Содержание

Введение **3**

1 Аналитическая часть **5**

1.1 Битонная сортировка 5

1.2 Блочная сортировка 5

1.3 Сортировка расческой 5

2 Конструкторская часть **7**

2.1 Разработка алгоритмов 7

2.2 Модель вычислений для проведения оценки трудоемкости . . . 11

2.3 Трудоёмкость алгоритмов 11

2.3.1 Алгоритм блочной сортировки 11

2.3.2 Алгоритм битонной сортировки 12

2.3.3 Алгоритм сортировки расческой 13

3 Технологическая часть **14**

3.1 Средства реализации 14

3.2 Список модулей 14

3.3 Функциональные тесты 17

4 Исследовательская часть **19**

4.1 Технические характеристики 19

4.2 Демонстрация работы программы 19

4.3 Время выполнения алгоритмов 21

Заключение **27**

Список источников **28**

Введение

Сортировка – перегруппировка некой последовательности, или кортежа, в определенном порядке. Это одна из главных процудур обработки структурированных данных. Расположение элементов в определенном порядке позволяет более эффективно проводить работу с последовательностью данных, в частности при поиске некоторых данных.

Существует множество алгоритмов сортировки, но любой алгоритм сортировки имеет:

- сравнение, которое определяет, как упорядочена пара элементов;
- перестановка для смены элементов местами;
- алгоритм сортировки, использующий сравнение и перестановки.

Что касается самого поиска, то при работе с отсортированным набором данных время, которое нужно на нахождение элемента, пропорционально логарифму количства элементов. Последовательность, данные которой расположены в хаотичном порядке, занимает время, которое пропорционально количству элементов, что куда больше логарифма.

Цель работы: исследование трудоемкости алгоритмов сортировки.

Задачи работы.

1. Изучить и реализовать алгоритмы сортировки: битонная, блочная и расческой.
2. Провести тестирование по времени для выбранных сортировок.
3. Провести сравнительный анализ трудоемкости алгоритмов на основе теоретических расчетов.
4. Провести сравнительный анализ реализаций алгоритмов по затраченному процессорному времени.

5. Описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В этом разделе будут рассмотрены алгоритмы сортировок – битонная, блочная, расческой.

1.1 Битонная сортировка

Битонная сортировка – размещение элемента входной последовательности на подходящее место выходной последовательности. [1]

Набор данных условно разделяется на входную последовательность и выходную. В начале отсортированная часть пуста. Каждый i-ый элемент, начиная с $i = 2$, входной последовательности размещается в уже отсортированную часть до тех пор, пока изначальные данные не будут исчерпаны.

1.2 Блочная сортировка

Блочная сортировка – сортировка, которая является модификацией сортировки пузырьком. Различие состоит в том, что в рамках одной итерации происходит проход по массиву в обоих направлениях. В сортировке пузырьком просходит только проход слева-направо, то есть в одном направлении. [2]

Суть сортировки – сначала идет обычный проход слева-направо, как при обычном пузырьке. Затем, начиная с элемента, который находится перед последним отсортированным, начинается проход в обратном направлении. Здесь также сравниваются элементы меняются местами при необходимости.

1.3 Сортировка расческой

Сортировка расческой. В сортировке пузырьком, когда сравниваются два элемента, промежуток (расстояние друг от друга) равен 1. Основная идея сортировки расчёской в том, что этот промежуток может быть гораздо больше, чем единица. [3]

Основная идея – устранить маленькие значения в конце списка, которые крайне замедляют сортировку пузырьком (большие значения в начале списка, не представляют проблемы для сортировки пузырьком).

Вывод

В данной работе необходимо реализовать алгоритмы сортировки, описанные в данном разделе, а также провести их теоритическую оценку и проверить ее экспериментально.

2 Конструкторская часть

В данном разделе будут рассмотрены схемы алгоритмов сортировок (битонная, блочная и расческой), а также найдена их трудоемкость.

Требования к программному обеспечению

Ряд требований к программе:

- на вход подается массив целых чисел в диапазоне от -10000 до 10000;
- возвращается отсортированный по месту массив, который был задан в предыдущем пункте.

2.1 Разработка алгоритмов

На рисунках 2.1, 2.2 и 2.3 представлены схемы алгоритмов сортировки – битонная, блочная и расческой.

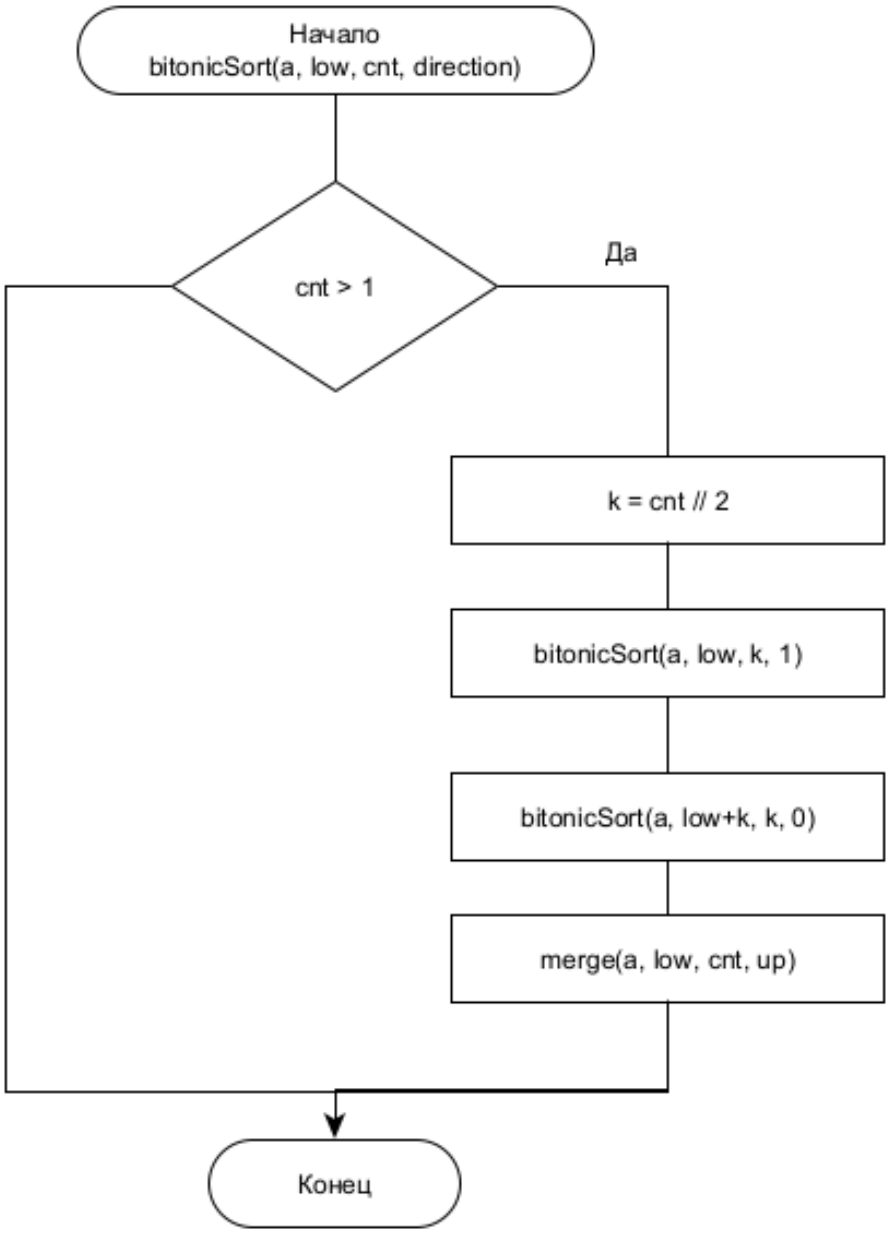

Рисунок 2.1 – Схема алгоритма битонной сортировки

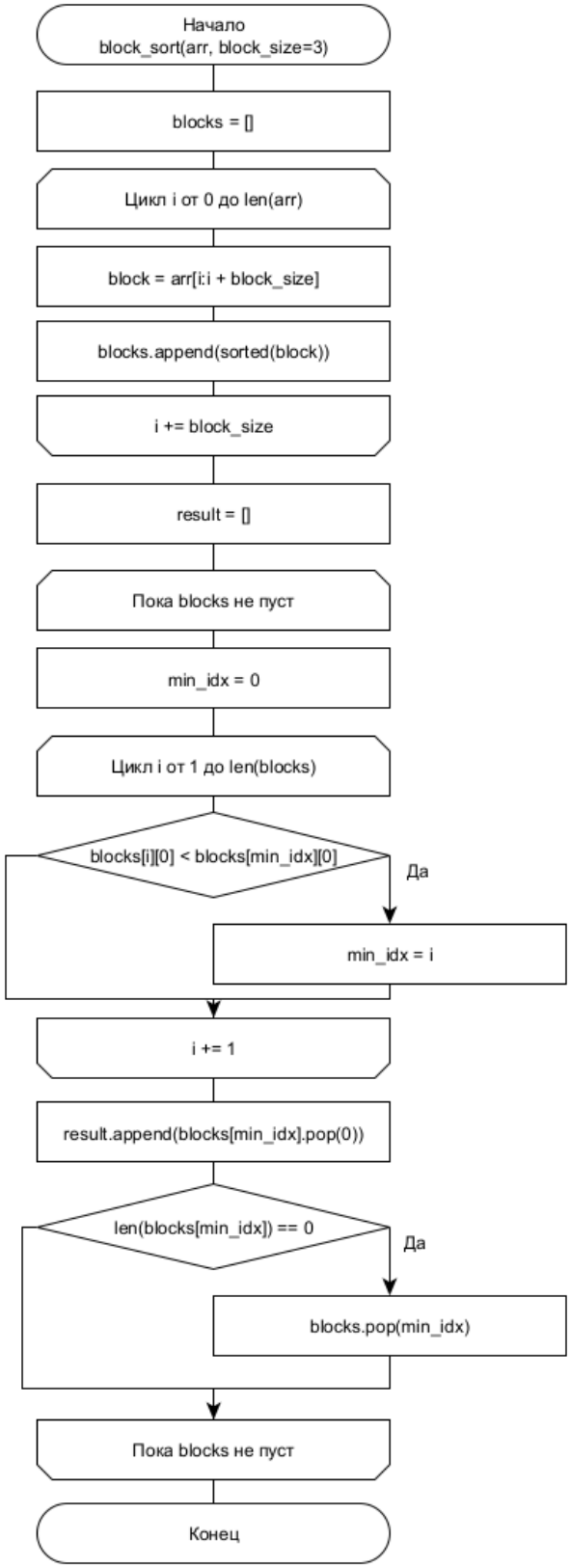

Рисунок 2.2 – Схема алгоритма блочной сортировки

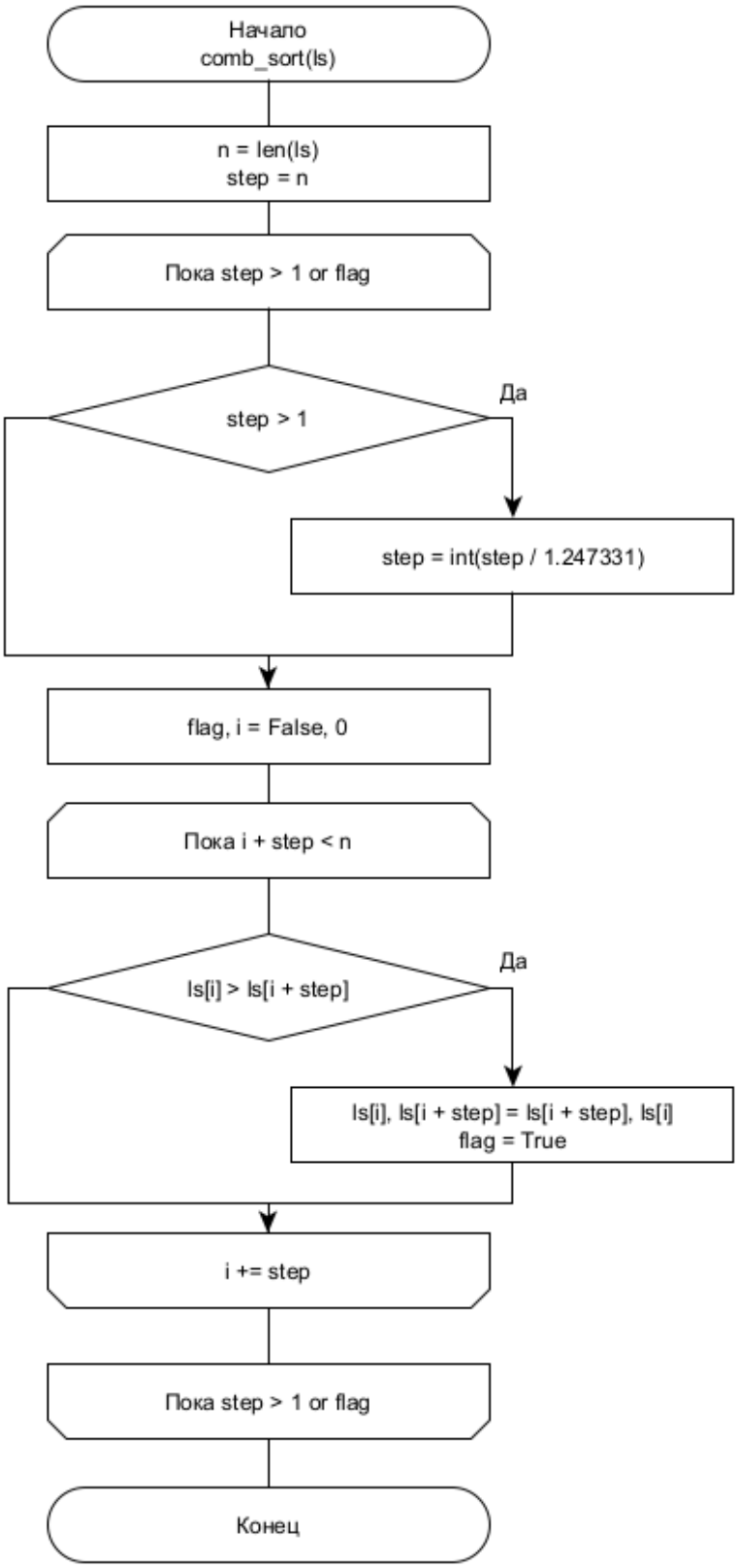

Рисунок 2.3 – Схема алгоритма сортировки расческой

2.2 Модель вычислений для проведения оценки трудоемкости

Введем модель вычислений [4], которая потребуется для определния трудоемкости каждого отдельно взятого алгоритма сортировки:

1. операции из списка (2.1) имеют трудоемкость равную 1;

$$+, -, /, *, \%, =, +=, -=, *=, /=, \%=, ==, !=, <, >, <=, >=, [], ++, --$$
(2.1)

2. трудоемкость оператора выбора `if условие then A else B` рассчитывается, как (2.2);

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

3. трудоемкость цикла рассчитывается, как (2.3);

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

4. трудоемкость вызова функции равна 0.

2.3 Трудоёмкость алгоритмов

Определим трудоемкость выбранных алгоритмов сортировки по коду.

2.3.1. Алгоритм блочной сортировки

- Трудоёмкость каждого из двух циклов, которая вычисляется по формуле (2.4).

$$f_{outer1} = 2 + 2 \cdot (N - 1). \quad (2.4)$$

- Суммарная трудоёмкость первого цикла, количество итераций которых меняется в промежутке $[1..N-1]$, которая вычисляется по формуле (2.7).

$$f_{inner1} = 2(N-1) + \frac{2 \cdot (N-1)}{2} \cdot (4 + f_{if1}). \tag{2.5}$$

- Трудоёмкость условия в первом цикле, которая равна (2.6).

$$f_{if1} = 4. \tag{2.6}$$

- Суммарная трудоёмкость второго цикла, количество итераций которых меняется в промежутке $[1..N-1]$, которая вычисляется по формуле (2.7).

$$f_{inner} = 2 \cdot (N-1) + f_{ins1}. \tag{2.7}$$

- Трудоёмкость внешнего цикла сортировки вставками, которая вычисляется по формуле(2.8).

$$f_{ins} = 3 \cdot (N-1) + f_{inner1}. \tag{2.8}$$

Трудоёмкость в **лучшем** случае (2.9).

$$f_{best} = 4 + \frac{3}{2}N + f_{ins} \approx K = O(N+K). \tag{2.9}$$

Трудоёмкость в **худшем** случае (2.10).

$$f_{worst} = 4 + \frac{3}{2}N + f_{ins} \approx K = O(N+K). \tag{2.10}$$

2.3.2. Алгоритм битонной сортировки

bitonicSort делит массив пополам и рекурсивно вызывает себя. Делить массив можно $logN$ раз.

$$f_{bitonicSort} = logN \cdot f_{merge} \tag{2.11}$$

При каждом вызове bitonicSort вызывает merge.

$$f_{merge} = 2 + (const) + logN \approx logN \tag{2.12}$$

$$f_{bitonicSort} = logN \cdot logN = log^2 N \tag{2.13}$$

Трудоёмкость в лучшем случае (2.14):

$$f_{best} = O(log^2(N)) \tag{2.14}$$

Трудоёмкость в худшем случае (2.15):

$$f_{best} = O(log^2(N)) \tag{2.15}$$

2.3.3. Алгоритм сортировки расческой

Трудоёмкость в лучшем случае (при уже отсортированном массиве) (2.16):

$$f_{best} = O(N \cdot logN) \tag{2.16}$$

Трудоёмкость в худшем случае (при массиве, отсортированном в обратном порядке) (2.17):

$$f_{worst} = O(N^2) \tag{2.17}$$

Вывод

Были разработаны схемы всех трех алгоритмов сортировки. Также для каждого из них были рассчитаны и оценены лучшие и худшие случаи.

Проведённая теоретическая оценка трудоемкости алгоритмов показала, что трудоёмкость выполнения алгоритма блочной сортировки составляет $O(N + K)$, расческой – от $O(N \cdot logN)$ до $O(N^2)$, битонной – $O(log^2(N))$.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги сортировок.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования $Python$[5]. В текущей лабораторной работе требуется замерить процессорное время для выполняемой программы, а также построить графики. Все эти инструменты присутствуют в выбранном языке программирования.

Время работы было замерено с помощью функции *process_time(...)* из библиотеки $time$[6].

3.2 Список модулей

Программа состоит из 4 модулей:

- $main.py$ – файл, содержащий весь служебный код;
- $bitonic.py$ – файл, содержащий код битонной сортировки;
- $block.py$ – файл, содержащий код блочной сортировки;
- $comb.py$ – файл, содержащий код сортировки расческой.

В листингах 3.1, 3.2, 3.3 представлены реализации алгоритмов сортировок (битонная, блочная, расческой).

Листинг 3.1 – Битонная сортировка

```
1
2  def compAndSwap(a, i, j, direction):
3      if (direction == 1 and a[i] > a[j]) or (direction == 0 and a[i]
       ↪  > a[j]):
4          a[i], a[j] = a[j], a[i]
5
6  # It recursively sorts a bitonic sequence in ascending order,
7  # if dir = 1, and in descending order otherwise (means dir=0).
8  # The sequence to be sorted starts at index position low,
9  # the parameter cnt is the number of elements to be sorted.
10 def merge(a, low, cnt, direction):
11     if cnt > 1:
12         k = cnt//2
13         for i in range(low, low+k):
14             compAndSwap(a, i, i+k, direction)
15         merge(a, low, k, direction)
16         merge(a, low+k, k, direction)
17
18 # This function first produces a bitonic sequence by recursively
19 # sorting its two halves in opposite sorting orders, and then
20 # calls merge to make them in the same order
21 def bitonicSort(a, low, cnt, direction):
22     if cnt > 1:
23         k = cnt//2
24         bitonicSort(a, low, k, 1)
25         bitonicSort(a, low+k, k, 0)
26         merge(a, low, cnt, direction)
27
28
29 def bitonic(a, up=True):
30     low = 0
31     cnt = len(a)
32     if cnt > 1:
33         k = cnt//2
34         bitonicSort(a, low, k, 1)
35         bitonicSort(a, low+k, k, 0)
36         merge(a, low, cnt, up)
37     return a
```

Листинг 3.2 – Блочная сортировка

```
def block_sort(arr, block_size=3):
    # Create an empty list to hold the sorted blocks
    blocks = []

    # Divide the input array into blocks of size block_size
    for i in range(0, len(arr), block_size):
        block = arr[i:i + block_size]

        # Sort each block and append
        # it to the list of sorted blocks
        blocks.append(sorted(block))

    # Merge the sorted blocks into
    # a single sorted list
    result = []
    while blocks:
        # Find the smallest element in the first block of each
        ↪ sorted block
        min_idx = 0
        for i in range(1, len(blocks)):
            if blocks[i][0] < blocks[min_idx][0]:
                min_idx = i

        # Remove the smallest element and
        # append it to the result list
        result.append(blocks[min_idx].pop(0))

        # If the block is now empty, remove
        # it from the list of sorted blocks
        if len(blocks[min_idx]) == 0:
            blocks.pop(min_idx)
    return result
```

Листинг 3.3 – Сортировка расческой

```
def comb_sort(ls):
    n = len(ls)
    step = n
    while step > 1 or flag:
        if step > 1:
            step = int(step / 1.247331)
        flag, i = False, 0
        while i + step < n:
            if ls[i] > ls[i + step]:
                ls[i], ls[i + step] = ls[i + step], ls[i]
                flag = True
            i += step
    return ls
```

3.3 Функциональные тесты

В таблице 3.2 приведены тесты для функций, реализующих алгоритмы сортировки. Тесты *для всех сортировок* пройдены успешно.

Таблица 3.1 – Функциональные тесты (блочная, расческой)

Входной массив	Ожидаемый результат	Результат
$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$
$[5, 4, 3, 2, 1]$	$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$
$[9, 7, -5, 1, 4]$	$[-5, 1, 4, 7, 9]$	$[-5, 1, 4, 7, 9]$
$[5]$	$[5]$	$[5]$
$[]$	$[]$	$[]$

Таблица 3.2 – Функциональные тесты (битонная)

Входной массив	Ожидаемый результат	Результат
$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$
$[5, 4, 3, 2, 1]$	$[2, 3, 4, 5, 1]$	$[2, 3, 4, 5, 1]$
$[9, 7, -5, 1, 4]$	$[-5, 1, 7, 9, 4]$	$[-5, 1, 7, 9, 4]$
$[5]$	$[5]$	$[5]$
$[]$	$[]$	$[]$

Результаты по битонной сортировке выведены в отдельную таблицу, так как она сортирует массив не исключительно по возрастанию или убыванию как обычные сортировки.

Вывод

Были разработаны схемы всех трех алгоритмов сортировки. Для каждого алгоритма была вычислена трудоемкость и оценены лучший и худший случаи.

4 Исследовательская часть

В данном разеделе будут приедены примеры работы программа, а также проведен сравнительный анализ адгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее:

- операционная система: Debian 12 [7] x86_64;
- память: 32 Гб;
- процессор: Процессор: AMD® Ryzen™ 2600 CPU @ 3.40 ГГц [8].

При тестировании ноутбук был включен в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

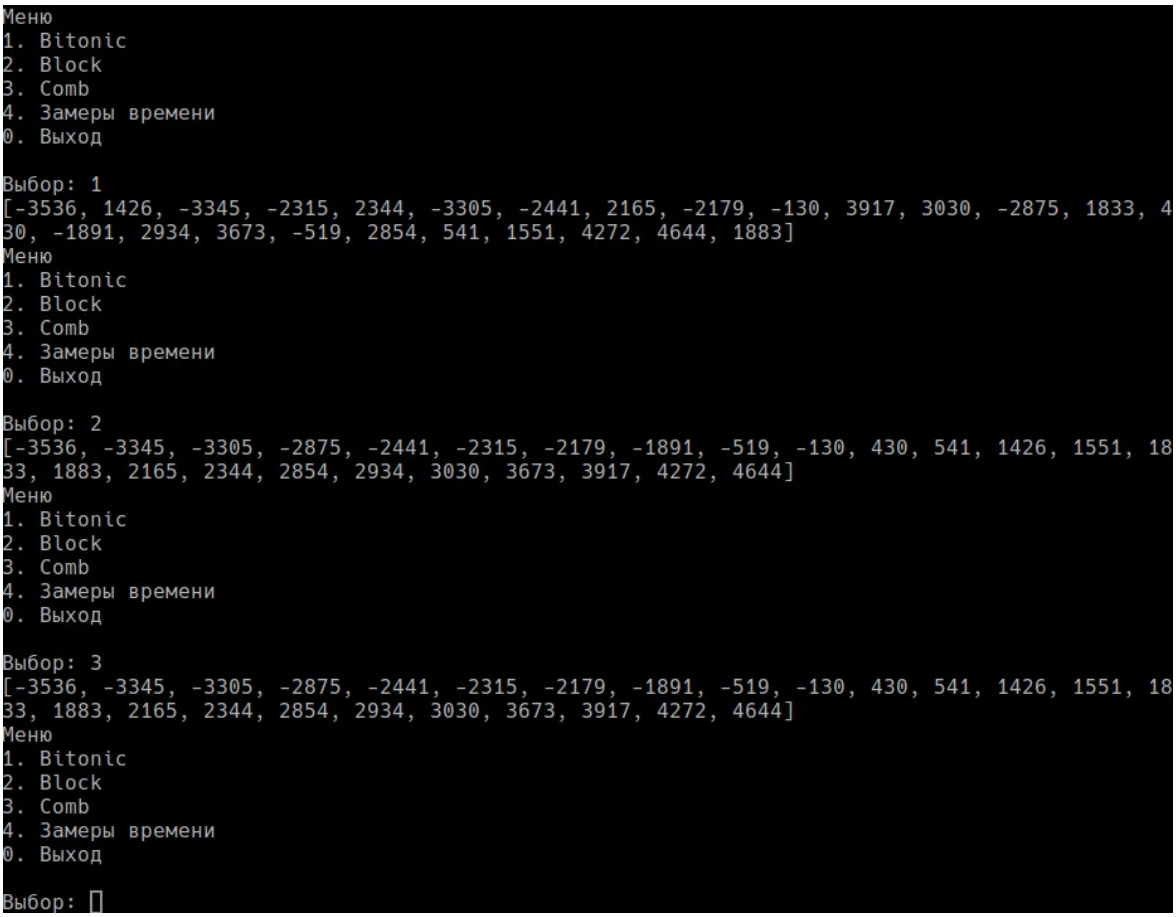

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Как было сказано выше, используется функция замера процессорного времени process_time(...) из библиотеки time на Python. Функция возвращает пользовательское процессорное временя типа float.

Использовать функцию приходится дважды, затем из конечного времени нужно вычесть начальное, чтобы получить результат.

Результаты замеров времени работы алгоритмов сортировки на различных входных данных (в мс) приведены в таблицах 4.1, 4.2 и 4.3.

Таблица 4.1 – Время сортировки, отсортированные данные

Количество чисел	Битонная, мс	Блочная, мс	Расческой, мс
100	0.1350	0.3011	0.0236
200	0.3604	0.4592	0.0315
300	0.6552	1.1661	0.0438
400	1.0601	1.1447	0.0675
500	1.6418	1.2287	0.0813
600	2.3249	2.4254	0.0924
700	3.0429	2.5413	0.1256
800	4.0808	2.7666	0.1251
900	5.0620	2.9390	0.1715
1000	6.3559	3.0283	0.1621

Таблица 4.2 – Время сортировки, отсортированные в обратном порядке данные

Количество чисел	Битонная, мс	Блочная, мс	Расческой, мс
100	0.1004	0.1829	0.6547
200	0.3147	0.4868	4.1152
300	0.6363	0.9823	6.4202
400	1.1146	1.1372	11.8419
500	1.7217	1.3165	18.7539
600	2.4530	2.4302	26.7358
700	3.3249	2.5866	37.1811
800	4.3909	3.1956	51.1194
900	5.5471	2.9448	65.7719
1000	7.0853	3.2796	80.3029

Таблица 4.3 – Время сортировки, случайные данные

Количество чисел	Битонная, мс	Блочная, мс	Расческой, мс
100	0.2819	0.1879	0.9221
200	0.4190	0.4661	2.0348
300	0.8330	0.9892	5.0758
400	1.4994	1.1524	9.2072
500	2.2670	1.2960	15.0499
600	3.4714	2.4753	21.7946
700	4.5544	2.7143	31.6451
800	6.5106	2.9621	43.2024
900	8.4504	3.2055	58.5832
1000	10.4828	3.2364	70.7920

Также на рисунках 4.2, 4.3, 4.4 приведены графические результаты замеров работы сортировок в зависимости от размера входного массива.

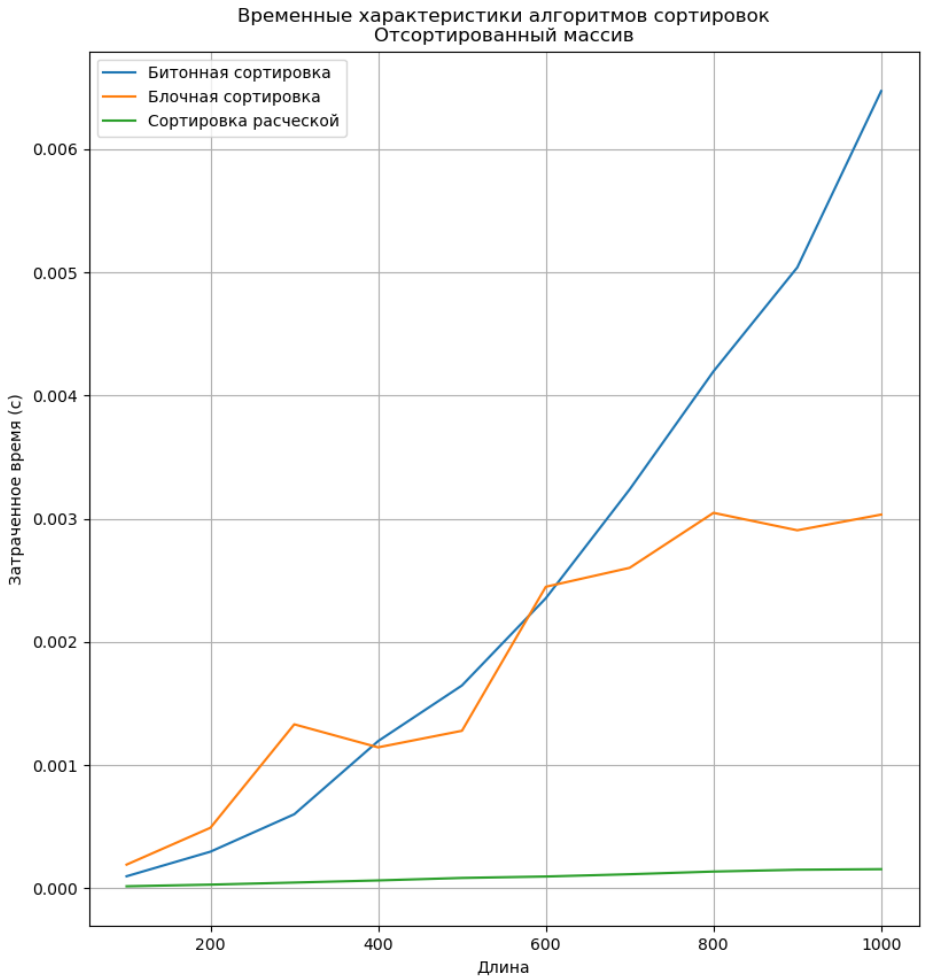

Рисунок 4.2 – Отсортированный массив

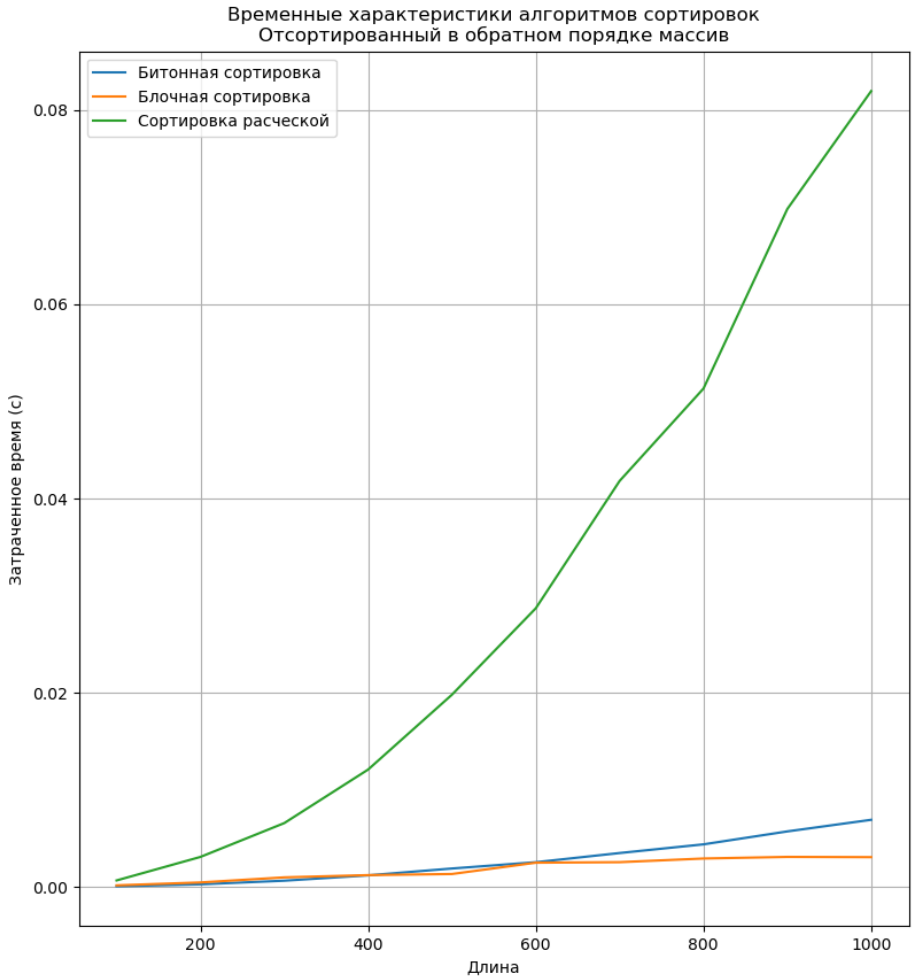

Рисунок 4.3 – Отсортированный в обратном порядке массив

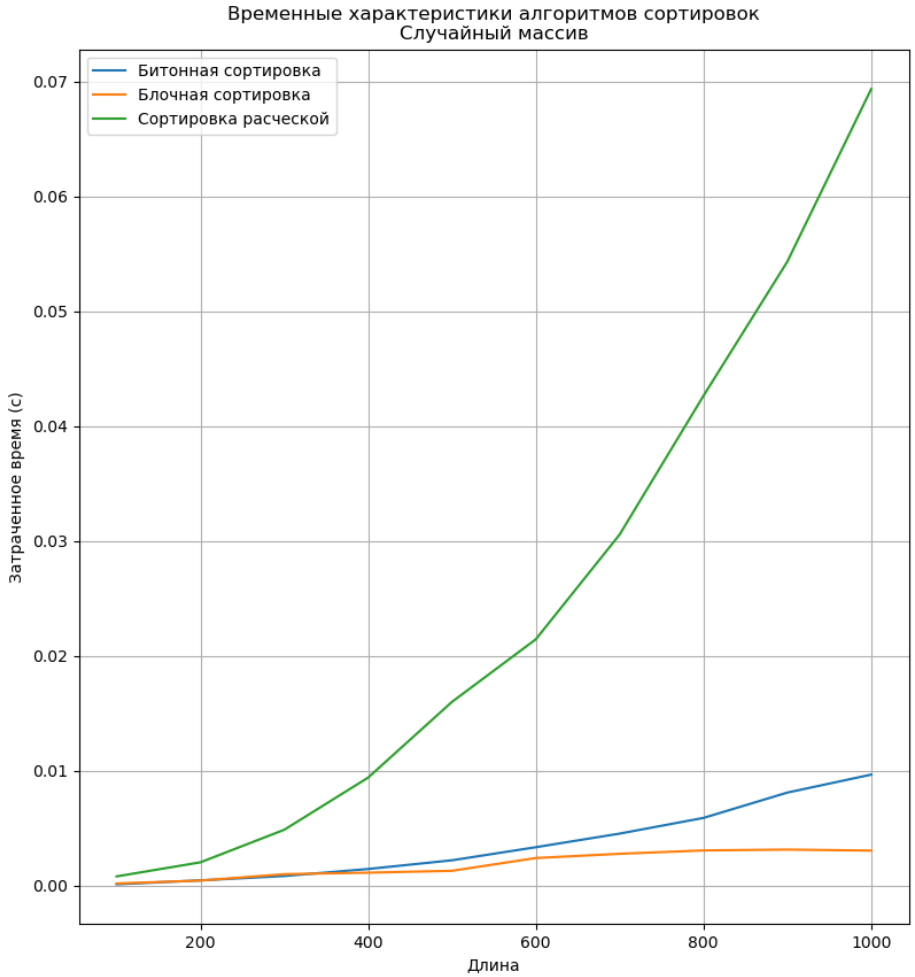

Рисунок 4.4 – Случайный массив

Вывод

Исходя из полученных результатов, сортировка расческой при случайно заполненном массиве, а также при обратно отсортированном работает намного дольше, чем блочная или битонная (т.к. имеет квадратичную сложность), при этом блочная сортировка показала себя лучше всех.

Что касается уже отсортированных данных, то лучше всего себя здесь показала сортировка расческой, в то время, как битонная была хуже всех.

Теоретические результаты замеров и полученные практически результаты совпадают.

Заключение

В результате исследования была рассчитана трудоемкость алгоритмов, проведённая теоретическая оценка трудоемкости алгоритмов показала, что трудоёмкость выполнения алгоритма блочной сортировки составляет $O(N + K)$, расческой – от $O(N \cdot logN)$ до $O(N^2)$, битонной – $O(log^2(N))$.

Исходя из полученных результатов замеров по времени, сортировка расческой лучше только на отсортированном массиве. В других случаях время ее работы значительно дольше блочной и битонной сортировок.

Сравнение результатов замеров по времени битонной сортировки и блочной сортировки показали, что блочная сортировка работает 3 раза быстрее, чем битонная сортировка на случайных данных, на отсортированном массиве битонная сортировка работает в 2 раза хуже, чем блочная сортировка, но на отсортированных в обратном порядке данных битонная сортировка незначительно отстает по скорости от блочной сортировки.

Цель, которая была поставлена в начале лабораторной работы была достигнута, а также в ходе выполнения лабораторной работы были решены следующие задачи:

- были реализованы и рассчитаны по трудоемкости алгоритмов сортировки: битонной, блочной и расческой;
- была выбрана модель вычисления и проведен сравнительный анализ трудоемкостей выбранных алгоритмов сортировки;
- на основе экспериментальных данных проведено сравнение выбранных алгоритмов сортировки;
- подготовлен отчет о лабораторной работе.

Список источников

[1] Laxmikant V. Kalé, Edgar Solomonik, Encyclopedia of Parallel Computing, с. 1855 - 1861. Springer, 2011.

[2] Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. Глава 8. Сортировка за линейное время // Алгоритмы: построение и анализ, 2-е издание. «Вильямс», 2005. - с. 230 - 234.

[3] Lacey S., Box R. A Fast, Easy Sort: A novel enhancement makes a bubble sort into one of the fastest sorting routines. Byte. - Апрель 1991. - с. 315 - 320.

[4] М. В. Ульянов Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ. 2007.

[5] Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org` (дата обращения: 30.10.2023).

[6] time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html#functions` (дата обращения: 30.10.2023).

[7] Debian 12 [Электронный ресурс]. `https://www.debian.org/` (дата обращения: 23.10.2023).

[8] Процессор: AMD® Ryzen™ 2600 CPU @ 3.40ГГц. `https://www.amd.com/en/product/7666` (дата обращения: 23.10.2023).