальная часть **22**

4.1. Технические характеристики 22

4.2. Измерение процессорного времени выполнения реализаций алгоритмов . . . 22

4.2.1. Худший случай 22

4.2.2. Лучший случай 23

4.2.3. Случайный случай 24

4.3. Измерение объёма потребляемой памяти реализаций алгоритмов 26

Заключение **27**

Список использованных источников **29**

Введение

В современном мире важную роль выполняют задачи обработки данных, в частности, алгоритмы сортировки, которые используются во многих сферах, требующих работу с информацией. Сортировка — процесс установки последовательности сравнимых элементов в определенном порядке. Задача сортировки не теряет своей актуальности и сейчас, так как еще не найден идеальный алгоритм сортировки, а объемы обрабатываемой информации постоянно растут [1].

Цель работы

Получение навыков кодирования программного продукта, тестирования и проведения замерного эксперимента работы программы на различных данных. Все это на примере решения задачи сортировки.

Задачи работы

1) изучение алгоритмов сортировки:
   - блочная сортировка;
   - пирамидальная сортировка;
   - сортировка подсчетом;
2) анализ трудоемкости данных алгоритмов;
3) кодирование данных алгоритмов;
4) проведение замерного эксперимента для данных алгоритмов, с измерением времени работы и использования памяти;
5) проведение сравнительного анализа алгоритмов на основе полученных данных.

1. Аналитическая часть

В данном разделе рассмотрены теоретические выкладки по алгоритмам сортировки. В дальнейшем в рамках данной работы предполагается, что происходит сортировка набора целых чисел по возрастанию.

1.1. Блочная сортировка

Блочная сортировка — алгоритм сортировки, в котором происходит разбиение входного набора элементов на блоки, причем значения элементов в каждом блоке больше, чем в предыдущем. После разбиения входного набора элементов на блоки, происходит сортировка каждого блока по отдельности, либо рекурсивно с использованием блочной сортировки, либо с использованием другого алгоритма сортировки. После сортировки блоков, так как значения в каждом блоке больше, чем в предыдущем, происходит слияние блоков, в результате которого получается отсортированный набор.

Пример разбиения на блоке представлен на рис. 1.1.

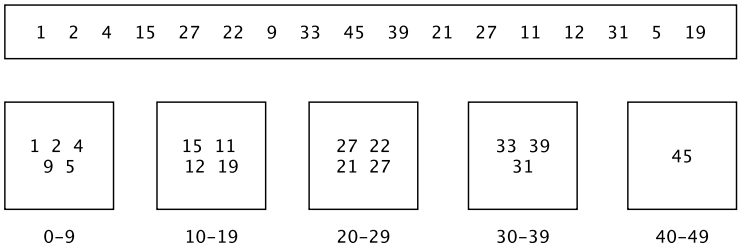

Рисунок 1.1 — Пример разбиения входного набора чисел на блоки

1.2. Пирамидальная сортировка

Пирамидальная сортировка — алгоритм сортировки, в котором на заданном массиве происходит создание такой структуры данных, как пирамида [2]. Пирамида определяется как последовательность элементов $a_l, a_{l+1}, \ldots, a_r$ такая, что

$$\forall i = l \ldots \frac{r}{2} : \begin{cases} a_i \leq a_{2 \cdot i}, \\ a_i \leq a_{2 \cdot i + 1}. \end{cases} \tag{1.1}$$

Если представлять пирамиду как древовидную структуру, то в корневой вершине дерева будет находиться минимальный элемент. Соответственно, если построить такую структуру на входном наборе данных, то минимальный элемент встанет на свою позицию, так как в результате построения пирамиды, минимальный элемент будет найден. Для дальнейшей сортировки требуется рассмотреть пирамидальную структуру на оставшемся наборе данных.

1.3. Сортировка подсчетом

Сортировка подсчетом — алгоритм сортировки, в котором происходит подсчет количества вхождений каждого элемента в исходный набор данных. На основе собранной статистики, происходит создание отсортированного массива. Особенностью данного алгоритма является то, что он работает только с наборами целых чисел.

2. Конструкторская часть

В данном разделе представлены схемы алгоритмов сортировки и произведена оценка трудоемкости алгоритмов сортировки.

2.1. Разработка алгоритма блочной сортировки

На рис. 2.1 – 2.2 представлена схема алгоритма блочной сортировки.

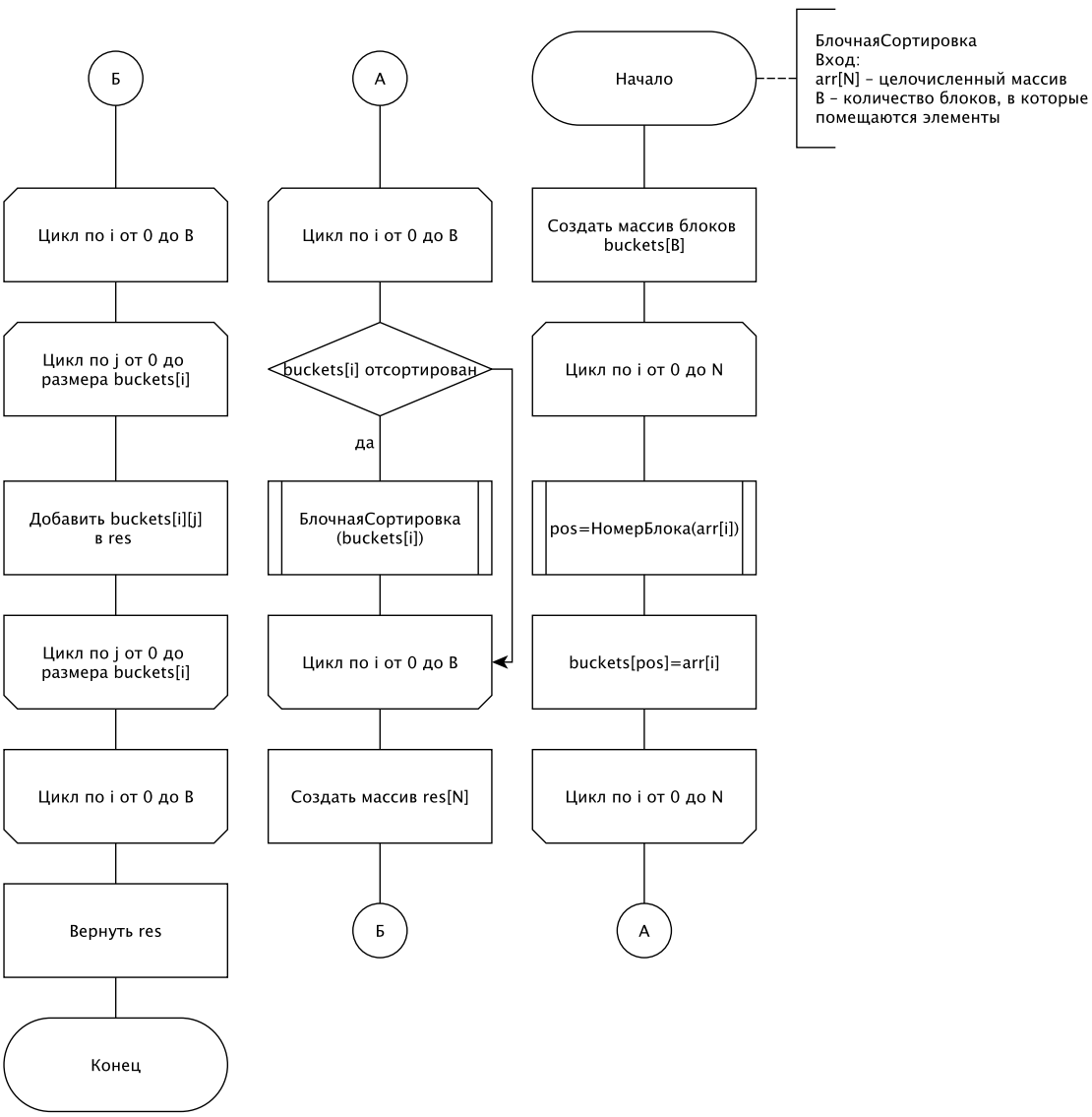

Рисунок 2.1 — Схема алгоритма блочной сортировки

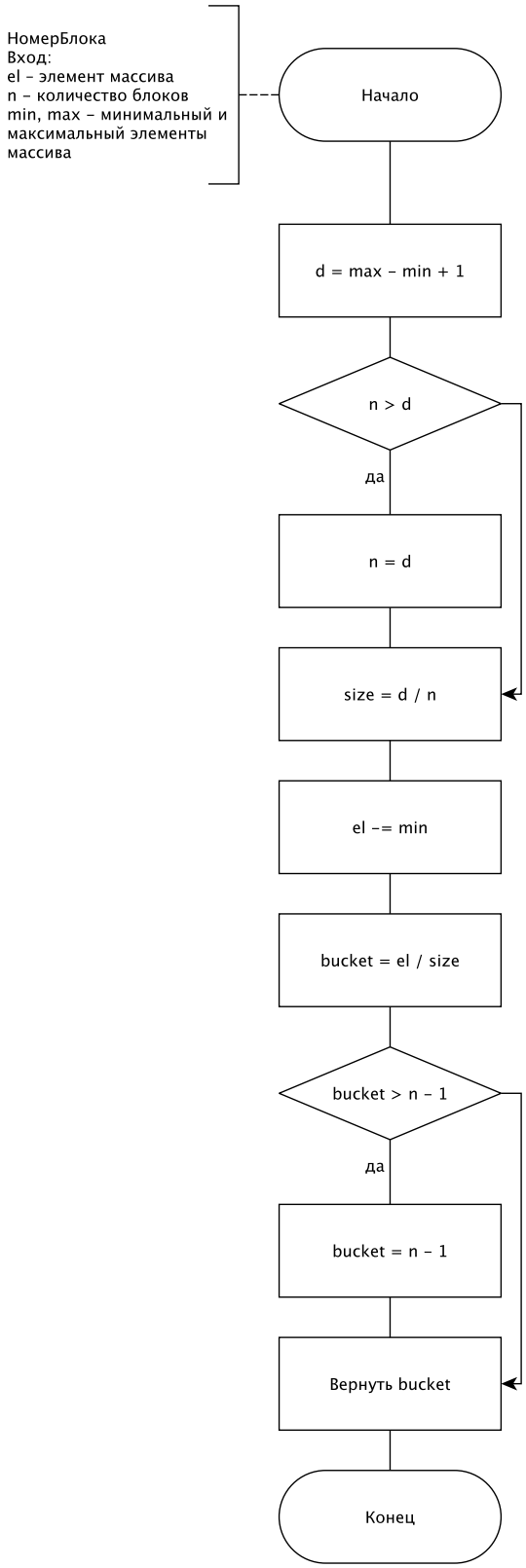

Рисунок 2.2 — Схема нахождения номера блока для элемента массива

2.2. Разработка алгоритма пирамидальной сортировки

На рис. 2.3 – 2.4 представлена схема алгоритма пирамидальной сортировки.

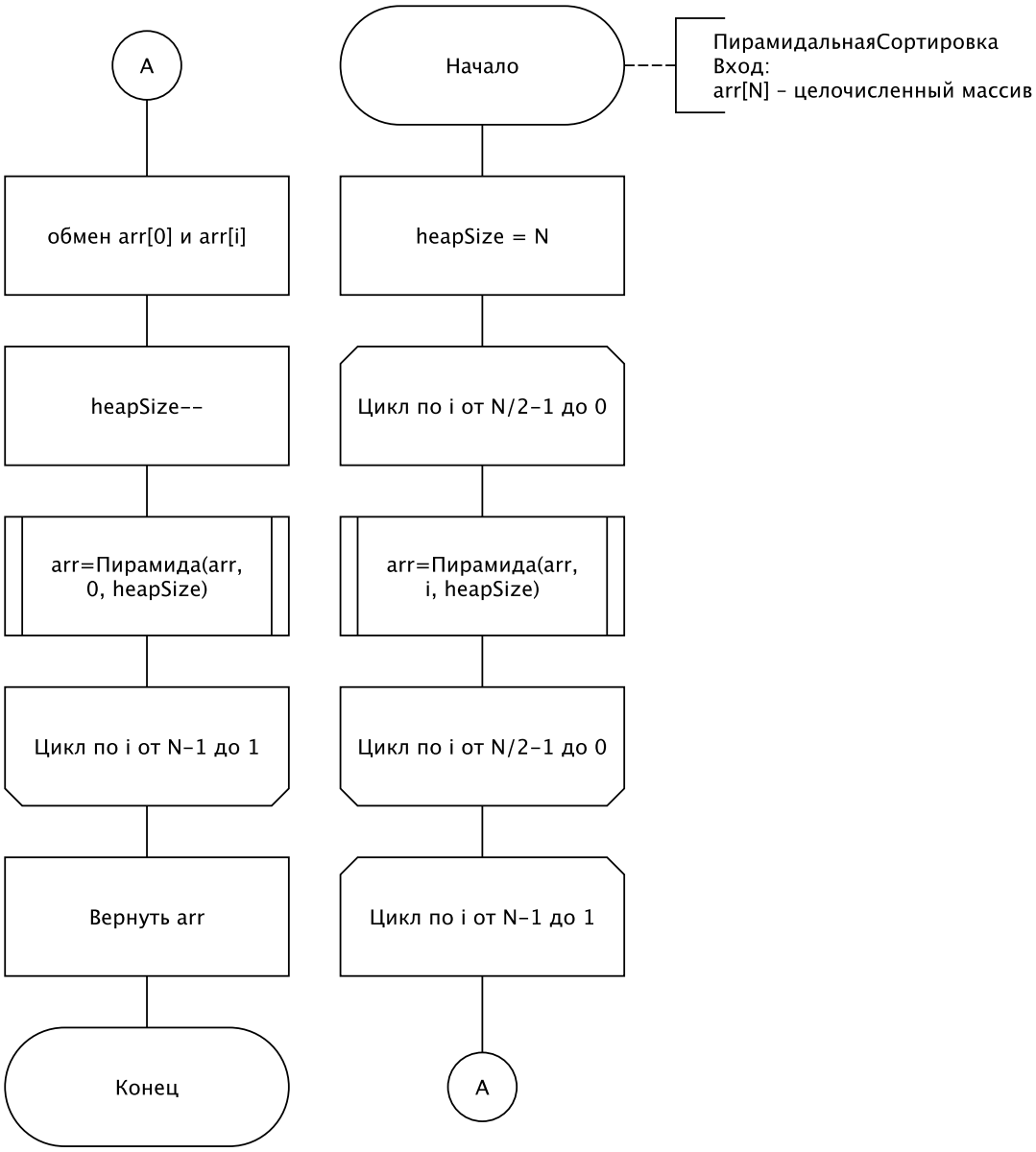

Рисунок 2.3 — Схема алгоритма пирамидальной сортировки

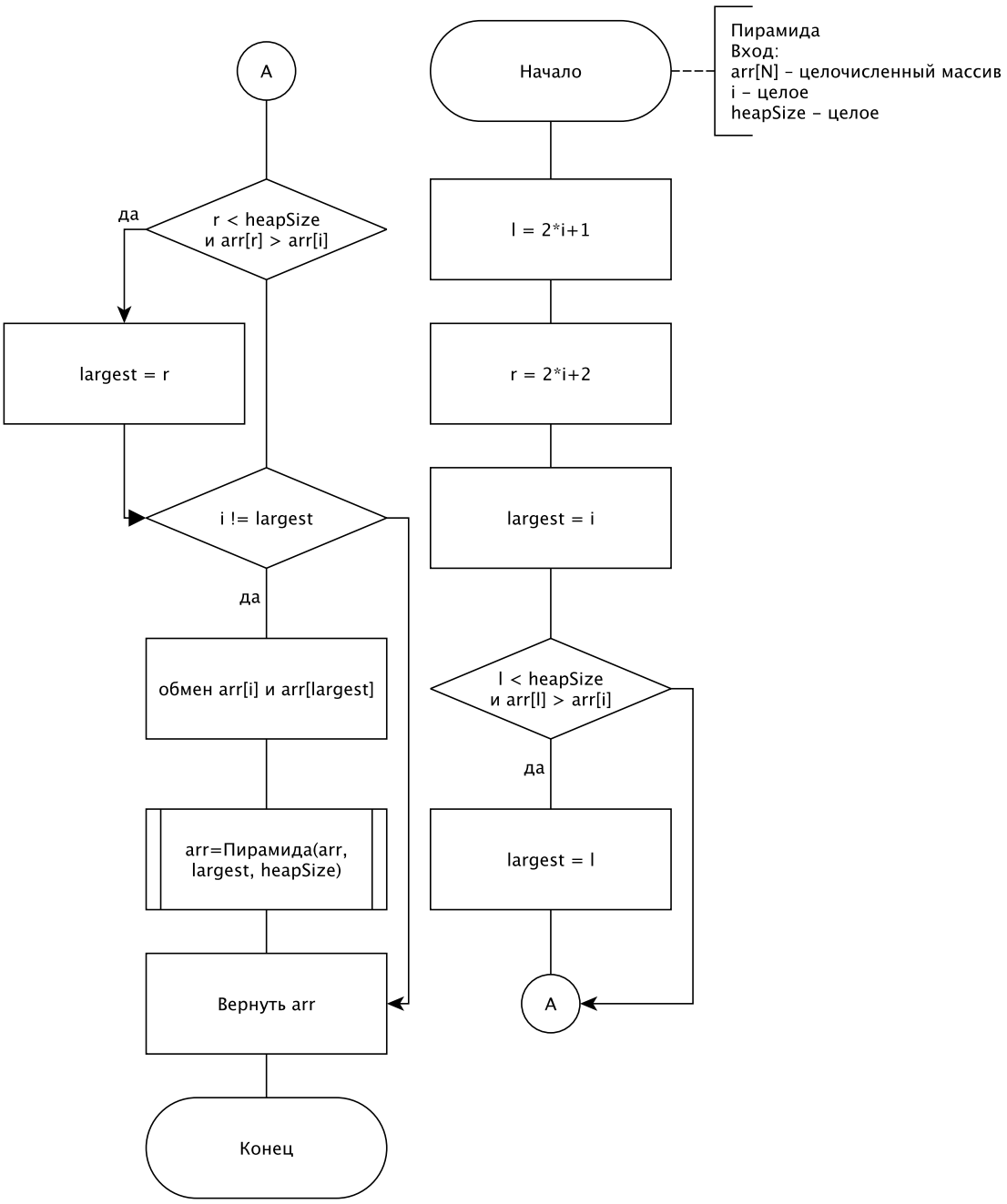

Рисунок 2.4 — Схема алгоритма преобразование массива в пирамиду

2.3. Разработка алгоритма сортировки подсчетом

На рис. 2.5 представлена схема алгоритма сортировки подсчетом.

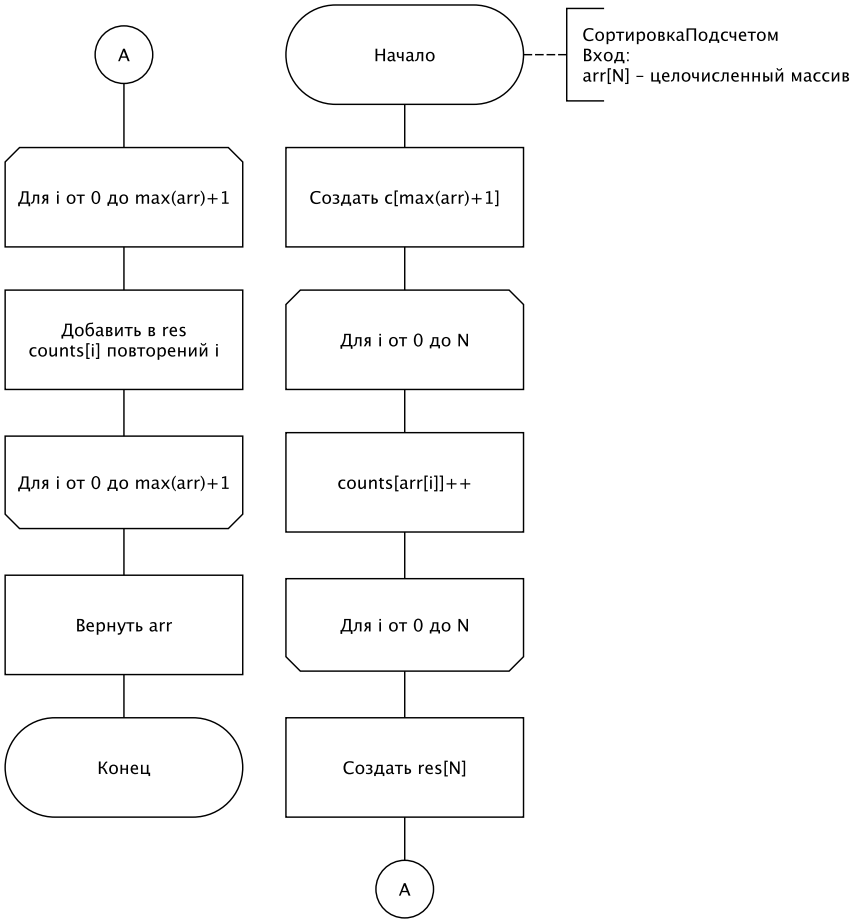

Рисунок 2.5 — Схема алгоритма сортировки подсчетом

2.4. Оценка трудоемкости алгоритмов сортировки

2.4.1. Модель вычислений

Для вычисления трудоемкости исследуемых алгоритмов необходимо ввести модель вычислений.

Если обозначить трудоемкость некоторой операции a, как f_a, то можно ввести таблицу соответствия значения трудоемкости для базовых операций:

Трудоемкость условного блока можно ввести следующим образом:

Таблица 2.1 — Таблица значений тредоемкости

Операция	Трудоемкость
:=	1
+ =	1
− =	1
+	1
−	1
<<	1
>>	1
[]	1
++	1
−−	1
>=	1
<=	1
==	1
! =	1
<	1
>	1
*	2
/	2
%	2
вызов функции	0

$$f_{if} = f_{cond} + \begin{cases} min(f_{in\_if}, f_{in\_else}) & \text{в лучшем случае,} \\ max(f_{in\_if}, f_{in\_else}) & \text{в худшем случае,} \end{cases} \tag{2.1}$$

где

— f_{if} — трудоемкость условного блока;

— f_{cond} — трудоемкость вычисления условия;

— f_{in_if} — трудоемкость фрагмента после if;

— f_{in_else} — трудоемкость фрагмента после $else$.

Трудоемкость цикла можно ввести следующим образом:

$$f_{loop} = f_{init} + f_{cmp} + n \cdot (f_{body} + f_{cmp} + f_{inc}), \tag{2.2}$$

где

— f_{loop} — трудоемкость цикла;

— f_{init} — трудоемкость инициализирующего выражения;

— f_{cmp} — трудоемкость сравнения цикла;

— f_{body} — трудоемкость тела цикла;

— f_{inc} — трудоемкость инкремента.

2.4.2. Трудоемкость алгоритма блочной сортировки

Здесь и далее будут использоваться следующие обозначения:

— N — количество элементов массива;

— M — максимальный элемент массива;

— B — количество элементов в одном блоке, известно, что

$$B \cdot M = N. \tag{2.3}$$

Трудоемкости нахождения максимума и минимума в массиве равны:

$$f_{max} = 1 + 2 + 1 + 1 + N \cdot \left(1 + 1 + 2 + \begin{cases} 1, & \text{если элемент больше максимума,} \\ 0, & \text{иначе,} \end{cases} \right) \tag{2.4}$$

$$f_{max} = \begin{cases} 5 + 5 \cdot N, & \text{в худшем случае,} \\ 5 + 4 \cdot N, & \text{в лучшем случае.} \end{cases} \tag{2.5}$$

$$f_{min} = \begin{cases} 5 + 5 \cdot N, & \text{в худшем случае,} \\ 5 + 4 \cdot N, & \text{в лучшем случае.} \end{cases} \tag{2.6}$$

Так как поиск максимальных и минимальных элементов отличаются только знаком сравнения элементов, причем эти знаки противоположны, то лучший случай для поиска максимума является худшим случаем для поиска минимума. Таким образом, можно полагать, что их суммарная трудоемкость будет равна

$$f_{min,max} = 10 + 9 \cdot N. \tag{2.7}$$

Трудоемкость расчета параметров для расчета размеров блока равна

$$f_1 = f_{min,max} + 3 + 1 + 1 + 1 + 3 + 1 = 20 + 9 \cdot N. \tag{2.8}$$

Трудоемкость размещения элементов по блокам равна

$$f_2 = 2 + N \cdot (5 + 6 + 2) = 2 + N \cdot 13. \tag{2.9}$$

Таким образом, трудоемкость блока с рекурсивным вызовом можно оценить, как

$$f_3 = 2 + M \cdot (2 + 6 \cdot B) = 2 + 2 \cdot M + 6 \cdot N. \tag{2.10}$$

Трудоемкость размещения отсортированных элементов из блоков в результирующий массив равна

$$f_4 = 2 + M \cdot (2 + 2 + B \cdot (2 + 3)) = 2 + 4 \cdot M + 5 \cdot N. \tag{2.11}$$

Трудоемкость одного вызова функции сортировки равна

$$f_5 = f_1 + f_2 + f_3 + f_4, \tag{2.12}$$

$$f_5 = 26 + 33 \cdot N + 6 \cdot M, \tag{2.13}$$

Для алгоритма блочной сортировки худшим случаем является тот, в котором в исходном массиве лежат отсортированные по убыванию слабо различающиеся элементы, которые в результате размещения будут расположены всего в двух блоках. Из-за проверки на отсортированность блоков, лучшим случаем будет являться отсортированный по возрастанию массив.

Так как функция сортировки будет содержать в себе рекурсивный вызов, то для расчета итоговой трудоемкости нужно умножить трудоемкость тела функции сортировки на глубину рекурсии. Глубина рекурсии в данном случае будет равна

$$d = \begin{cases} 1, & \text{в лучшем случае,} \\ N, & \text{в худшем случае.} \end{cases} \tag{2.14}$$

Таким образом, трудоемкость алгоритма блочной сортировки равна

$$f = d \cdot f_5 = \begin{cases} 33 \cdot N + 6 \cdot M + 26, & \text{в лучшем случае,} \\ 33 \cdot N^2 + 6 \cdot M \cdot N + 26 \cdot M, & \text{в худшем случае.} \end{cases} \tag{2.15}$$

На основе вышеуказанных расчетов можно сделать вывод, что асимптотика алгоритма блочной сортировки равна $O(N)$ в лучшем случае, и $O(N^2)$ в худшем случае.

2.4.3. Трудоемкость алгоритма пирамидальной сортировки

Лучшим случаем для данного алгоритма можно назвать случай, когда все элементы массива равны.

Трудоемкость тела построения пирамиды равна

$$f_{pyrbody} = \begin{cases} 6 + 1 + 5 + 1 + 5 + 1 + 8, & \text{в лучшем случае,} \\ 6 + 1 + 5 + 5, & \text{иначе.} \end{cases} \tag{2.16}$$

$$f_{pyrbody} = \begin{cases} 17, & \text{в лучшем случае,} \\ 27, & \text{иначе.} \end{cases} \tag{2.17}$$

Так как в функции есть рекурсивный вызов, то ее трудоемкость следует считать, как трудоемкость тела функции, умноженную на глубину рекурсии. Глубина рекурсии будет равна

$$d = \log_2 N. \tag{2.18}$$

Таким образом, трудоемкость построения пирамиды будет равна

$$f_{pyr} = d * f_{pyrbody} = \begin{cases} 17 \cdot \log_2 N, & \text{в лучшем случае,} \\ 27 \cdot \log_2 N, & \text{иначе.} \end{cases} \tag{2.19}$$

Трудоемкость начального цикла построения пирамиды равна

$$f_1 = 2 + \frac{N}{2} \cdot (2 + 1 + f_{pyr}), \tag{2.20}$$

$$f_1 = \begin{cases} 2 + \frac{1}{2} \cdot 3 \cdot N + \frac{1}{2} \cdot 17 \cdot N \cdot \log_2 N, & \text{в лучшем случае,} \\ 2 + \frac{1}{2} \cdot 3 \cdot N + \frac{1}{2} \cdot 27 \cdot N \cdot \log_2 N, & \text{иначе.} \end{cases} \tag{2.21}$$

Трудоемкость непосредственно цикла сортировки равна

$$f_2 = 2 + N \cdot (7 + 1 + 2 + 1 + f_{pyr}), \tag{2.22}$$

$$f_2 = \begin{cases} 2 + 11 \cdot N + 17 \cdot N \cdot \log_2 N, & \text{в лучшем случае,} \\ 2 + 11 \cdot N + 27 \cdot N \cdot \log_2 N, & \text{иначе.} \end{cases} \tag{2.23}$$

Трудоемкость пирамидальной сортировки будет равна

$$f = f_1 + f_2 = \begin{cases} 4 + 12,5 \cdot N + 25.5 \cdot N \cdot \log_2 N, & \text{в лучшем случае,} \\ 4 + 12,5 \cdot N + 40.5 \cdot N \cdot \log_2 N, & \text{иначе.} \end{cases} \tag{2.24}$$

Таким образом, асимптотика алгоритма пирамидальной сортировки равна $O(N \cdot \log N)$.

2.4.4. Трудоемкость алгоритма сортировки подсчетом

Трудоемкость заполнения вспомогательного массива равна

$$f_1 = 2 + 4 \cdot N. \tag{2.25}$$

Трудоемкость размещения элементов в результирующий массив равна

$$f_2 = 2 + R \cdot (2 + Q \cdot (2 + 3)), \tag{2.26}$$

где R — максимальное число в массиве, Q — количество элементов, равных данному числу.

$$f_2 = 2 + 2 \cdot R + 5 \cdot Q \cdot R = 2 + 2 \cdot R + 5 \cdot N. \tag{2.27}$$

Трудоемкость алгоритма сортировки подсчетом будет равна

$$f = f_1 + f_2 = 4 + 9 \cdot N + 2 \cdot R. \tag{2.28}$$

В данном случае расположение элементов не влияет на скорость алгоритма, влияет только величина элеменов. Таким образом, худшим случаем для данного алгоритма будет сортировки больших чисел. Лучшим случаем будет сортировка одинаковых элементов небольшого размера, например, сортировка нулей. Асимптотика данного алгоритма равна $O(N + R)$.

3. Технологическая часть

В данном разделе представлены реализации алгоритмов сортировки. Кроме того, указаны требования к ПО и средства реализации.

3.1. Требования к ПО

— программа позволяет вводить имя файла, содержащего информацию о массиве, с помощью аргументов командной строки;

— программа аварийно завершается в случае ошибок, выводя сообщение о соответствующей ошибке;

— программа выполняет замеры времени работы реализаций алгоритмов;

— программа строит зависимости времени работы реализаций алгоритмов от размеров входных данных;

— программа принимает на вход непустые массивы, состоящие из целых чисел.

3.2. Средства реализации

Для реализации данной работы выбран язык программирования Go, так как он содержит необходимые для тестирования библиотеки, а также обладает достаточными инструментами для реализации ПО, удовлетворяющего требованиям данной работы [4].

3.3. Реализации алгоритмов

В листингах 3.1 – 3.5 представлены реализации алгоритмов сортировки.

Листинг 3.1 — Исходный код реализации алгоритма блочной сортировки

```
func (m *Bucket) getBucketFunc(max, min int) (func(int) int, int) {
        d := max - min + 1
        n := m.buckets
        if n > d {
                n = d
        }

        size := d / n
        if d%n != 0 {
                size++
        }

        return func(k int) int {
                k -= min
                bucket := k / size
                if bucket > n-1 {
                        bucket = n - 1
                }
                return bucket
        }, n
}

func (m *Bucket) isSorted(a []int) bool {
        for i := 1; i < len(a); i++ {
                if a[i] < a[i-1] {
                        return false
                }
        }

        return true
}
```

Листинг 3.2 — Исходный код реализации алгоритма блочной сортировки (продолжение листинга 3.1)

```
func (m *Bucket) sort(a []int) []int {
        if len(a) <= 1 {
                return a
        }

        f, n := m.getBucketFunc(utils.Max(a...),
        utils.Min(a...))

        buckets := make([][]int, n)

        for _, e := range a {
                buckets[f(e)] = append(buckets[f(e)], e)
        }

        for i := range buckets {
                if !m.isSorted(buckets[i]) {
                        buckets[i] = m.sort(buckets[i])
                }
        }

        res := make([]int, 0, len(a))
        for _, b := range buckets {
                for _, e := range b {
                        res = append(res, e)
                }
        }

        return res
}
```

Листинг 3.3 — Исходный код реализации алгоритма пирамидальной сортировки

```
func (m *Heap) heapify(a []int, i, heapSize int) []int {
        left := i<<1 + 1
        right := i<<1 + 2

        largest := i
        if left < heapSize && a[left] > a[i] {
                largest = left
        }

        if right < heapSize && a[right] > a[largest] {
                largest = right
        }

        if i != largest {
                tmp := a[i]
                a[i] = a[largest]
                a[largest] = tmp
                a = m.heapify(a, largest, heapSize)
        }

        return a
}

func (m *Heap) sort(a []int) []int {
        heapSize := len(a)

        for i := len(a)/2 - 1; i >= 0; i-- {
                a = m.heapify(a, i, heapSize)
        }
```

Листинг 3.4 — Исходный код реализации алгоритма пирамидальной сортировки (продолжение листинга 3.3)

```
    for i := len(a) - 1; i >= 1; i-- {
                tmp := a[0]
                a[0] = a[i]
                a[i] = tmp

                heapSize--a
                a = m.heapify(a, 0, heapSize)
        }

        return a
}
```

Листинг 3.5 — Исходный код реализации алгоритма сортировки подсчетом

```
func (m *Count) sort(a []int) []int {
        counts := make([]int, utils.Max(&m.stat, a...)+1)
        for _, e := range a {
                counts[e]++
        }

        res := make([]int, 0, len(a))
        for i, c := range counts {
                for j := 0; j < c; j++ {
                        res = append(res, i)
                }
        }

        return res
}
```

3.4. Тестирование

Тестирование проводилось по методологии чёрного ящика. **Тесты пройдены успешно.**

В таблице представлены тестовые данные для реализаций алгоритмов сортировки.

Таблица 3.1 — Тестовые данные для алгоритмов сортировки

№	Массив	Результат
1	[5, 6, 2, 6, 1, 7]	[1, 2, 5, 6, 6, 7]
2	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3	[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4	[1, 1, 1, 1, 1, 1]	[1, 1, 1, 1, 1, 1]
5	[1]	[1]
6	[]	[]

4. Экспериментальная часть

В данном разделе описаны замерные эксперименты и представлены результаты исследования.

4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование [6]:

— 8 ГБ оперативной памяти;

— процессор Apple M2 (тактовая частота — до 3.5ГГц);

— операционная система macOS Ventura 13.0.

4.2. Измерение процессорного времени выполнения реализаций алгоритмов

Для измерения процессорного времени выполнения реализаций алгоритмов была использована функция языка C — $clock\_gettime$, которая позволяет получить текущее процессорное время в наносекундах [7].

4.2.1. Худший случай

В таблице 4.1 представлены результаты измерений процессорного времени выполнения в зависимости от размера массива для худших случаев алгоритмов сортировок. На рисунке 4.1 представлена зависимость времени выполнения от длины массива.

Таблица 4.1 — Результаты замеров процессорного времени для худшего случая (в нс)

Длина массива	Блочная	Пирамидальная	Подсчетом
1000	806800	460230	849980
2000	1474230	1022930	840620
3000	2153590	1680120	894170
4000	2894810	2230680	816990
5000	3636540	2893940	880270
6000	4500900	3568040	835930
7000	5038730	4188360	840940
8000	5761830	4856270	846700
9000	6441740	5500050	847050
10000	7223430	6159050	850000

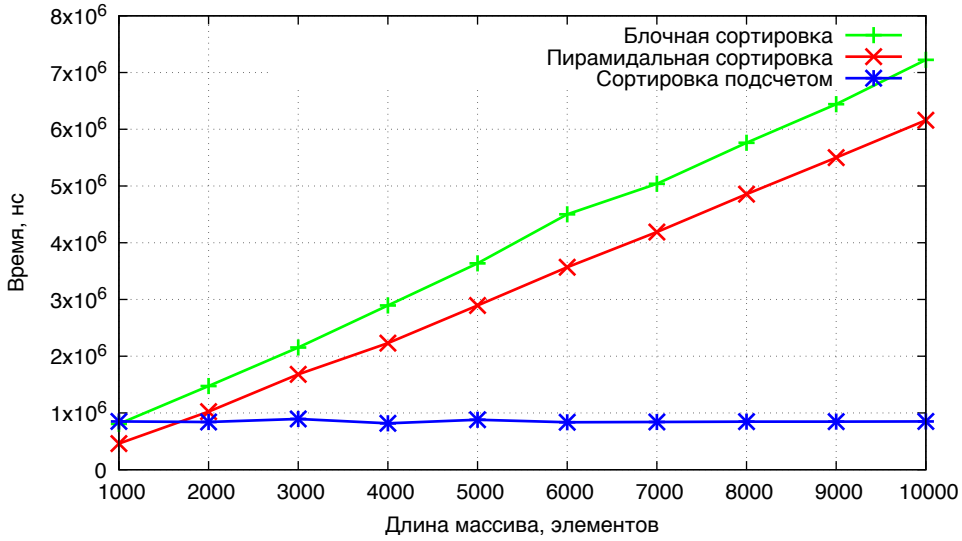

Рисунок 4.1 — Результаты замеров времени для худшего случая

4.2.2. Лучший случай

В таблице 4.2 представлены результаты измерений процессорного времени выполнения в зависимости от размера массива для лучших случаев алгоритмов сортировок. На рисунке 4.2 представлена зависимость времени выполнения от длины массива.

Таблица 4.2 — Результаты замеров процессорного времени для лучшего случая (в нс)

Длина массива	Блочная	Пирамидальная	Подсчетом
1000	178780	98770	13360
2000	244380	148500	10950
3000	302290	203300	15100
4000	400050	272240	18780
5000	497950	339660	25180
6000	616370	406350	26280
7000	705840	478020	33700
8000	792700	547480	35440
9000	904100	624620	42700
10000	997280	677370	47740

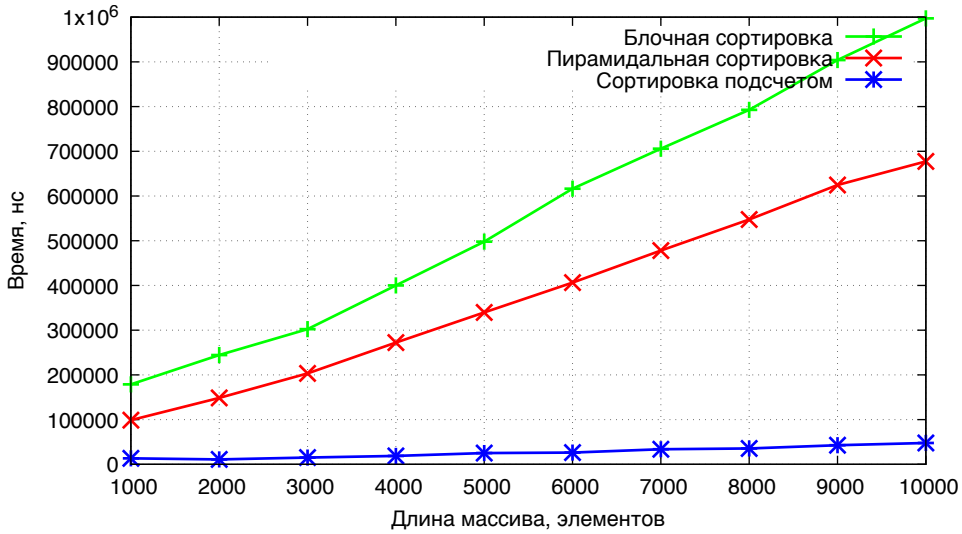

Рисунок 4.2 — Результаты замеров времени для лучшего случая

4.2.3. Случайный случай

В таблице 4.3 представлены результаты измерений процессорного времени выполнения в зависимости от размера массива для алгоритмов сортировок на случайных входных данных. На рисунке 4.3 представлена зависимость времени выполнения от длины массива.

Таблица 4.3 — Результаты замеров процессорного времени для случайного случая (в нс)

Длина массива	Блочная	Пирамидальная	Подсчетом
1000	499350	468690	20330
2000	1033070	1009490	22670
3000	1590200	1624070	29670
4000	2097670	2211810	40560
5000	2614480	2901950	48720
6000	3104580	3470970	55820
7000	3487130	4106930	59860
8000	3945090	4754490	66900
9000	4358250	5430200	69750
10000	4769100	6104990	74730

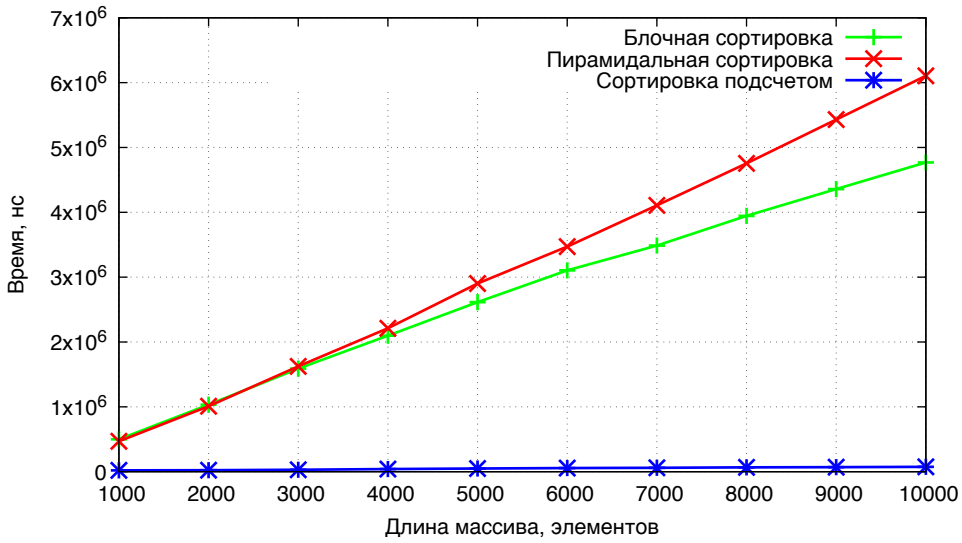

Рисунок 4.3 — Результаты замеров времени для случайного случая

4.3. Измерение объёма потребляемой памяти реализаций алгоритмов

В таблице 4.4 представлены результаты измерения потребляемой памяти в зависимости от длины массива. На рисунке 4.4 представлена зависимость потребляемой памяти от длины массива.

Таблица 4.4 — Результаты замеров потребляемой памяти (в байтах)

Длина массива	Блочная	Пирамидальная	Подсчетом
1000	24336	8056	32136
2000	48336	16056	56136
3000	72336	24056	80144
4000	96336	32056	104144
5000	120336	40056	128144
6000	144336	48056	152144
7000	168336	56056	176144
8000	192336	64056	200144
9000	216336	72056	224144
10000	240336	80056	248144

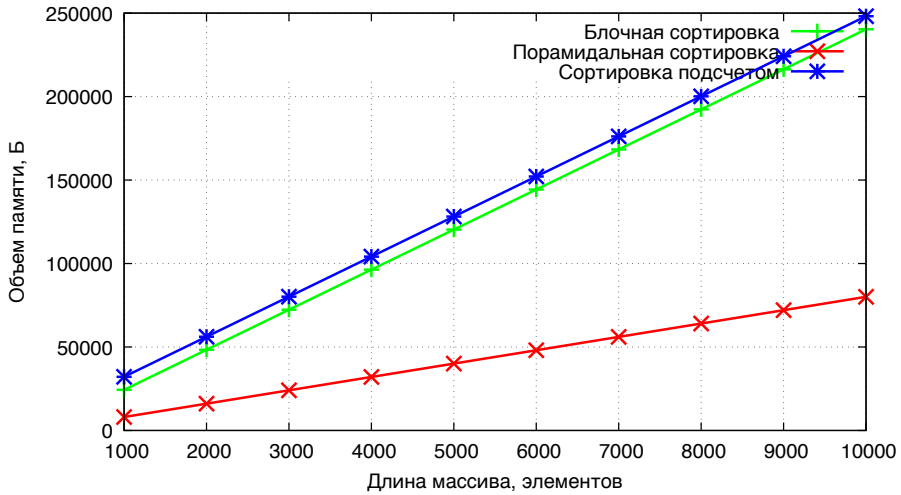

Рисунок 4.4 — Результаты замеров памяти

Заключение

Для алгоритмов сортировки были рассмотрены лучшие и худшие случаи.

Для алгоритма блочной сортировки худшим случаем является случай, когда элементы разбросаны в небольшое количество сильно удаленных в плане величины групп таким образом, что в результате деления на блоки большинство элементов попадает в один блок. Лучшим случаем для алгоритма блочной сортировки является отсортированный по возрастанию массив.

Для алгоритма пирамидальной сортировки нет конкретного худшего случая, он работает примерно одинаковое время на практически любых наборах данных, кроме массивов, в которых все элементы равны. В этом случае, количество сравнений и перестановок минимально — это лучший случай.

Для алгоритма сортировки подсчетом худшим случаем является случай, когда максимальный элемент массива очень большой. Лучшим случаем для алгоритма сортировки подсчетом является массив, состоящий их нулей.

Все дальнейшие выводы приведены исходя из анализов данных замеров для массива размером 10000, так как данная размерность позволяет получить достаточно показательные результаты для проведения сравнения.

По итогам замеров можно сказать, что в худшем случае по сравнению с лучшим алгоритмы сортировки заметно деградируют в плане производительности:

— время работы алгоритма блочной сортировки в худшем случае в 7.24 раз больше, чем в лучшем;

— время работы алгоритма пирамидальной сортировки в худшем случае в 9.09 раз больше, чем в лучшем;

— время работы алгоритма сортировки подсчетом в худшем случае в 17.81 раз больше, чем в лучшем.

В худшем случае, время работы алгоритма блочной сортировки больше времени работы алгоритма пирамидальной сортировки в 1.17 раз. Время работы алгоритма блочной сортировки больше времени работы алгоритма сортировки подсчетом в 8.49 раз.

В лучшем случае, время работы алгоритма блочной сортировки больше времени работы алгоритма пирамидальной сортировки в 1.47 раз. Время работы алгоритма блочной сортировки больше времени работы алгоритма сортировки подсчетом в 20.88 раз.

На наборе случайных элементов время работы алгоритма пирамидальной сортировки больше времени работы алгоритма блочной сортировки в 1.28 раз. Время работы алгоритма пирамидальной сортировки больше времени работы алгоритма сортировки подсчетом в 81.69 раз.

С точки зрения потребления памяти, наименее эффективной является сортировка подсчетом. Объем потребляемой памяти сортировки подсчетом больше, чем у блочной сортировки в 1.03 раз, и больше, чем у пирамидальной сортировки в 3.09 раз.

Цель работы была достигнута: были изучены алгоритмы сортировки. Были выполнены все задачи:

— изучены алгоритмы сортировки;

— проанализированы трудоемкости данных алгоритмов;

— кодированы данные алгоритмы;

— проведен замерный эксперимент для данных алгоритмов, с измерением времени работы и использования памяти;

— проведен сравнительный анализа алгоритмов на основе полученных данных.

Список использованных источников

1. Скворцов С. В., Пюрова Т. А. Параллельные алгоритмы сортировки данных и их реализация на платформе CUDA //Вестник Рязанского государственного радиотехнического университета. – 2016. – №. 58. – С. 42-48.

2. Удов Г. Г., Шалыто А. А. Классическая и автоматная реализации алгоритма пирамидальной сортировки набора чисел.

3. Зубков С. Assembler. Для DOS, Windows и Unix. – Litres, 2022.

4. Документация по языку программирования *Go* [Электронный ресурс]. Режим доступа: *https://go.dev/doc* (дата обращения: 07.10.2022).

5. Документация по пакетам языка программирования *Go* [Электронный ресурс]. Режим доступа: *https://pkg.go.dev* (дата обращения: 07.10.2022).

6. Техническая спецификация ноутбука *MacBookAir* [Электронный ресурс]. Режим доступа: *https://support.apple.com/kb/SP869* (дата обращения: 08.10.2022).

7. Документация по функции *clock_gettime* [Электронный ресурс]. Режим доступа: *https://man7.org/linux/man-pages/man3/clock_gettime.3.html* (дата обращения: 25.10.2022).