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 «Анализы Алгоритмов»

на тему: «Алгоритмы сортировки»

Студент группы ИУ7-56Б _____ (Подпись, дата) Мамврийский И. С. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Волкова Л. Л. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Строганов Ю. В.. (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

Сортировка – процесс перегруппировки заданной последовательности объектов в некотором определеннном порядке.

Целью алгоритмов сортировки является упорядочение последовательности элементов данных. 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.

Целью данной лабораторной работы является исследование алгоритмов сортировок.

Для достижения поставленной цели небходимо выполнить следующие задачи:

— Описать три алгоритма сортировки: блинная, пирамидальная, перемешиванием;

— Создать программное обеспечение, реализующее алгоритмы сортировки, указанные в варианте;

— Оценить трудоемкости сортировок;

— Провести анализ затрат работы программы по времени, выяснить влияющие на них характристики.

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов сортировки пирамидальная, блинная, перемешиванием.

1.1 Пирамидальная сортировка

Пирамида (англ. binary heap) определяется как структура данных, представляющая собой объект-массив, который можно рассматривать как почти полное бинарное дерево. Каждый узел этого дерева соответствует определенному элементу массива. На всех уровнях, кроме, может быть, последнего, дерево полностью заполнено (заполненным считается уровень, который содержит максимально возможное количество узлов). Последний уровень заполняется слева направо до тех пор, пока в массиве не закончатся элементы.[1]

В пирамиде, представленной на рисунке 1.1, число в окружности, представляющей каждый узел дерева, является значением, сортируемым в данном узле. Число над узлом — это соответствующий индекс массива. Линии, попарно соединяющие элементы массива, обозначают взаимосвязь вида “родитель-потомок”. Родительские элементы всегда расположены слева от дочерних. Данное дерево имеет высоту, равную 3; узел с индексом 4 (и значением 8) расположен на первом уровне.

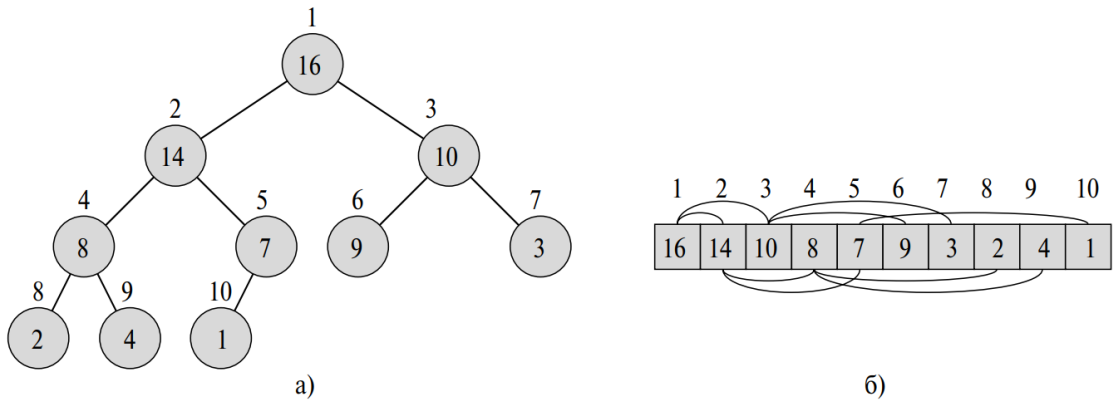

Рисунок 1.1 – Пирамида, представленная в виде а) бинарного дерева и б) массива

Обозначим, что:

1) Родительский элемент в массиве будет находится под индексом – $\frac{i}{2}$;

2) Правый потомок в массиве будет находится под индексом – $2 \cdot i$;

3) Левый потомок в массиве будет находится под индексом – $2 \cdot i + 1$.

Различают два вида бинарных пирамид: неубывающие и невозрастающие. В пирамидах обоих видов значения, расположенные в узлах, удовлетворяют свойству пирамиды (heap property), являющемуся отличительной чертой пирамиды того или иного вида. Свойство невозрастающих пирамид (max-heap property) заключается в том, что для каждого отличного от корневого узла с индексом i выполняется следующее неравенство

$$A[i/2] \geq A[i] \tag{1.1}$$

Другими словами, значение узла не превышает значение родительского по отношению к нему узла. Таким образом, в невозрастающей пирамиде самый большой элемент находится в корне дерева, а значения узлов поддерева, берущего начало в каком-то элементе, не превышают значения самого этого элемента. Принцип организации неубывающей пирамиды (min-heap) прямо противоположный. Свойство неубывающих пирамид (min-heap property) заключается в том, что для всех отличных от корневого узлов с индексом i выполняется такое неравенство:

$$A[i/2] \leq A[i] \tag{1.2}$$

Таким образом, наименьший элемент такой пирамиды находится в ее корне.

1.2 Сортировка перемешиванием

Алгоритм сортировки перемешиванем является модификацией алгоритма сортировки пузырьком. В отличие от сортировки пузырьком, где происходит обход последовательности только в одном направлении, в

алгоритме сортировки перемешиванием после достижения одной из границ рабочей части последовательности (то есть той части, которая еще не отсортирована и в которой происходит смена элементов) меняет направление движения. При этом при движении в одном направлении алгоритм перемещает к границе рабочей области максимальный элемент, а в другом направлении – минимальный элемент. Границы рабочей части последовательности устанавливаются в месте последнего обмена.[2]

1.3 Блинная сортировка

Блинная сортировка (pancake sort) – алгоритм сортировки массива, в котором сортировка осуществляется переворотом части массива.[3]

В этом алгоритме, к массиву, позволено применять только одну операцию – переворот части массива. И в отличии от других методов сортировки, где пытаются уменьшить количество сравнений, в этом нужно минимизировать количество переворотов.

Идея алгоритма заключается в том, чтобы за каждый проход, переместить максимальный элемент в конец массива.

Вывод

В данном разделе были рассомтотерены алгоритмы пирамидальной, блинной сортировок и алгоритм сортировки перемешиванием.

2 Конструкторская часть

В данном разделе будут рассмотрены схемы алгоритмов сортировок (блинная, перемешиванием и пирамидальная), а также найдена их трудоемкость.

2.1 Разработка алгоритмов

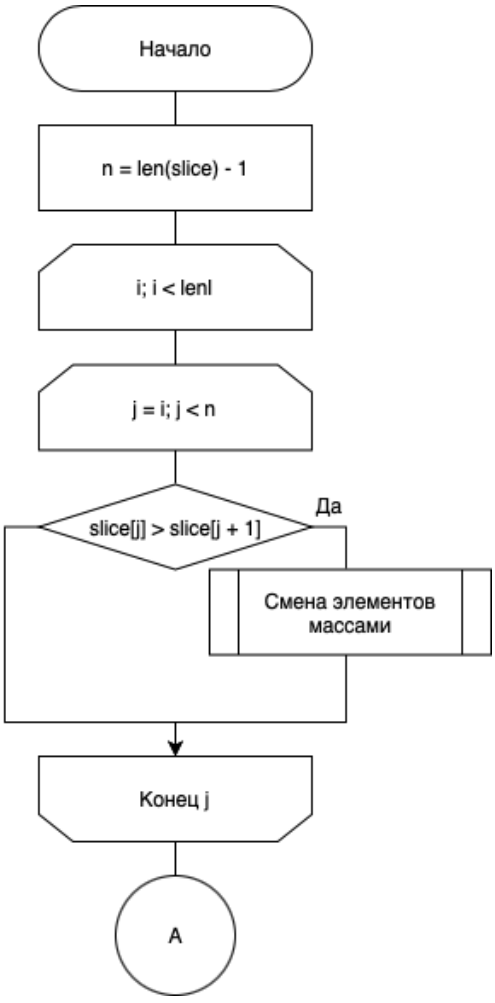

Рисунок 2.1 – Схема алгоритма сортировки Перемешиванем

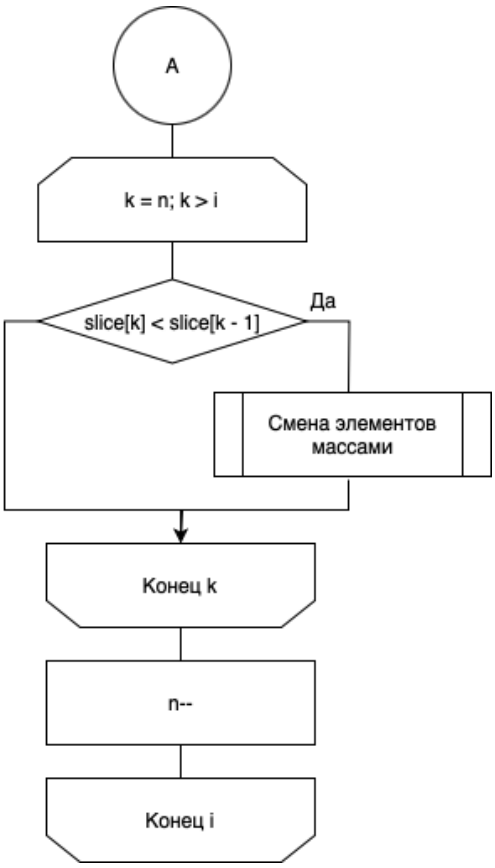

Рисунок 2.2 – Схема алгоритма сортировки Перемешиванем

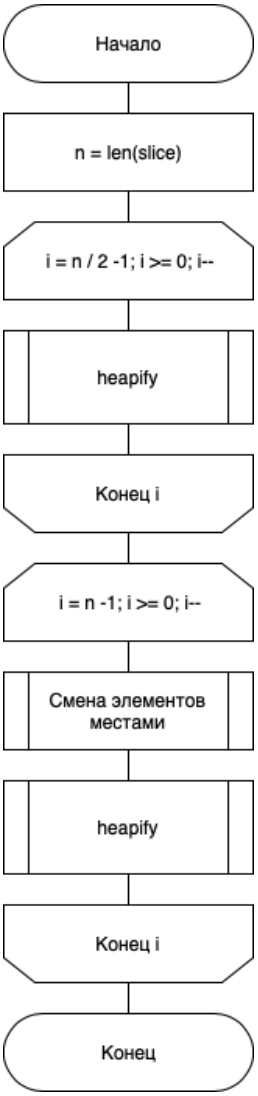

Рисунок 2.3 – Схема алгоритма Пирамидальной сортировки.

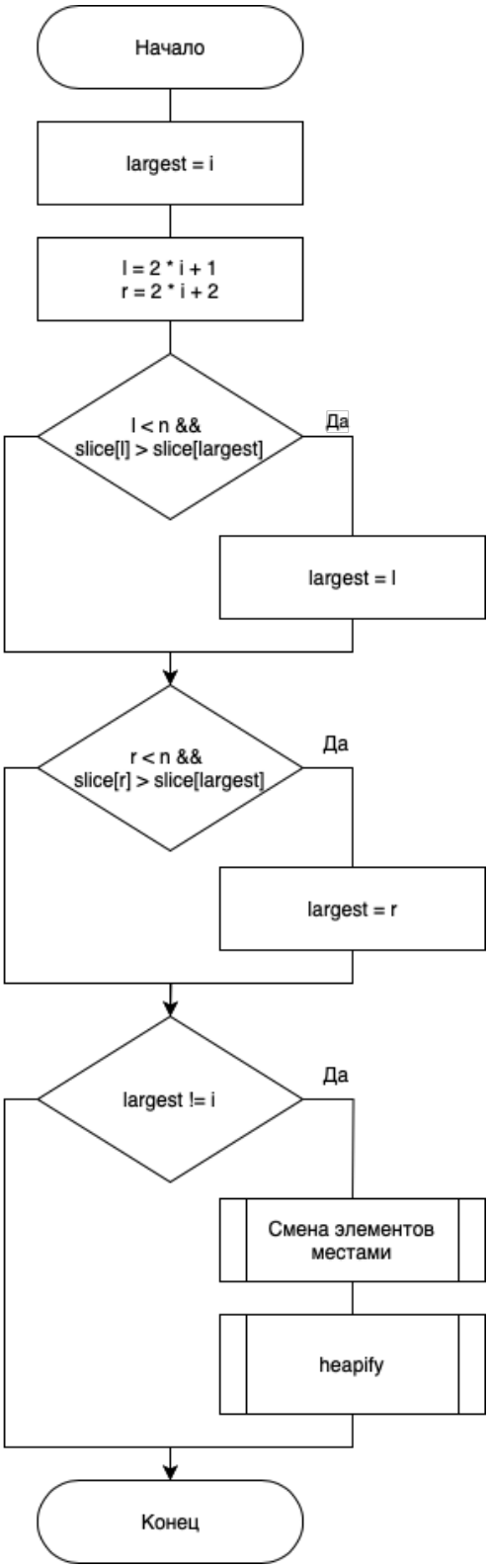

Рисунок 2.4 – Схема алгоритма Пирамидальной сортировки. Функция поиска максимального.

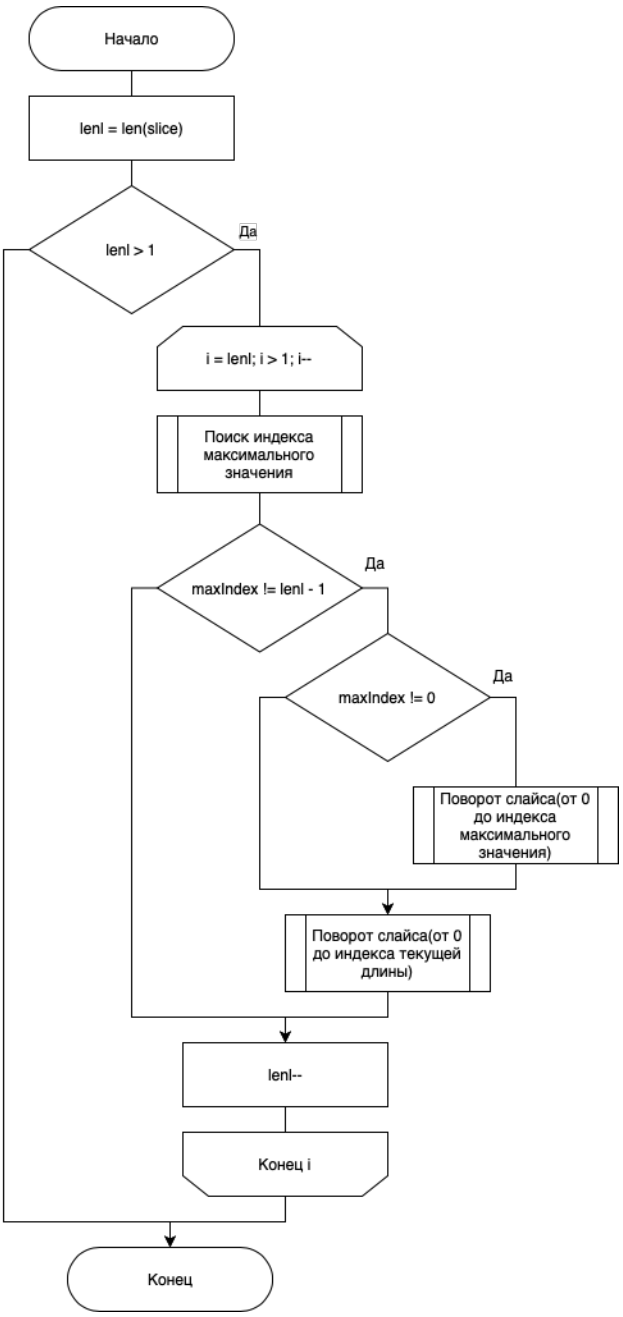

Рисунок 2.5 – Схема алгоритма Блинной сортировки.

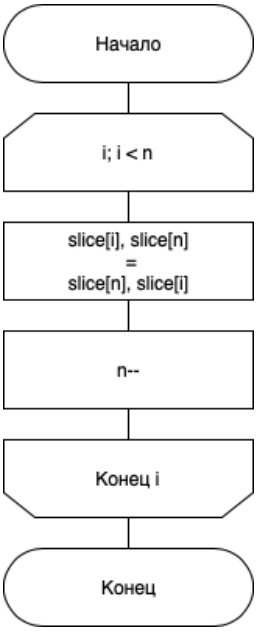

Рисунок 2.6 – Схема алгоритма Блинной сортировки. Функция разворота массива

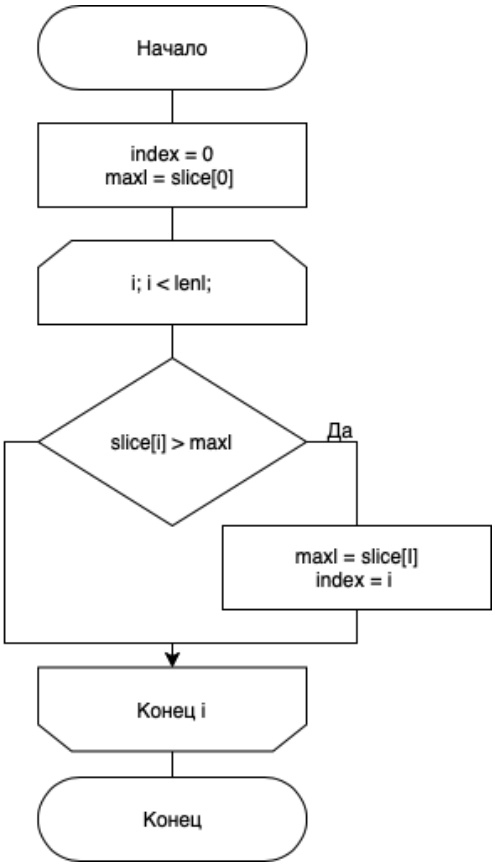

Рисунок 2.7 – Схема алгоритма Блинной сортировки. Функция поиска индекса максимального значения.

2.2 Оценка трудоемкости алгоритмов

В данном подразделе производится оценка трудоемкости каждого из алгоритмов.

2.2.1 Модель вычислений

Введем модель вычислений для оценки трудоемкости алгоритмов:

— операции из списка 2.1 имеют трудоемкость 2;

$$*,\ /,\ //,\ \%,\ *=,\ /=,\ //= \tag{2.1}$$

— операции из списка 2.2 имеют трудоемкость 1;

$$\begin{gathered} =,\ +,\ -,\ +=,\ -=,\ <,\ >,\ ==,\ !=, \\ >=,\ <=,\ [],\ <<,\ >>,\ ++,\ --, and, or \end{gathered} \tag{2.2}$$

— трудоемкость оператора выбора `if` условие `then` A `else` B рассчитывается по формуле 2.3:

$$f_{if} = f + \begin{cases} f_A, & \text{если условие выполняется;} \\ f_B, & \text{иначе} \end{cases} \tag{2.3}$$

— трудоемкость оператора цикла рассчитывется по формуле 2.4:

$$f_{\text{for}} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \tag{2.4}$$

— трудоемкость вызова функции равна 0.

2.2.2 Алгоритм сортировки перемешиванием

В лучшем случае алгоритму подается на вход упорядоченная последовательность. При таких входных данных первый внутренний цикл A осуществлет полный проход по последовательность в одну сторону и не находит обменов соседний элементов, поэтому на перой же итерации внешнего цикла правая граница становится равной левой границе, из-за чего второй внутренний цикл B не отрабатывает, произведя только ининциализацию и одну проверку условия, а внешний цикл завершает свою работу после первой итерации. Таким образом, в лучшем случае трудоемкость алгоритма сортировки перемешиваним равна (формула 2.5):

$$f = 4 + 1 + 2 + (N - 1)(2 + 4) + 1 + 2 + 1 + 1 = 6N + 6 = O(N) \quad (2.5)$$

В худшем случае алгоритму подается на вход упорядоченная в обратном порядке последовательность. Внешний цикл отрабатывает $\frac{N}{2}$ раз, так как за одну итерацию на свое место ставятся 2 элемента. Количество итераций каждого цикла зависит от четности размера подаваемой последовательнсти, если N — четно, то цикл A отработает $\frac{N-1+1}{2}$ итераций, а цикл B — $\frac{N-2+1}{2}$, если N — нечетно, то цикл A отработает $\frac{N-1+2}{2}$ итераций, а цикл B — $\frac{N-2}{2}$, однако и в том, и другом случае в общем два цикла отработают $\frac{2N-1}{2}$ итераций, а так как сложность тел обоих циклов одинаковая при вычислениях можно сразу пользоваться общим числом итераций. Таким образом, в худшем случае трудоемкость алгоритма сортировки перемешиванием равна (формула 2.6):

$$\begin{aligned} f = 4 + \frac{N}{2}(1 + 2 + 1 + 2 + 1 + \frac{2N - 1}{2}(2 + 4 + 9 + 1)) = \\ = 4 + \frac{N}{2}(16N - 1) = 8N^2 - \frac{N}{2} + 4 = O(N^2) \end{aligned} \quad (2.6)$$

2.2.3 Алгоритм пирамидальной сортировки

Трудоемкость в лучшем случае при отсортированном массиве по возрастанию выведена в формуле (2.7).

$$\begin{gathered} f_{best} = 3 + 4 + \frac{N}{2} \cdot (2 + 1 + f_{heapify\_best}) + \\ +3 + (N-1)(2 + 2 + f_{swap} + 1 + f_{heapify\_best}) \end{gathered} \tag{2.7}$$

Трудоемкость в худшем случае при отсортированном массиве по убыванию выведена в формуле (2.8).

$$\begin{gathered} f_{worst} = 3 + 4 + \frac{N}{2} \cdot (2 + 1 + f_{heapify\_worst}) + \\ +3 + (N-1)(2 + 2 + f_{swap} + 1 + f_{heapify\_worst}) \end{gathered} \tag{2.8}$$

Трудоемкость перестановки элементов будет равна, формула (2.9)

$$f_{swap} = 6 \tag{2.9}$$

Трудоемкость части программы сортировки пирамиды является функция heapify.

$$f_{heapify\_best} = 5 + logN \cdot (5+3+4+2+4) = 5 + 17 \cdot logN = O(logN) \tag{2.10}$$

$$f_{heapify\_worst} = 5 + logN \cdot (5+3+4+5+7) = 5 + 24 \cdot logN = O(logN) \tag{2.11}$$

Исходя из выведенных выше, то лучший и худший случаи будут равны:

$$\begin{gathered} f_{best} = 7 + \frac{N}{2} \cdot (3 + 5 + 17 \cdot logN) + 3 + (N-1) \cdot (5 + 6 + \\ +5 + 17 \cdot logN) = 23N + 24N \cdot logN - 13 \cdot logN - 3 = \\ = O(N \cdot logN) \end{gathered} \tag{2.12}$$

$$f_{worst} = 7 + \frac{N}{2} \cdot (3 + 5 + 24 \cdot logN) + 3 + (N - 1) \cdot (5 + 6 + \\ +5 + 24 \cdot logN) = 23N + 33N \cdot logN - 24 \cdot logN - 3 = \\ = O(N \cdot logN) \tag{2.13}$$

Таким образов из выведенных результатов в формулах (2.12) и (2.13) можно понять, что лучший и худший случай имеют трудоемкость $O(N \cdot logN)$.

2.2.4 Алгоритм блинной сортировки

Трудоемкость в случае, если элементы массива расположены в произвольном порядке.

$$f_{worst} = 1 + 1 + 2 + N(2 + f_{searchMaxIndex} + 2 + 1 + f_{flipSlice} * 2) \tag{2.14}$$

Трудоемкость в случае, если массив отсортирован по убыванию.

$$f_{middle} = 1 + 1 + 2 + N(2 + f_{searchMaxIndex} + 2 + 1 + f_{flipSlice}) \tag{2.15}$$

Трудоемкость в случае, если массив отсортирован по возрастанию.

$$f_{best} = 1 + 1 + 2 + N(2 + f_{searchMaxIndex} + 2) \tag{2.16}$$

Рассмотрим трудоемкость функции searchMaxIndex.

$$f_{searchMaxIndex} = 3 + 2 + N(2 + 2 + 2 + 1) = 5 + 7N \tag{2.17}$$

Рассмотрим трудоемкость функции flipSlice.

$$f_{flipSLice} = 2 + N(2 + 6 + 1) = 2 + 9N \tag{2.18}$$

Добавляя трудоемкости функций (2.17) и (2.18) в формулы (2.19), (2.20), (2.21) получим:

Трудоемкость в случае, если элементы массива расположены в произвольном порядке.

$$f_{worst} = 1 + 1 + 2 + N(2 + 5 + 7N + 2 + 1 + (2 + 9N) * 2) = \\ = 4 + N(14 + 25N) = 25N^2 + 14N + 4 = O(N^2) \tag{2.19}$$

Трудоемкость в случае, если массив отсортирован по убыванию.

$$f_{middle} = 1 + 1 + 2 + N(2 + 5 + 7N + 2 + 1 + 2 + 9N = \\ = 4 + N(12 + 16N) = 16N^2 + 12N + 4 = O(N^2) \tag{2.20}$$

Трудоемкость в случае, если массив отсортирован по возрастанию.

$$f_{best} = 1 + 1 + 2 + N(2 + 5 + 7N + 2) = \\ = 3 + N(9 + 7N) = 7N^2 + 9N + 3 = O(N^2) \tag{2.21}$$

2.3 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

— длина массива — целое число;

— массив — набор целых чисел;

2.4 Требования к ПО

Программа должна предоставлять следующие возможности:

— выбор режима работы: единичный эксперимент и замер времени работы алгоритмов сортировки;

— в режиме единичного эксперимента ввод размеров и содержимого сортируемого массив;

— в режиме замера времени происходит замер времени при различных размерах матриц и различных видах отсортированности(по убыванию, по возрастанию, рандомно).

Вывод

В данном разделе были разработаны алгоритмы сортировки блинная, перемешиванием и пирамидальная, также была произведена оценка трудоемкостей алгоритмов.

3 Технологическая часть

В данном разделе описаны требования к программному обеспечению, средства реализации, приведены листинги кода и данные, на которых будет проводиться тестирование.

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык GO[4]. Данный выбор обусловлен тем, что данный язык является типизированным, имеет инструменты для тестирования, профилирования и форматирования кода.

Замеры времени проводились при помощи getThreadCpuTimeNs функции, написанной на C, подключенной с помощью CGO.[5]

3.2 Реализация алгоритмов

В данном подразделе представлены листинги кода ранее описанных алгоритмов:

— алгоритм пирамидальной сортировки (листинг 3.1);

— алгоритм блинной сортировки (листинги 3.2);

— алгоритм сортировки перемешивания (листинги 3.3).

Листинг 3.1 – Реализация алгоритма пирамидальной сортировки.

```
1  func HeapSort(slice []int) {
2      n := len(slice)
3
4      for i := n / 2 - 1; i >= 0; i-- {
5          heapify(slice, n, i)
6      }
7
8      for i := n - 1; i >= 0; i-- {
9          slice[0], slice[i] = slice[i], slice[0]
10
11         heapify(slice, i, 0)
12     }
13 }
14
15 func heapify(slice []int, n int, i int) {
16     largest := i
17     l := 2 * i + 1
18     r := 2 * i + 2
19
20     if l < n && slice[l] > slice[largest] {
21         largest = l
22     }
23
24     if r < n && slice[r] > slice[largest] {
25         largest = r
26     }
27
28     if largest != i {
29         slice[i], slice[largest] = slice[largest], slice[i]
30
31         heapify(slice, n, largest)
32     }
33 }
```

Листинг 3.2 – Реализация алгоритма блинной сортировки.

```
1  func searchMaxIndex(slice []int, lenl int) int {
2      indx := 0
3      maxl := slice[0]
4
5      for i := 0; i < lenl; i++ {
6          if slice[i] > maxl {
7              maxl = slice[i]
8              indx = i
9          }
10     }
11
12     return indx
13 }
14
15 func flipSlice(slice []int, n int) {
16     for i := 0; i < n; i++ {
17         slice[i], slice[n] = slice[n], slice[i]
18         n--
19     }
20 }
21
22 func PancakeSort(slice []int) {
23     lenl := len(slice)
24
25     if lenl > 1 {
26         for i := lenl; i > 1; i-- {
27             maxIndex := searchMaxIndex(slice, lenl)
28
29             if maxIndex != lenl - 1 {
30                 if maxIndex != 0 {
31                     flipSlice(slice, maxIndex)
32                 }
33                 flipSlice(slice, lenl - 1)
34             }
35
36             lenl--
37         }
38     }
39 }
```

Листинг 3.3 – Реализация алгоритма сортировки перемешивания.

```
func ShakerSort(slice []int) {
    n := len(slice) - 1
    for i := 0; i < n + 1; i++ {
        for j := i; j < n; j++ {
            if slice[j] > slice[j + 1] {
                slice[j], slice[j + 1] = slice[j + 1], slice[j]
            }
        }

        for k := n; k > i; k-- {
            if slice[k] < slice[k - 1] {
                slice[k], slice[k - 1] = slice[k - 1], slice[k]
            }
        }

        n--
    }
}
```

3.3 Функциональные тесты

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

Таблица 3.1 – Функциональные тесты.

Входной массив	Ожидаемый результат	Результат
$[1, 2, 3, 4]$	$[1, 2, 3, 4]$	$[1, 2, 3, 4]$
$[5, 4, 3, 2, 1]$	$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$
$[3, 2, -5, 0, 1]$	$[-5, 0, 0, 2, 3]$	$[-5, 0, 0, 2, 3]$
$[1]$	$[1]$	$[1]$
$[]$	$[]$	$[]$

Алгоритмы, реализованные в данной лабораторной работе, функциональные тесты прошли успешно.

Вывод

Были разработаны схемы всех трех алгоритмов сортировки. Для дальнейшей проверки правильности работы программы были выделены классы эквивалентности тестов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени, представлены далее:

— Процессор – 2 Гц 4-ядерный процессор Intel Core i5;

— Оперативная память – 16 ГБайт;

— Операционная система – macOS Venura 13.5.2.

4.2 Пример работы программы

В данном подразделе представлен пример 4.1 работы программы:

Листинг 4.1 – Пример работы программы

```
ivanmamvriyskiy@MacBook-Pro-Ivan-2 main % go run main.go
Выберите пункт меню:
1)Одиночный случай;
2)Запуск тестов;
3)Выход из программы.
1

Введите размер слайса: 3

Введите 1 элемент слайса: 1
Введите 2 элемент слайса: 2
Введите 3 элемент слайса: 3

Пирамидальная сортировка: 1 2 3
Блинная сортировка: 1 2 3
Сортировка перемешиванием: 1 2 3
```

4.3 Время выполнения алгоритмов

В таблицах 4.1, 4.2, 4.3 представлены результаты замера времени работы алгоритмов в зависимости от отсортированности и длины среза. На рисунках 4.1, 4.2, 4.3 представлены графики зависимостей времени работы алгоритмов при разной отсортированности взависимости от длины среза.

Таблица 4.1 – Время выполнения работы алгоритмов при отсортированном по возрастанию срезе в зависимости от длины среза(нс).

Длина среза	Блинная	Пирамидальная	Перемешиванием
10	400	700	400
100	14 000	6 300	9 100
200	52 000	13 700	32 100
300	114 500	21 800	66 800
400	192 500	26 900	97 100
500	261 700	34 800	147 700
600	331 000	44 300	201 100
700	394 700	44 800	221 400
800	445 000	47 900	251 800
900	472 800	46 300	269 100
1000	575 300	52 700	322 700
2000	2 347 800	120 600	1 292 500
3000	5 260 300	192 300	2 858 500
4000	9 207 500	277 400	5 126 600
5000	14 667 500	340 100	8 006 700

Таблица 4.2 – Время выполнения работы алгоритмов при отсортированном по убыванию срезе в зависимости от длины среза(нс).

Длина среза	Блинная	Пирамидальная	Перемешиванием
10	100	300	100
100	6 500	43 00	4 000
200	25 200	6 400	21 400
300	65 200	15 800	45 800
400	104 600	22 800	75 500
500	171 600	40 800	106 700
600	237 100	26 300	184 600
700	299 200	51 500	159 400
800	381 100	54 100	276 600
900	486 100	72 500	313 500
1000	566 900	69 900	336 100
2000	2 453 600	127 300	1 342 100
3000	5 321 100	248 500	2 926 000
4000	9 185 700	264 500	5 037 100
5000	15 129 900	350 900	8 189 500

Таблица 4.3 – Время выполнения работы алгоритмов при рандомном расположении чисел в срезе в зависимости от длины среза(нс).

Длина среза	Блинная	Пирамидальная	Перемешиванием
10	500	100	400
100	11 500	700	4 900
200	42 300	1 900	22 700
300	99 000	2 500	44 400
400	131 500	2 200	65 200
500	243 800	3 100	114 400
600	354 300	5 100	140 700
700	387 500	3 500	159 000
800	515 500	5 600	232 200
900	678 600	4 600	283 100
1000	808 800	5 800	331 800
2000	3 413 700	10 200	1 331 300
3000	6 737 100	14 800	2 869 200
4000	11 649 900	19 600	5 008 700
5000	19 142 300	36 300	8 176 300

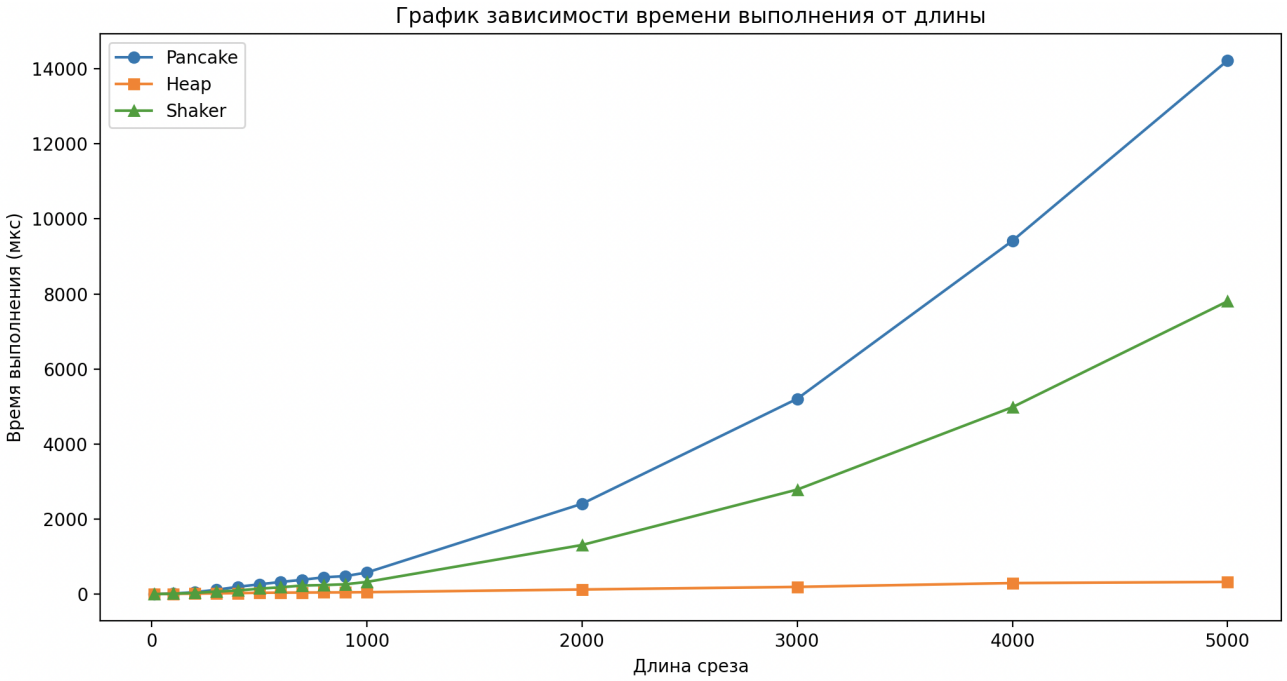

Рисунок 4.1 – Время выполнения работы алгоритма при отсортированном по возрастанию срезе в зависимости от длины среза.

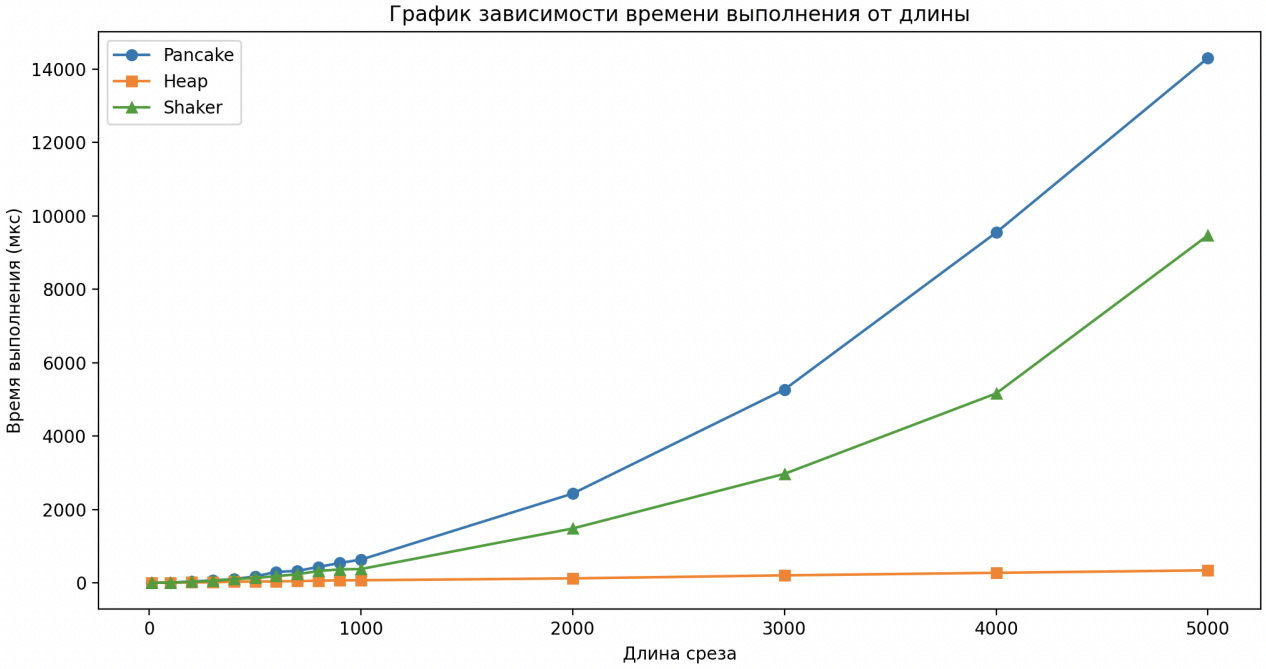

Рисунок 4.2 – Время выполнения работы алгоритма при отсортированном по убыванию срезе в зависимости от длины среза.

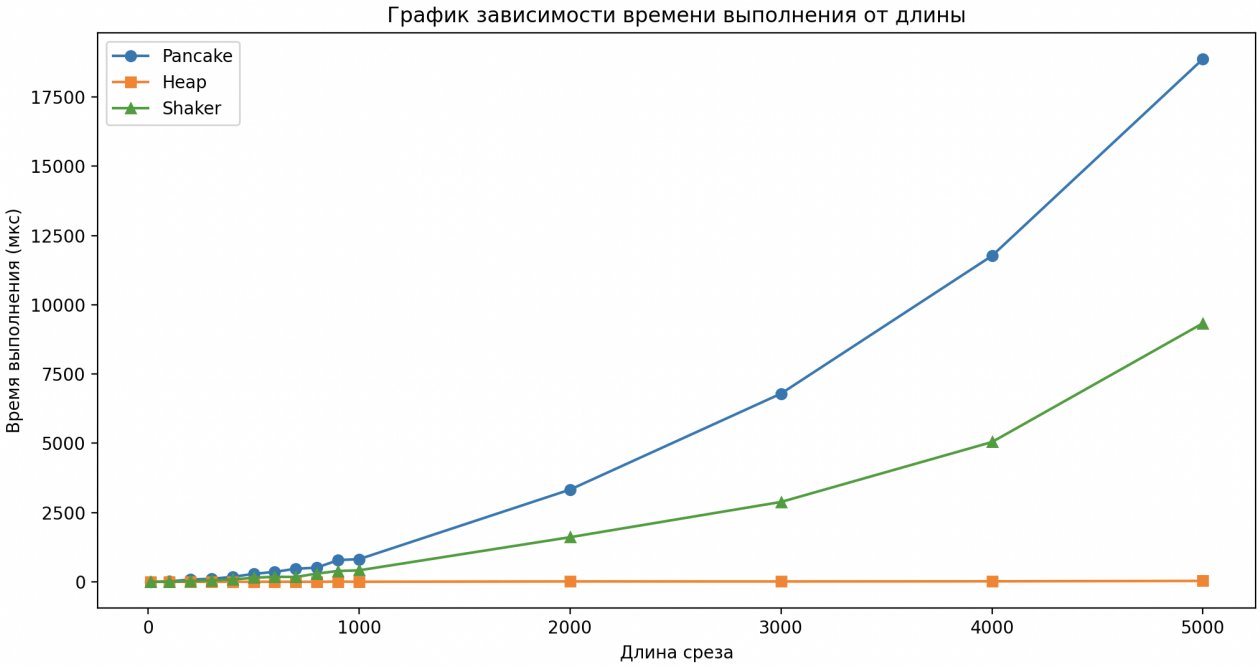

Рисунок 4.3 – Время выполнения работы алгоритма при рандомном расположении чисел в срезе в зависимости от длины среза.

По результатам эксперимента можно сделать следующие выводы:

— Алгоритм блинной сортировки работает медленнее всех при любых способах расположения элементов в последовательности, а пирамидальный алгоритм сортировки – быстрее всех.

— Любой из алгоритмов быстрее всего работает при отсортированной по возрастанию последовательности.

— Блинная сортировка быстрее всего работает при отсортированной последовательности по возрастанию, так как не происходит двойного поворота при нахождении индекса максимального числа не равного текущей длине.

— Пирамидальная сортировка при любом расположении элементов происходит примерно одинаково. Выигрыш по времени происходит в случаях, когда максимальное значение находится в родительской ячейке, так как не происходит обмена.

— Сортировка перемешиванием быстрее выполнятеся, когда последовательность отсортирована по возрастанию, так как не происходит перетаскивания максимальных и минимальных значений. Дольше всего выполнеятся, когда последовательность отсортирована по убыванию, так как на каждом шаге происходит протаскивание максимальных и минимальных значений.

4.4 Вывод

Таким образом, для более быстрой сортировки последовательности независимо от расположения в ней элементов необходимо использовать пирамидальную сортировку.

Заключение

В ходе исследования был проведен сравнительный анализ алгоритмов, в результате которого было выяснено, что пирамидальная сортировка при рандомном расположении элементов в срезе работает быстрее сортировки перемешиванием примерно в 4 – 30 раз при длине среза от 10 – 600 и в 53 – 225 раз быстрее при длине среза 700 – 1000. Данное превосходство резко уменьшается при отсортированной последовательности, выигрыш по времени при таком расположении элементов в 3 – 30 раз. Если сравнивать блинную сортировку и сортировку перемешиванием, то вторая работает быстрее примерно в 1 – 3 раза независимо от расположения элементов.

Из результатов оценки трудоемкости следует, что пирамидальная сортировка имеет наименьшую сложность, которая составляет $O(N \cdot logN)$, далее следует сортировка перемешиванием, которая при лучшем случае имеет сложность O(N), а при худшем случае O(N^2). Наибольшую трудоемкость имеет блинная сортировка, у которой сложность O(N^2). Лучший случай данной сортировки имеет примерно такую же трудоемксоть, как худший случай сортировки перемешиванием, а худший случай примерно в 4 раза имеет большую трудоемкость, чем худший случай сортировки перемешиванием.

Исходя из данных показателей пирамидальная сортировка работает быстрее всех, а блинная дольше всех независимо от расположения элементов в последовательности.

Цель данной лабораторной работы, которая заключается в исследовании алгоритмов сортировок, выполнена. Также были выполнены следующие задачи:

— описаны алгоритмы пирамидальной, блинной сортировки и сортировки перемешиванием;

— создано программное обеспечение, реализующее алгоритмы данных сортировок;

— произведена оценка трудоемкости каждого из алгоритмов;

— по экспериментальным данным сделаны выводы об эффективности по времени каждого из реализованных алгоритмов, которые

были подтверждены теоретическими расчетами трудоемкости алгоритмов.

Список литературы

1. Д.В. Шагбазян А.А. Штанюк Е.В. Малкина. Алгоритмы сортировки. Анализ, реализация, применение: учебное пособие. — Нижний Новгород: Нижегородский государственный университет, 2019. С. 22–25.

2. Шейкерная сортировка (перемешиванием) [Электронный ресурс]. Режим доступа: `https://kvodo.ru/shaker-sort.html` (дата обращения: 15.10.2023).

3. Блинная сортировка [Электронный ресурс]. Режим доступа: `https://programm.top/c-sharp/algorithm/array-sort/pancake-sort/` (дата обращения: 15.10.2023).

4. Документация Go [Электронный ресурс]. Режим доступа: `https://go.dev` (дата обращения: 20.09.2023).

5. Документация CGO [Электронный ресурс]. Режим доступа: `https://pkg.go.dev/cmd/cgo` (дата обращения: 15.10.2023).