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 Гурова Н.А.

Группа ИУ7-54Б

Оценка (баллы)

Преподаватель Волкова Л.Л., Строганов Ю.В.

Москва — 2022 г.

Оглавление

Введение **3**

1 Аналитическая часть **4**

1.1 Гномья сортировка 4

1.2 Плавная сортировка 4

1.3 Сортировка бинарным деревом 4

2 Конструкторская часть **6**

2.1 Требования к вводу 6

2.2 Требования к программе 6

2.3 Разработка алгоритмов 6

2.4 Модель вычислений (оценки трудоемкости) 11

2.5 Трудоёмкость алгоритмов 11

2.5.1 Алгоритм гномьей сортировки 11

2.5.2 Алгоритм плавной сортировки 12

2.5.3 Алгоритм сортировки бинарным деревом 12

3 Технологическая часть **14**

3.1 Требования к ПО 14

3.2 Средства реализации 14

3.3 Сведения о модулях программы 14

3.4 Реализация алгоритмов 15

3.5 Функциональные тесты 18

4 Исследовательская часть **19**

4.1 Технические характеристики 19

4.2 Время выполнения алгоритмов 19

Заключение **22**

Список использованных источников **23**

Введение

Сортировка – процесс перегруппировки заданной последовательности объектов в определенном порядке. Это одна из главных процедур обработки структурированной информации.

Существует множество различных методов сортировки данных, однако любой алгоритм сортировки имеет:

- сравнение, определяющее упорядочность пары элементов;
- перестановка, меняющая местами пару элементов;
- алгоритм сортировки, использующий сравнение и перестановки.

Алгоритмы сортировки имеют большое значение, так как позволяют эффективнее проводить работу с последовательностью данных. Например, возьмем задачу поиска элемента в последовательности – 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 количеству элементов, что куда больше логарифма.

Задачи данной лабораторной работы:

1) изучить и реализовать три алгоритма сортировки:
   (a) гномья сортировка;
   (b) плавная сортировка;
   (c) сортировка бинарным деревом;
2) провести сравнительный анализ трудоемкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
3) провести сравнительный анализ алгоритмов на основе экспериментальных данных по времени выполнения программы;
4) 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 представлены описания алгоритмов гномьей сортировки, плавной сортировки и сортировки двоичным деревом поиска.

1.1 Гномья сортировка

Гномья сортировка [1] – алгоритм сортировки, который использует только один цикл, что является редкостью. В этой сортировке массив просматривается слева-направо, при этом сравниваются и, если нужно, меняются соседние элементы. Если происходит обмен элементов, то происходит возвращение на один шаг назад. Если обмена не было - алгоритм продолжает просмотр массива в поисках неупорядоченных пар.

1.2 Плавная сортировка

Плавная сортировка [1] – алгоритм сортировки выбором, разновидность пирамидальной сортировки. От классического алгоритма пирамидальной сортировки отличается тем, что его сложность зависит от степени изначальной упорядоченности входного массива, на основе которого строится "двоичная куча". Эта сортировка подразумевает два основных шага: формирование последовательности куч и получение на основе на этой последовательности отсортированного массива.

1.3 Сортировка бинарным деревом

Сортировка бинарным деревом [1] – 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.

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.

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 этом разделе будут приведены требования к вводу и прогамме, а также схемы алгоритмов и вычисления трудоемкости данных алгоритмов.

2.1 Требования к вводу

На вход подаются массив объектов и функция, которая позвояет сравнить два объекта масссива между собой.

2.2 Требования к программе

При вводе пустого масссива программа не должна аварийно завершатся. Вывод программы - отсортированый по возрастанию массив.

2.3 Разработка алгоритмов

На рисунках 2.1, 2.2, 2.3 и 2.4 представлены схемы алгоритмов гномьей сортировки, плавной сортировки, построения двоичного дерева поиска и сортировки двоичным деревом соответственно.

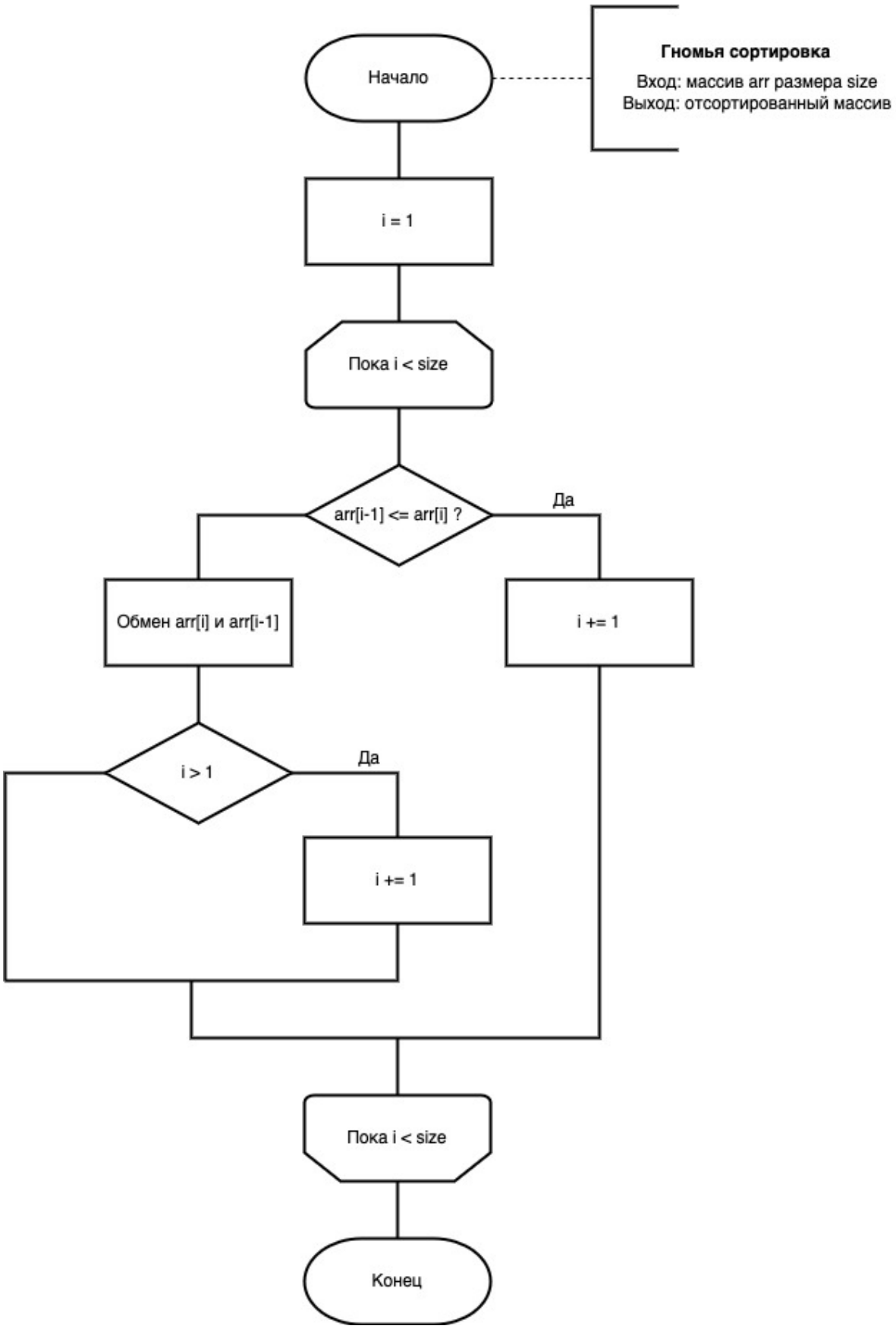

Рисунок 2.1 – Схема алгоритма гномьей сортировки

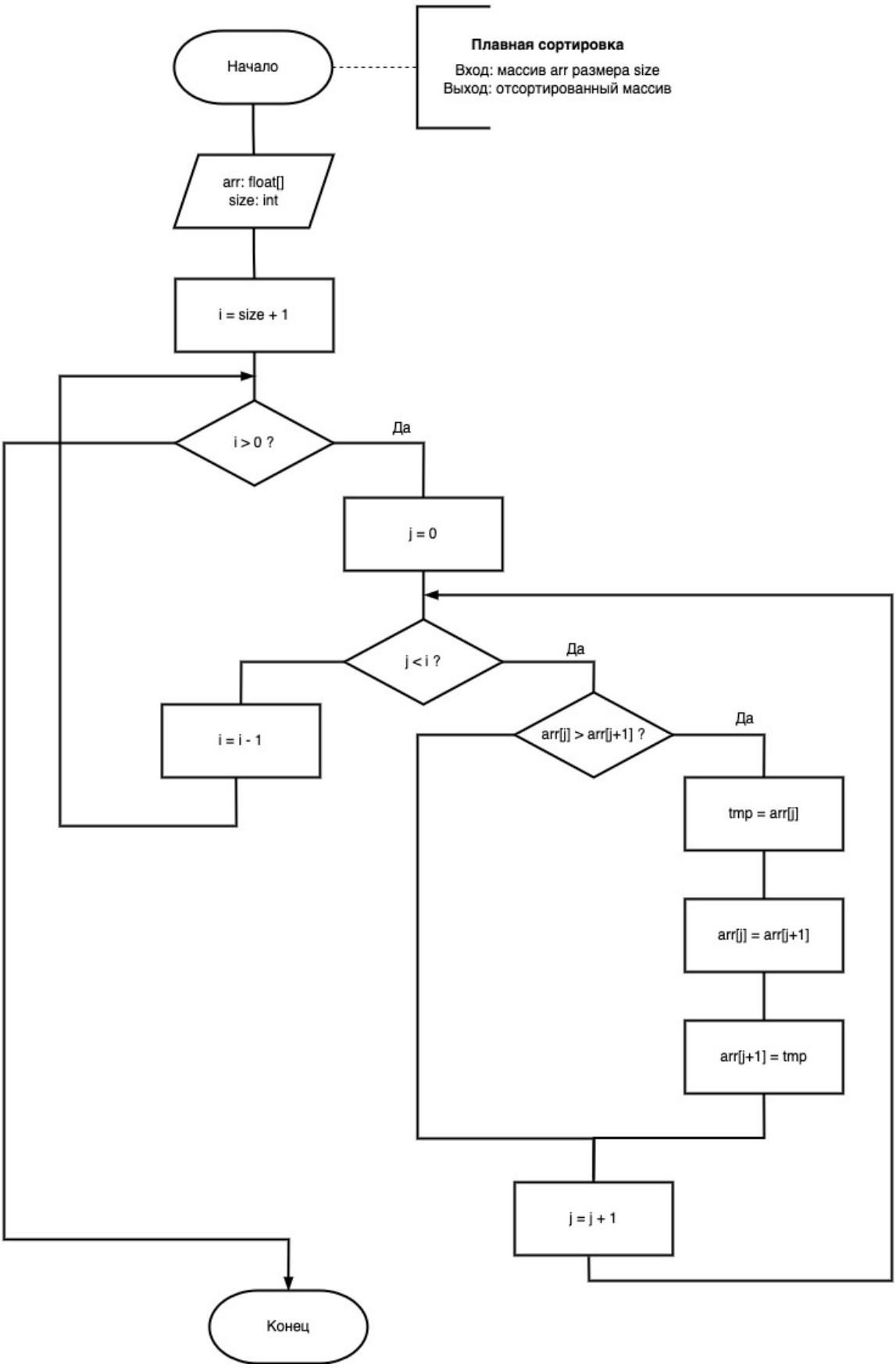

Рисунок 2.2 – Схема алгоритма плавной сортировки

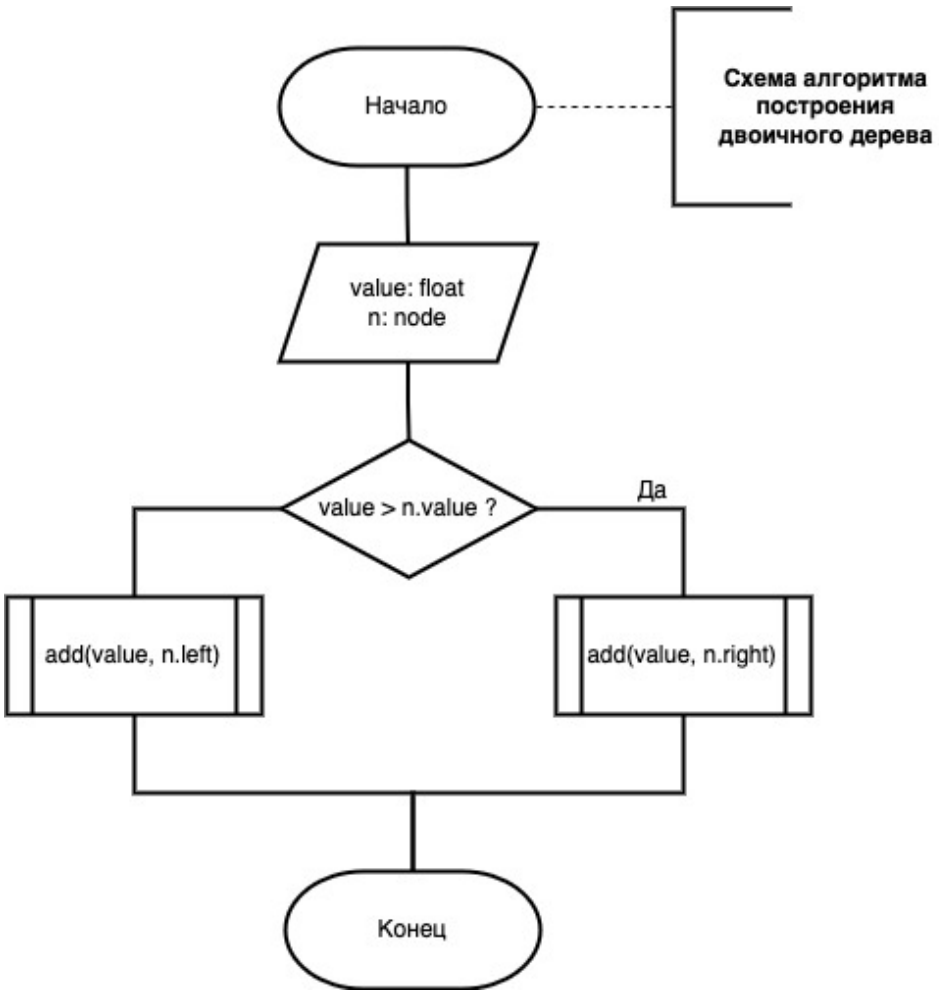

Рисунок 2.3 – Схема алгоритма построения двоичного дерева поиска

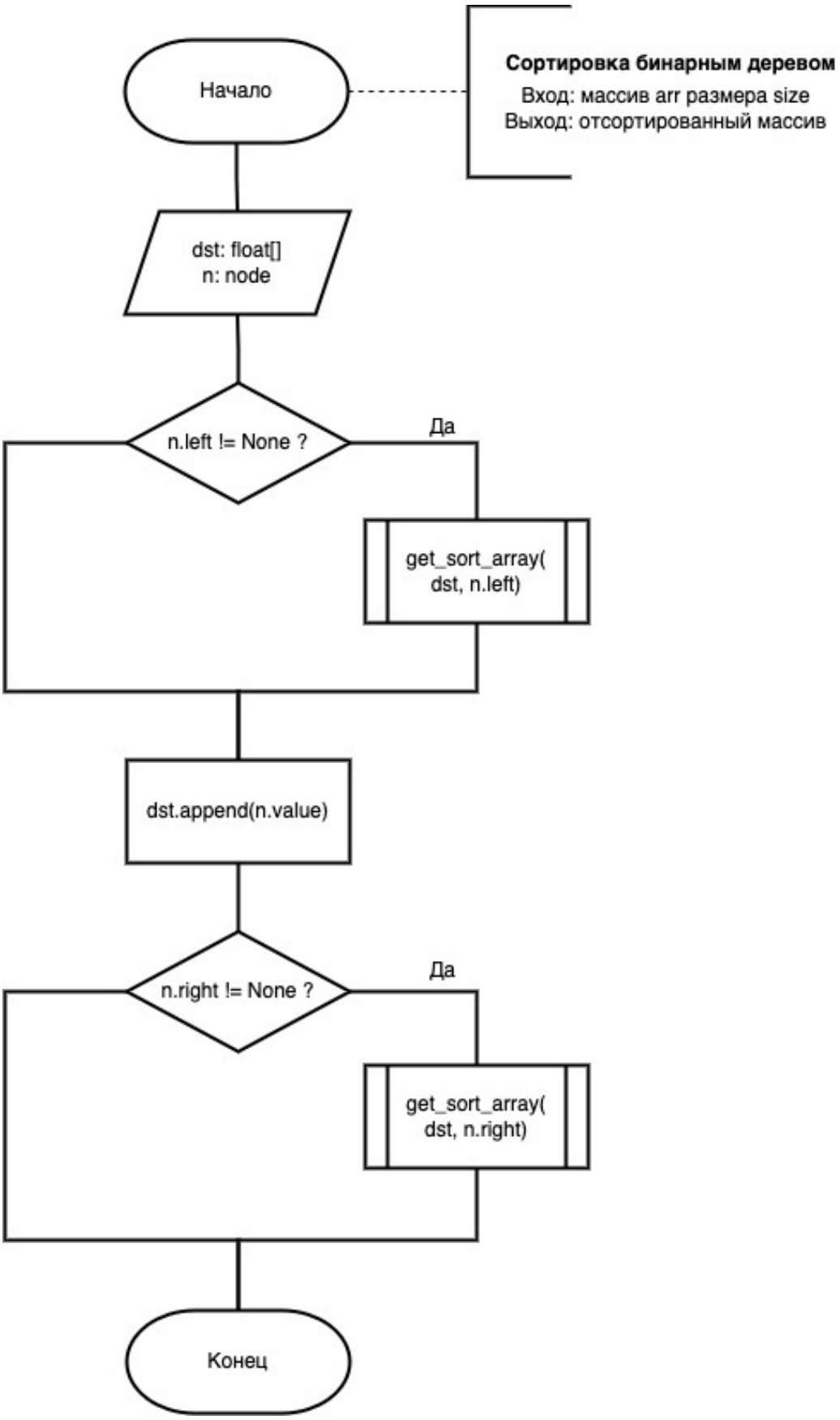

Рисунок 2.4 – Схема алгоритма сортировки двоичным деревом поиска

2.4 Модель вычислений (оценки трудоемкости)

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

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

$$+, -, /, *, \%, =, ==, != , <, >, <=, >=, [] \tag{2.1}$$

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

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

3. трудоемкость цикла рассчитывается, как (2.3);

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

4. трудоемкость вызова функции равна 0.

2.5 Трудоёмкость алгоритмов

Определим трудоемкость выбранных алгоритмов сортировки по коду. Во всех последующих вычислениях обозначим размер массивов как N.

2.5.1 Алгоритм гномьей сортировки

Трудоёмкость в лучшем случае (при уже отсортированном массиве) (2.4):

$$f_{best} = 1 + N(4 + 1) = 5N + 1 = O(N) \tag{2.4}$$

Трудоёмкость в худшем случае (при массиве, отсортированном в обратном порядке) (2.5):

$$f_{worst} = 1 + N(4 + (N-1) * (7+2)) = 9N^2 - 5N + 1 = O(N^2) \quad (2.5)$$

2.5.2 Алгоритм плавной сортировки

Трудоемкость алгоритма плавной сортировки состоит из:

- Трудоемкости построения Леонардовых деревьев, которая равна (2.6).

$$f_{leo\_trees} = 13N \cdot \log(N) + 7 \quad (2.6)$$

- Трудоемкости восстановления порядка элеменотов массива, которая равна (2.7).

$$f_{main\_loop} = 17N \quad (2.7)$$

Таким образом общая трудоемкость алгоритма выражается как (2.8).

$$f_{total} = f_{leo\_trees} + f_{main\_loop} \quad (2.8)$$

Трудоемкость алгоритма в лучшем случае (2.9).

$$f_{best} = O(N) \quad (2.9)$$

Трудоемкость алгоритма в худщем случае (2.10).

$$f_{worst} = O(N \log(N)) \quad (2.10)$$

2.5.3 Алгоритм сортировки бинарным деревом

Трудоемкость алгоритма сортировки бинарным деревом состоит из:

- Трудоемкости построения бинарного дерева, которая равна (2.11).

$$f_{make\_tree} = (5 \cdot \log(N) + 3) * N = N \cdot \log(N) \quad (2.11)$$

- Трудоемкости восстановления порядка элементов массива, которая равна (2.12):

$$f_{main\_loop} = 7N \tag{2.12}$$

Таким образом общая трудоемкость алгоритма выражается как (2.13).

$$f_{total} = f_{make\_tree} + f_{main\_loop} \tag{2.13}$$

Трудоемкость алгоритма в лучшем случае (2.14).

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

Трудоемкость алгоритма в худшем случае (2.15).

$$f_{worst} = O(N \log(N)) \tag{2.15}$$

Вывод

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

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программа принимает на вход массив сравнимых элементов. В качестве результата возвращается массив в отсортированном порядке. Программа не должна аварийно завершаться при некорректном вводе.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран ЯП Python [2].

Данный язык имеет все небходимые инструменты для решения поставленной задачи.

Время работы алгоритмов было замерено с помощью функции time() из библиотеки time [3].

3.3 Сведения о модулях программы

Программа состоит из четырех модулей:

1. algorithms.py - хранит реализацию алгоритмов сортировок;
2. unit_tests.py - хранит реализацию тестирующей системы и тесты;
3. time_memory.py - хранит реализацию системы замера памяти и времени;
4. tools.py - хранит реализацию вспомогательных функций.

3.4 Реализация алгоритмов

В листингах 3.1, 3.2, 3.3 представлены реализации алгоритмов сортировок (гномьей, плавной и бинарным деревом).

Листинг 3.1 – Алгоритм гномьей сортировки

```
def gnome_sort(data):
    i, j, size = 1, 2, len(data)
    while i < size:
        if data[i - 1] <= data[i]:
            i, j = j, j + 1
        else:
            data[i - 1], data[i] = data[i], data[i - 1]
            i -= 1
            if i == 0:
                i, j = j, j + 1
    return data
```

Листинг 3.2 – Алгоритм плавной сортировки

```
def _leonardo_numbers(hi):
    a, b = 1, 1
    numbers = []
    while a <= hi:
        numbers.append(a)
        a, b = b, a + b + 1
    return numbers


def _restore_heap(lst, i, heap, leo_nums):
    current = len(heap) - 1
    k = heap[current]

    while current > 0:
        j = i - leo_nums[k]
        if (lst[j] > lst[i] and
                (k < 2 or lst[j] > lst[i - 1] and lst[j] > lst[i
                   - 2])):
            lst[i], lst[j] = lst[j], lst[i]
            i = j
            current -= 1
```

```
            k = heap[current]
        else:
            break

    while k >= 2:
        t_r, k_r, t_l, k_l = _get_child_trees(i, k, leo_nums)
        if lst[i] < lst[t_r] or lst[i] < lst[t_l]:
            if lst[t_r] > lst[t_l]:
                lst[i], lst[t_r] = lst[t_r], lst[i]
                i, k = t_r, k_r
            else:
                lst[i], lst[t_l] = lst[t_l], lst[i]
                i, k = t_l, k_l
        else:
            break


def _get_child_trees(i, k, leo_nums):
    t_r, k_r = i - 1, k - 2
    t_l, k_l = t_r - leo_nums[k_r], k - 1
    return t_r, k_r, t_l, k_l


def smooth_sort(lst):
    leo_nums = _leonardo_numbers(len(lst))
    heap = []

    for i in range(len(lst)):
        if len(heap) >= 2 and heap[-2] == heap[-1] + 1:
            heap.pop()
            heap[-1] += 1
        else:
            if len(heap) >= 1 and heap[-1] == 1:
                heap.append(0)
            else:
                heap.append(1)
        _restore_heap(lst, i, heap, leo_nums)

    for i in reversed(range(len(lst))):
        if heap[-1] < 2:
            heap.pop()
```

```
62          else:
63              k = heap.pop()
64              t_r, k_r, t_l, k_l = _get_child_trees(i, k, leo_nums)
65              heap.append(k_l)
66              _restore_heap(lst, t_l, heap, leo_nums)
67              heap.append(k_r)
68              _restore_heap(lst, t_r, heap, leo_nums)
69
70      return lst
```

Листинг 3.3 – Алгоритм сортировки бинарным деревом поиска

```
1  class Node:
2      def __init__(self, val, left=None, right=None):
3          self.val = val
4          self.left = left
5          self.right = right
6
7      def add(self, val):
8          if self.val > val:
9              if self.left is None:
10                 self.left = Node(val)
11             else:
12                 self.left.add(val)
13         else:
14             if self.right is None:
15                 self.right = Node(val)
16             else:
17                 self.right.add(val)
18
19
20 def _one_node_sort(node, dst_list):
21     if node.left is not None:
22         _one_node_sort(node.left, dst_list)
23
24     dst_list.append(node.val)
25
26     if node.right is not None:
27         _one_node_sort(node.right, dst_list)
28
29
30 def binary_tree_sort(arr):
```

```
31      if len(arr) == 0:
32          return arr
33
34      head = Node(arr[0])
35      for i in range(1, len(arr)):
36          head.add(arr[i])
37
38      dst = []
39      _one_node_sort(head, dst)
40      return dst
```

3.5 Функциональные тесты

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

Таблица 3.1 – Функциональные тесты

Входной массив	Ожидаемый результат	Результат
[]	[]	[]
[1]	[1]	[1]
[0, 1]	[0, 1]	[0, 1]
[1, 0]	[0, 1]	[0, 1]
[2, −2]	[−2, 2]	[−2, 2]
[2, −2, 2]	[−2, 2, 2]	[−2, 2, 2]
[0, 0, 1, 2]	[0, 0, 1, 2]	[0, 0, 1, 2]
[5, 2, 1, 8, 9, 10]	[1, 2, 5, 8, 9, 10]	[1, 2, 5, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7]	[1, 2, 3, 4, 5, 6, 7]	[1, 2, 3, 4, 5, 6, 7]
[9, 2, 1]	[1, 2, 9]	[1, 2, 9]
[9, −10000, −20000]	[−20000, −10000, 9]	[−20000, −10000, 9]
[5, 4, 4, 3]	[3, 4, 4, 5]	[3, 4, 4, 5]

Вывод

Были выбраны средства реализации и замера времени выполнения алгоритмов сортировки. Были реализованы алгоритмы трех сортировок и программа, реализующая замер времени работы этих алгоритмов.

4 Исследовательская часть

В данном разделе приводятся результаты замеров алгоритмов по процессорному времени.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: macOS Monterey 12.5.1
- Память: 16 Гб.
- Процессор: 2,3 ГГц 4-ядерный процессор Intel Core i5.

Во время тестирования устройство было подключено к сети электропитания, нагружено приложениями окружения и самой системой тестирования.

4.2 Время выполнения алгоритмов

Алгоритмы тестировались при помощи функции time() из библиотеки time языка Python. Данная функция возвращает количество секунд, прошедших с начала эпохи, типа float.

Контрольная точка возвращаемого значения не определна, поэтому допустима только разница между результатами последовательных вызовов.

Замеры времени для каждой длины массива проводились 1000 раз. В качестве результата взято среднее время работы алгоритма на данной длине массива. При каждом запуске алгоритма, на вход подавались случайно сгенерированные массивы. Тестовые пакеты создавались до начала замера времени.

Результаты замеров приведены на рисунках 4.1, 4.2, 4.3 (в микросекундах).

Массив не отсортирован

N	Гномья сортировка	Плавная сортировка	Сортировка бинарным деревом
0	0	1	0
1	0	3	2
5	1	11	7
10	1	16	14
50	6	75	221
100	12	147	859
250	29	382	5281

Рисунок 4.1 – Результаты замеров времени для неотсортированного массива (в микросекундах)

Массив отсортирован

N	Гномья сортировка	Плавная сортировка	Сортировка бинарным деревом
0	0	1	0
1	0	1	1
5	1	8	4
10	1	15	13
50	5	72	218
100	11	145	850
250	30	390	5412

Рисунок 4.2 – Результаты замеров времени для отсортированного массива (в микросекундах)

Массив отсортирован в обратном порядке

N	Гномья сортировка	Плавная сортировка	Сортировка бинарным деревом
0	0	1	0
1	0	1	1
5	1	7	4
10	1	15	12
50	6	72	219
100	14	148	838
250	43	370	5291

Рисунок 4.3 – Результаты замеров времени для отсортированного в обратном порядке массива (в микросекундах)

Вывод

Алгоритм гномьей сортировки работает быстрее алгоритмов плавной сортировки и сортировки бинарным деревом независимо от того, как расположены объекты в сортируемом массиве. Напротив, сортировка бинарным деревом проигрывает двум другим алгоритмам по времени, так как использует рекурсию. Также можно заметить, что алгоритму плавной сортировки не важно, как расположены объекты в массиве, он всегда будет давать примерно одинаковые временные результаты.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- изучены и реализованы 3 алгоритма сортировки: гномья, плавная, бинарным деревом;
- проведен сравнительный анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
- проведен сравнительный анализ алгоритмов на основе экспериментальных данных;
- подготовлен отчет о лабораторной работе.

Теоретически и экспериментально было подтверждено различие во временной эффективности различных алгоритмов сортировок, в частности алгоритма гномьей сортировки, алгоритма плавной сортировки и алгоритма сортировки бинарным деревом. Было получено, что гномья сортировка наиболее быстрая среди этих трех алгоритмов (работает в 4-5 раз быстрее), а сортировка бинарным деревом наиболее медленная (проигрывает в 3-4 раза).

Список использованных источников

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

[2] Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org` (дата обращения: 04.09.2021).

[3] time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html#functions` (дата обращения: 04.09.2021).