МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМ. Н. Э. БАУМАНА

УДК _____
№ госрегистрации _____
Инв. №_____

УТВЕРЖДАЮ

Преподаватель

_____

«_____» _____ 2020 г.

ДИСЦИПЛИНА АНАЛИЗ АЛГОРИТМОВ

ОТЧЁТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №3

Алгоритмы сортировки
(промежуточный)

Студент _____ Ф.М. Набиев

Преподаватели Л.Л. Волкова, Ю.В. Строганов

Москва, 2020

СОДЕРЖАНИЕ

Введение 4

1 Аналитический раздел 5

1.1 Описание алгоритмов 5

1.1.1 Быстрая сортировка 5

1.1.2 Сортировка вставками 6

1.1.3 Шейкерная сортировка 6

1.2 Вывод 7

2 Конструкторский раздел 8

2.1 Функциональная модель 8

2.2 Разработка алгоритмов 8

2.3 Трудоемкость алгоритмов 12

2.3.1 Быстрая сортировка 12

2.3.2 Сортировка вставками 14

2.3.3 Шейкерная сортировка 14

2.4 Вывод 14

3 Технологический раздел 15

3.1 Требования к программному обеспечению 15

3.2 Средства реализации 15

3.3 Листинги кода 16

3.4 Примеры работы 18

3.5 Описание тестирования 19

3.6 Вывод 21

4 Исследовательский раздел 22

4.1 Эксперименты по замеру времени 22

4.2 Вывод 24

Заключение 25

Список использованных источников 26

ВВЕДЕНИЕ

В настоящее время необходимо сортировать большие объемы данных. Для этой цели существуют алгоритмы сортировки, которые упорядочивают элементы в списке [1]. Алгоритмы сортировки могут использоваться для ускорения поиска элементов среди большого количества информации, например для баз данных или математических программ. Также сортировки могут быть полезны в сферах бизнеса и бухгалтерии.

Целью данной лабораторной работы является сравнительное изучение трёх существующих алгоритма сортировки.

Обозначим задачи:

— реализовать три различных алгоритма сортировки;

— теоретически вычислить эффективность алгоритмов;

— сравнить алгоритмы по времени.

1 Аналитический раздел

В данном разделе будет определена теоретическая база, необходимая для реализации поставленных задач.

1.1 Описание алгоритмов

Сортировка - это процесс упорядочения некоторого множества элементов, на котором определены отношения порядка $>$, $<$, $>=$, $<=$ (по возрастанию или убыванию) [2]. При выборе алгоритма сортировки необходимо выбрать тот алгоритм, который будет проделывать минимум операций над данными и тем самым максимально быстро получать необходимый результат - отсортированный список.

Трудоемкость алгоритма - это зависимость количества операций от количества данных, с которыми алгоритм работает. Список действий, цена которых считается за 1:

$$+, -, *, /, \%, =, ==, !=, <, >, <=, >=, [], +=$$

За последние 70 лет появилось множество алгоритмов сортировок для компьютера[1].

1.1.1 Быстрая сортировка

Описание алгоритма:

а) выбирается элемент, называемый опорным;

б) остальные элементы сравниваются с опорным, на основании сравнения меньшие опорного перемещаются левее него, а большие или равные - правее;

в) рекурсивно упорядочиваются подмассивы, лежащие слева и справа от опорного элемента.

Задачи о выборе опорного элемента и разбиении массива на подмассивы относительно опорного элемента могут быть решены разными способами, а

эффективность их решения напрямую влияет на эффективность сортировки. В данной лабораторной работе будет использоваться разбиение Ломуто, а в качестве опорного элемента будет выбираться последний элемент.

1.1.2 Сортировка вставками

Описание алгоритма:

а) выбирается один из элементов входных данных;

б) выбранный элемент вставляется на нужную позицию в уже отсортированной последовательности;

в) п 1,2 выполняются, пока набор входных данных не будет исчерпан.

1.1.3 Шейкерная сортировка

Данный алгоритм является разновидностью сортировки простыми обменами, суть которой заключается в повторяющихся до полной сортированности проходах по по сортируемому массиву, за каждый из которых элементы попарно сравниваются и меняются местами, если их порядок неверный.

Сортировка простыми обменами обладает следующими особенностями:

— если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, её можно исключить из рассмотрения;

— при движении от конца массива к началу элемент минимальной ценности переходит на первую позицию, а наиболее ценный элемент сдвигается только на одну позицию вправо.

Алгоритм шейкреной сортировки предлагает учитывать вышеописанные особенности и вносит следующие изменения:

— границы рабочей части массива устанавливаются в месте последнего обмена на каждой итерации;

— массив просматривается поочередно справа налево и слева направо.

1.2 Вывод

В данной работе стоит задача реализации 3 алгоритмов сортировки: быстрая сортировка, сортировка вставками и шейкерная сортировка. Необходимо теоретически оценить трудоемкость этих алгоритмов и проверить все вычисления экспериментально.

2 Конструкторский раздел

В данном разделе будет проведена конкретизация поставленных задач, составлены и проанализированы алгоритмы.

2.1 Функциональная модель

На рисунке 2.1 представлена функциональная модель IDEF0 уровня 1.

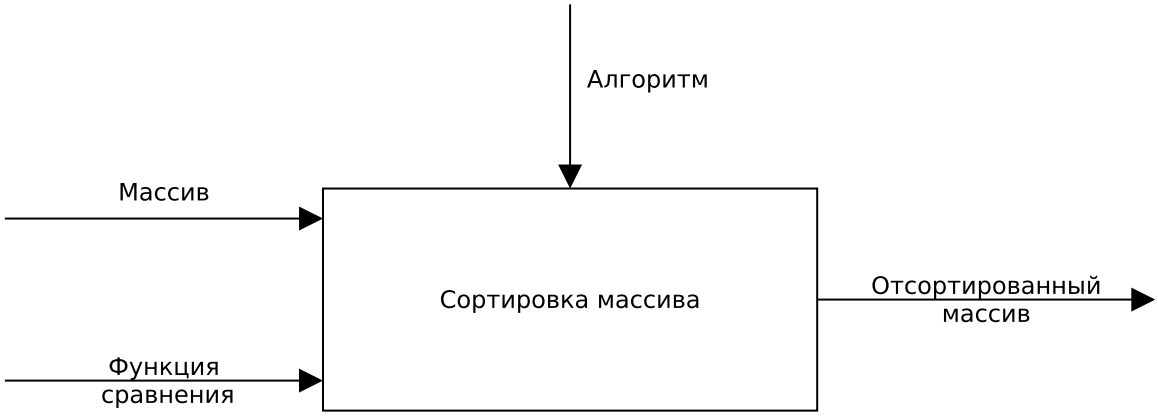

Рисунок 2.1 — Функциональная IDEF0 модель уровня 1

2.2 Разработка алгоритмов

Для непосредственной реализации вышеописанных алгоритмов важно иметь их некоторые упрощённые визуальные представления, так как чтение таких представлений упрощает написание кода. Подходящим для этого вариантом визуализации являются схемы алгоритмов.

На рисунках 2.2, 2.3, 2.4 изображены схемы алгоритмов быстрой сортировки, сортировки вставками и шейкерной сортировки.

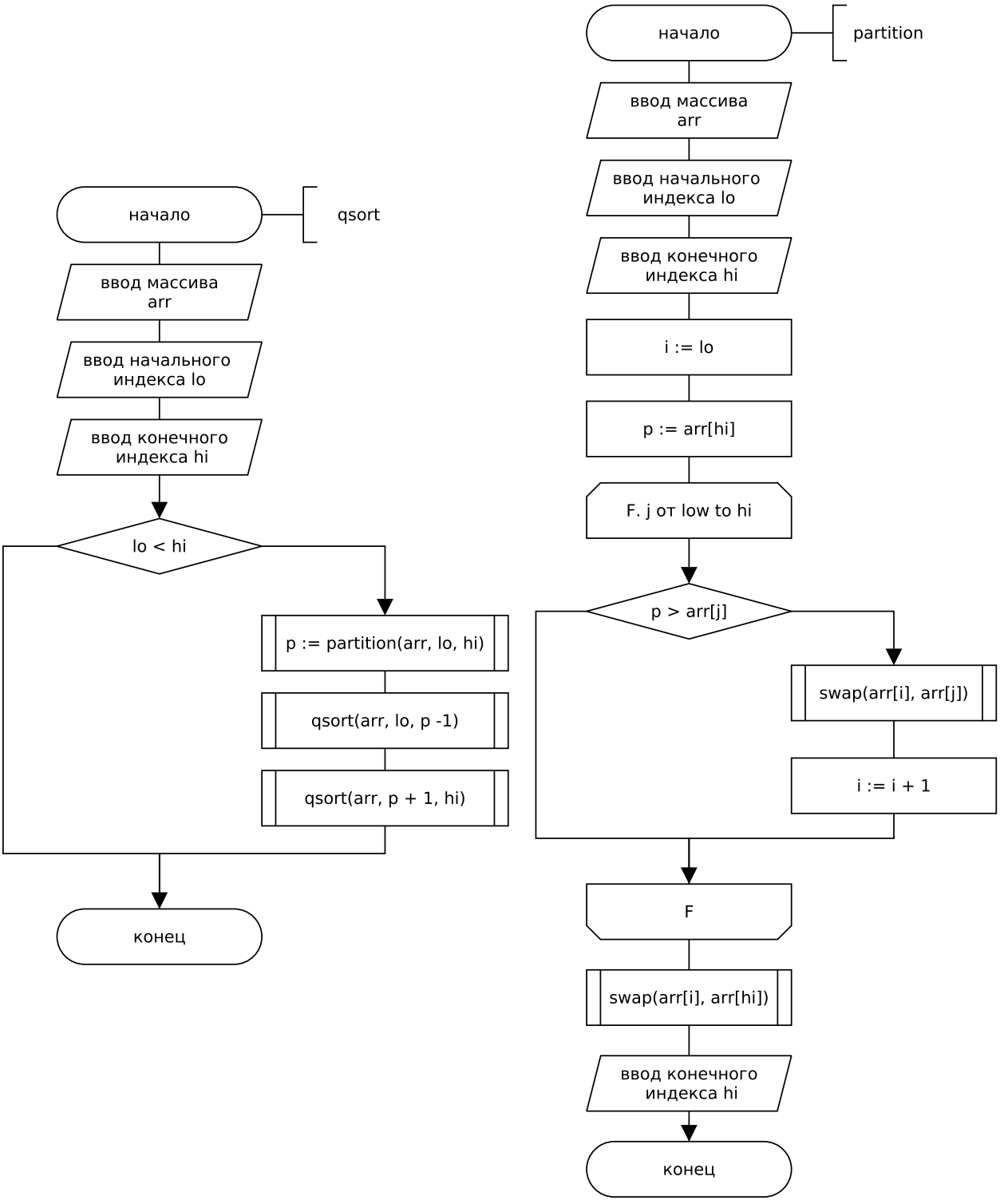

Рисунок 2.2 — Быстрая сортровка

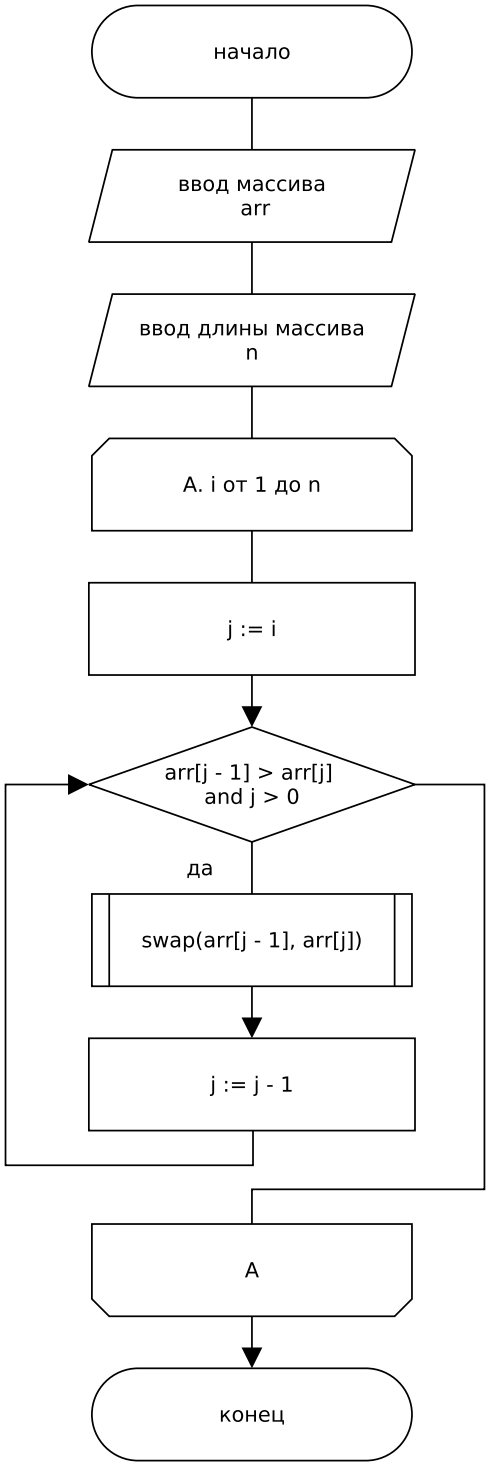

Рисунок 2.3 — Сортировка вставками

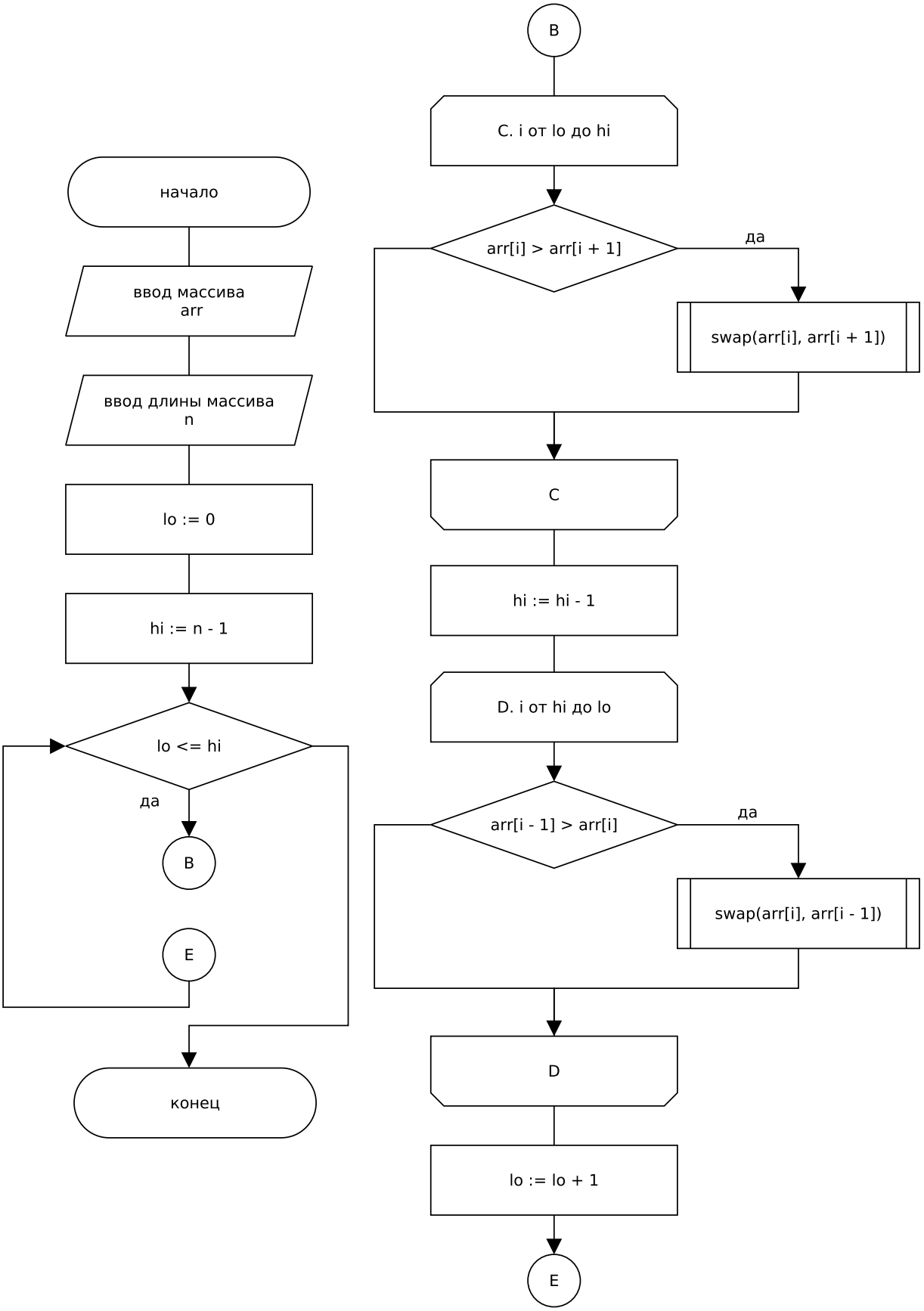

Рисунок 2.4 — Шейкерная сортировка

2.3 Трудоемкость алгоритмов

Проведём оценку сложности рассматриваемых алгоритмов сортировки.

2.3.1 Быстрая сортировка

Быстрая сортировка является рекурсивным алгоритмом, значит при подсчёте стоимости производимых операций будет получено рекуррентное отношение. Для упрощения задачи, подсчитаем асимптотическую сложность лучшего и худшего случаев.

Запишем рекуррентное отношение, соответствующее быстрой сортировке:

$$T_{qs}(n) = T_{qs}(m) + T_{qs}(s) + f_{partition}(n) \tag{2.1}$$

где n - размер входного массива, m - размер левого подмассива, s - размер правого подмассива, $f_{partition}(n)$ - объём работы, который необходимо совершить, чтобы произвести разбиение массива длины n.

Так как мы вычисляем асимптотическую сложность всего алгоритма, то вычислять точную трудоёмкость функции разбиения, его составляющей, не имеет смысла. Разбиение Ломуто заключается в единственном линейном обходе входного массива, а выбор опорного элемента имеет постоянную сложность, потому что это всегда последний элемент. Таким образом имеем сложность функции разбиения $O(n)$, другими словами:

$$f_{partition}(n) = n \tag{2.2}$$

Лучшим случаем для быстрой сортировки будет тот, при котором глубина дерева рекурсивных вызовов будет минимальна. Это условие будет достигнуто, если на каждом этапе рекурсии входной массив будет разбиваться на два равных (или практически равных) по длине подмассива. Тогда имеем следующие рекуррентное отношение:

$$T_{qs}(n) = T_{qs}(\frac{n}{2}) + T_{qs}(\frac{n}{2}) + n = 2 \cdot T_{qs}(\frac{n}{2}) + n \tag{2.3}$$

Что напоминает вид основной теоремы о рекуррентных соотношениях:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n) \tag{2.4}$$

где $a = 2$, $b = 2$, $f(n) = n$. Это означает, что применив эту теорему мы получим искомую трудоёмкость. Очевидно, первый варианты основной теоремы рекуррентных соотношений не подходит, так как:

$$f(n) = n = n^1 = n^c \Rightarrow c = 1 = log_2(2) = log_b(a) \Rightarrow c \not< log_b(a) \tag{2.5}$$

Проверим второй вариант:

$$f(n) = n = n^1 = n^1 \cdot log^0(n) = n^c \cdot log^k(n) \tag{2.6}$$

где $c = 1 = log_b(a)$, $k = 0$, а значит условия второго варианта основной теоремы соблюдены. Таким образом имеем асимптотическую сложность быстрой сортировки с разбиением Ломуто в лучшем случае:

$$T_{qs} = O(n \cdot log(n)) \tag{2.7}$$

Рассмотрим худший случай. Условия его наступления полностью противоположны условиям лучшего случая: глубина дерева рекурсивных вызовов должна быть максимальна. Это означает, что в ходе первого разбиения будут получены подмассивы длин 1 и $n - 1$, а на всех последующих этапах рекурсии длина большего подмассива будет уменьшаться только на 1, а общее число операций разбиения составит $n - 1$. Тогда можем подсчитать трудоёмкость:

$$\sum_{i=0}^{n-1}(n - i) \approx O(n^2) \tag{2.8}$$

Таким образом сортировка полностью отсортированного массива или обратно отсортированного массива является худшим случаем для быстрой сортировки с выбором последнего элемента в качестве опорного, а её сложность равна $O(n^2)$.

Сложность выполнения быстрой сортировки в среднем случае имеет вероятностный характер, так как в произвольном массиве отношение порядка, в котором находятся соседние элементы, случайно. Имеем трудоёмкость[1]:

$$O(n \cdot log(n)) \tag{2.9}$$

2.3.2 Сортировка вставками

Изображённый на рисунке 2.3 алгоритм сортировки вставками состоит из двух вложенных циклов. В лучшем случае внутренний цикл вырождается, а в худшем - всегда выполняется.

Трудоёмкость лучшего случая:

$$2 + (n-1) \cdot (4+5) = 2 + (n-1) \cdot 9 = 9n - 7 \tag{2.10}$$

Тут n - это длина сортируемого массива.

Трудоёмкость худшего случая:

$$2 + (n-1) \cdot (4 + (n-1) \cdot (9+3)) = 11n^2 - 18n + 9 \tag{2.11}$$

Значит, асимптотическая сложность для лучшего случая - $O(n)$, для худшего - $O(n^2)$, а для среднего - $O(n^2)$[[1].

2.3.3 Шейкерная сортировка

Шейкерная сортировка в любых случаях имеет асимптотическую сложность $O(n^2)$[1].

2.4 Вывод

Быстрая сортировка имеет лучшую среднюю сложность, что говорит о ней, как о потенциально наиболее универсальной сортировке. Шейкреная сортировка имеет наихудшую трудоёмкость во всех случаях, а значит его применение нецелесообразно в принципе.

3 Технологический раздел

В данном разделе будут составлены требования к программному обеспечению, выбраны средства реализации и определены тестовые данные.

3.1 Требования к программному обеспечению

Требования к вводу:

— размер массива;

— элементы массива, разделенные произвольным количеством пробельных символов.

Требования к выводу:

— отсортированный массив.

Требования к программе:

— выбор алгоритма происходит через аргументы командной строки путём передачи его номера:

1) быстрая сортировка;

2) сортировка вставками;

3) шейкерная сортировка.

3.2 Средства реализации

Для реализации программы вычисления редакционного расстояния мной был выбран язык программирования C++. В рамках текущей задачи данный язык программирования имеет ряд существенных преимуществ:

— Статическая типизация;

— Близость к низкоуровневому C при наличии многих возможностейвысоко уровненных языков;

— Встроенная библиотека std::chrono, позволяющая измерять процессорное время [3].

3.3 Листинги кода

В листингах 3.2, 3.1, 3.3 приведены текста функций быстрой сортировки, сортировки вставками и шейкерной сортировки соответственно.

Листинг 3.1 — Сортировка вставками

```
1  #define swp(val1, val2) { auto tmp = val1; val1 = val2; val2 = tmp; }
2
3  template <typename Type>
4  void insertionsort(Type *arr, int len, bool (*comp)(Type, Type))
5  {
6      for (int i = 1; i < len; i++)
7      {
8          for (int j = i; j > 0 && comp(arr[j - 1], arr[j]); j--)
9          {
10             swp(arr[j], arr[j - 1]);
11         }
12     }
13 }
14
15 #undef swp
```

Листинг 3.2 — Бастрая сортировка

```
1  #define swp(val1, val2) { auto tmp = val1; val1 = val2; val2 = tmp; }
2
3  template <typename Type>
4  static int parition(Type *arr, int low, int high, bool (*comp)(Type, Type))
5  {
6      int i = low;
7      Type pivot = arr[high];
8
9      for (int j = low; j < high; j++)
10     {
11         if (comp(pivot, arr[j]))
12         {
13             swp(arr[i], arr[j]);
14             i++;
15         }
16     }
17
18     swp(arr[i], arr[high]);
19
20     return i;
21 }
22
23 template <typename Type>
24 static void qsrec(Type *arr, int low, int high, bool (*comp)(Type, Type))
25 {
26     if (low < high)
27     {
28         int p = parition(arr, low, high, comp);
29
30         qsrec(arr, low, p - 1, comp);
31         qsrec(arr, p + 1, high, comp);
32     }
33 }
34
35 template <typename Type>
36 void quicksort(Type *arr, int len, bool (*comp)(Type, Type))
37 {
38     qsrec(arr, 0, len - 1, comp);
39 }
40
41 #undef swp
```

Листинг 3.3 — Шейкреная сортировка

```
1  #define swp(val1, val2) { auto tmp = val1; val1 = val2; val2 = tmp; }
2
3  template <typename Type>
4  void shakersort(Type *arr, int len, bool (*comp)(Type, Type))
5  {
6      int low = 0;
7      int high = len - 1;
8
9      while (low <= high)
10     {
11         for (int idx = low; idx < high; idx++)
12         {
13             if (comp(arr[idx], arr[idx + 1]))
14             {
15                 swp(arr[idx], arr[idx + 1]);
16             }
17         }
18         high--;
19
20         for (int idx = high; idx > low; idx--)
21         {
22             if (comp(arr[idx - 1], arr[idx]))
23             {
24                 swp(arr[idx], arr[idx - 1]);
25             }
26         }
27         low++;
28     }
29 }
30
31 #undef swp
```

3.4 Примеры работы

На рисунках 3.1-3.7 приведены примеры работы реализованной программы.

```
 # ./fn_aa_lab_03 1
5
1
```

```
 # ./fn_aa_lab_03 2
5
1
```

```
 # ./fn_aa_lab_03 3
5
1
```

Рисунок 3.1 — Преждевременное завершение ввода

```
# ./fn_aa_lab_03 1
```

```
# ./fn_aa_lab_03 2
```

```
# ./fn_aa_lab_03 3
```

Рисунок 3.2 — Пустой ввод

```
# ./fn_aa_lab_03 1
4.2
```

```
# ./fn_aa_lab_03 2
4.2
```

```
# ./fn_aa_lab_03 3
4.2
```

Рисунок 3.3 — Некорректный ввод

```
# ./fn_aa_lab_03 1
5
1 5 3 4 2

1 2 3 4 5
```

```
# ./fn_aa_lab_03 2
5
1 5 3 4 2

1 2 3 4 5
```

```
# ./fn_aa_lab_03 3
5
1 5 3 4 2

1 2 3 4 5
```

Рисунок 3.4 — Массив произвольной степени сортированности

```
# ./fn_aa_lab_03 1
5
1 2 3 4 5

1 2 3 4 5
```

```
# ./fn_aa_lab_03 2
5
1 2 3 4 5

1 2 3 4 5
```

```
# ./fn_aa_lab_03 3
5
1 2 3 4 5

1 2 3 4 5
```

Рисунок 3.5 — Отсортированный массив

```
# ./fn_aa_lab_03 1
5
5 4 3 2 1

1 2 3 4 5
```

```
# ./fn_aa_lab_03 2
5
5 4 3 2 1

1 2 3 4 5
```

```
# ./fn_aa_lab_03 3
5
5 4 3 2 1

1 2 3 4 5
```

Рисунок 3.6 — Обратно отсортированный массив

```
# ./fn_aa_lab_03 1
5
0 0 0 0 0

0 0 0 0 0
```

```
# ./fn_aa_lab_03 2
5
0 0 0 0 0

0 0 0 0 0
```

```
# ./fn_aa_lab_03 3
5
0 0 0 0 0

0 0 0 0 0
```

Рисунок 3.7 — Массив из одинаковых чисел

3.5 Описание тестирования

Для тестирования программы были подготовлены тестовые данные, указанные в таблице 3.1.

Таблица 3.1 — Тестовые данные

Входной массив	Ожидаемый результат
1, 2, 3, 4, 5, 6, 7, 8, 9	1, 2, 3, 4, 5, 6, 7, 8, 9
9, 8, 7, 6, 5, 4, 3, 2, 1	1, 2, 3, 4, 5, 6, 7, 8, 9
4, 2, 7, 4, 8, 2, 4, 8, 1	1, 2, 2, 4, 4, 4, 7, 8, 8
5, 5, 5, 5, 5, 5, 5, 5, 5	5, 5, 5, 5, 5, 5, 5, 5, 5

Результаты тестирования приведены в таблицах 3.2, 3.3, 3.4. Все тесты были успешно пройдены.

Таблица 3.2 — Результаты тестирования быстрой сортировки

Входной массив	Результат
1, 2, 3, 4, 5, 6, 7, 8, 9	1, 2, 3, 4, 5, 6, 7, 8, 9
9, 8, 7, 6, 5, 4, 3, 2, 1	1, 2, 3, 4, 5, 6, 7, 8, 9
4, 2, 7, 4, 8, 2, 4, 8, 1	1, 2, 2, 4, 4, 4, 7, 8, 8
5, 5, 5, 5, 5, 5, 5, 5, 5	5, 5, 5, 5, 5, 5, 5, 5, 5

Таблица 3.3 — Результаты тестирования сортировки вставками

Входной массив	Результат
1, 2, 3, 4, 5, 6, 7, 8, 9	1, 2, 3, 4, 5, 6, 7, 8, 9
9, 8, 7, 6, 5, 4, 3, 2, 1	1, 2, 3, 4, 5, 6, 7, 8, 9
4, 2, 7, 4, 8, 2, 4, 8, 1	1, 2, 2, 4, 4, 4, 7, 8, 8
5, 5, 5, 5, 5, 5, 5, 5, 5	5, 5, 5, 5, 5, 5, 5, 5, 5

Таблица 3.4 — Результаты тестирования шейкреной сортировки

Входной массив	Результат
1, 2, 3, 4, 5, 6, 7, 8, 9	1, 2, 3, 4, 5, 6, 7, 8, 9
9, 8, 7, 6, 5, 4, 3, 2, 1	1, 2, 3, 4, 5, 6, 7, 8, 9
4, 2, 7, 4, 8, 2, 4, 8, 1	1, 2, 2, 4, 4, 4, 7, 8, 8
5, 5, 5, 5, 5, 5, 5, 5, 5	5, 5, 5, 5, 5, 5, 5, 5, 5

3.6 Вывод

Таким образом, были сформулированы требования к реализуемой программе, выбран язык программирования C++ в качестве основного инструмента разработки, и успешно проведены тесты, подтверждающие правильность работы разработанной программы.

4 Исследовательский раздел

В данном разделе будет произведено исследование разработанной программы.

4.1 Эксперименты по замеру времени

На рисунках 4.1, 4.2, 4.3 приведены графики зависимости времени выполнения сортировки от длины сортируемого массива для отсортированных, отсортированных в обратном порядке и произвольных массивов соответственно.

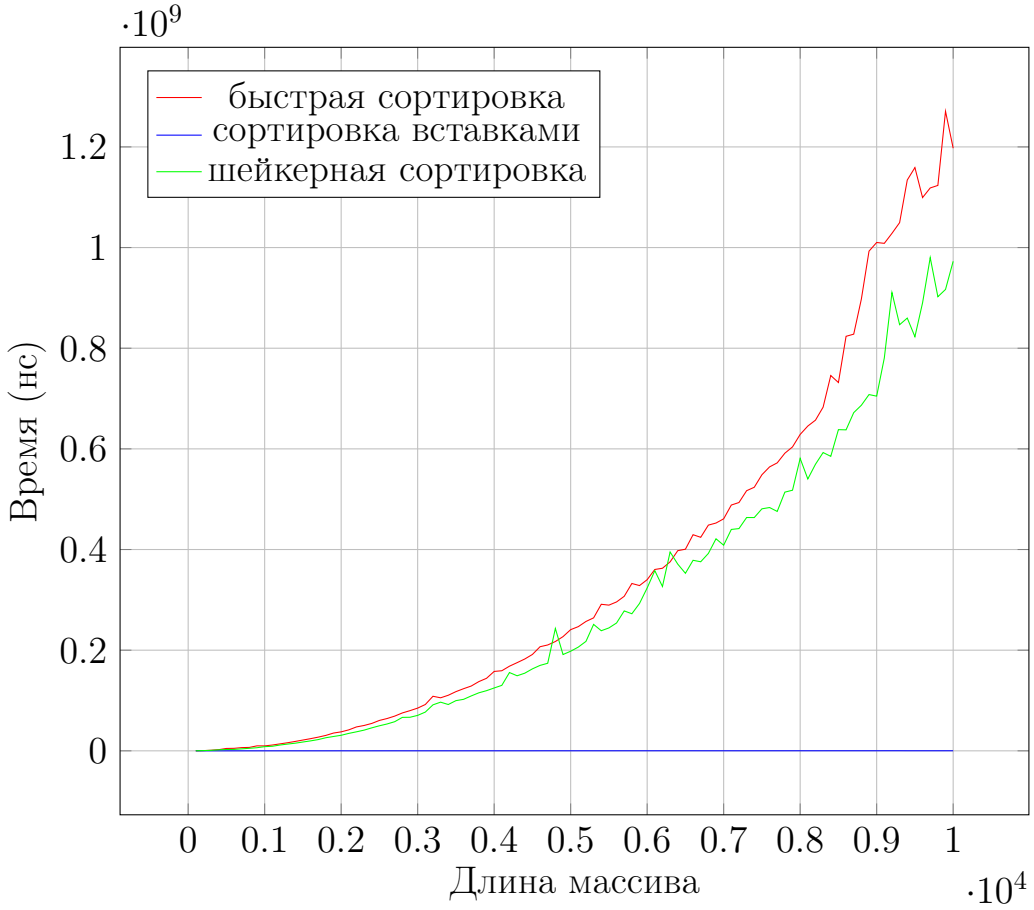

Рисунок 4.1 — Отсортированный массив

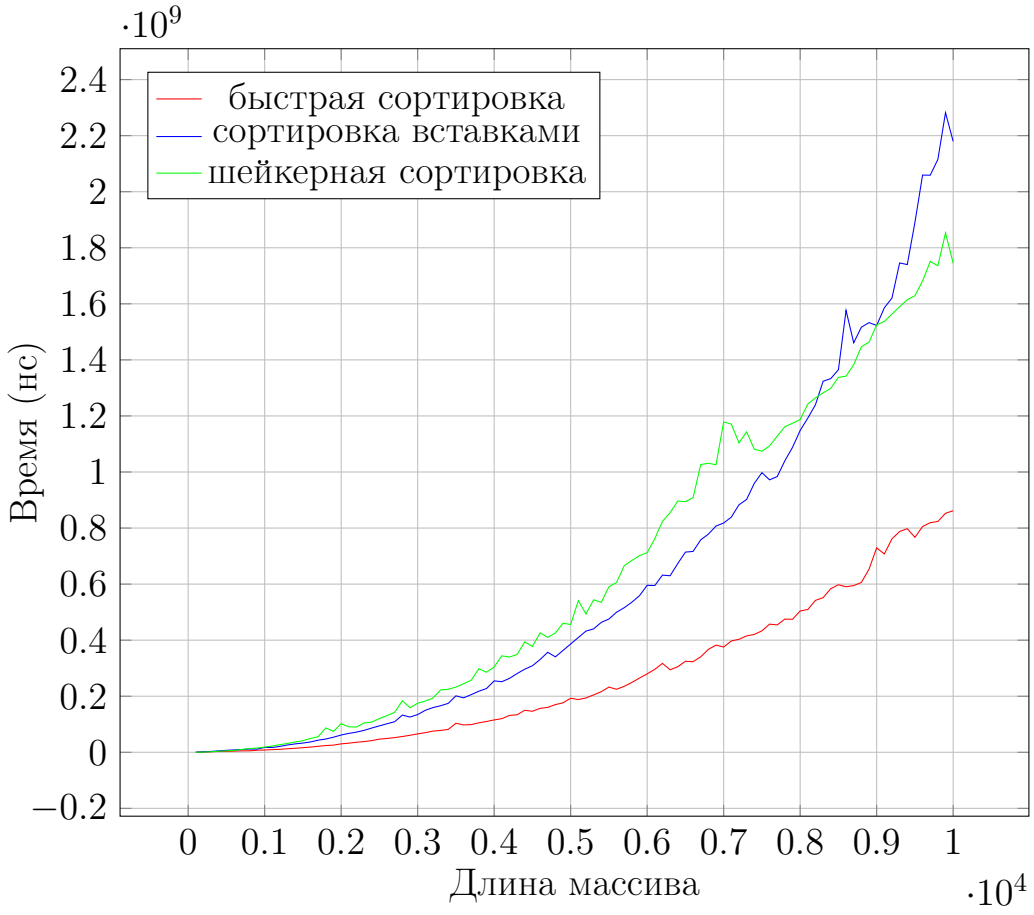

Рисунок 4.2 — Обратно отсортированный массив

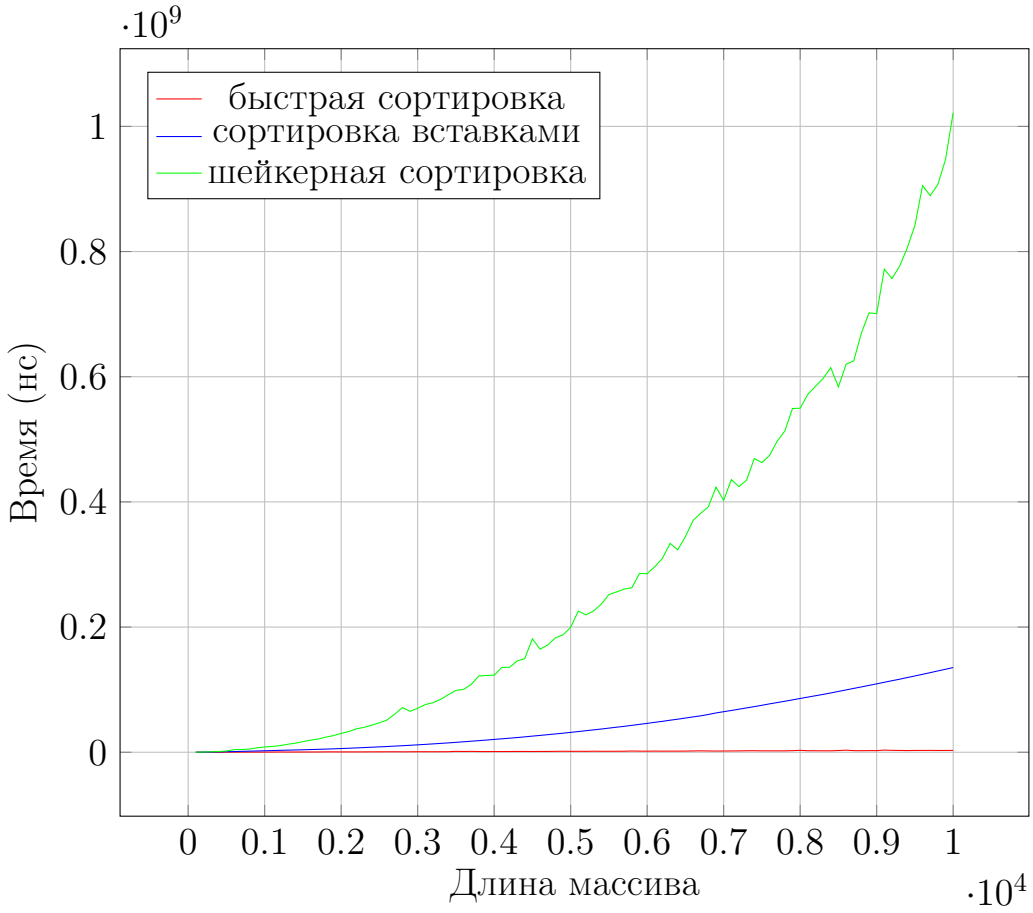

Рисунок 4.3 — Произвольный массив

Данный эксперимент проводился на ноутбуке, подключённом к сети питания. Модель процессора ноутбука: Intel i5-8400H с максимальной тактовой частотой 2.500 ГГц в обычном режиме.

4.2 Вывод

Из полученных графиков видно, что скорость работы быстрой сортировки превосходит две другие в случаях обратно отсортированных и произвольных массивов. При этом шейкерная сортировка уступает сортировке вставками в случаях произвольных и отсортированных массивов.

ЗАКЛЮЧЕНИЕ

В ходе данной работы было проведено сравнение трех алгоритмов сортировки: быстрая сортровка, сортировка вставками, шейкерная сортировка. Были сделаны следующие выводы:

— быстрая сортировка является универсальным решением при необходимости упорядочевания;

— шейкерная сортировка является худшим выбором в рассмотренных случаях.

Все задачи, поставленные в данной работе были выполнены:

а) реализовано три различных алгоритма сортировки;

б) теоретически вычислена эффективность алгоритмов;

в) алгоритмы сравнены по времени.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Кнут Д. - "Исскуство программирования для ЭВМ"

2. Павловская Т.А. - "C/C++. Программирование на языке высокого уровня"

3. Володин Ф. Основные концепции библиотеки chrono (C++) [Электронный ресурс]. URL: https://habr.com/ru/post/324984/ Дата обращения: 20.10.2019