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 Малышев И.Н.

Группа ИУ7-54Б

Оценка (баллы)

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

Москва — 2022 г.

Оглавление

Введение

Одной из важнейших процедур обработки структурированной информации является сортировка.

Сортировка – это процесс перегруппировки заданной последовательности (кортежа) объектов в некотором определеннном порядке. Такой опредленный порядок позволяет, в некоторых случаях, эффективнее и удобнее работать с заданной последовательностью. В частности, одной из целей сортировки является облегчение задачи поиска элемента в отсортированном множестве.

Алгоритмы сортировки используются практически в любой программой системе. Целью алгоритмов сортировки является упорядочение последовательности элементов данных. Поиск элемента в последовательности отсортированных данных занимает время, пропорциональное логарифму количества элементов в последовательности, а поиск элемента в последовательности не отсортированных данных занимает время, пропорциональное воличеству элементов в последовательности, то есть намного большее. Существует множество различных методов сортировки данных. Однако любой алгоритм сортировки можно разбить на три основные части:

- сравнение, определяющее упорядочность пары элементов;
- перестановка, меняющая местами пару элементов;
- собственно сортирующий алгоритм, который осуществляет сравнение и перестановку элементов данных до тех пор, пока все эти элементы не будут упорядочены.

Одной из важнейшей характеристикой любого алгоритма сортировки является скорость его работы, которая определяется функциональной зависимостью среднего времени сортировки последовательностей элементов данных, определенной длины, от этой длины.

Задачи данной лабораторной:

1) изучить и реализовать три алгоритма сортировки:
   (a) пирамидальная 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 пирамидальной сортировки, плавной сортировки и сортировки двоичным деревом поиска.

1.1 Пирамидальная сортировка

Пирамидальная сортировка [1] – это метод сортировки сравнением, основанный на такой структуре данных как "двоичная куча". Двоичная куча (binary heap) – структура данных, позволяющая быстро (за логарифмическое время) добавлять элементы и извлекать элемент с максимальным приоритетом (например, максимальный по значению).

Общие идеи алгоритма:

1) построить двоичную кучу из входных данных;

2) поменять местами первый и последний элемент кучи;

3) уменьшить размер кучи на 1;

4) если размер кучи больше 0, перейти к пункту 1.

1.2 Плавная сортировка

Плавная сортировка [1] – алгоритм сортировки выбором, разновидность пирамидальной сортировки. От класиического алгоритма пирамидальной сортировки отличается тем, что его сложность зависит от степени изначальной упорядоченности входного массива, на основе которого строится "двоичная куча".

Шаги алгоритма:

1) сформировать последовательность куч;

2) сформировать отсортированный массив.

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

Необходимо реализовать три алгоритма сортировки: пирамидальную сортировку, плавную сортировку и сортировку бинарным деревом, а также выполнить теоретическую оценку сложности этих алгоритмов и сравнить ее с эксперементальными показателями.

2 Конструкторская часть

В этом разделе будут приведены схемы алгоритмов и вычисления трудоемкости данных алгоритмов.

2.1 Разработка алгоритмов

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

2.2 Модель вычислений (оценки трудоемкости)

Для последующего вычисления трудоемкости необходимо ввести модель вычислений:

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.

2.3 Трудоёмкость алгоритмов

Обозначим во всех последующих вычислениях размер массивов как N.

2.3.1 Алгоритм пирамидальной сортировки

Трудоемкость алгоритма пирамидальной сортировки состоит из:

- Трудоемкости полной "просейки"всего массива, которая равна (2.4).

$$f_{full\_sift} = 2 + N \cdot f_{sift} \tag{2.4}$$

- Трудоемкости единоразовой "просейки"f_{sift}, которая участвует в расчете трудоемкости полной "просейки"(2.4) и основного цикла (2.6), которая равна (2.5).

$$f_{sift} = 4 + \begin{cases} 4, & \text{л.с.} \\ 22 \cdot \log(N), & \text{х.с.} \end{cases} \tag{2.5}$$

- Трудоемкости основного цикла, которая равна (2.6).

$$f_{main\_loop} = 1 + (8 + f_{sift}) \cdot N \tag{2.6}$$

Таким образом общая трудоемкость алгоритма выражается как (2.7).

$$f_{total} = f_{full\_sift} + f_{main\_loop} \tag{2.7}$$

Минимально возможная трудоемкость алгоритма (2.8).

$$f_{heap\_min} = 12N + 3 + 4 \cdot \log(N) = O(N \log(N)) \tag{2.8}$$

Максимально возможная трудоемкость алгоритма (2.9).

$$f_{heap\_max} = 16 \cdot \log(N) + 3 \cdot \log(N) = O(N \log(N)) \tag{2.9}$$

2.3.2 Алгоритм плавной сортировки

Трудоемкость алгоритма плавной сортировки состоит из:

- Трудоемкости построения Леонардовых деревьев, которая равна (2.10).

$$f_{leo\_trees} = 13N \cdot \log(N) + 7 \tag{2.10}$$

- Трудоемкости восстановления порядка элеменотов массива, которая равна (2.11).

$$f_{main\_loop} = 17N \tag{2.11}$$

Таким образом общая трудоемкость алгоритма выражается как (2.12).

$$f_{total} = f_{leo\_trees} + f_{main\_loop} \tag{2.12}$$

Минимально возможная трудоемкость алгоритма (2.13).

$$f_{min} = O(N) \tag{2.13}$$

Максимально возможная трудоемкость алгоритма (2.14).

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

2.3.3 Алгоритм сортировки бинарным деревом

Трудоемкость алгоритма сортировки бинарным деревом состоит из:

- Трудоемкости построения бинарного дерева, которая равна (2.15).

$$f_{make\_tree} = (5 \cdot \log(N) + 3) * N = N \cdot \log(N) \tag{2.15}$$

- Трудоемкости восстановления порядка элементов массива, которая равна (2.16):

$$f_{main\_loop} = 7N \tag{2.16}$$

Таким образом общая трудоемкость алгоритма выражается как (2.17).

$$f_{total} = f_{make\_tree} + f_{main\_loop} \tag{2.17}$$

Минимально возможная трудоемкость алгоритма (2.18).

$$f_{min} = O(N \log(N)) \tag{2.18}$$

Максимально возможная трудоемкость алгоритма (2.19).

$$f_{max} = O(N \log(N)) \tag{2.19}$$

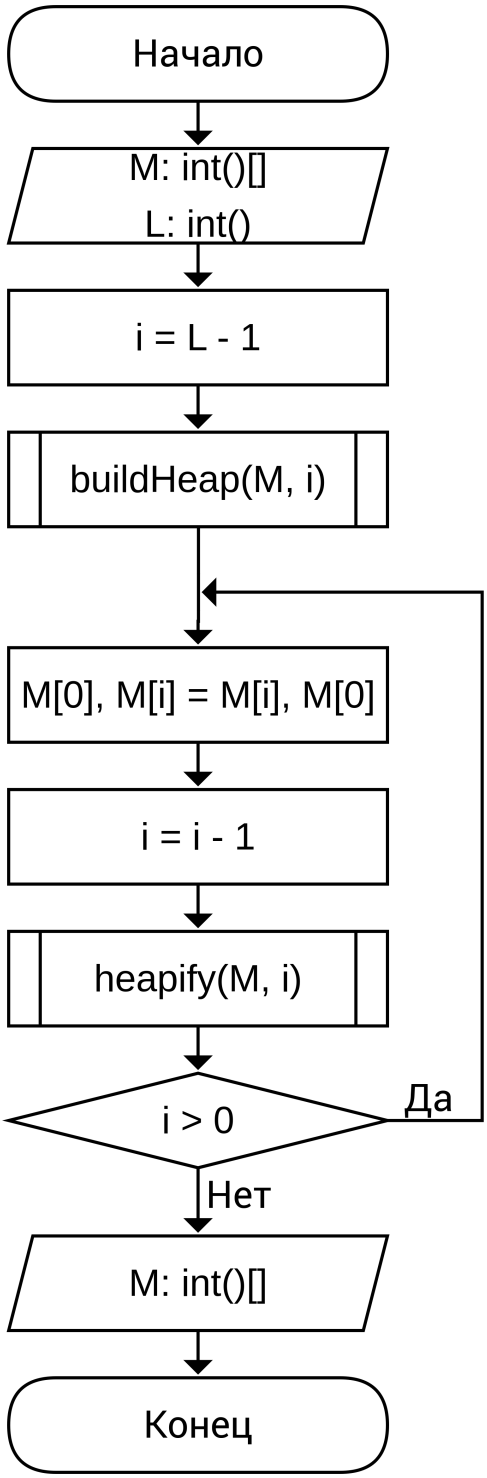

Рисунок 2.1 – Схема алгоритма пирамидальной сортировки

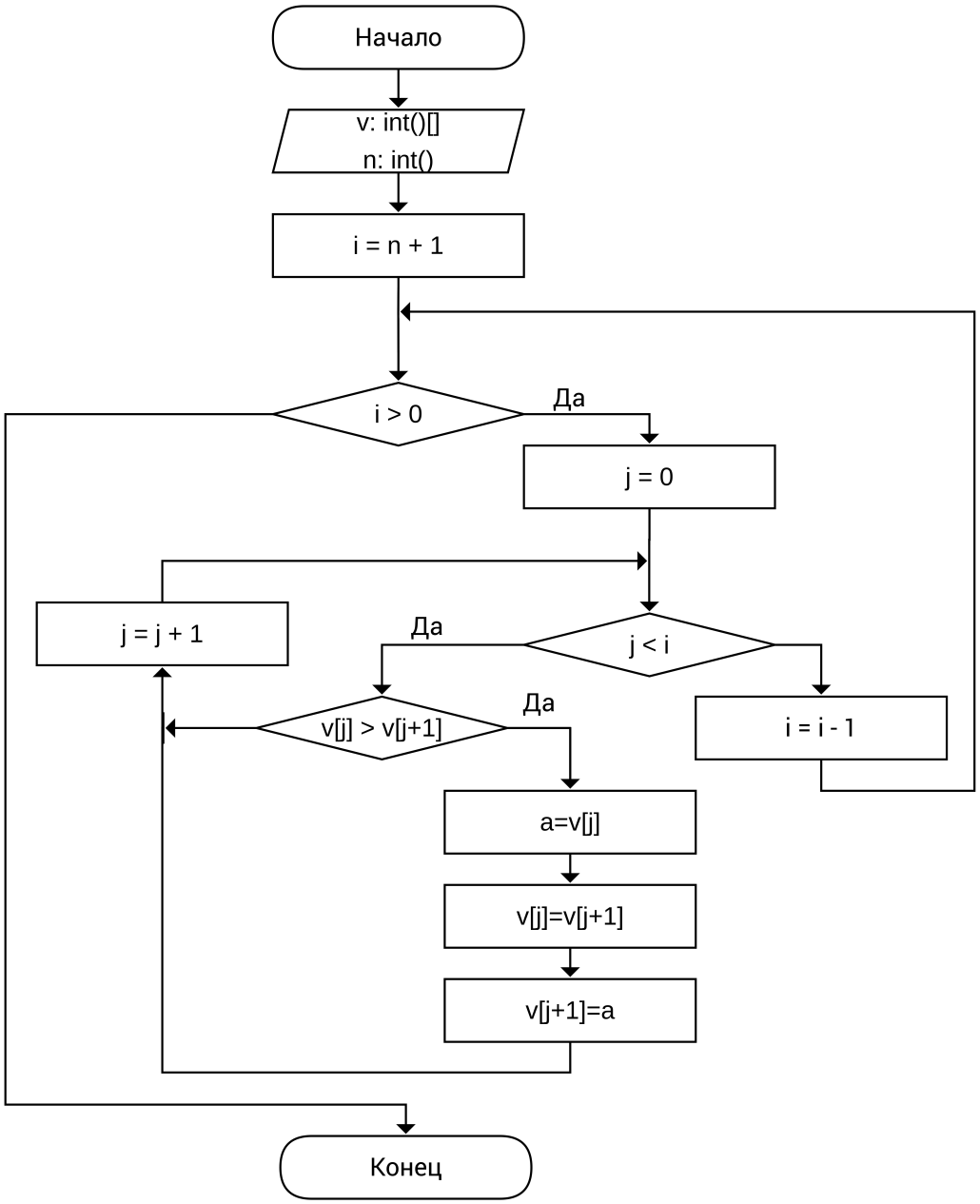

Рисунок 2.2 – Схема алгоритма плавной сортировки

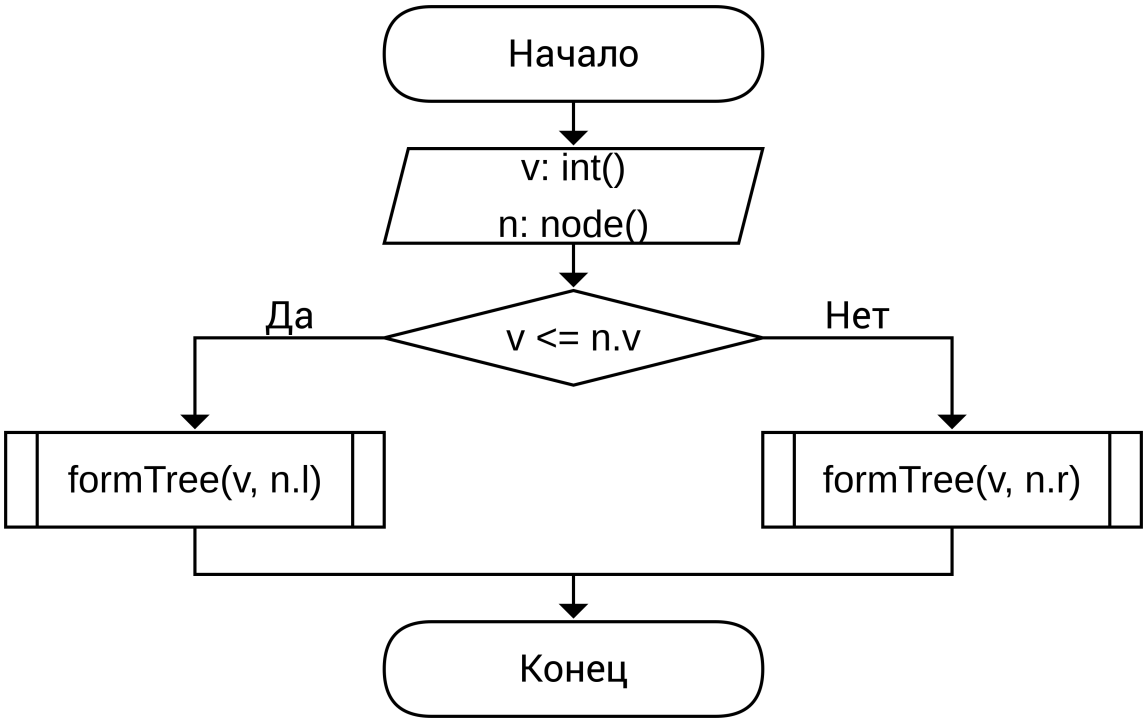

Рисунок 2.3 – Схема алгоритма построения двоичного дерева поиска

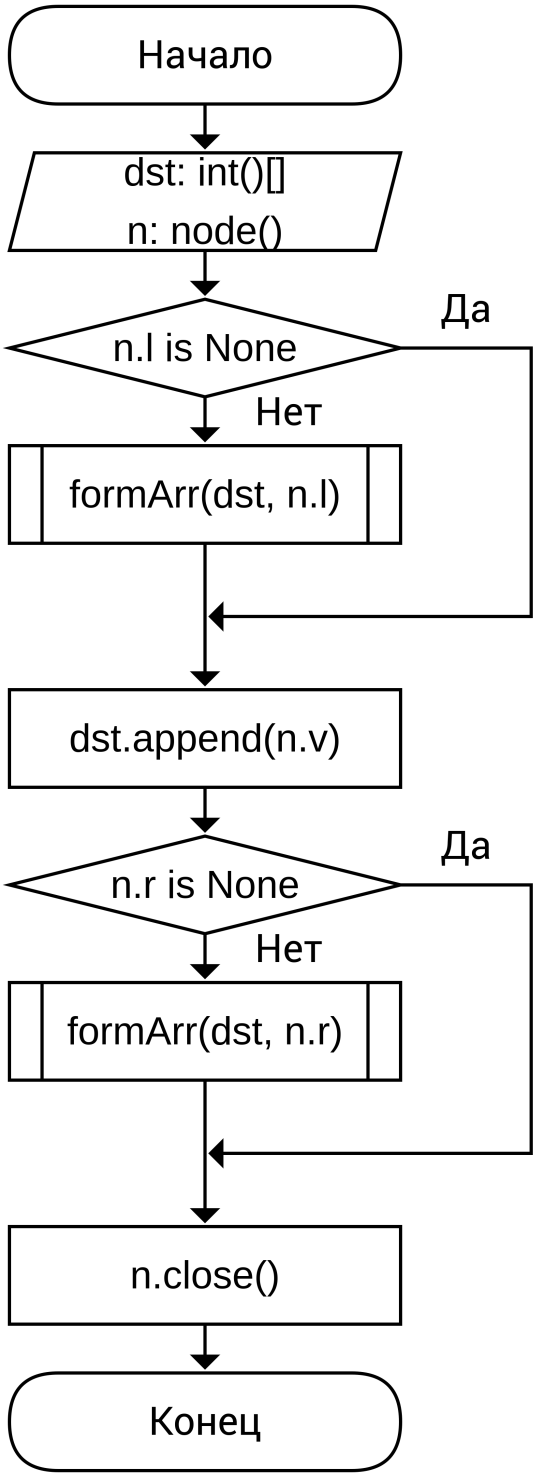

Рисунок 2.4 – Схема алгоритма сортировки двоичным деревом

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 программе предъявляется ряд требований:

- на вход подаётся массив сравнимых элементов (целые числа);
- на выходе — тот же массив, но 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 язык программирования (ЯП) Python [2].

Данный ЯП предоставляет широкий набор библиотек и руководств по их использованию, что значительно облегчает выполнения большинства задач в области программирования.

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

3.3 Сведения о модулях программы

Программа состоит из шести модулей:

1. beenary_tree_sort.py – файл, содержащий реализацию бинарной сортировки;
2. heap_sort.py – файл, содержащий реализаци пирамидальной сортировки;

3. smooth_sort.py – фаайл, содержащий реализацию плавной сортировки;

4. perfomance_tests.py – файл, содержащий функциональные тесты;

5. time_tests.py – файл, содержащий программу, выполняющую замеры времени работы сортировок;

6. memory_tests.py – файл, содержащий программу, выполняющую замеры выделяемой памяти.

3.4 Реализация алгоритмов

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

Листинг 3.1 – Алгоритм пирамидальной сортировки

```
def heap_sort(data):
    for start in range((len(data) - 2) // 2, -1, -1):
        HeapSift(data, start, len(data) - 1)
    for end in range(len(data) - 1, 0, -1):
        data[end], data[0] = data[0], data[end]
        HeapSift(data, 0, end - 1)
    return data

def HeapSift(data, start, end):
    root = start
    while True:
        child = root * 2 + 1
        if child > end: break
        if child + 1 <= end and data[child] < data[child + 1]:
            child += 1
        if data[root] < data[child]:
            data[root], data[child] = data[child], data[root]
            root = child
        else:
            break
```

Листинг 3.2 – Алгоритм плавной сортировки

```
def smoothsort(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()
        else:
            k = heap.pop()
            t_r, k_r, t_l, k_l = get_child_trees(i, k, leo_nums)
            heap.append(k_l)
        restore_heap(lst, t_l, heap, leo_nums)
        heap.append(k_r)
        restore_heap(lst, t_r, heap, leo_nums)

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
```

```
42              (k < 2 or lst[j] > lst[i-1] and lst[j] > lst[i-2])):
43              lst[i], lst[j] = lst[j], lst[i]
44              i = j
45              current -= 1
46              k = heap[current]
47          else:
48              break
49
50      while k >= 2:
51          t_r, k_r, t_l, k_l = get_child_trees(i, k, leo_nums)
52          if lst[i] < lst[t_r] or lst[i] < lst[t_l]:
53              if lst[t_r] > lst[t_l]:
54                  lst[i], lst[t_r] = lst[t_r], lst[i]
55                  i, k = t_r, k_r
56              else:
57                  lst[i], lst[t_l] = lst[t_l], lst[i]
58                  i, k = t_l, k_l
59          else:
60              break
61
62  def get_child_trees(i, k, leo_nums):
63      t_r, k_r = i - 1, k - 2
64      t_l, k_l = t_r - leo_nums[k_r], k - 1
65      return t_r, k_r, t_l, k_l
66
67  def smooth_sort(data):
68      smoothsort(data)
69      return data
```

Листинг 3.3 – Алгоритм сортировки бинарным деревом

```
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 insert(self, val):
8           if self.val > val:
9               if self.left is None:
10                  self.left = Node(val)
11              else:
```

```
12                  self.left.insert(val)
13          else:
14              if self.right is None:
15                  self.right = Node(val)
16              else:
17                  self.right.insert(val)
18
19      def get_sorted(self, dst):
20          if self.left is not None:
21              self.left.get_sorted(dst)
22          dst.append(self.val)
23          if self.right is not None:
24              self.right.get_sorted(dst)
25
26      def sort(self):
27          dst = []
28          self.get_sorted(dst)
29          return dst
30
31
32  def beenary_tree_sort(arr):
33      if len(arr) == 0:
34      return arr
35
36      head = Node(arr[0])
37      for i in range(1, len(arr)):
38      head.insert(arr[i])
39
40      return head.sort()
```

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
$[1, 2, 3, 4]$	$[1, 2, 3, 4]$	$[1, 2, 3, 4]$
$[5, 4, 3, 2, 1]$	$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$
$[3, 2, -5, 0, 1]$	$[-5, 0, 0, 2, 3]$	$[-5, 0, 0, 2, 3]$
$[4]$	$[4]$	$[4]$
$[]$	$[]$	$[]$

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 будут приведены примеры работы программ, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование, следующие:

- Операционная система Ubuntu 22.04.1 [4] Linux x86_64.
- Память: 8 ГБ.
- Процессор: AMD® Ryzen 5 3500u.

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно 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.

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

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

Результаты замеров приведены в таблицах 4.1, 4.2 и 4.3.

Таблица 4.1 – Результаты замеров времени для отсортированного массива (в микросекундах).

Длина массива	Пирамидальная	Плавная	Бинарным деревом
1	1	2	1
26	60	55	86
51	85	73	199
76	129	105	406
101	186	139	703
126	236	172	1076
151	296	208	1536
176	358	245	2085
201	422	276	2699

Таблица 4.2 – Результаты замеров времени для отсортированного в обратном порядке массива (в микросекундах).

Длина массива	Пирамидальная	Плавная	Бинарным деревом
1	1	2	1
26	48	141	90
51	68	208	188
76	109	315	403
101	156	437	686
126	202	578	1059
151	254	730	1547
176	307	849	2058
201	357	992	2868

Таблица 4.3 – Результаты замеров времени для случайных данных (в микросекундах).

Длина массива	Пирамидальная	Плавная	Бинарным деревом
1	1	2	2
26	60	109	66
51	90	202	77
76	145	324	118
101	203	454	164
126	260	593	210
151	324	736	260
176	388	886	309
201	456	1036	366

Вывод

Алгоритм плавной сортировки работает быстрее на изначально упорядоченном массиве, чем остальные алгоритмы. Алгоритм пирамидальной сортировки работает быстрее в случае, если данные упорядоченны в обратном направлении. Алгоритм сортировки бинарным деревом работает быстрее на случайно упорядоченных данных.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- изучены и реализованы 3 алгоритма сортировки: пирамидальная, 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.

Поставленная цель достигнута.

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).

[4] Ubuntu 20.04.3 LTS (Focal Fossa) [Электронный ресурс]. Режим доступа: `https://releases.ubuntu.com/20.04/` (дата обращения: 04.09.2021).