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 Пронина Л. Ю.

Группа ИУ7-54Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2023 г.

Содержание

Введение

Сортировка — одна из основных операций в алгоритмике и программировании, которая позволяет упорядочить данные по заданному критерию. Однако, сортировка может быть времязатратной операцией, особенно при работе с большими объемами данных. Поэтому важно понимать и анализировать трудоемкость различных алгоритмов сортировки.

В данной лабораторной работе будет рассмотрено несколько известных алгоритмов сортировки, таких как сортировка расческой, сортировка бинарным деревом и сортировка слиянием. Будет проведена теоретическая оценка их трудоемкости. Для подтверждения результатов будет выполнен экспериментальный анализ, в ходе которого мы проанализируем время выполнения каждого алгоритма на разных наборах данных.

Целью данной работы является реализация и исследование алгоритмов сортировки — сортировка расческой, сортировка бинарным деревом и сортировка слиянием. Для достижения поставленной цели необходимо выполнить следующие задачи:

1) описать и реализовать алгоритмы сортировки слиянием, бинарным деревом и расческой;

2) провести экспиремент, чтобы измерить время выполнения для выбранных сортировок;

3) провести сравнительный анализ трудоемкости алгоритмов на основе теоретических расчетов;

4) провести сравнительный анализ процессорного времени реализаций алгоритмов;

5) описать и обосновать полученные результаты в отчете о выполненной лабораторной работе.

1 Аналитическая часть

В этом разделе будут рассмотрены алгоритмы сортировок — расческой, бинарным деревом и слиянием.

1.1 Сортировка расческой

Сортировка расческой является модификацией сортировки пузырьком, и конкурирует с алгоритмами, подобными быстрой сортировке. Основная идея — устранить маленькие значения в конце списка, которые крайне замедляют сортировку пузырьком. В сортировке пузырьком, когда сравниваются два элемента, промежуток равен 1. В сортировке расческой этот промежуток начинается с большого значения и уменьшается в 1.3 раза каждую итерацию, пока не станет единицей. В среднем алгоритм работает лучше, чем пузырьковая сортировка. Худший случай, однако остается $O(n^2)$ [1].

1.2 Сортировка бинарным деревом

Сортировка бинарным деревом — универсальный алгоритм сортировки, заключающийся в построении двоичного дерева поиска по ключам массива, с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке следования ключей [2].

Шаги алгоритма:

1) построить двоичное дерево поиска по ключам массива;

2) собрать результирующий массив путём обхода узлов дерева поиска в необходимом порядке следования ключей;

3) вернуть, в качестве результата, отсортированный массив.

1.3 Сортировка слиянием

Данная сортировка была разработана Джоном фон Нейманом в 1945 году и относится к классу рекурсивных алгоритмов [3].

Алгоритм работает следующим образом.

1) Массив рекурсивно разбивается на 2 равные части, и каждая из частей делится до тех пор, пока размер очередного подмассива не станет равным единице.

2) Далее выполняется операция алгоритма, называемая слиянием. Два единичных массива сливаются в общий результирующий массив, при этом из каждого выбирается меньший элемент (при сортировке по возрастанию) и записывается в свободную левую ячейку результирующего массива. После чего из двух результирующих массивов собирается третий общий отсортированный массив, и так далее. В случае, если один из массивов закончится, элементы другого дописываются в собираемый массив.

3) В конце операции слияния, элементы перезаписываются из результирующего массива в исходный.

Вывод

В данном разделе были описаны идеи рассматриваемых алгоритмов сортировки: расческой, бинарным деревом и слиянием.

2 Конструкторская часть

В данном разделе будут рассмотрены схемы алгоритмов сортировок, а также найдена их трудоемкость.

2.1 Требования к ПО

Ряд требований к программе:

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

2.2 Разработка алгоритмов

На рисунках 2.1-2.5 представлены схемы алгоритмов сортировки - расческой, бинарным деревом и слиянием.

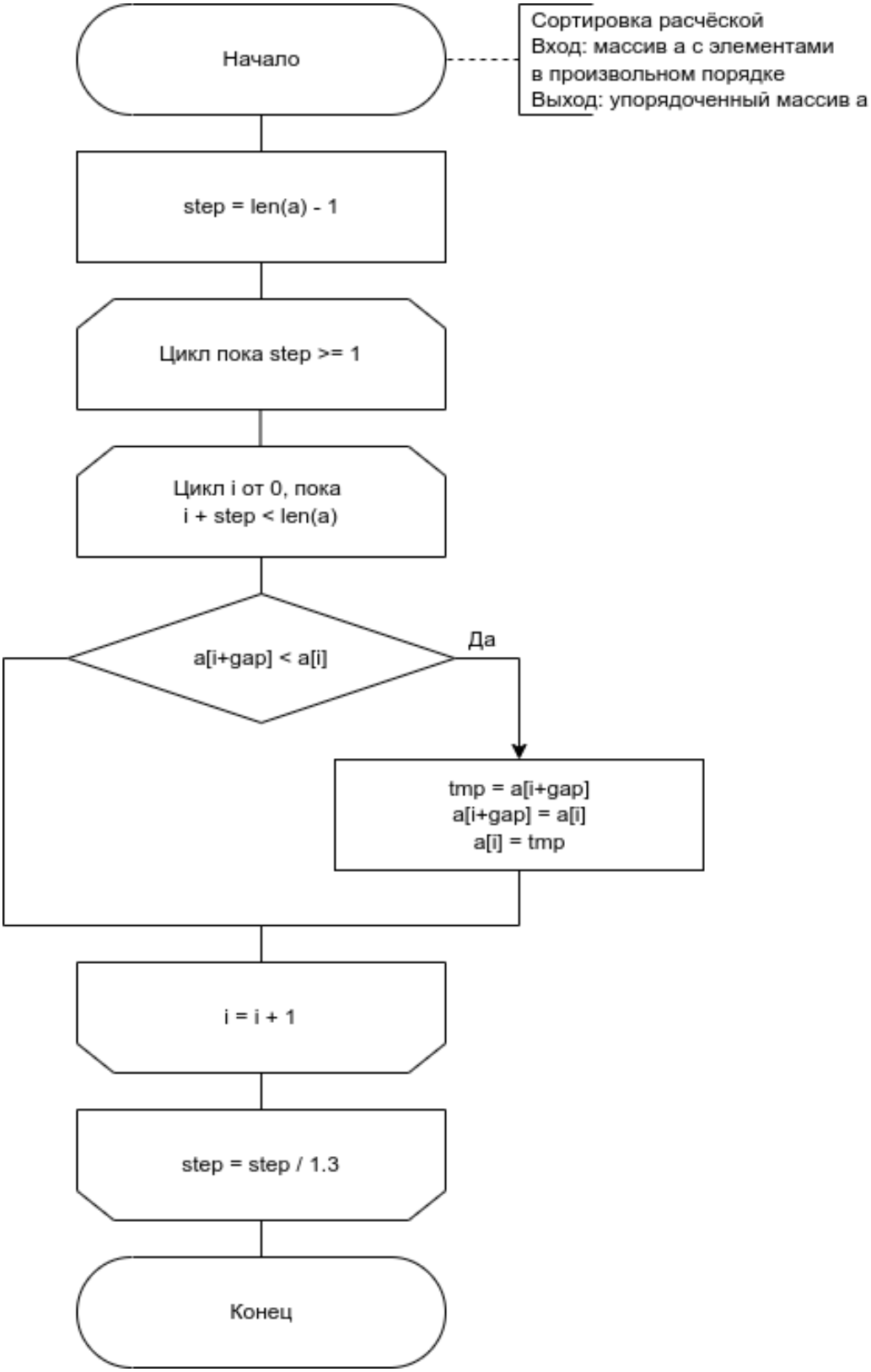

Рисунок 2.1 – Схема алгоритма сортировки расческой

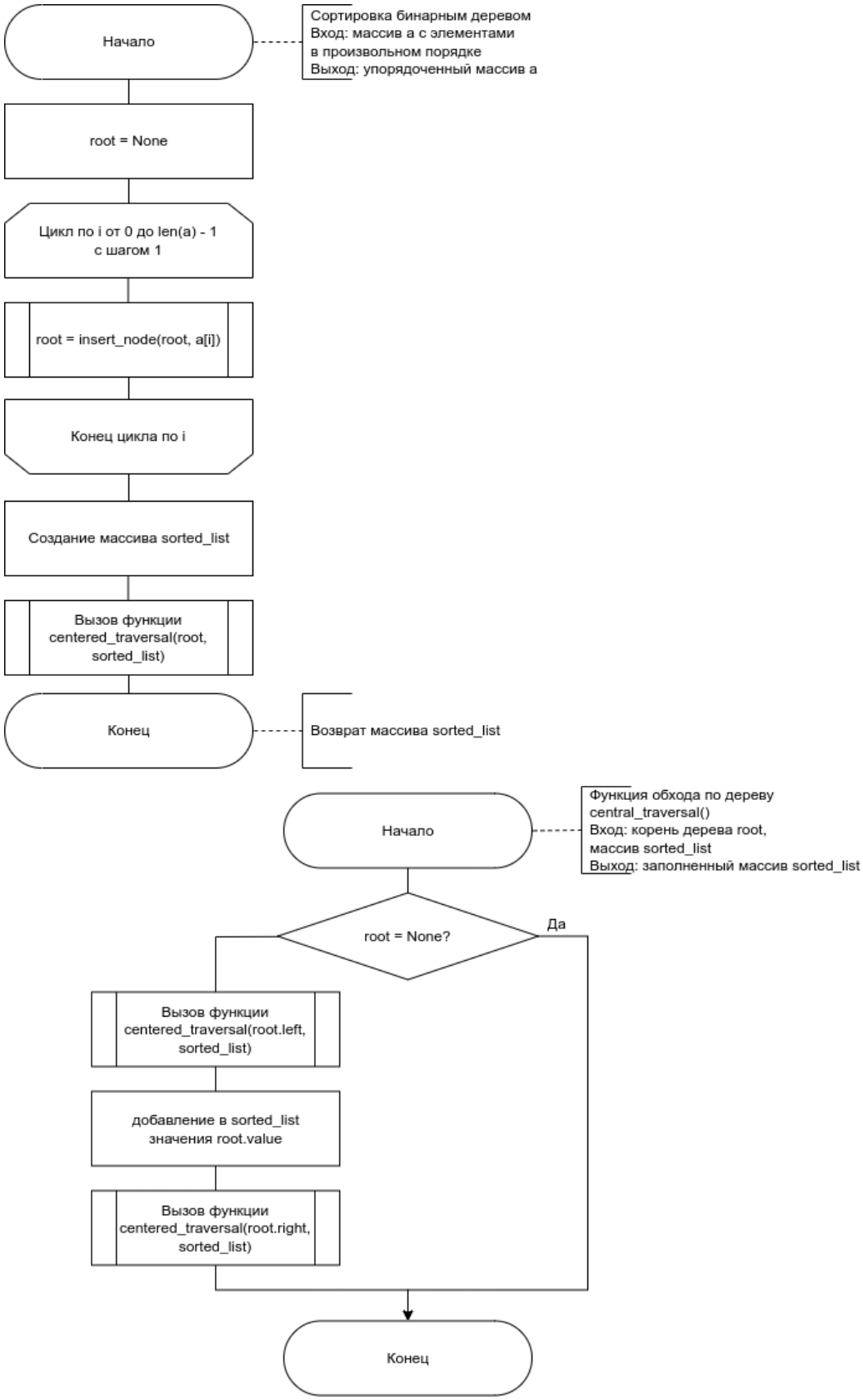

Рисунок 2.2 – Схема алгоритма сортировки бинарным деревом - 1 часть

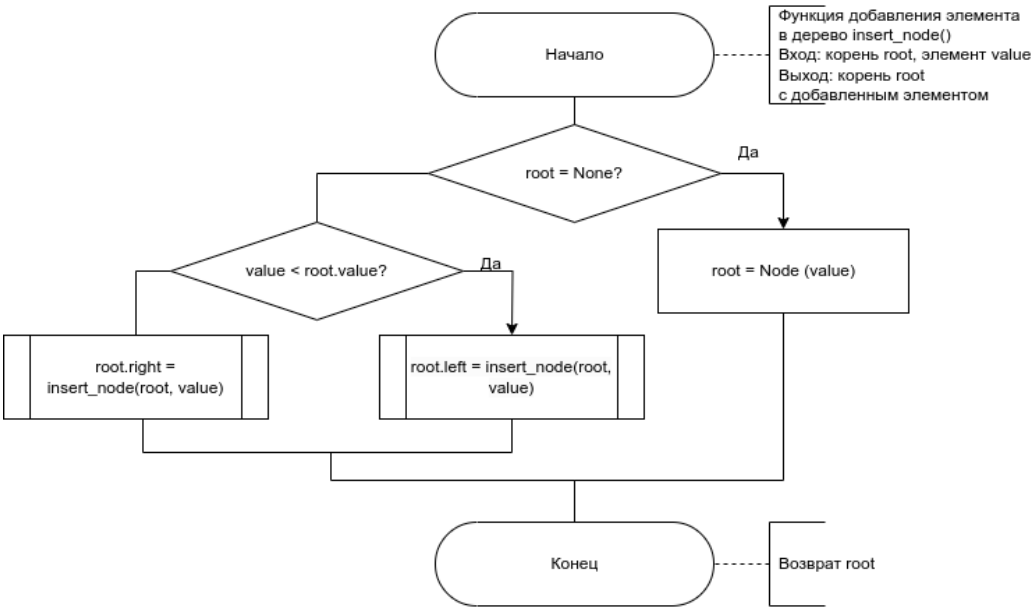

Рисунок 2.3 – Схема алгоритма сортировки бинарным деревом - 2 часть

Рисунок 2.4 – Схема алгоритма сортировки слиянием - 1 часть

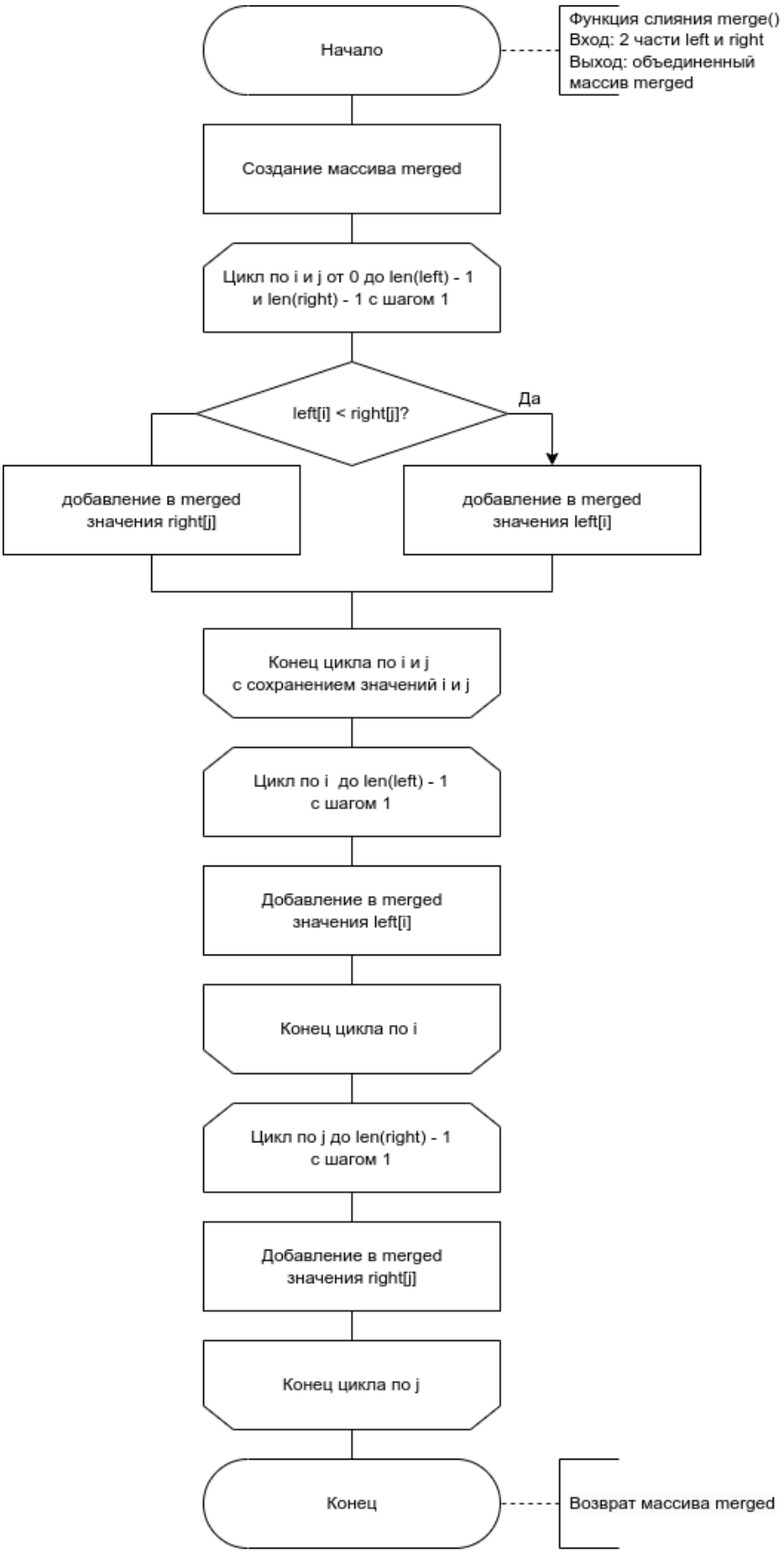

Рисунок 2.5 – Схема алгоритма сортировки слиянием - 2 часть

2.3 Модель вычислений для проведения оценки трудоемкости

Чтобы провести вычисление трудоемкости алгоритмов умножения матриц, введем модель вычислений [4]:

1) Трудоемкость следующих базовых операций единична: +, -, =, +=, -=, ==, !=, <, >, <=, >=, [], ++, –, «, ». Операции *, %, / имеют трудоемкость 2.

2) трудоемкость оператора выбора `if условие then A else B` рассчитывается, как (2.1);

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.1}$$

3) трудоемкость цикла рассчитывается, как (2.2);

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \tag{2.2}$$

4) трудоемкость передачи параметров в функцию и возврат из нее равны 0.

2.4 Трудоёмкость алгоритмов

Определим трудоемкость выбранных алгоритмов сортировки. Введем обозначение N - количество элементов в массиве.

2.4.1 Алгоритм сортировки расческой

Трудоемкость сортировки расческой состоит из:

1) предварительных расчетов трудоемкостью (2.3);

$$f(N) = 3 \tag{2.3}$$

2) цикла, трудоемкость которого равна (2.4), где t — фактор, в нашем случае 1.247.

$$f(N) = 1 + log_t(N) \cdot (1 + 2 + 9N + 1) + 2 \tag{2.4}$$

Трудоемкость при лучшем случае (отсортированный массив) определяется формулой (2.5).

$$f(N) = log_t(N) \cdot (4 + N) + 3 \tag{2.5}$$

Однако стоит учитывать, что при факторе ≈ 1.3, трудоемкость аппроксимируется как в формуле (2.6), где C — некая константа.

$$log_t(N) \cdot (4 + N) + 3 \approx C \cdot log(N) \cdot N \approx O(N \cdot log(N)) \tag{2.6}$$

Трудоемкость при худшем случае (2.7).

$$f(N) = log_t(N) \cdot (4 + 9N) + 3 \approx O(N^2) \tag{2.7}$$

2.4.2 Алгоритм сортировки бинарным деревом

Трудоемкость алгоритма сортировки бинарным деревом состоит из:

1) Трудоемкости построения бинарного дерева, которая равна (2.8);

$$f_{make\_tree} = (5 \cdot \log(N) + 3) * N = N \cdot \log(N) \tag{2.8}$$

2) Трудоемкости восстановления порядка элементов массива, которая равна (2.9).

$$f_{main\_loop} = 7N \tag{2.9}$$

Таким образом общая трудоемкость алгоритма выражается как (2.10).

$$f_{total} = f_{make\_tree} + f_{main\_loop} \tag{2.10}$$

Для наилучшего и наихудшего случая общая трудоемкость алгоритма совпадает и равна (2.11).

$$f = O(N \log(N)) \tag{2.11}$$

2.4.3 Алгоритм сортировки слиянием

Трудоемкость алгоритма сортировки слиянием состоит из трудоемкости функций *merge* и *merge_sort*.

Трудоемкость функции merge состоит из

1) предварительной инициализации, трудоемкость которой равна 2;

2) основного цикла, трудоемкость которого (2.12);

$$f_{main\_loop} = 2 + N \cdot (3 + 8) \tag{2.12}$$

3) 2 дополнительных циклов, трудоемкость которых (2.13)

$$f_{extra\_loop} = (k_1 + k_2) \cdot 6 \tag{2.13}$$

где $k_1 < N/2$ и $k_2 < N/2$.

Общая трудоемкость: $2 + 2 + N * (3 + 8) + (k1 + k2) * 6 = 11N + 4 + 6(k1 + k2) = O(N)$, где $k1 < N/2, k2 < N/2$

Вычислим трудоемкость *merge_sort*. Разобьём все “слияния”, выполненные в процессе сортировки массива на “слои”. Слияние двух частей изначального массива - первый слой, слияния частей каждой из этих двух частей (четвертей оригинального массива) - второй слой, и т.д. Необходимо заметить, что количество элементов в каждом слое равна N. Тогда на каждом слое все слияния выполняются за $O(N)$. Известно, что количество таких строк, называемое глубиной рекурсивного дерева, будет $log(N)$.

Для наилучшего и наихудшего случая общая сложность mergeSort совпадает и равна: $O(N * log(N))$ [3].

Вывод

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

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 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие типы данных:

- количество элементов в массиве — целое число типа *int*;
- массив — список типа *int*.

3.3 Сведения о модулях программы

Программа состоит из двух модулей:

- *main.py* — файл, содержащий весь служебный код;
- *sorts.py* — файл, содержащий код всех сортировок.

3.4 Реализация алгоритмов

В листингах 3.1-3.3 представлены реализации алгоритмов сортировок (расческой, бинарным деревом и слиянием).

Листинг 3.1 – Алгоритм сортировки бинарным деревом

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
def insert_node(root, value):
    if root is None:
        return Node(value)
    if value < root.value:
        root.left = insert_node(root.left, value)
    else:
        root.right = insert_node(root.right, value)
    return root
def centered_traversal(root, sorted_list):
    if root:
        centered_traversal(root.left, sorted_list)
        sorted_list.append(root.value)
        centered_traversal(root.right, sorted_list)
def binary_tree_sort(arr):
    root = None
    for element in arr:
        root = insert_node(root, element)
    sorted_list = []
    centered_traversal(root, sorted_list)
    return sorted_list
```

Листинг 3.2 – Алгоритм сортировки расческой

```
def comb_sort(arr):
    gap = len(arr)
    shrink_factor = 1.3
    while gap >= 1:
        gap = int(gap / shrink_factor)
        i = 0
        while i + gap < len(arr):
            if arr[i] > arr[i + gap]:
                arr[i], arr[i + gap] = arr[i + gap], arr[i]
            i += 1
    return arr
```

Листинг 3.3 – Алгоритм сортировки слиянием

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)
def merge(left, right):
    merged = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    while i < len(left):
        merged.append(left[i])
        i += 1
    while j < len(right):
        merged.append(right[j])
        j += 1
    return merged
```

3.5 Функциональные тесты

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

Таблица 3.1 – Функциональные тесты

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

Вывод

Были представлены средства реализации и сами реализации каждого алгоритма.

4 Исследовательская часть

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

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялся эксперимент представлены далее:

- операционная система — Ubuntu 22.04.3 [7] Linux x86_64;
- память — 16 Гб;
- процессор — Intel® Core™ i5-1135G7 @ 2.40ГГц.

При эксперименте ноутбук не был включен в сеть электропитания.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы, на котором выводится меню и выполняется каждая из 3 сортировок на предварительно подготовленном массиве чисел.

```
 Меню

1. Сортировка расческой
2. Сортировка бинарным деревом
3. Сортировка слиянием
4. Замеры времени
0. Выход

    Выбор:     1
[-3536, -3345, -3305, -2875, -2441, -2315, -2179, -1891, -519, -130, 430, 541, 1426, 1551, 1833, 1883, 2165, 2
344, 2854, 2934, 3030, 3673, 3917, 4272, 4644]


 Меню

1. Сортировка расческой
2. Сортировка бинарным деревом
3. Сортировка слиянием
4. Замеры времени
0. Выход

    Выбор:     2
[-3536, -3345, -3305, -2875, -2441, -2315, -2179, -1891, -519, -130, 430, 541, 1426, 1551, 1833, 1883, 2165, 2
344, 2854, 2934, 3030, 3673, 3917, 4272, 4644]


 Меню

1. Сортировка расческой
2. Сортировка бинарным деревом
3. Сортировка слиянием
4. Замеры времени
0. Выход

    Выбор:     3
[-3536, -3345, -3305, -2875, -2441, -2315, -2179, -1891, -519, -130, 430, 541, 1426, 1551, 1833, 1883, 2165, 2
344, 2854, 2934, 3030, 3673, 3917, 4272, 4644]
```

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.

Функция используется дважды: перед началом выполнения алгоритма и после завершения, затем из конечного времени вычитается начальное, чтобы получить результат.

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

Таблица 4.1 – Отсортированные данные

Размер	Расческой	Бинарным деревом	Слиянием
100	96.68	555.99	95.46
200	220.52	2,263.25	222.92
300	417.48	5,007.74	349.33
400	577.89	8,962.32	471.86
500	794.07	14,073.85	604.17
600	984.98	20,987.65	802.39
700	1,286.17	29,853.94	938.30
800	1,486.94	38,480.89	1,073.84
900	1,774.24	49,576.84	1,199.23

Таблица 4.2 – Отсортированные в обратном порядке данные

Размер	Расческой	Бинарным деревом	Слиянием
100	102.80	557.37	105.87
200	231.22	2,224.29	224.09
300	445.14	5,045.33	352.55
400	601.85	9,001.34	463.68
500	810.16	14,074.22	606.80
600	1,045.80	20,323.24	764.61
700	1,282.02	27,838.25	907.07
800	1,473.24	39,628.49	1,161.71
900	1,888.61	50,180.55	1,242.45

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

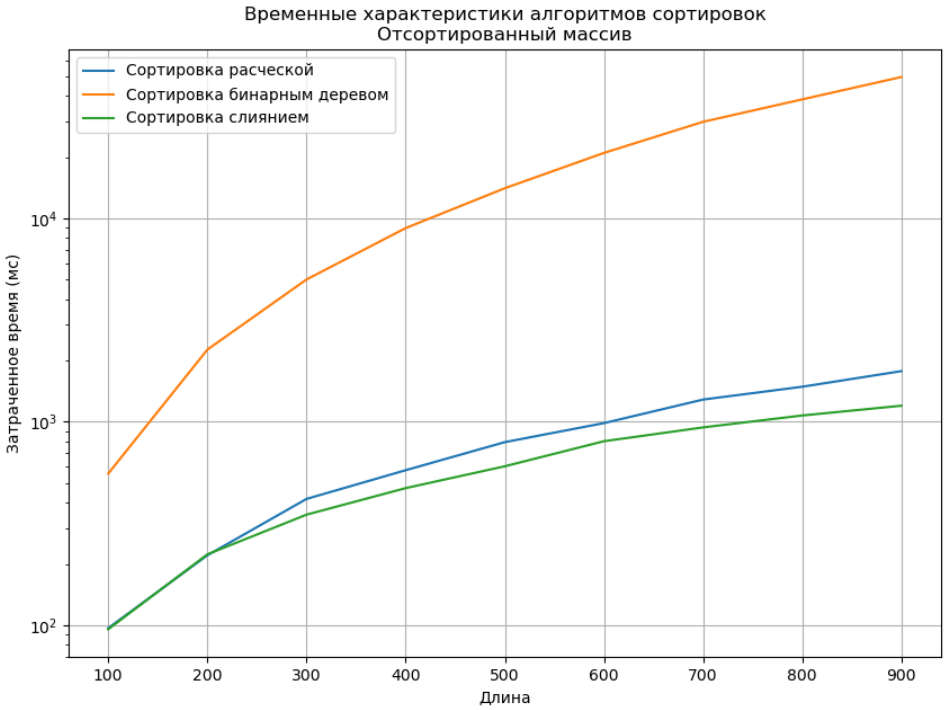

Рисунок 4.2 – Отсортированный массив

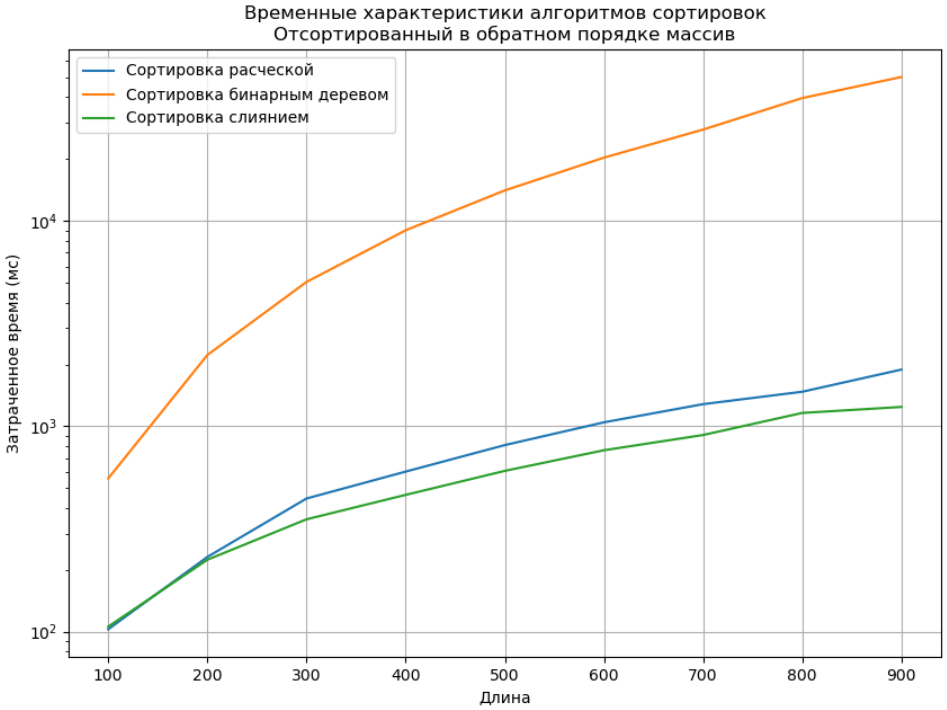

Рисунок 4.3 – Отсортированный в обратном порядке массив

Таблица 4.3 – Случайные данные

Размер	Расческой	Бинарным деревом	Слиянием
100	108.12	99.93	107.90
200	255.87	253.64	249.29
300	470.66	400.53	400.66
400	678.91	585.54	539.63
500	922.55	763.61	690.55
600	1,110.81	935.66	856.70
700	1,370.43	1,202.98	1,016.88
800	1,576.80	1,398.82	1,181.66
900	1,864.26	1,618.94	1,371.91

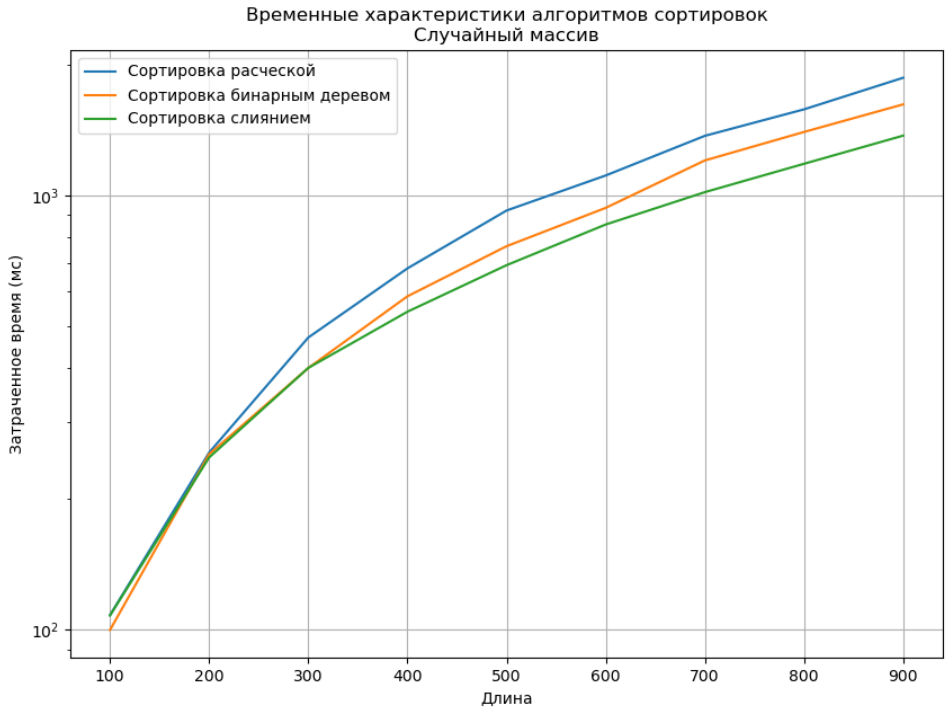

Рисунок 4.4 – Случайный массив

Вывод

Исходя из полученных результатов, сортировка бинарным деревом при массиве заполненном отсортированными в любом порядке данными работает дольше других примерно в 15 раз, но на случайных значениях она примерно в 1.13 раз медленнее, чем сортировка расческой и в 1.2 раза быстрее сортировки слиянием. Так же из эксперимента видно, что сортировка слиянием показала

себя лучше всех при размере массива более 200 на любых данных, а при отсортированных в прямом и обратном порядках данных она быстрее, чем сортировка расческой примерно в 1.3 раза.

Заключение

В результате вычисления трудоемкости алгоритмов сортировок было определено, что алгоритм сортировки бинарным деревом и алгоритм сортировки слиянием и в наилучшем, и в наихудшем случаях имеют трудоемкость, равную $Nlog(N)$, а сортировка расческой в лучшем случае имеет трудоемкость, равную тоже $Nlog(N)$, а в худшем случае N^2.

В результате эксперимента было получено, что сортировка бинарным деревом при массиве заполненном отсортированными в любом порядке данными работает дольше других примерно в 15 раз, но на случайных значениях она примерно в 1.13 раз медленнее, чем сортировка расческой и в 1.2 раза быстрее сортировки слиянием.

Так же из эксперимента видно, что сортировка слиянием показала себя лучше всех при размере массива более 200 на любых данных, а при отсортированных в прямом и обратном порядках данных она быстрее, чем сортировка расческой примерно в 1.3 раза.

В ходе лабораторной работы были описаны и реализованы алгоритмы сортировки слиянием, бинарным деревом и расческой, проведен экспиремент, чтобы измерить время выполнения для этих сортировок. Также был проведен сравнительный анализ трудоемкости алгоритмов на основе теоретических расчетов, анализ процессорного времени реализаций алгоритмов и составлен отчет с описанием и обоснованием полученных результатов.

Поставленная цель лабораторной работы была достигнута.

Список используемых источников

[1] A comparative Study of Sorting Algorithms Comb, Cocktail and Counting Sorting [Электронный ресурс]. Режим доступа: `https://www.irjet.net/archives/V4/i1/IRJET-V4I1249.pdf` (дата обращения: 01.11.2023).

[2] Кнут Дональд. Сортировка и поиск. Вильямс, 2000. Т. 3 из *Искусство программирования*. с. 834.

[3] Cormen T. H. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.

[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: 01.11.2023).

[6] time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html#functions` (дата обращения: 01.11.2023).

[7] Ubuntu 20.04.3 LTS (Focal Fossa) [Электронный ресурс]. Режим доступа: `https://releases.ubuntu.com/20.04/` (дата обращения: 01.11.2023).