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 Жаворонкова А. А.

Группа ИУ7-56Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2023 г.

Содержание

Введение

В программировании алгоритмы сортировок занимают отдельное важное место. Их существует огромное количество, и все они обладают различными свойствами, такими как трудоемкость и объем требуемой памяти. Упорядочивание элементов в массиве необходимо для решения множества задач в разных сферах, связанных с математикой, физикой, компьютерной графикой и т. д. В данном отчете будут рассмотрены следующие методы сортировки:

— блочная сортировка;

— плавная сортировка;

— сортировка слиянием.

Целью данной работы является описание, реализация и исследование алгоритмов сортировок — блочной, плавной и слиянием. Для достижения поставленной цели необходимо выполнить следующие задачи:

— описать алгоритмы сортировок — блочной, плавной, слиянием;

— реализовать алгоритмы указанных сортировок;

— провести тестирование по методу черного ящика для реализаций указанных алгоритмов сортировок;

— провести сравнительный анализ по времени и по памяти реализаций указанных алгоритмов сортировок;

— описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчетно-пояснительная записка к работе.

1 Аналитическая часть

В данном разделе будут рассмотрены блочная сортировка, плавная сортировка и сортировка слиянием.

1.1 Блочная сортировка

Алгоритм блочной (корзинной) сортировки [4] разделяет элементы массива входных данных на некоторое количество блоков — k, количество блоков зависит от количества исходного множества данных. Далее каждый из таких блоков сортируется либо другой сортировкой, либо рекурсивно тем же методом разбиения. После сортировок внутри каждых блоков данные записываются в исходный массив в порядке разбиения на блоки.

1.2 Плавная сортировка

При плавной сортировке [1] в массив накапливается куча из данных, которые затем сортируются путем непрерывного удаления максимума из кучи. В отличие от пирамидальной сортировки, здесь используется не двоичная куча, а специальная, полученная с помощью чисел Леонардо.

Числа Леонардо — последовательность чисел, задаваемая зависимостью:

$$L(n) = \begin{cases} 1, n = 0 \\ 1, n = 1 \\ L(n-1) + L(n-2) + 1, n > 1 \end{cases} \tag{1.1}$$

Куча состоит из последовательности куч, размеры которых равны одному из чисел Леонардо, а корни хранятся в порядке возрастания. Преимущества таких специальных куч перед двоичными состоят в том, что если последовательность отсортирована, её создание и разрушение займёт $O(n)$ времени, что будет быстрее.

1.3 Сортировка слиянием

При сортировке слиянием [5], массив разделяется пополам до тех пор, пока каждый участок не станет длиной в один элемент. Затем эти участки возвращаются на место (сливаются) в правильном порядке. В итоге работы алгоритма, исходный массив данных преобразуется в отсортированный, путем сравнивания разделенных элементов между собой. Стоит отметить, что в отличие от линейных алгоритмов сортировки, сортировка слиянием будет делить и склеивать массив вне зависимости от того, был он отсортирован изначально или нет. Поэтому, несмотря на то, что в худшем случае алгоритм отработает быстрее, чем линейный, в лучшем случае, производительность алгоритма будет ниже, чем у линейного. Поэтому сортировка слиянием — не самое лучшее решение, когда необходимо отсортировать частично упорядоченный массив.

1.4 Вывод

В данном разделе были теоретически разобраны алгоритмы блочной сортировки, плавной сортировки и сортировки слиянием.

К разрабатываемой программе предъявляются следующие требования.

1. Программа должна предоставлять функциональность сортировки массива алгоритмами блочной сортировки, плавной сортировки, сортировки слиянием.

2. Реализуемое ПО будет работать в двух режимах — пользовательском, в котором можно выбрать алгоритм и вывести для него отсортированный массив, а также экспериментальном режиме, в котором можно произвести сравнение реализаций алгоритмов по времени работы на различных входных данных.

3. В первом режиме в качестве входных данных в программу будет подаваться массив, также реализовано меню для вызова алгоритмов и замеров времени. Программа должна корректно обрабатывать случай ввода массива нулевой длины.

4. Во втором режиме будет происходить измерение процессорного времени работы программы, будут построены зависимости времени работы от размера массива. Входной массив будет сгенерирован автоматически для заданного размера массива.

2 Конструкторская часть

В этом разделе будут представлены схемы алгоритмов блочной сортировки, плавной сортировки и сортировки слиянием.

2.1 Разработка алгоритмов

На рисунках 2.1 — 2.7 представлены схемы алгоритмов сортировок.

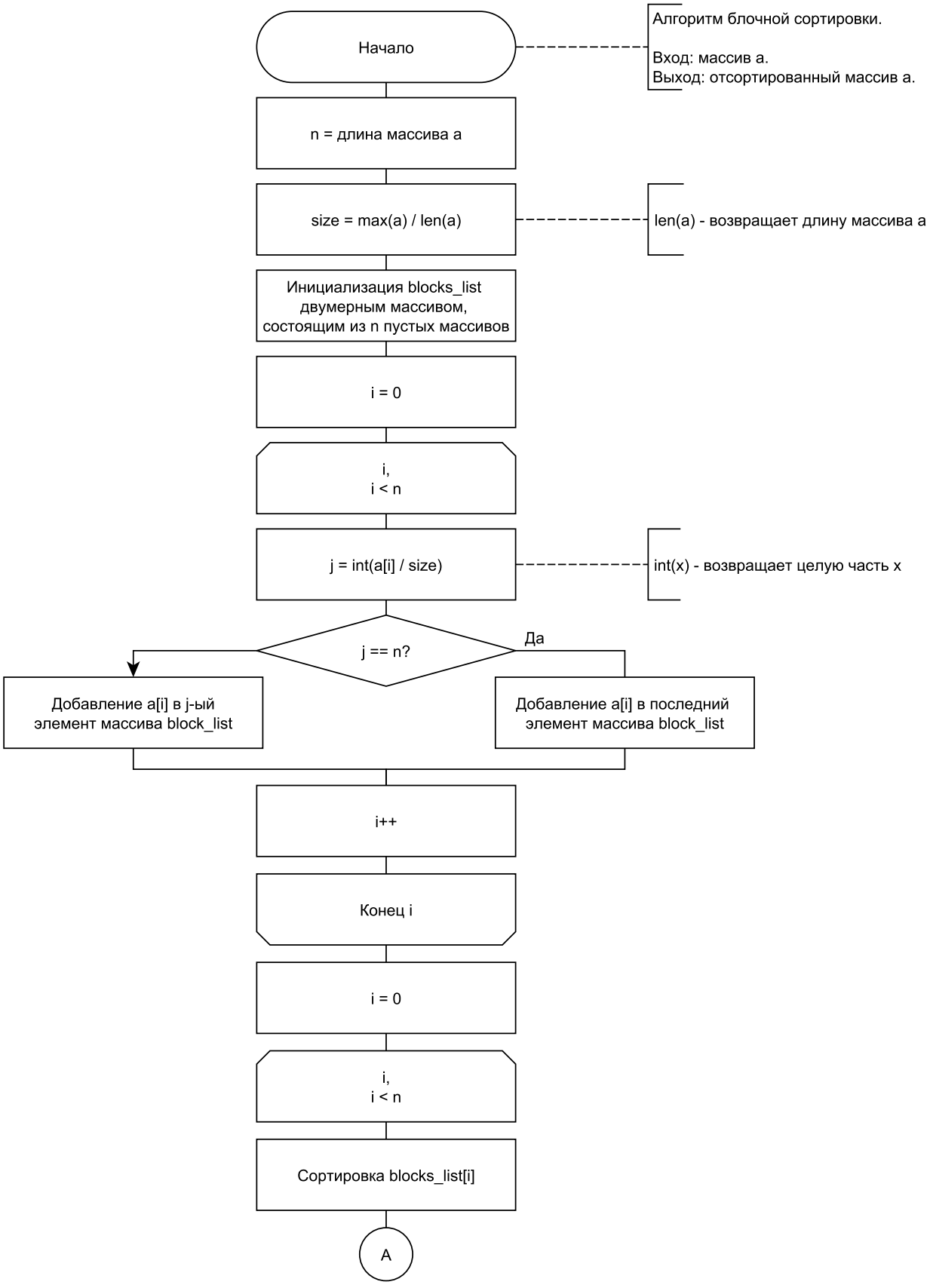

Рисунок 2.1 – Схема алгоритма блочной сортировки (часть 1)

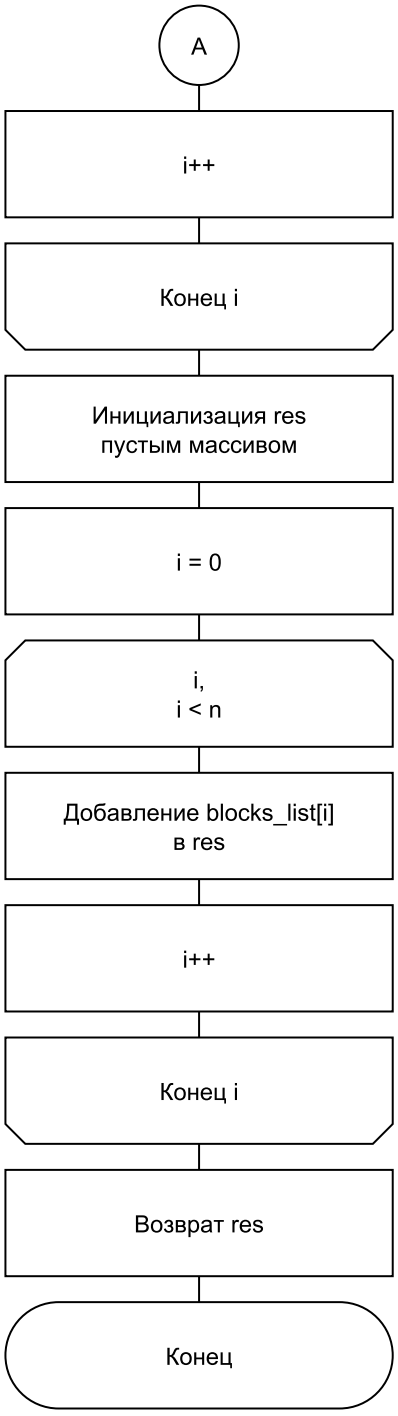

Рисунок 2.2 – Схема алгоритма блочной сортировки (часть 2)

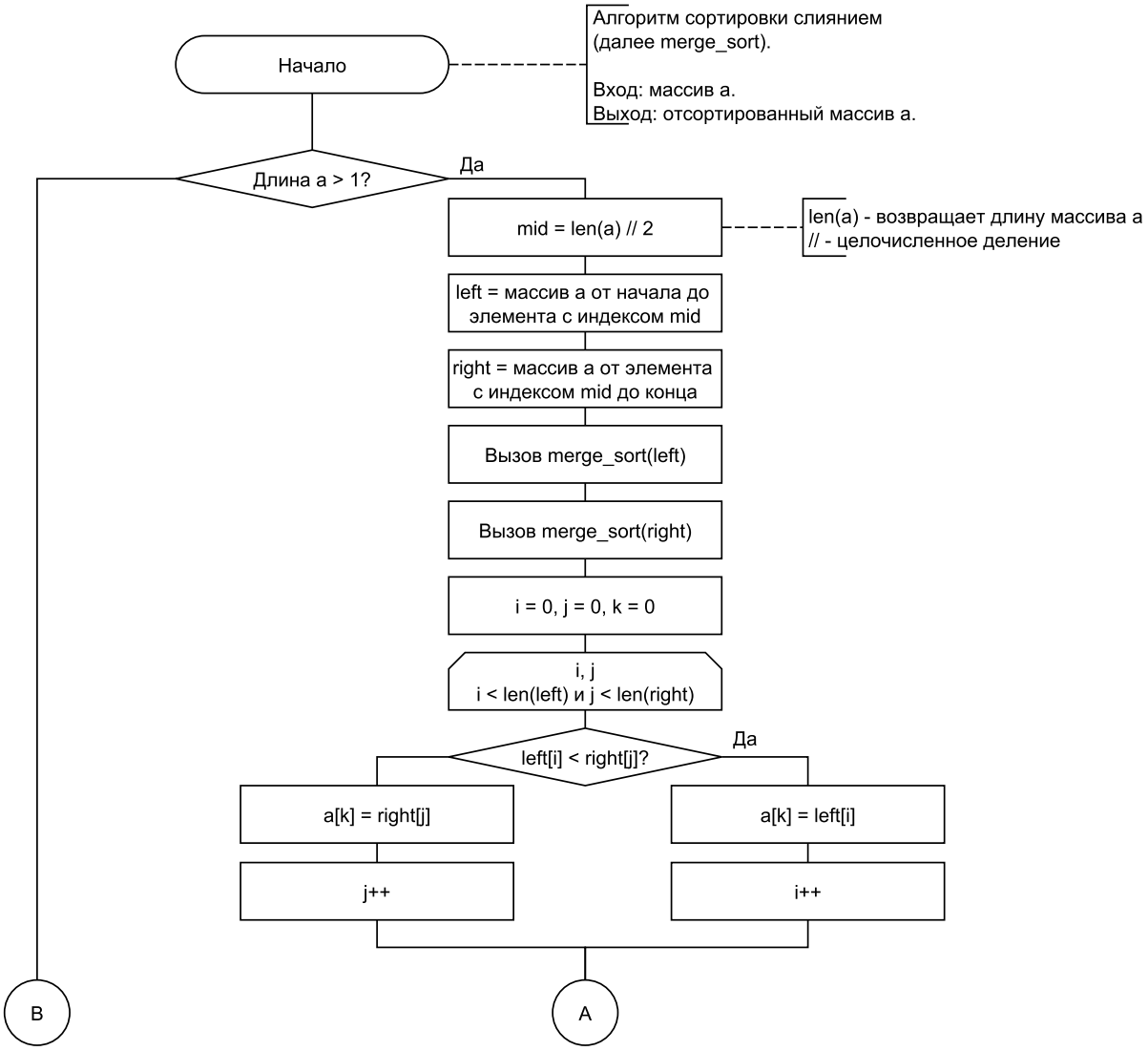

Рисунок 2.3 – Схема алгоритма сортировки слиянием (часть 1)

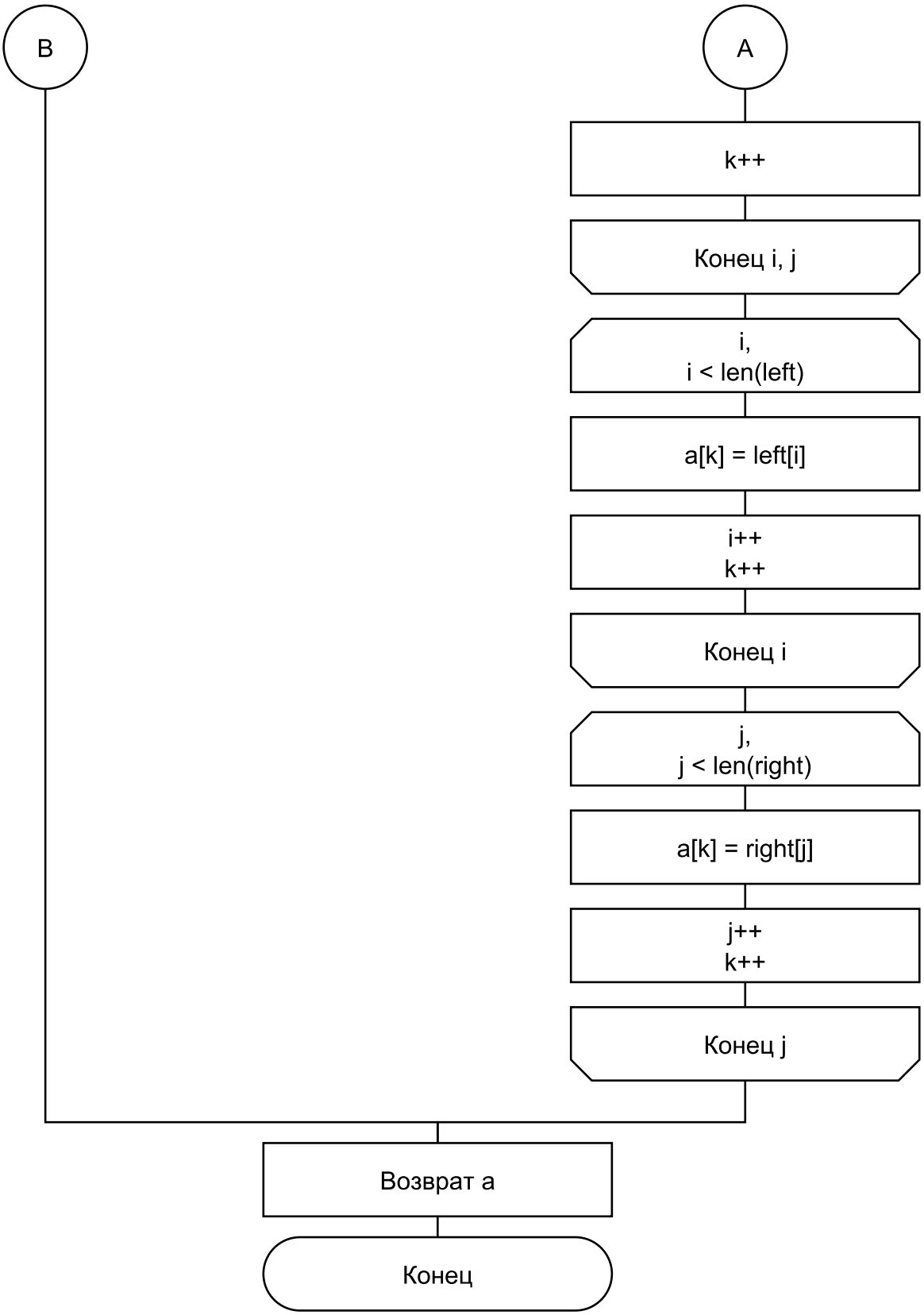

Рисунок 2.4 – Схема алгоритма сортировки слиянием (часть 2)

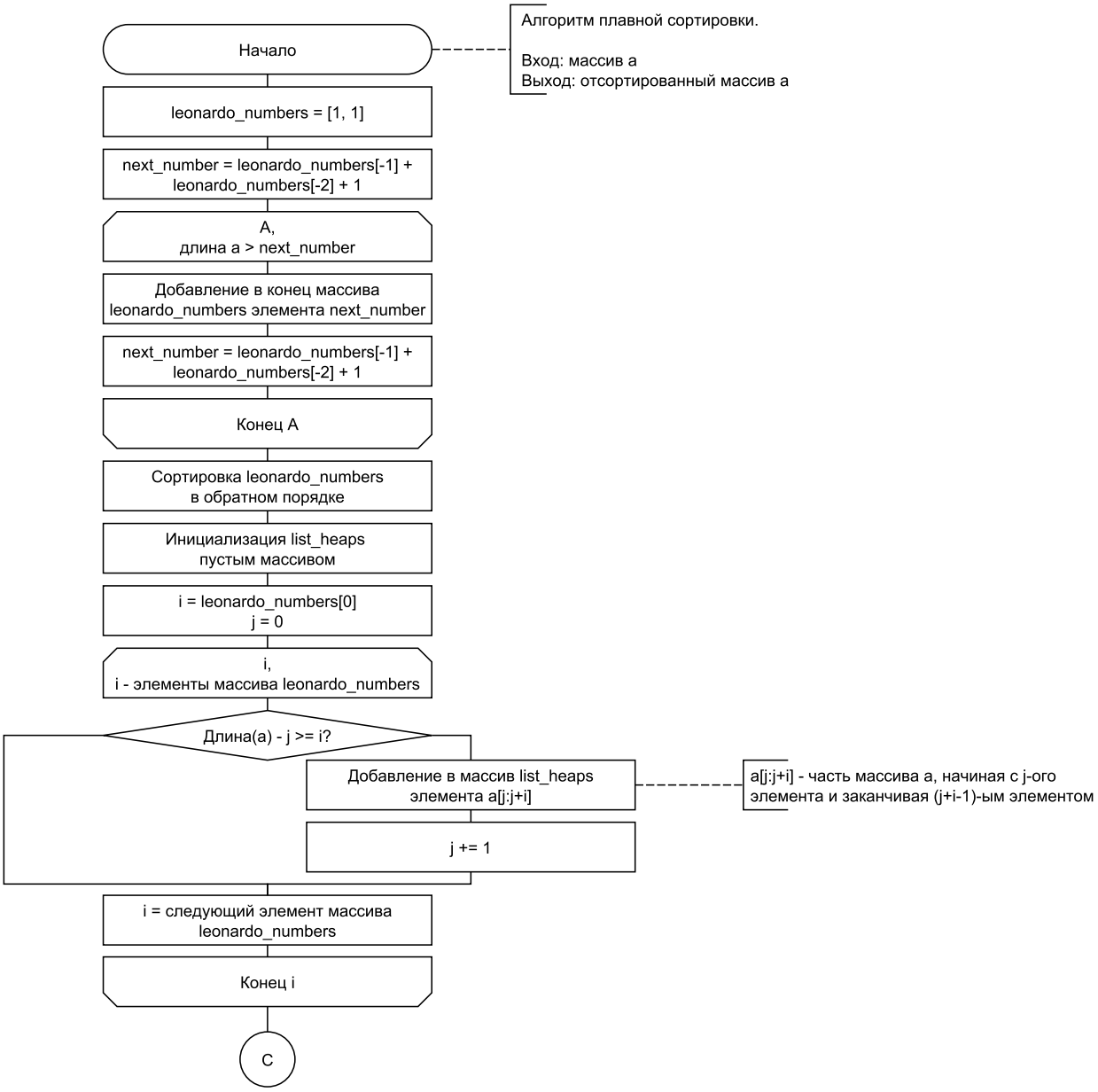

Рисунок 2.5 – Схема алгоритма плавной сортировки (часть 1)

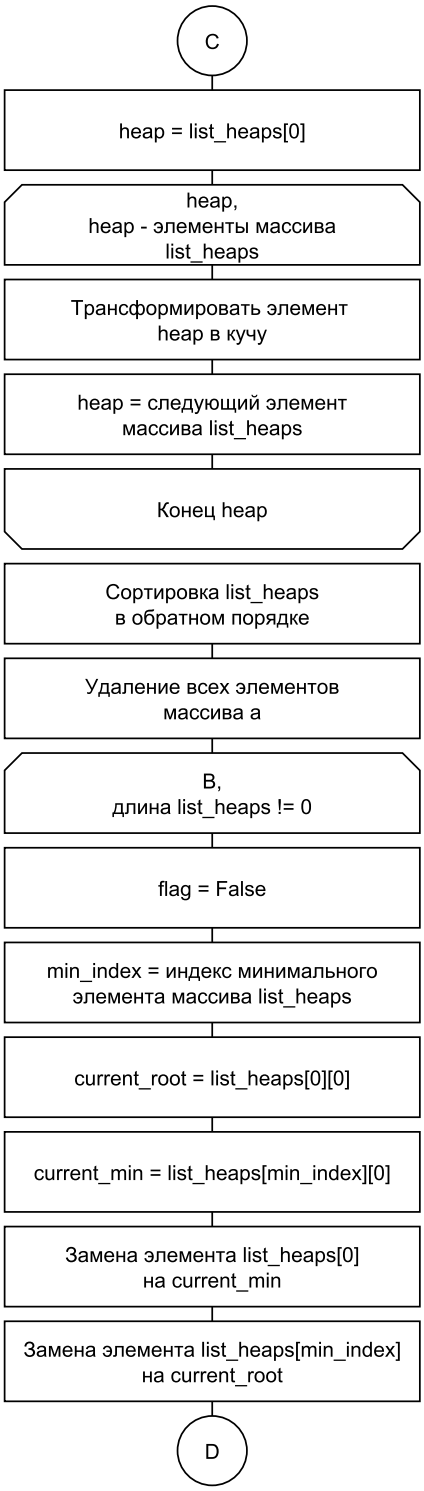

Рисунок 2.6 – Схема алгоритма плавной сортировки (часть 2)

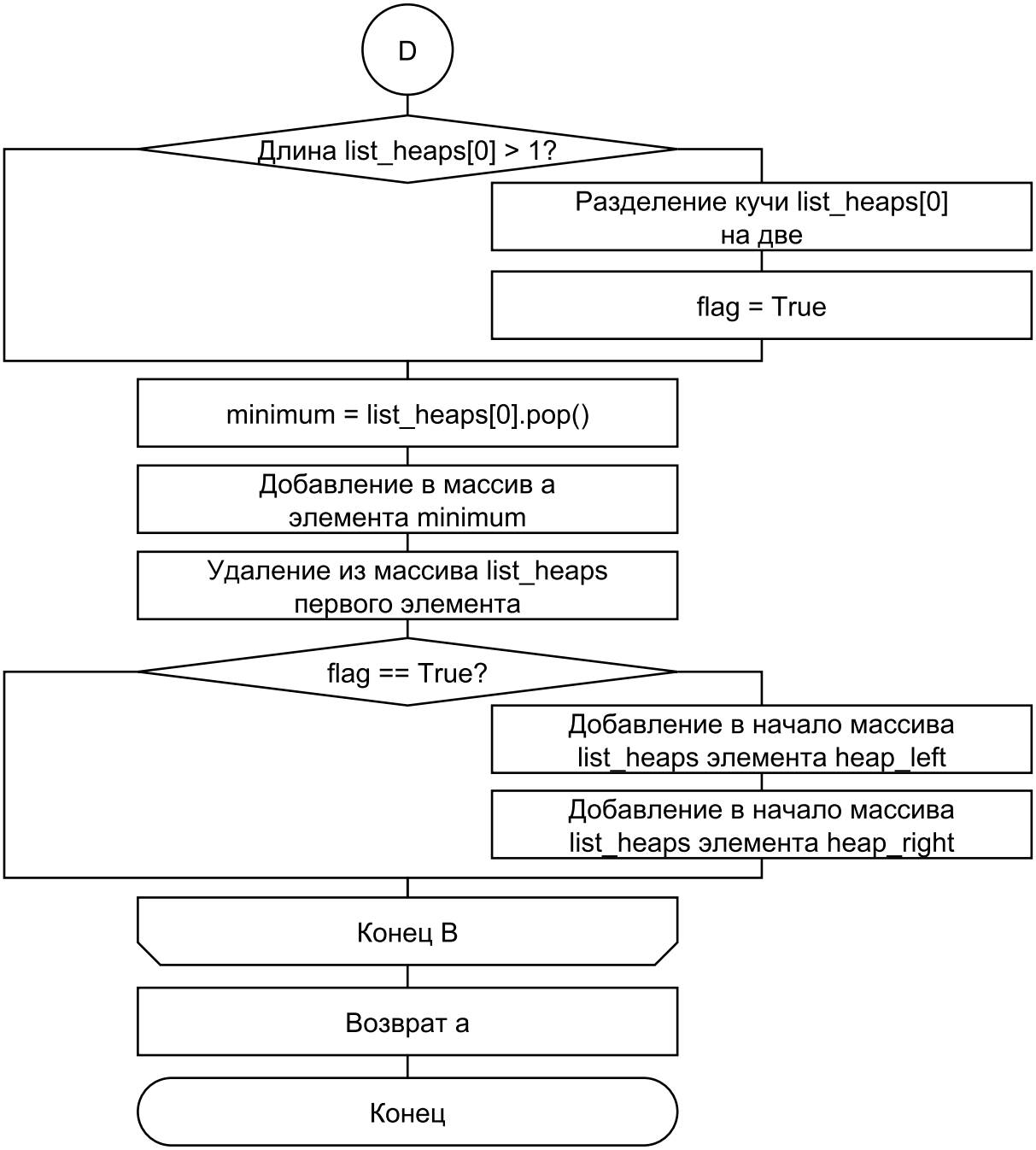

Рисунок 2.7 – Схема алгоритма плавной сортировки (часть 3)

2.2 Модель вычислений

1. Трудоемкость базовых операций.

   Следующие операторы имеют трудоемкость 1:

$$+, -, + =, - =, =, ==, ! =, >=, <=, >, <,$$

$$>>, <<, [], \&, |, \&\&, ||, ++, --$$

Следующие операторы имеют трудоемкость 2:

$$*, /, \%, * =, / =$$

2. Условный оператор.

   Для конструкций вида:

```
1 if (условие)
2 {
3     Блок 1;
4 }
5 else
6 {
7     Блок 2;
8 }
```

   Пусть трудоемкость блока 1 — f_1, блока 2 — f_2. Пусть также трудоемкость условного перехода — 0.

   Тогда трудоемкость условного оператора:

$$f_{if} = f_{\text{вычисления условия}} + \begin{bmatrix} \min(f_1, f_2), \text{лучший случай} \\ \max(f_1, f_2), \text{худший случай} \end{bmatrix} \tag{2.1}$$

3. Трудоемкость циклов.

   Трудоемкость циклов вычисляется по следующей формуле:

$$\begin{aligned} f_{\text{цикла}} = & f_{\text{инициализации}} + f_{\text{сравнения}} + \\ & + M_{\text{шагов}} \cdot (f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \end{aligned} \tag{2.2}$$

2.3 Трудоемкость алгоритмов

Рассчитаем трудоемкость алгоритмов сортировок.

2.3.1 Алгоритм блочной сортировки

В таблице 2.1 представлена построчная оценка трудоемкости блочной сортировки.

Таблица 2.1 – Построчная оценка трудоемкости блочной сортировки

Строка кода	Вес
n = len(a)	2
size = max(a) / n	4
blocks_list = []	1
for i in range(n):	2
blocks_list.append([])	1
for i in range(n):	2
j = int(a[i] / size)	5
if j != n:	1
blocks_list[j].append(a[i])	3
else:	0
blocks_list[-1].append(a[i])	3
for i in range(n):	2
blocks_list[i] = sorted(blocks_list[i])	4
res = []	1
for i in range(n):	2
res += blocks_list[i]	2
return res	0

В лучшем случае, когда массив отсортирован по возрастанию, трудоемкость:

$$f = O(N) \tag{2.3}$$

В худшем случае, когда массив отсортирован по убыванию, трудоемкость:

$$f = O(N^2) \tag{2.4}$$

2.3.2 Алгоритм плавной сортировки

В таблице 2.2 представлена построчная оценка трудоемкости плавной сортировки.

Таблица 2.2 – Построчная оценка трудоемкости плавной сортировки

Строка кода	Вес
leonardo_numbers = [1, 1]	1
next_number = leonardo_numbers[-1] + leonardo_numbers[-2] + 1	5
while len(a) > next_number:	2
leonardo_numbers.append(next_number)	1
next_number = leonardo_numbers[-1] + leonardo_numbers[-2] + 1	5
leonardo_numbers.reverse()	1
list_heaps, j = [], 0	2
for i in leonardo_numbers:	2
if len(a) - j >= i:	3
list_heaps.append(a[j:j + i])	2
j += i	1
for heap in list_heaps:	2
heapq.heapify(heap)	1
list_heaps.reverse()	1
a.clear()	1
while len(list_heaps):	2
flag = False	1
min_index = list_heaps.index(min(list_heaps))	3
current_root = list_heaps[0][0]	3
current_min = list_heaps[min_index][0]	3
heapq.heapreplace(list_heaps[0], current_min)	2
heapq.heapreplace(list_heaps[min_index], current_root)	2
if len(list_heaps[0]) > 1:	3
heap_left, heap_right = heap_division(list_heaps[0])	4
flag = True	1
minimum = heapq.heappop(list_heaps[0])	3
a.append(minimum)	1
list_heaps.pop(0)	1
if flag:	1
list_heaps.insert(0, heap_left)	1
list_heaps.insert(0, heap_right)	1
return a	0

В лучшем случае, когда массив отсортирован по возрастанию, трудоемкость:

$$f = O(N) \tag{2.5}$$

В худшем случае, когда массив отсортирован по убыванию, трудоемкость:

$$f = O(NlogN) \tag{2.6}$$

2.3.3 Алгоритм сортировки слиянием

В таблице 2.3 представлена построчная оценка трудоемкости сортировки слиянием.

Таблица 2.3 – Построчная оценка
трудоемкости сортировки слиянием

Строка кода	Вес
if len(a) > 1:	2
mid = len(a) // 2	4
left = a[:mid]	2
right = a[mid:]	2
merge_sort(left)	0
merge_sort(right)	0
i = j = k = 0	3
while i < len(left) and j < len(right):	5
if left[i] < right[j]:	3
a[k] = left[i]	3
i += 1	1
else:	0
a[k] = right[j]	3
j += 1	1
k += 1	1
while i < len(left):	2
a[k] = left[i]	3
i += 1	1
k += 1	1
while j < len(right):	2
a[k] = right[j]	3
j += 1	1
k += 1	1
return a	0

В лучшем случае, когда массив отсортирован по возрастанию, трудоемкость:

$$f = O(NlogN) \tag{2.7}$$

В худшем случае, когда массив отсортирован по убыванию, трудоемкость:

$$f = O(NlogN) \tag{2.8}$$

2.4 Классы эквивалентности тестирования

Для тестирования выделены следующие классы эквивалентности:

1. массив пустой;
2. массив состоит из одного элемента;
3. массив отсортирован;
4. массив отсортирован в обратном порядке;
5. все элементы массива одинаковые.

2.5 Использование памяти

Пусть n — число элементов входного массива. Тогда рассчитаем затраты памяти для каждого алгоритма сортировки.

1. Алгоритм блочной сортировки:
   - результирующий массив: $n \cdot sizeof(int)$;
   - массив блоков: $n \cdot sizeof(int)$;
   - дополнительные переменные: $sizeof(double) + sizeof(int)$.

   Таким образом, необходимая память:

$$sizeof(int) \cdot (2n + 1) + sizeof(double) \tag{2.9}$$

2. Алгоритм плавной сортировки:
   - массив чисел Леонардо: $k \cdot sizeof(int)$, где k — количество чисел Леонардо;

— массив куч: $k \cdot n \cdot sizeof(int)$;

— дополнительные переменные: $8 \cdot sizeof(int)$.

Таким образом, необходимая память:

$$sizeof(int) \cdot (k(1+n)+8) \tag{2.10}$$

3. Алгоритм сортировки слиянием:

— левый и правый подмассивы массива a: $\log_2 n \cdot n \cdot sizeof(int)$;

— дополнительные переменные: $\log_2 n \cdot 4 \cdot sizeof(int)$.

Таким образом, необходимая память для всего дерева рекурсивных вызовов, высотой $\log_2 n$:

$$\log_2 n \cdot sizeof(int) \cdot (n+4) \tag{2.11}$$

Вывод

Сравнивая формулы (2.9), (2.10) и (2.11), можно сделать вывод, что меньше всего памяти необходимо для реализации алгоритма блочной сортировки, а больше всего — для реализации алгоритма сортировки слиянием, поскольку она выполняется рекурсивно.

2.6 Вывод

В данном разделе были представлены схемы алгоритмов, рассматриваемых в лабораторной работе.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги реализаций алгоритмов блочной сортировки, плавной сортировки и сортировки слиянием.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования *Python* [3]. 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* [2].

3.2 Описание используемых типов данных

При реализации будут использованы следующие типы и структуры данных:

— тип *int* для количества элементов массива;

— *массив ячеек типа int* для входного массива.

3.3 Сведения о модулях программы

Программа состоит из двух модулей:

— *main.py* — файл, содержащий весь служебный код;

— *sort.py* — файл, содержащий реализации алгоритмов.

3.4 Реализация алгоритмов

В листингах 3.1 — 3.3 представлены реализации алгоритмов блочной сортировки, плавной сортировки и сортировки слиянием.

Листинг 3.1 – Реализация алгоритма блочной сортировки

```
def block_sort(a):
    n = len(a)
    size = max(a) / n
    blocks_list = []
    for i in range(n):
        blocks_list.append([])

    for i in range(n):
        j = int(a[i] / size)
        if j != n:
            blocks_list[j].append(a[i])
        else:
            blocks_list[-1].append(a[i])

    for i in range(n):
        blocks_list[i] = sorted(blocks_list[i])

    res = []
    for i in range(n):
        res += blocks_list[i]
    return res
```

Листинг 3.2 – Реализация алгоритма плавной сортировки

```
def smooth_sort(a):
    leonardo_numbers = [1, 1]
    next_number = leonardo_numbers[-1] + leonardo_numbers[-2] + 1
    while len(a) > next_number:
        leonardo_numbers.append(next_number)
        next_number = leonardo_numbers[-1] + leonardo_numbers[-2] +
            1
    leonardo_numbers.reverse()

    list_heaps = []
```

```
    j = 0
    for i in leonardo_numbers:
        if len(a) - j >= i:
            list_heaps.append(a[j:j + i])
            j += i

    for heap in list_heaps:
        heapq.heapify(heap)
    list_heaps.reverse()

    a.clear()
    while len(list_heaps):
        flag = False

        min_index = list_heaps.index(min(list_heaps))
        current_root = list_heaps[0][0]
        current_min = list_heaps[min_index][0]

        heapq.heapreplace(list_heaps[0], current_min)
        heapq.heapreplace(list_heaps[min_index], current_root)

        if len(list_heaps[0]) > 1:
            heap_left, heap_right = heap_division(list_heaps[0])
            flag = True

        minimum = heapq.heappop(list_heaps[0])
        a.append(minimum)

        list_heaps.pop(0)
        if flag:
            list_heaps.insert(0, heap_left)
            list_heaps.insert(0, heap_right)
    return a
```

Листинг 3.3 – Реализация алгоритма сортировки слиянием

```
def merge_sort(a):
    if len(a) > 1:
        mid = len(a) // 2
        left = a[:mid]
        right = a[mid:]
        merge_sort(left)
```

```
7           merge_sort(right)
8           i = j = k = 0
9           while i < len(left) and j < len(right):
10              if left[i] < right[j]:
11                  a[k] = left[i]
12                  i += 1
13              else:
14                  a[k] = right[j]
15                  j += 1
16              k += 1
17
18          while i < len(left):
19              a[k] = left[i]
20              i += 1
21              k += 1
22          while j < len(right):
23              a[k] = right[j]
24              j += 1
25              k += 1
26
27      return a
```

3.5 Функциональные тесты

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы описанных сортировок. Тесты *для всех алгоритмов* пройдены успешно.

Таблица 3.1 – Функциональные тесты

		Ожидаемый результат		
№	Массив	Блочная сорт.	Плавная сорт.	Сорт. слиянием
1	[]	[]	[]	[]
2	[5]	[5]	[5]	[5]
3	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]
4	[5, 4, 3, 2, 1]	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]
5	[1, 1, 1]	[1, 1, 1]	[1, 1, 1]	[1, 1, 1]

3.6 Вывод

Были представлены реализации алгоритмов блочной сортировки, плавной сортировки и сортировки слиянием, которые были описаны в предыдущем разделе. Также в данном разделе была приведена информация о выбранных средствах для разработки алгоритмов.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программы, а также проведен сравнительный анализ алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее:

— операционная система: Windows 11, x64;

— оперативная память: 8 Гб;

— процессор: AMD Ryzen 5 5500U с видеокартой Radeon Graphics 2.10 ГГц.

Во время замеров времени ноутбук был нагружен только встроенными приложениями окружения.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

```
Меню:
1) Блочная сортировка
2) Плавная сортировка
3) Сортировка слиянием
4) Построить графики

0) Выход

Введите пункт меню: 1
Введите элементы массива через пробел
4 3 7 9 10 -1 2
[-1, 2, 3, 4, 7, 9, 10]
Меню:
1) Блочная сортировка
2) Плавная сортировка
3) Сортировка слиянием
4) Построить графики

0) Выход

Введите пункт меню: 2
Введите элементы массива через пробел
1 5 7 9 12 -3 -10
[-10, -3, 1, 5, 7, 9, 12]
Меню:
1) Блочная сортировка
2) Плавная сортировка
3) Сортировка слиянием
4) Построить графики

0) Выход

Введите пункт меню: 3
Введите элементы массива через пробел
5 5 7 1 1 0 7 2
[0, 1, 1, 2, 5, 5, 7, 7]
```

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 время типа float.

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

Замеры проводились для размеров массива от 100 до 500 с шагом 50 по 5000 раз на различных входных данных.

Результаты замеров приведены в таблицах 4.1— 4.3 (время в мс).

Таблица 4.1 – Результаты замеров времени для произвольных массивов

Размер массива	Блочная сорт., мс	Плавная сорт., мс	Сорт. слиянием, мс
100	0.263	0.075	0.163
150	0.428	0.097	0.278
200	0.603	0.150	0.375
250	0.803	0.119	0.528
300	0.922	0.106	0.634
350	1.134	0.188	0.722
400	1.297	0.200	0.825
450	1.422	0.216	0.925
500	1.647	0.256	1.084

Таблица 4.2 – Результаты замеров времени для отсортированных массивов

Размер массива	Блочная сорт., мс	Плавная сорт., мс	Сорт. слиянием, мс
100	0.166	0.066	0.241
150	0.256	0.081	0.400
200	0.334	0.109	0.559
250	0.375	0.144	0.744
300	0.509	0.178	1.019
350	0.625	0.197	1.078
400	0.731	0.178	1.194
450	0.912	0.234	1.325
500	0.981	0.231	1.641

Таблица 4.3 – Результаты замеров времени для отсортированных в обратном порядке массивов

Размер массива	Блочная сорт., мс	Плавная сорт., мс	Сорт. слиянием, мс
100	0.150	0.081	0.269
150	0.241	0.109	0.416
200	0.369	0.122	0.613
250	0.403	0.144	0.681
300	0.491	0.119	0.950
350	0.616	0.188	1.019
400	0.756	0.216	1.234
450	0.769	0.203	1.359
500	0.909	0.212	1.616

На рисунках 4.2— 4.4 приведена визуализация результатов замеров.

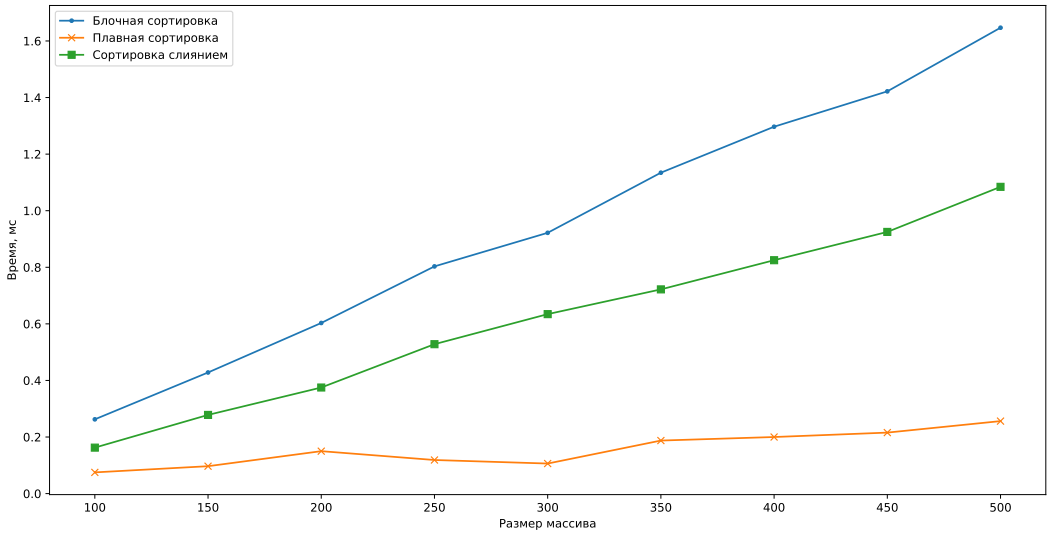

Рисунок 4.2 – Визуализация результатов замеров для произвольных массивов

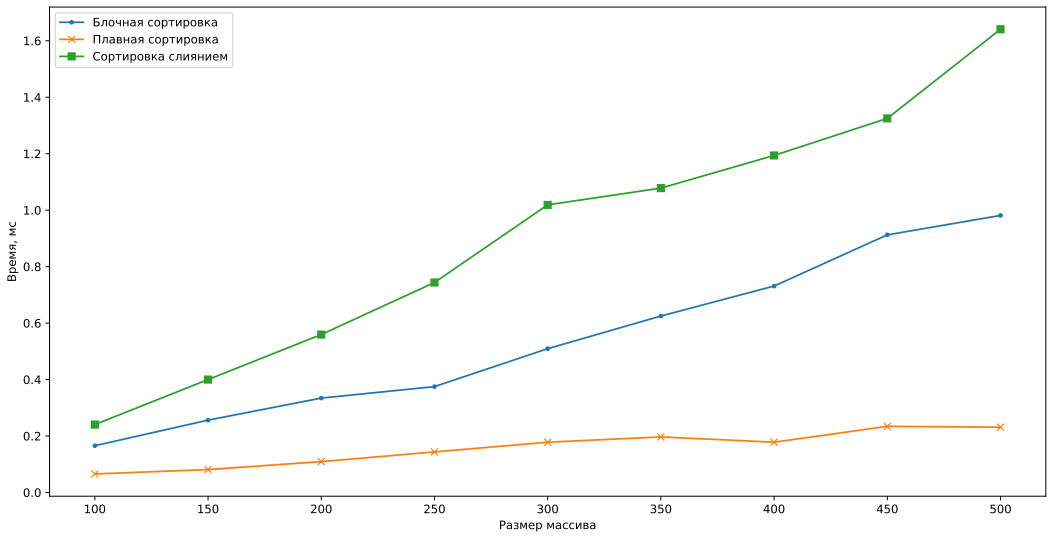

Рисунок 4.3 – Визуализация результатов замеров для отсортированных массивов

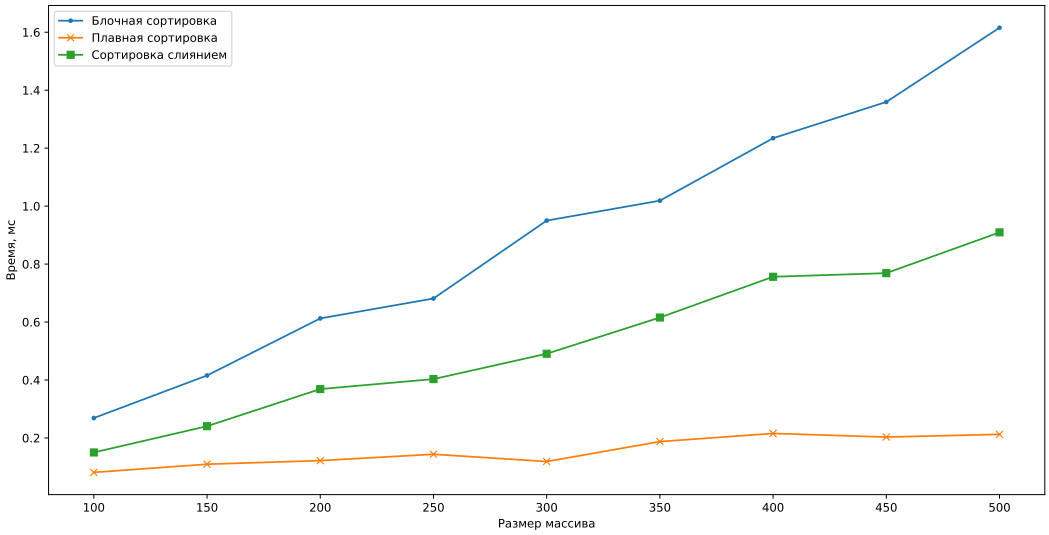

Рисунок 4.4 – Визуализация результатов замеров для отсортированных в обратном порядке массивов

4.4 Вывод

В результате эксперимента было получено, что наиболее эффективной по времени является реализация плавной сортировки. Она дает преимущество при любых входных данных: произвольных, отсортированных, отсортированных в обратном порядке. Например, при количестве элементов массива равном 500 реализация алгоритма плавной сортировки дает преимущество в 84% перед реализацией алгоритма блочной сортировки и в 76% перед реализацией алгоритма сортировки слиянием.

Заключение

В результате исследования было определено, что наиболее эффективной по памяти реализацией является реализация алгоритма блочной сортировки. Однако среди рассмотренных реализаций алгоритмов она является наименее эффективной по времени. Реализация, имеющая наименьшее время работы — реализация алгоритма плавной сортировки. А наименее эффективной по памяти является реализация алгоритма сортировки слиянием.

Цель, которая была поставлена в начале лабораторной работы была достигнута: описаны, реализованы и исследованы алгоритмы блочной сортировки, плавной сортировки и сортировки слиянием. В ходе выполнения были решены все задачи:

— описаны алгоритмы сортировок — блочной, плавной, слиянием;

— реализованы алгоритмы указанных сортировок;

— проведено тестирование по методу черного ящика для реализаций указанных алгоритмов сортировок;

— проведен сравнительный анализ по времени и по памяти реализаций указанных алгоритмов сортировок;

— описаны и обоснованы полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчетно-пояснительная записка к работе.

Список используемых источников

1. Smoothsort Demystified [Электронный ресурс]. — URL: https://www.keithschwarz.com/smoothsort (дата обр. 17.10.2023).
2. time — Time access and conversions [Электронный ресурс]. — URL: https://docs.python.org/3/library/time.html#functions (дата обр. 17.10.2023).
3. Welcome to Python [Электронный ресурс]. — URL: https://www.python.org (дата обр. 17.10.2023).
4. Алгоритмы сортировки данных [Электронный ресурс]. — URL: http://p96555me.beget.tech/materials/sort-methods/#2.4 (дата обр. 17.10.2023).
5. Алгоритмы сортировки данных [Электронный ресурс]. — URL: http://p96555me.beget.tech/materials/sort-methods/#2.1 (дата обр. 17.10.2023).