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»

на тему: «Трудоемкость сортировок»

Студент группы ИУ7-56Б _____ (Подпись, дата) Тарба А. В. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Строганов Д. В. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Строганов Ю. В.. (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

Целью данной лабораторной работы является исследование алгоритмов сортировки.

Для реализации поставленной цели необходимо выполнить следующие задачи:

— описать блочный алгоритм сортировки, быстрый и расческой;

— создать программное обеспечение, реализующее указанные алгоритмы сортировки;

— оценить трудоемкости сортировок;

— измерить процессорное время работы реализаций алгоритмов.

1 Аналитическая часть

В данном разделе будут рассмотрены три алгоритма сортировок: блочная, быстрая, пузырьком и расческой.

1.1 Блочная сортировка

Блочный алгоритм сортировки основан на разделении исходного массива на небольшие блоки и последующей их сортировке. Сначала осуществляется подготовка массива блоков, который инициализируется пустыми. Затем происходит распределение элементов исходного массива по соответствующим блокам в зависимости от их значений, используя различные стратегии, например, по интервалам или каким-то другим критериям.

Далее каждый блок сортируется по отдельности с использованием другого алгоритма сортировки, такого как сортировка вставками или слиянием. После этого отсортированные блоки объединяются обратно в один отсортированный массив.

Суть процесса работы алгоритма блочной сортировки состоит в том, что массив данных разбивается на блоки, которые сортируются по отдельности, а затем объединяются в общий отсортированный массив. Этот подход позволяет упростить задачу сортировки путем разделения данных на более мелкие части и последующей их сортировки, что в конечном итоге приводит к ускорению процесса сортировки.

1.2 Быстрая сортировка

Процесс работы алгоритма быстрой сортировки начинается с выбора опорного элемента из массива. Обычно в качестве такого элемента используется средний элемент массива или случайный элемент. Затем остальные элементы массива сравниваются с выбранным опорным элементом и перераспределяются таким образом, чтобы все элементы с меньшим значением находились слева от опорного элемента, а с большим значением — справа от него.

Далее массив разбивается на две части, используя опорный элемент в качестве разделителя. Каждая из новых частей подмассива снова сортируется путем выбора нового опорного элемента и его перемещения на нужное место.

Этот процесс продолжается рекурсивно для каждого из подмассивов, пока размер каждого подмассива не станет равен 1 или 0. Затем отсортированные подмассивы объединяются в общий отсортированный массив.

Суть процесса работы алгоритма быстрой сортировки такова: разделение массива на подзадачи, сортировка каждой из подзадач и объединение отсортированных подмассивов.

1.3 Сортировка пузырьком

Суть алгоритма сортировки пузырьком состоит в сравнении пар соседних элементов массива и их последовательном обмене до тех пор, пока массив не будет отсортирован. На каждой итерации больший элемент «всплывает» вверх. Этот процесс повторяется до тех пор, пока не будет произведено ни одного обмена элементов на одной итерации.

Каждая итерация начинается с прохода по массиву сравнением пары соседних элементов. Если элементы не упорядочены (меньший элемент находится справа от большего), происходит их обмен. После этого алгоритм переходит к следующей паре элементов и повторяет процесс. Это продолжается до тех пор, пока в ходе одной итерации не произойдет ни одного обмена, что означает, что массив уже отсортирован.

Суть работы алгоритма сортировки пузырьком заключается в последовательном проходе по массиву и сравнении соседних элементов до тех пор, пока все элементы не будут находиться в правильном порядке.

1.3.1 Сортировка расческой

Сортировка расческой — улучшение сортировки пузырьком. Ее идея состоит в том, чтобы ускорить работу алгоритма при появлении элементов с небольшими значениями в конце массива, которые замедляют работу

алгоритма.

В отличие от сортировки пузырьком сортировка расческой производит сравнение не соседних элементов, а элементов находящихся на некотором расстоянии друг от друга. Это расстояние выбирается не случайно, а определяется специальным образом. Для этого вводится специальная величина $factor = 1.247$. Начальное расстояние определяется как $dist = \frac{size}{factor}$, где $size$ - размер сортируемого массива. При каждом следующем проходе расстояние уменьшается в $factor$ раз. Таким образом расстояние сравнения элементов на i-ом шаге описывается по формуле 1.1

$$dist_i = \frac{dist_{i-1}}{factor} \tag{1.1}$$

Вывод

В данном разделе были рассмотрены алгоритмы блочной, быстрой сортировки, сортировки пузырьком и ее усовершенствование — сортировка расческой.

2 Конструкторская часть

В данном разделе приведены схемы алгоритмов сортировок: блочная, быстрая, расческой.

2.1 Разработка алгоритмов

На вход алгоритмов подаются указатель на массив $array$ и размер массива n - целое положительное число.

На рисунках 2.1 – 2.3 представлены схемы алгоритмов сортировок: блочная, быстрая, расческой.

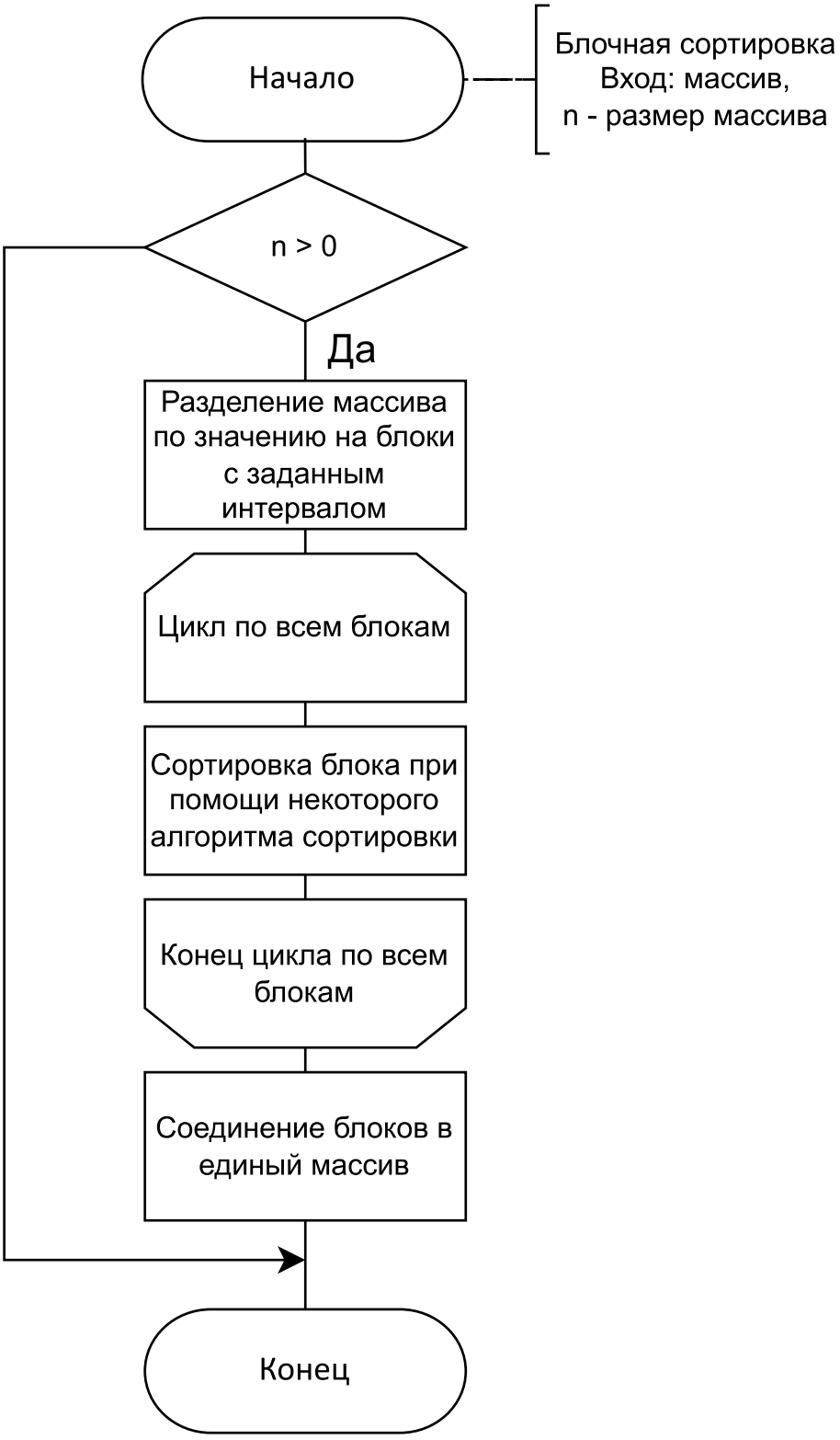

Рисунок 2.1 – Схема алгоритма блочной сортировки

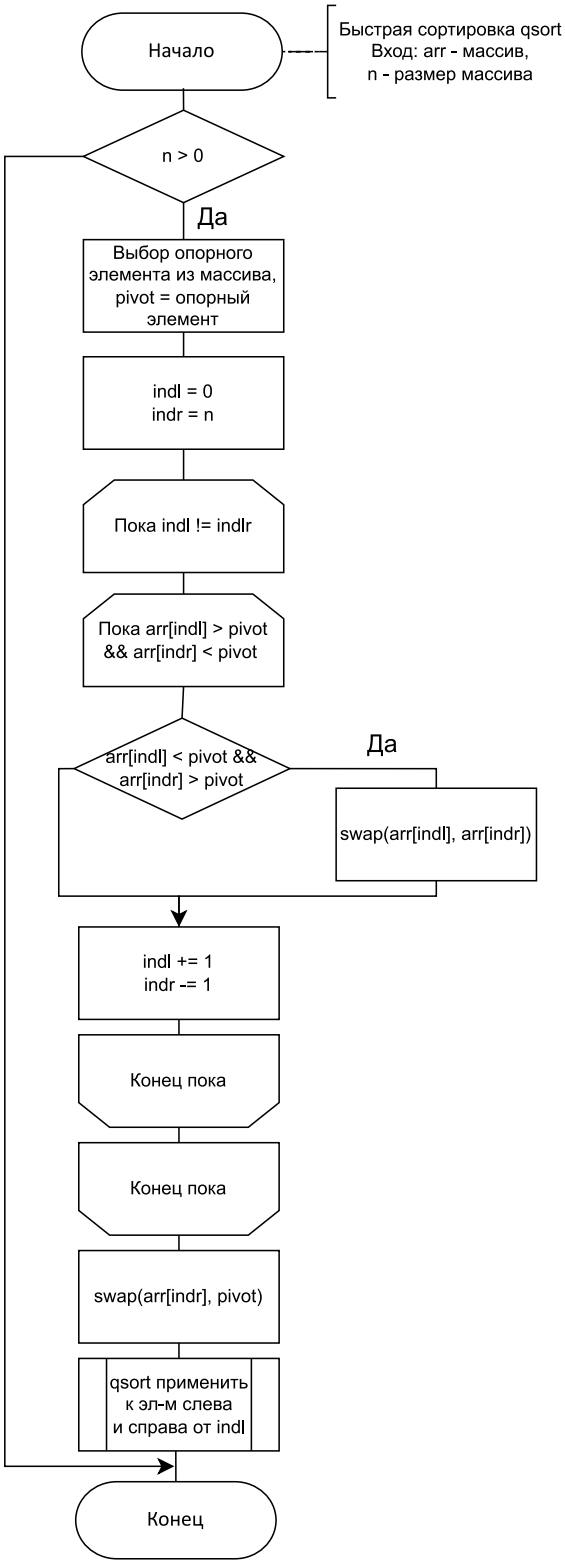

Рисунок 2.2 – Схема алгоритма быстрой сортировки

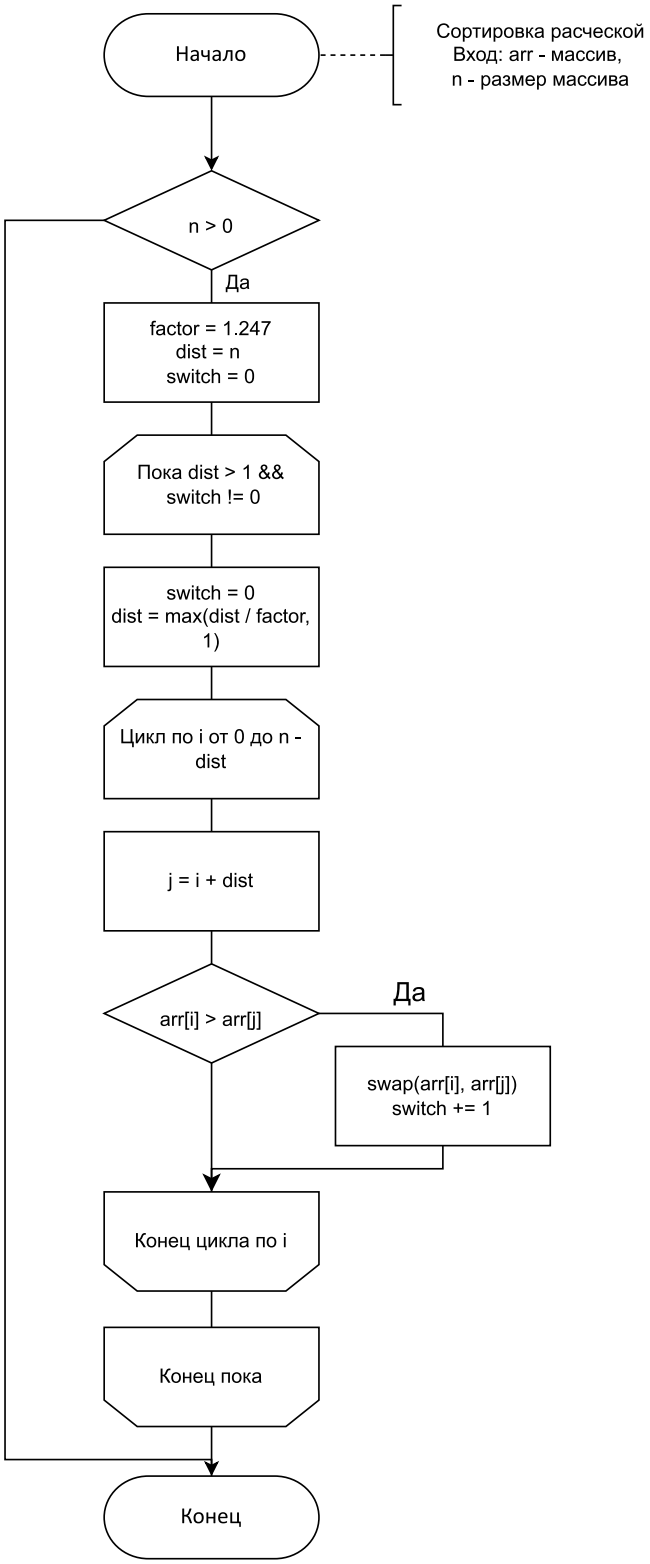

Рисунок 2.3 – Схема алгоритма сортировки расческой

2.2 Модель вычислений для проведения оценки трудоемкости

Введем модель вычислений, примем трудоемкость операций следующим образом:

- $+, -, =, + =, - =, ==, ! =, <, >, <=, >=, [], ++, --, \&\&, >>, <<, ||, \&, |$ равной 1;
- $*, /, \%, * =, / =, \% =$ равной 2;
- трудоемкость условного оператора

$$f_{if} = f_{\text{условия}} + \begin{cases} min(f_1, f_2), & \text{лучший случай} \\ max(f_1, f_2), & \text{худший случай,} \end{cases}$$

  где f_1 - трудоемкость при выполненном условии, f_2 - трудоемкость при невыполненном условии.

- трудоемкость цикла:

$$f_{for} = f_{\text{инициализация}} + f_{\text{сравнения}} + M_{\text{итераций}} \cdot (f_{\text{тело}} + f_{\text{инкремент}} + f_{\text{сравнения}})$$

- Трудоемкость передачи параметра в функции и возврат из функции равны 0.

2.3 Трудоемкость алгоритмов сортировки

2.3.1 Алгоритм блочной сортировки

Трудоемкость в лучшем случае при отсортированном массиве, когда ничего не обменивается, но все же данные рассматриваются. Выведена по

формуле (2.1).

$$f_{best} = 3 + 4 + \frac{N}{4} \cdot (3 + 2 + log(N) \cdot (2 + 4 + 4)) = \\ = 7 + \frac{5N}{4} + \frac{5N \cdot log(N)}{2} = O(N \cdot log(N)) \tag{2.1}$$

Трудоемкость в худшем случае при отсортированном массиве в обратном порядке. Выведена формуле (2.2).

$$f_{worst} = 7 + \frac{N}{4} \cdot (5 + log(N) \cdot (10 + log(N) \cdot (6 + 4))) = \\ = 7 + \frac{5N}{4} + \frac{5N \cdot log(N)}{2} + \frac{5N \cdot log^2(N)}{2} = O(N \cdot log^2(N)) \tag{2.2}$$

2.3.2 Алгоритм быстрой сортировки

Трудоемкость в лучшем случае при отсортированном массиве описана формулой (2.3).

$$f_{best} = log_2 N(1 + 2 + \frac{N-1}{2}(1 + 2) + 2 + \frac{N-1}{2}(1 + 2) + 3 + 2) = \\ = log_2 N(1 + 9 + 3(N-1)) = log_2 N(7 + 3N) = \\ = 7log_2 N + 3N log_2 N = O(NlogN) \tag{2.3}$$

Трудоемкость в худшем случае при отсортированном массиве в обратном порядке описана формулой (2.4).

$$f_{worst} = N(1 + 2 + \frac{N-1}{2}(1 + 2) + 2 + \frac{N-1}{2}(1 + 2) + 3 + 2) \\ = N(1 + 9 + 3(N-1)) = N(7 + 3N) = \\ = 7N + 3N^2 = O(N^2) \tag{2.4}$$

2.3.3 Алгоритм сортировки расческой

Трудоемкость в лучшем случае при отсортированном массиве и худшем случае при неотсортированном массиве в обратном порядке описана

формулами 2.5 и 2.6.

$$f_{best} = 2 + log_{shrink}N(2 + 1 + 1 + 2 + (N - 1)(2 + 4 + 1) + 2) = \\ = 2 + log_{shrink}N + log_{shrink}N = O(NlogN) \tag{2.5}$$

$$f_{worst} = 2 + (N - 1)(2 + 1 + 1 + 2(N - 1)(2 + 4 + 1 + 6 + 1) + 2) = \\ = 2 + (N - 1)(8 + 14(N - 1)) = 2 + (N - 1)(14N - 6) = \\ = 2 + 14N^2 - 6N - 14N + 6 = 14N^2 - 20N + 8 = O(N^2) \tag{2.6}$$

Вывод

В данном разделе на основе теоретических данных были построены схемы требуемых алгоритмов, выбраны используемые типы данных. Для каждого алгоритма сортировки были выведены трудоемкости худшего и лучшего случаев.

3 Технологическая часть

В данном разделе будут приведены средства реализации, листинг кода и функциональные тесты.

3.1 Средства реализации

Для реализации данной лабораторной работы был выбран язык *go* [1]. Данный выбор обусловлен наличием у языка встроенного utf8, позволяющего хранить как кириллические символы, так и латинские, что позволит правильно обрабатывать строки на нескольких языках. Также в *go* есть поддержка вставок кода на языке *C*, что будет использовано для замеров процессорного времени при помощи вызова функции языка *C* *clock_gettime* [2].

3.2 Сведения о модулях программы

Данная программа представляет из себя набор следующих модулей:

— `main.go` — основной файл, реализующий интерфейс для взаимодействия с алгоритмами;

— `algs.go` — файл содержит реализации алгоритмов сортировок;

— `utils.go` — файл содержит вспомогательные функции;

— `time.go` — файл содержит функции, для измерения времени.

3.3 Реализация алгоритмов

В листингах 3.1 – 3.3 реализованы алгоритмы сортировки, а именно блочная, быстрая и расческой.

Листинг 3.1 – Функция быстрой сортикровки

```
1    func quickSort(arr []int) []int {
2        if len(arr) < 2 {
3            return arr
4        }
5
6        pivotIndex := len(arr) / 2
7        pivot := arr[pivotIndex]
8
9        var less, greater, equal []int
10       for _, num := range arr {
11           if num < pivot {
12               less = append(less, num)
13           } else if num > pivot {
14               greater = append(greater, num)
15           } else {
16               equal = append(equal, num)
17           }
18       }
19
20       // Рекурсивно сортируем подмассивы и объединяем результа
             ты
21       sortedArr := append(quickSort(less), equal...)
22       sortedArr = append(sortedArr, quickSort(greater)...)
23
24       return sortedArr
25   }
```

Листинг 3.2 – Функция блочной сортировки

```
func blockSort(arr []int, bucketSize int) []int {
    if len(arr) == 0 {
        return arr
    }

    // Находим минимальное и максимальное значение в массиве
    min := arr[0]
    max := arr[0]
    for _, num := range arr {
        if num < min {
            min = num
        } else if num > max {
            max = num
        }
    }
    // Инициализируем карманы
    bucketCount := (max-min)/bucketSize + 1
    buckets := make([][]int, bucketCount)
    for i := 0; i < bucketCount; i++ {
        buckets[i] = make([]int, 0)
    }

    // Распределяем значения по карманам
    for _, num := range arr {
        bucketIndex := (num - min) / bucketSize
        buckets[bucketIndex] = append(buckets[bucketIndex], num)
    }

    // Сортируем карманы и объединяем отсортированные значения
    sortedArr := make([]int, 0)
    for i := 0; i < bucketCount; i++ {
        if len(buckets[i]) > 1 {
            // sort.Ints(buckets[i])
            buckets[i] = combSort(buckets[i])
        }
        sortedArr = append(sortedArr, buckets[i]...)
    }

    return sortedArr
}
```

Листинг 3.3 – Функция сортировки расческой

```
func combSort(arr []int) []int {
    n := len(arr)
    if n == 0 {
        return arr
    }

    gap := n
    shrink := 1.3
    swapped := true

    for gap > 1 || swapped {
        // Вычисляем новый интервал
        gap = int(float64(gap) / shrink)

        if gap < 1 {
            gap = 1
        }

        swapped = false
        // Совершаем проход по массиву
        for i := 0; i+gap < n; i++ {
            if arr[i] > arr[i+gap] {
                // Меняем местами элементы, если это необходимо
                arr[i], arr[i+gap] = arr[i+gap], arr[i]
                swapped = true
            }
        }
    }

    return arr
}
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для разработанных алгоритмов сортировки. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Массив	**Размер**	**Ожидаемый р-т**	**Фактический результат**
41 67 34 0 69	5	0 34 41 67 69	0 34 41 67 69
31 57 24 -10 59	5	-10 24 31 57 59	-10 24 31 57 59
1 2 3 4 5	5	1 2 3 4 5	1 2 3 4 5
100 88 76 65 43	5	43 65 76 88 100	43 65 76 88 100
-59 -33 -66 -100 -31	5	-100 -66 -59 -33 -31	-100 -66 -59 -33 -31

Вывод

Были реализованы алгоритмы: блочная, быстрая сортировка и сортировка расческой.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее.

— Процессор: AMD Ryzen 7 4800H with Radeon Graphics 2.90 ГГц [3].

— Оперативная память: 16 ГБайт.

— Операционная система: Ubuntu 22.04.3 LTS [4].

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы программы на примере сортировки массива из пяти элементов блочной сортировкой.

```
1. Ввести массив.
2. Выбрать алгоритм.
3. Замерить процессорное время.
4. Сгенерировать CSV с замерами времени.
5. Выйти из программы.
Введите номер команды: 1
Введите размер массива: 5
Введите элемент 1: 5
Введите элемент 2: 4
Введите элемент 3: 3
Введите элемент 4: 2
Введите элемент 5: 1
1. Ввести массив.
2. Выбрать алгоритм.
3. Замерить процессорное время.
4. Сгенерировать CSV с замерами времени.
5. Выйти из программы.
Введите номер команды: 2
1. Блочная сортировка.
2. Сортировка расческой.
3. Быстрая сортировка.
5. Выйти назад.
Выберете алгоритм: 1
Отсортированный массив:
[1, 2, 3, 4, 5]
```

Рисунок 4.1 – Демонстрация работы программы

4.3 Временные характеристики

Результаты эксперимента замеров по времени приведены в таблице 4.1.

В таблице 4.1 приведены результаты замеров по времени для блочной сортировки, быстрой сортировки, и сортировки расческой.

Таблица 4.1 – Замеры по времени лучшего случая для сортировок, размер которых от 50 до 1000.

Количество элементов	**Время, нс**		
	Блочная	**Расческой**	**Быстрая**
50	27 325	1 796	15 122
100	21 526	1 691	31 925
200	32 425	3 062	69 585
300	35 646	4 623	109 676
400	47 683	7 082	145 337
500	51 852	8 953	152 219
600	57 140	9 193	215 313
700	67 613	12 110	246 304
800	73 004	12 345	295 223
900	81 659	14 515	305 741
1000	91 106	29 501	475 382

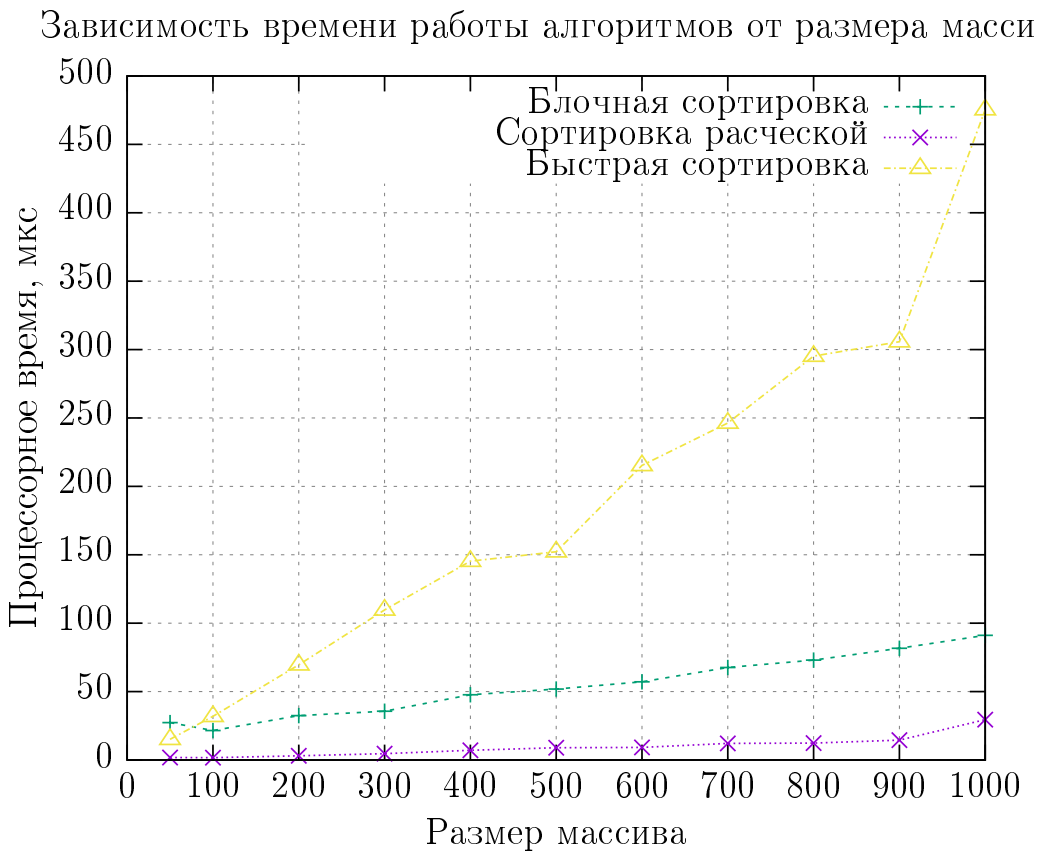

Рисунок 4.2 – Сравнение по памяти блочной, быстрой сортировок и сортировки расческой

4.4 Характеристики по памяти

Результаты замеров памяти приведены в таблице 4.2.

В таблице 4.2 приведены результаты замеров по памяти для блочной сортировки, быстрой сортировки, и сортировки расческой.

Таблица 4.2 – Замеры затраченной памяти сортировками для размеров массивов от 100 до 1000.

Количество элементов	**Память, байт**		
	Блочная	**Быстрая**	**Расческой**
100	19 288	3 168	15 848
200	22 304	6 368	38 008
300	25 448	9 568	69 240
400	28 672	12 768	88 897
500	33 368	15 968	96 497
600	38 192	19 168	164 882
700	40 104	22 368	178 474
800	47 504	25 568	199 682
900	53 400	28 768	207 442
1000	55 384	31 968	215 498

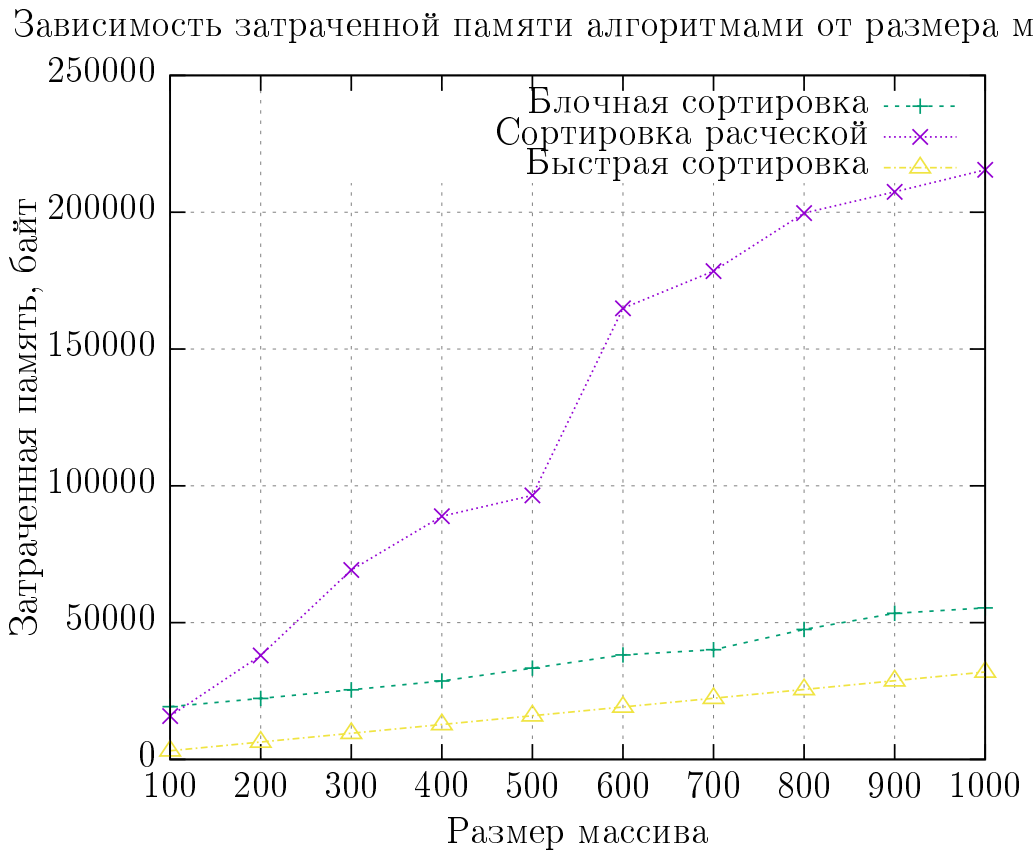

Рисунок 4.3 – Сравнение по времени блочной, быстрой сортировок и сортировки расческой

4.5 Вывод

В данном разделе было произведено сравнение количества затраченного времени вышеизложенных алгоритмов. Наименее затратным по времени оказалась сортировка расческой, наихудшее время показала быстрая сортировка, что может быть связано с ее рекурсивной реализацией.

Таким образом сортировка расческой превосходит блочную сортировку в 3 раза по процессорному времени работы на размере массива в 1000 элементов, а быструю сортировку в 16 раз на тех же размерах.

Наиболее эффективным алгоритмом с точки зрения затрачиваемой им памяти является алгоритм быстрой сортировки, он использует меньше памяти, чем блочная и расческой на всех размерах массива.

Заключение

Исходя из полученных результатов замеров по времени, быстрая сортировка показывает меньшую эффективность по процессорному времени работы на размерах массива, начиная с 100, чем сортировка расческой и блочная сортировка. Это связано с рекурсивной реализацией быстрой сортировки.

Таким образом при необходимости выбора алгоритма сортировки работающего быстрее остальных целесообразно использовать алгоритм сортировки расческой.

Наиболее эффективным алгоритмом по памяти является алгоритм быстрой сортировки он использует меньше дополнительной памяти, чем все остальные рассмотренные алгоритмы. На размерах массива равном 1000 он превосходит блочную сортировку в 1.7 раз, а сортировку расческой в 6.7 раз.

Целью данной лабораторной работы является исследование алгоритмов сортировки.

По результатам оценки трудоемкостей алгоритмов было выявлено, что трудоемкость в лучшем случае у всех алгоритмов совпадает и оценивается как $O(NlogN)$, однако лучшей оценкой трудоемкости в худшем случае обладает алгоритм блочной сортировки сортировки $O(Nlog^2N)$, а быстрая сортировка и сортировка расческой обладают трудоемкостью в худшем случае равной $O(N^2)$. Однако в результате исследование было показано, что с учетом особенностей реализаций алгоритмов наилучшим образом себя показал алгоритм сортировки расческой.

Для достижения поставленной цели были выполнены следующие задачи:

— описать блочный алгоритм сортировки, быстрый и расческой;

— создать программное обеспечение, реализующее указанные алгоритмы сортировки;

— оценить трудоемкости сортировок;

— измерить процессорное время работы реализаций алгоритмов.

Список использованных источников

1. Документация по golang [Электронный ресурс]. — Режим доступа: https://go.dev/doc/ (дата обращения: 19.10.2023).

2. C library function clock_gettime() [Электронный ресурс]. — Режим доступа: https://www.opennet.ru/man.shtml?topic=clock_gettime (дата обращения: 19.10.2023).

3. AMD [Электронный ресурс]. — Режим доступа: https://www.amd.com/en/products/apu/amd-ryzen-7-4800h (дата обращения: 19.10.2023).

4. Ubuntu 22.04.3 LTS [Электронный ресурс]. — Режим доступа: https://ubuntu.com/download/desktop (дата обращения: 19.10.2023).