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»

на тему: «Трудоёмкость сортировок»

Студент ИУ7-54Б _____ Писаренко Д. П.
(Группа) (Подпись, дата) (И. О. Фамилия)

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

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Сортировка данных является фундаментальной задачей в области информатики и алгоритмов. Независимо от конкретной области применения, эффективные алгоритмы сортировки существенно влияют на производительность программных систем. От правильного выбора алгоритма зависит как время выполнения программы, так и затраты ресурсов компьютера [1].

Алгоритмы сортировки находят применение в следующих сферах:

— базы данных;

— анализ данных и статистика;

— алгоритмы машинного обучения;

— криптография.

Цель данной лабораторной работы — исследовать алгоритмы сортировки. Для достижения поставленной цели необходимо выполнить следующие задачи:

— описать алгоритмы блочной, быстрой сортировок и сортировки выбором;

— разработать программное обеспечение, реализующее алгоритмы сортировок;

— выбрать инструменты для реализации и замера процессорного времени выполнения реализаций алгоритмов;

— проанализировать затраты реализаций алгоритмов по времени и памяти.

1 Аналитический раздел

Сортировкой называют перестановку объектов, при которой они располагаются в порядке возрастания или убывания [1].

В данном разделе будут описаны три алгоритма сортировок: блочная, быстрая и выбором.

1.1 Алгоритм блочной сортировки

Идея заключается в разбиении входных данных на «блоки» одинакового размера, после чего данные в блоках сортируются и результаты сортировок объединяются. Отсортированная последовательность получается путём последовательного перечисления элементов каждого блока. Для деления данных на блоки, алгоритм предполагает, что значения распределены равномерно, и распределяет элементы по блокам равномерно. Например, предположим, что данные имеют значения в диапазоне от 1 до 100 и алгоритм использует 10 блоков. Алгоритм помещает элементы со значениями 1–10 в первый блок, со значениями 11–20 во второй, и т.д. Если элементы распределены равномерно, в каждый блок попадает примерно одинаковое число элементов. Если в списке N элементов, и алгоритм использует N блоков, в каждый блок попадает всего один элемент, поэтому возможно отсортировать элементы за конечное число шагов.

1.2 Алгоритм быстрой сортировки

Данный алгоритм можно разделить на следующие шаги [2]:

— выбрать опорный элемент;

— разбить массив относительно опорного элемента: элементы меньше опорного поместить перед ним, больше - после него;

— рекурсивно применить алгоритм к подмассивам;

— объединить подмассивы в один отсортированный массив.

1.3 Алгоритм сортировки выбором

Данный алгоритм можно разделить на следующие шаги [3]:

— взять первый элемент последовательности A[i], здесь i – номер элемента, для первого i равен 1;

— найти минимальный (максимальный) элемент последовательности и запомнить его номер в переменную key;

— если номер первого элемента и номер найденного элемента не совпадают, т. е. если key != 1, тогда два этих элемента обменять значениями;

— увеличить i на 1 и продолжить сортировку оставшейся части массива, а именно с элемента с номером 2 по N, так как элемент A[1] уже занимает свою позицию.

Вывод

В данном разделе были описаны три алгоритма сортировок: блочная, быстрая и выбором.

2 Конструкторский раздел

В этом разделе будет представлено описание используемых типов данных, а также схематические изображения алгоритмов сортировок: блочной, быстрой и выбором.

2.1 Требования к программному обеспечению

Программа должна поддерживать два режима работы: режим массового замера времени и режим сортировки введенного массива.

Режим массового замера времени должен обладать следующей функциональностью:

— генерировать массивы различного размер для проведения замеров;

— осуществлять массовый замер, используя сгенерированные данные;

— результаты массового замера должны быть представлены в виде таблицы и графика.

К режиму сортировки выдвигается следующий ряд требований:

— возможность работать с массивами разного размера, которые вводит пользователь;

— наличие интерфейса для выбора действий;

— на выходе программы массив, отсортированный тремя алгоритмами по возрастанию.

2.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры и типы данных:

— целое число представляет количество элементов в массиве;

— список целых чисел;

2.3 Разработка алгоритмов

На рисунке 2.1 приведена схема алгоритма блочной сортировки.

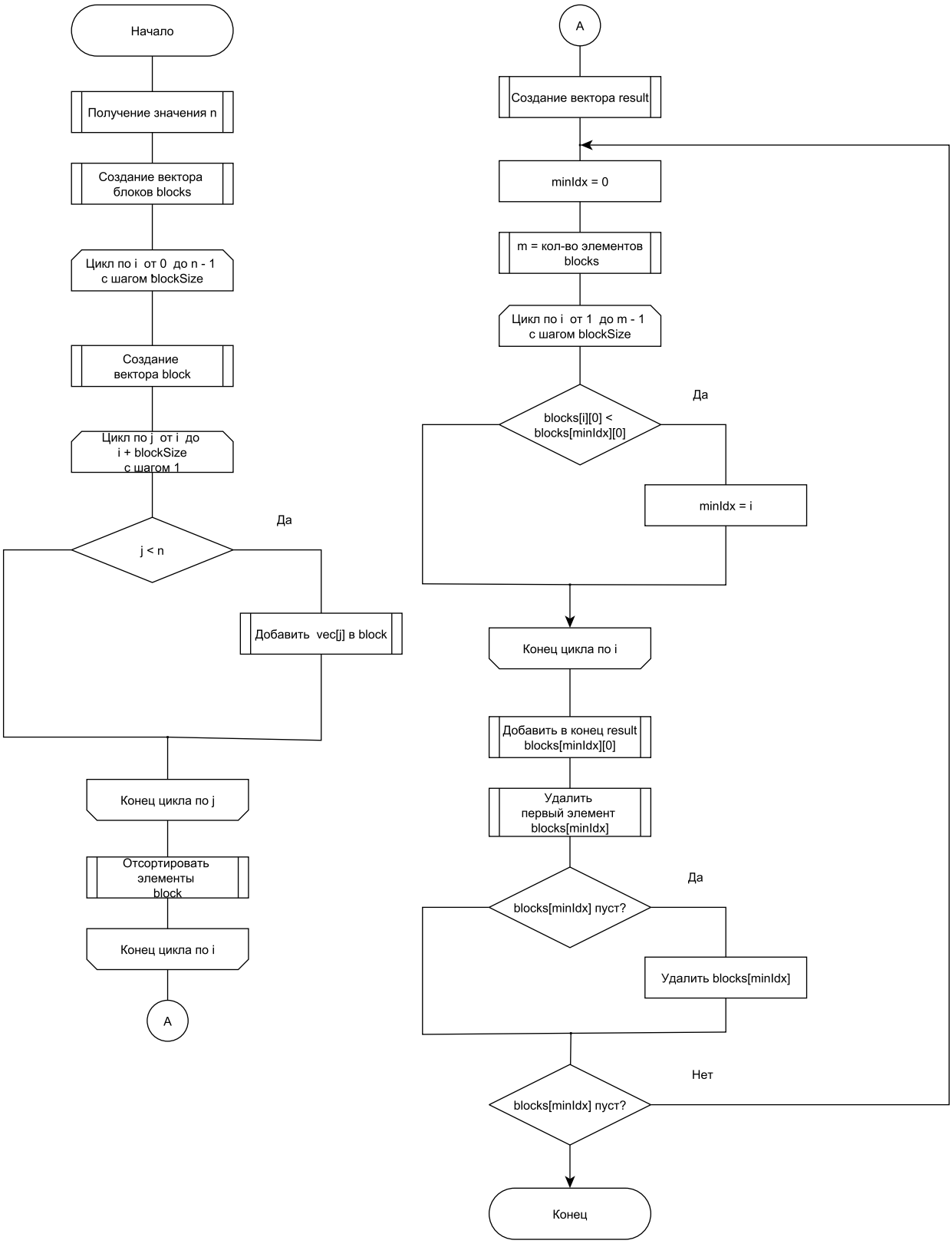

Рисунок 2.1 – Схема алгоритма блочной сортировки

На рисунке 2.2 приведена схема алгоритма быстрой сортировки.

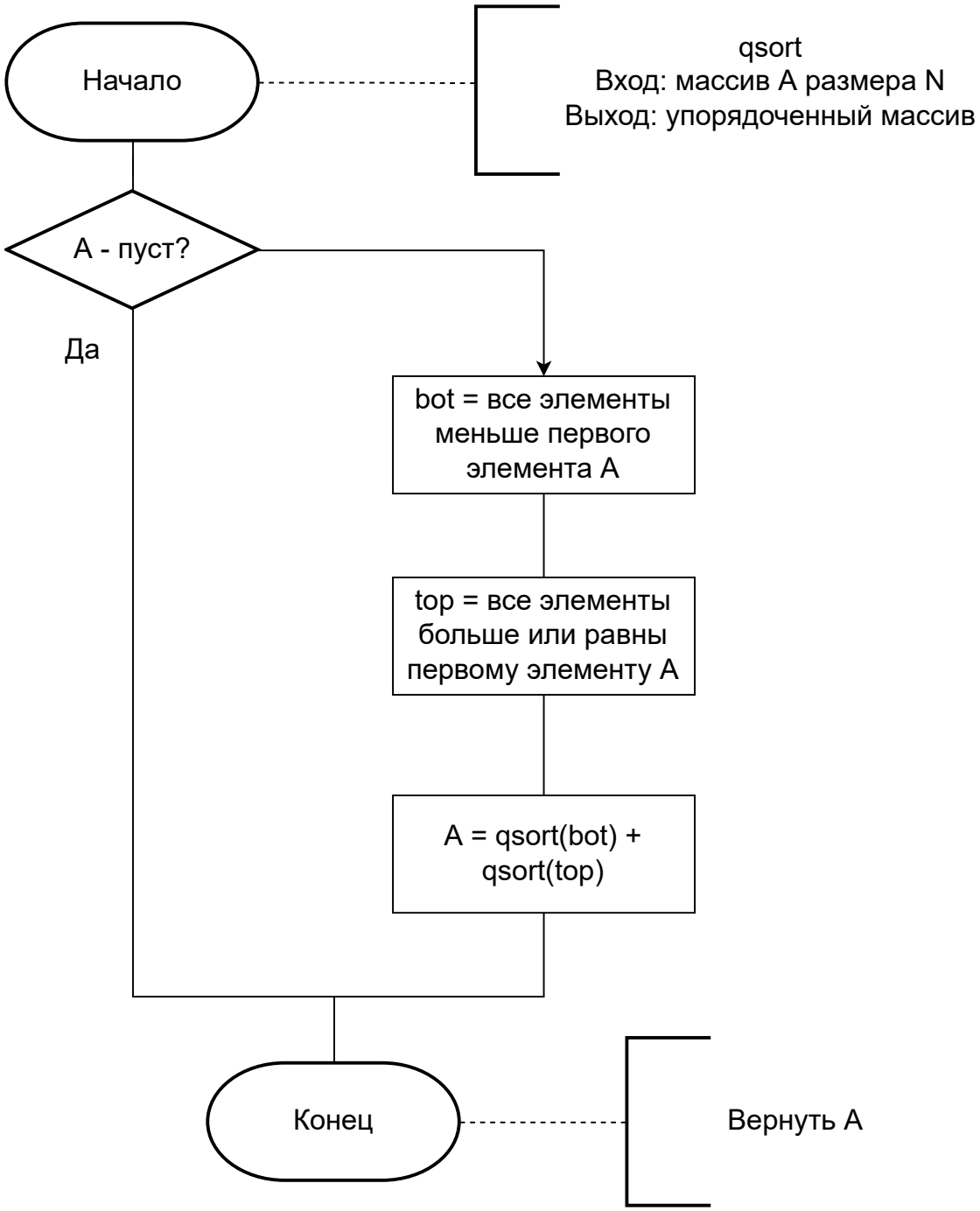

Рисунок 2.2 – Схема алгоритма быстрой сортировки

На рисунке 2.3 приведена схема алгоритма сортировки выбором.

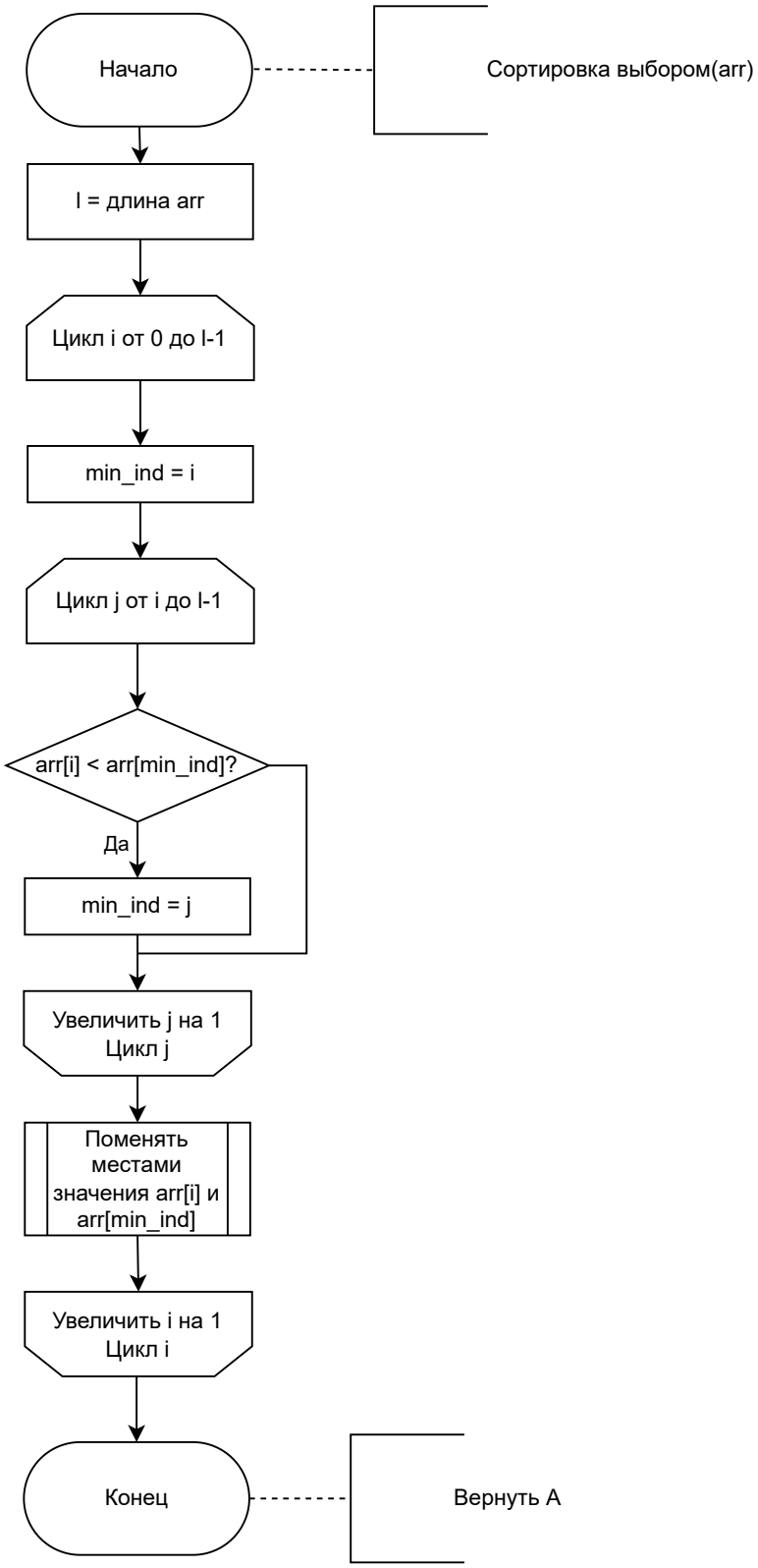

Рисунок 2.3 – Схема алгоритма сортировки выбором

2.4 Оценка трудоемкости алгоритмов

Модель для оценки трудоемкости алгоритмов состоит из шести пунктов:

1) $+, -, =, + =, - =, ==, ||, \&\&, <, >, <=, >=, <<, >>, []$ — считается, что эти операции обладают трудоемкостью в 1 единицу;

2) $*, /, * =, / =, \%$ — считается, что эти операции обладают трудоемкостью в 2 единицы;

3) трудоемкость условного перехода принимается за 0;

4) трудоемкость условного оператора рассчитывается по следующей формуле

$$f_{if} = f_{\text{условия}} + \begin{cases} min(f_1, f_2), & \text{лучший случай} \\ max(f_1, f_2), & \text{худший случай} \end{cases}, \tag{2.1}$$

где f_1 — трудоемкость блока, который вычисляется при выполнении условия, а f_2 — трудоемкость блока, который вычисляется при невыполнении условия;

5) трудоемкость цикла рассчитывается по следующей формуле

$$\begin{aligned} f_{for} = f_{\text{инициализация}} + f_{\text{сравнения}} + M_{\text{итераций}} \cdot (f_{\text{тело}} + \\ + f_{\text{инкремент}} + f_{\text{сравнения}}); \end{aligned} \tag{2.2}$$

6) вызов подпрограмм и передача параметров принимается за 0.

2.4.1 Трудоемкость алгоритма блочной сортировки

В данной реализации размер блока обозначается как k, трудоемкость операции добавления и удаления элемента из вектора равна 2.

Лучший случай: массив отсортирован, элементы распределены равномерно (все блоки содержат одинаковое число элементов), расчет трудоемкости данного случая приведен в следующей формуле

$$
\begin{aligned}
f_{best} = 1 + 1 + \frac{n}{k} \cdot (1 + 2 + f_{shaker} + 2 + 1 + 4 + \\
+ k \cdot (3 + 1 + 4)) + 1 + 1 + \\
+ \frac{n}{k} \cdot (1 + 4 + 1 + 1 + 5 + 1 + 4 + 1 + 1 + n \cdot (5)) = \\
= 4 + \frac{29 \cdot n + n \cdot f_{shaker} + 5 \cdot n^2}{k} + 8 \cdot n = \\
= 4 + 8 \cdot n + 29 \cdot \frac{n}{k} + n \cdot (14.5 + \frac{k}{2}) + \frac{5 \cdot n^2}{k} = \\
= 4 + 8 \cdot n + 29 \cdot \frac{n}{k} + 14.5 \cdot n + \frac{n \cdot k}{2} + \frac{5 \cdot n^2}{k} = \\
= \frac{5 \cdot n^2}{k} + 22.5 \cdot n + 29 \cdot \frac{n}{k} + \frac{n \cdot k}{2} + 4.
\end{aligned}
\tag{2.3}
$$

Худший случай: большое количество пустых блоков, массив отсортирован в обратном порядке (худший случай сортировки перемешиванием, которая используется в блочной сортировке), расчет трудоемкости приведен в следующей формуле

$$
\begin{aligned}
f_{worst} = 1 + 1 + \frac{n}{k} \cdot (1 + 2 + f_{shaker} + 2 + 1 + 4 + \\
+ k \cdot (3 + 1 + 4)) + 1 + 1 + \\
+ \frac{n}{k} \cdot (1 + 4 + 1 + 1 + 5 + 1 + 4 + 1 + 1 + k \cdot (6)) = \\
= 4 + \frac{29 \cdot n + n \cdot f_{shaker} + 6 \cdot n^2}{k} + 8 \cdot n = \\
= 4 + 8 \cdot n + 29 \cdot \frac{n}{k} + n \cdot (19.5 + \frac{k}{2}) + \frac{6 \cdot n^2}{k} = \\
= 4 + 8 \cdot n + 29 \cdot \frac{n}{k} + 19.5 \cdot n + \frac{n \cdot k}{2} + \frac{6 \cdot n^2}{k} = \\
= \frac{6 \cdot n^2}{k} + 27.5 \cdot n + 29 \cdot \frac{n}{k} + \frac{n \cdot k}{2} + 4.
\end{aligned}
\tag{2.4}
$$

2.4.2 Трудоемкость алгоритма быстрой сортировки

Чтобы вычислить трудоемкость алгоритма быстрой сортировки, нужно учесть следующее:

— трудоемкость условного оператора на проверку *pivot* в теле цикла вычисляется по следующей формуле

$$f_{if\_pivot} = 1 + \begin{cases} 2 + 1, \\ 0 \end{cases} \tag{2.5}$$

— трудоемкость цикла, в теле которого условный оператор на проверку $pivot$, вычисляется по следующей формуле

$$f_{for\_low\_high} = 1 + M \cdot f_{if\_pivot} \tag{2.6}$$

— трудоемкость условного оператора на проверку low в теле цикла вычисляется по следующей формуле

$$f_{if\_low} = 1 + \begin{cases} 1, \\ 0 \end{cases} \tag{2.7}$$

— трудоемкость цикла, в теле оператор ветвления f_{if_low} и $f_{for\_low\_high}$, вычисляется по следующей формуле

$$f_{while\_stack} = 1 + N \cdot (2 + f_{if\_low} + 2 + f_{for\_low\_high} + 2 + 2) \tag{2.8}$$

— трудоемкость условного оператора на проверку длины массива вычисляется по формуле

$$f_{if\_len} = 1 + \begin{cases} 1, \\ 1 + f_{while\_stack} \end{cases} \tag{2.9}$$

В итоге, трудоемкость быстрой сортировки вычисляется по формуле

$$\begin{gathered} f_{quick\_sort} = 1 + f_{if\_len} = 1 + 1 + N \cdot (2 + f_{if\_low} + 2 + f_{for\_low\_high}) = \\ 1 + 1 + N \cdot (2 + 2 + 2 + 1 + M \cdot f_{if\_pivot} + 2 + 2) = \\ 1 + 1 + N \cdot (2 + 2 + 2 + 1 + M \cdot (3 + 2 + 2) + 1 + 3 + 2 + 1 = \\ 2 + 7N + 7MN + 7N = 14N + 7MN \end{gathered} \tag{2.10}$$

2.4.3 Трудоемкость алгоритма сортировки выбором

Трудоемкость сортировки выбором в худшем случае $O(N^2)$.
Трудоемкость сортировки выбором в лучшем случае $O(N^2)$.
Худший и лучший случаи совпадают [3].

Вывод

На основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов. Была введена модель оценки трудоемкости алгоритма, были рассчитаны трудоемкости алгоритмов в соответствии с этой моделью.

В результате теоретической оценки трудоемкостей алгоритмов выяснилось, что лучшей асимптотической оценкой во всех случаях обладает быстрая сортировка. Худшей асимптотической оценкой во всех случаях обладает сортировка выбором $O(N^2)$.

3 Технологический раздел

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинг кода и функциональные тесты.

3.1 Средства реализации

Для реализации данной работы был выбран язык *Python* [4]. Данный выбор обусловлен следующим:

— язык поддерживает все структуры данных, которые выбраны в результате проектирования;

— язык позволяет реализовать все алгоритмы, выбранные в результате проектирования;

— язык позволяет замерять процессорное время с помощью модуля *time*.

Процессорное время было замерено с помощью функции *process_time()* из модуля *time* [5].

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

— *main.py* — файл, содержащий функцию *main*;

— *functions.py* — файл, содержащий вспомогательные функции (ввод массива с клавиатуры, рандомно, с файла и т.д.)

— *sorts.py* — файл, содержащий код реализаций всех алгоритмов сортировок;

— *tests.py* — файл, в котором содержатся функции для замера и вывода времени выполнения реализаций алгоритмов.

3.3 Реализация алгоритмов

В листингах 3.1 – 3.3 приведены реализации блочной, быстрой сортировок и сортировки выбором.

Листинг 3.1 – Реализация блочной сортировки

```
def block_sort(arr):
    block_size = 10
    blocks = []

    i = 0
    while i < len(arr):
        block = []
        j = i
        while j < i + block_size and j < len(arr):
            block.append(arr[j])
            j += 1
        block.sort()
        blocks.append(block)
        i += block_size

    arr_ind = 0
    while blocks:
        min_ind = 0
        for i in range(1, len(blocks)):
            if blocks[i][0] < blocks[min_ind][0]:
                min_ind = i

        arr[arr_ind] = blocks[min_ind][0]
        arr_ind += 1
        blocks[min_ind].pop(0)

        if not blocks[min_ind]:
            blocks.pop(min_ind)

    return arr
```

Листинг 3.2 – Реализация быстрой сортировки

```
import random

def quick_sort(arr):
    less = []
    equal = []
    greater = []

    if len(arr) > 1:
        pivot = arr[random.randint(0, len(arr) -1)]
        for x in arr:
            if x < pivot:
                less.append(x)
            elif x == pivot:
                equal.append(x)
            elif x > pivot:
                greater.append(x)
        return quick_sort(less) + equal + quick_sort(greater)

    else:
        return arr
```

Листинг 3.3 – Реализация сортировки выбором

```
def selection_sort(arr):
    size = len(arr)

    for ind in range(size):
        min_ind = ind

        for j in range(ind + 1, size):
            if arr[j] < arr[min_ind]:
                min_ind = j
        arr[ind], arr[min_ind] = arr[min_ind], arr[ind]

    return arr
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для разработанных алгоритмов сортировок. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Массив	**Ожидаемый результат**	**Фактический 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]
[ ]	[ ]	[ ]
[1]	[1]	[1]
[4, 1, 2, 3]	[1, 2, 3, 4]	[1, 2, 3, 4]
[2, 1]	[1, 2]	[1, 2]
[31, 57, 24, -10, 59]	[-10, 24, 31, 57, 59]	[-10, 24, 31, 57, 59]

Вывод

Были разработаны и протестированы спроектированные алгоритмы сортировок: блочная, быстрая и выбором.

4 Исследовательский раздел

В данном разделе будут приведены: пример работы программы, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы разработанного программного обеспечения, а именно показаны результаты сортировки массива $[3, 5, -3, 1, 2, 4, -6, 2]$.

```
МЕНЮ:
1. Блочная сортировка
2. Быстрая сортировка
3. Сортировка выбором
4. Измерить время
0. Выход

Выберите пункт: 1
Введите элементы массива (в одну строку через пробел):
3 5 -3 1 2 4 -6 2

Результат: [-6, -3, 1, 2, 2, 3, 4, 5]


МЕНЮ:
1. Блочная сортировка
2. Быстрая сортировка
3. Сортировка выбором
4. Измерить время
0. Выход

Выберите пункт: 2
Введите элементы массива (в одну строку через пробел):
3 5 -3 1 2 4 -6 2

Результат: [-6, -3, 1, 2, 2, 3, 4, 5]


МЕНЮ:
1. Блочная сортировка
2. Быстрая сортировка
3. Сортировка выбором
4. Измерить время
0. Выход

Выберите пункт: 3
Введите элементы массива (в одну строку через пробел):
3 5 -3 1 2 4 -6 2

Результат: [-6, -3, 1, 2, 2, 3, 4, 5]
```

Рисунок 4.1 – Демонстрация работы программы при сортировке массива

4.2 Технические характеристики

Технические характеристики компьютера, на котором проводился замерный эксперимент:

— процессор Intel Core i5-10400F (6 ядер) [6];

— 16 Гб оперативная память DDR4;

— операционная система Windows 10 Pro [7].

Во время проведения исследования компьютер был нагружен только системными приложениями и целевой программой.

4.3 Время выполнения реализаций алгоритмов

Результаты замеров времени выполнения реализаций алгоритмов сортировок приведены в таблицах 4.1 – 4.3. Замеры времени проводились на массивах одного размера и усреднялись для каждого набора одинаковых экспериментов.

В таблицах 4.1 – 4.3 используются следующие обозначения:

— Блочная — реализация алгоритма блочной сортировки;

— Быстрая — реализация алгоритма быстрой сортировки;

— Выбором — реализация алгоритма сортировки выбором.

Таблица 4.1 – Время работы реализации алгоритмов на неотсортированных массивах (в мс)

Размер массива	Блочная	Быстрая	Выбором
100	0.078	0.125	0.203
200	0.203	0.266	0.906
300	0.406	0.500	1.781
400	0.766	0.547	3.484
500	1.078	0.734	5.625
600	1.391	0.891	7.813
700	1.813	1.156	10.609
800	2.234	1.250	14.250
900	2.984	1.297	18.063
1000	3.734	1.500	22.516

Таблица 4.2 – Время работы реализации алгоритмов на отсортированных в обратном порядке массивах (в мс)

Размер массива	Блочная	Быстрая	Выбором
100	0.078	0.125	0.203
200	0.203	0.250	0.766
300	0.406	0.391	1.938
400	0.641	0.547	3.348
500	0.969	0.688	5.375
600	1.328	0.875	7.609
700	1.750	1.047	10.641
800	2.609	1.141	14.047
900	2.938	1.250	17.828
1000	3.719	1.438	22.266

Таблица 4.3 – Время работы реализации алгоритмов на отсортированных массивах (в мс)

Размер массива	Блочная	Быстрая	Выбором
100	0.063	0.109	0.203
200	0.203	0.313	0.828
300	0.391	0.438	1.859
400	0.641	0.578	3.453
500	1.000	0.688	5.313
600	1.313	0.859	7.906
700	1.891	1.031	10.469
800	2.281	1.125	13.813
900	2.938	1.313	18.109
1000	3.438	1.438	22.422

На рисунках 4.2 – 4.4 изображены графики зависимостей времени выполнения реализаций сортировок от размеров массивов.

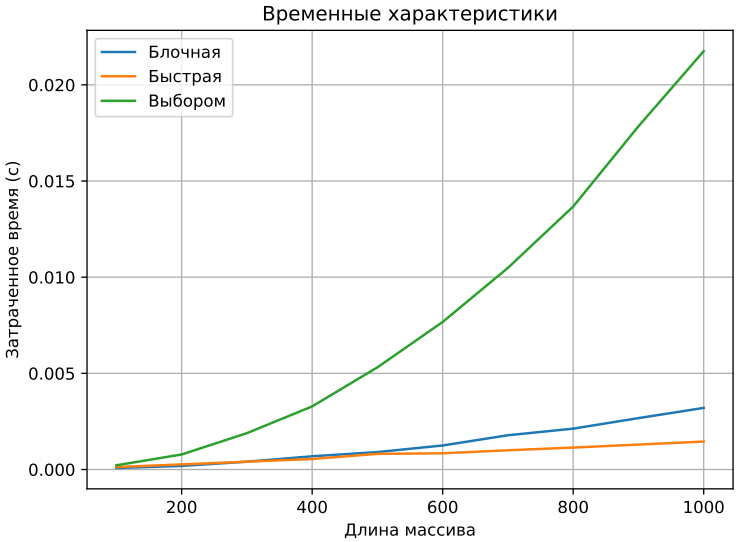

Рисунок 4.2 – Сравнение реализаций алгоритмов по времени выполнения на неотсортированных массивах

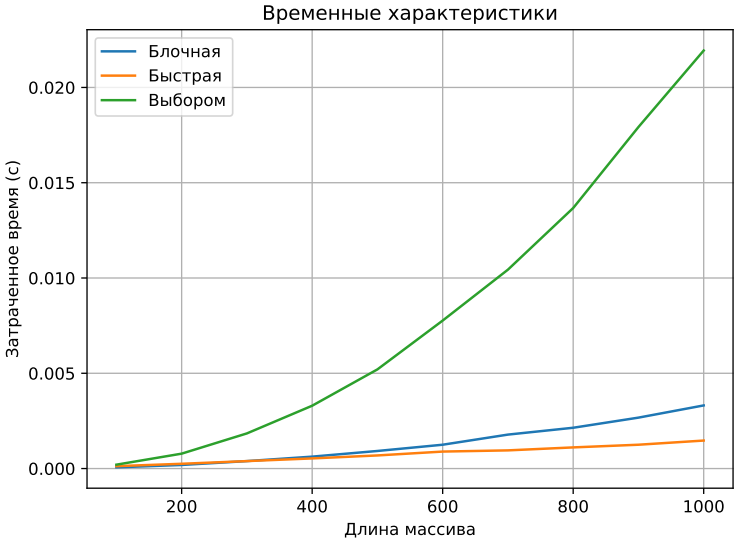

Рисунок 4.3 – Сравнение реализаций алгоритмов по времени выполнения на отсортированных в обратном порядке массивах

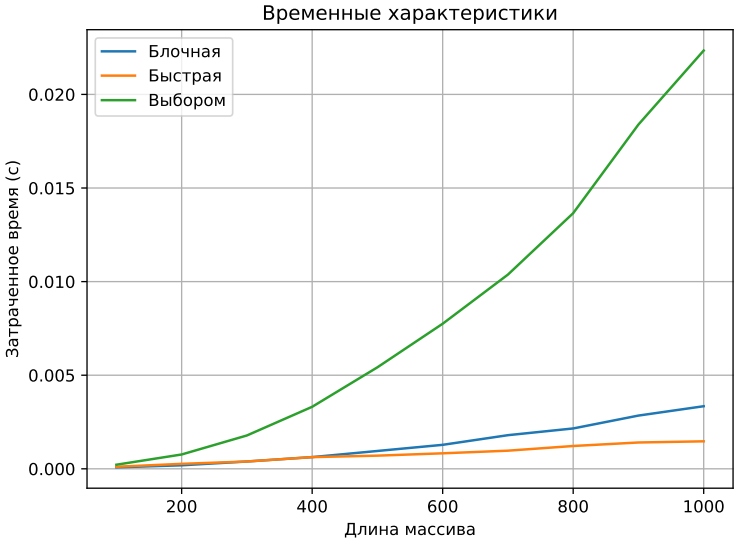

Рисунок 4.4 – Сравнение реализаций алгоритмов по времени выполнения на отсортированных массивах

4.4 Затраты по памяти реализаций алгоритмов

Введем следующие обозначения:

— size(a) — функция, вычисляющая размер входного параметра a в байтах;

— int — целочисленный тип данных.

Теоретически оценим объем используемой алгоритмами памяти при сортировке массива размером N.

4.4.1 Блочная сортировка

Оценка используемой блочной сортировкой памяти приведена в формуле

$$M_{BlockSort} = 5 \cdot \text{size}(int) + N \cdot \text{size}(int) + 8 \tag{4.1}$$

где

— $5 \cdot \text{size}(int)$ — дополнительные переменные;

— $N \cdot \text{size}(int)$ — массив блоков;

— 8 — указатель на массив, переданный в качестве параметра.

4.4.2 Быстрая сортировка

Оценка используемой быстрой сортировкой памяти приведена в формуле

$$M_{QuickSort} = M_{call} \cdot \begin{cases} \log_2 N, & \text{лучший случай,} \\ N, & \text{худший случай} \end{cases} \tag{4.2}$$

где

— $M_{call} = (8+3 \cdot \text{size}(int)+8$ — память, затрачиваемая на один рекурсивный вызов (8 — адрес возврата, $3 \cdot \text{size}(int)$ —дополнительные переменные, 8 — указатель на массив);

— $\log_2 N$ — глубина стека вызовов в лучшем случае;

— N — глубина стека вызовов в худшем случае.

Тогда итоговая формула

$$M_{QuickSort} = (8 + 3 \cdot \text{size}(int) + N \cdot \text{size}(int)) \cdot \begin{cases} \log_2 N, & \text{лучший случай,} \\ N, & \text{худший случай} \end{cases} \quad (4.3)$$

4.4.3 Сортировка выбором

Оценка используемой сортировкой выбором памяти приведена в формуле

$$M_{ChoiceSort} = 3 \cdot \text{size}(int) + 8 \quad (4.4)$$

где

— $3 \cdot \text{size}(int)$ — дополнительные переменные;

— 8 — указатель на массив, переданный в качестве параметра.

Вывод

В результате замеров времени выполнения реализаций различных алгоритмов было выявлено, что для массивов длиной 1000, отсортированных в обратном порядке, реализация алгоритма быстрой сортировки по времени оказалась в 2.59 раз лучше, чем реализация блочной сортировки, и в 15.49 раз лучше реализации сортировки выбором. В свою очередь, реализация блочной сортировки оказалась лучше в 5.99 раз по времени выполнения, чем реализация сортировки выбором.

Для отсортированных массивов длиной 1000 реализация быстрой сортировки оказалась лучше по времени в 2.39 раз, чем реализация блочной сортировки, и в 15.6 раз лучше, чем реализация сортировки выбором. В свою очередь, реализация блочной сортировки оказалась лучше в 6.52 раза по времени выполнения, чем реализация сортировки выбором.

Для случайно упорядоченных массивов длиной 1000 реализация быстрой сортировки оказалась лучше по времени в 2.49 раз, чем реализация блочной сортировки, и в 15.01 раза лучше, чем реализация сортировки выбором. В свою очередь, реализация блочной сортировки на случайно упорядоченных массивах оказалась лучше в 6.03 раза по времени выполнения, чем реализация сортировки выбором.

Стоит заметить, что для массивов длиной менее 300, реализация блочной сортировки была лучше или такой же по времени выполнения по сравнению с быстрой сортировкой, но на массивах длиной более 300 быстрая сортировка становилась лучше по времени выполнению.

В результате теоретической оценки алгоритмов по памяти можно сделать вывод о том, что алгоритм сортировки выбором является наименее ресурсозатратным. Алгоритм быстрой сортировки, напротив, требует больше всего памяти, что объясняется тем, что алгоритм рекурсивный, и на каждый вызов функции требуется выделение памяти на стеке для сохранения информации, связанной с этим вызовом.

ЗАКЛЮЧЕНИЕ

Цель данной лабораторной работы была достигнута, а именно были исследованы алгоритмы сортировок.

Для достижения поставленной цели были выполнены следующие задачи.

— описаны алгоритмы блочной, быстрой сортировок и сортировки выбором;

— разработано программное обеспечение, реализующее алгоритмы сортировок;

— выбраны инструменты для реализации алгоритмов и замера процессорного времени их выполнения;

— проведен анализ затрат реализаций алгоритмов по времени и памяти.

В результате исследования реализаций различных алгоритмов было получено, что для массивов длиной 1000, отсортированных в обратном порядке, реализация алгоритма быстрой сортировки по времени оказалась в 2.59 раз лучше, чем реализация блочной сортировки, и в 15.49 раз лучше реализации сортировки выбором. В свою очередь, реализация блочной сортировки оказалась лучше в 5.99 раз по времени выполнения, чем реализация сортировки выбором.

Для отсортированных массивов длиной 1000 реализация быстрой сортировки оказалась лучше по времени в 2.39 раз, чем реализация блочной сортировки, и в 15.6 раз лучше, чем реализация сортировки выбором. В свою очередь, реализация блочной сортировки оказалась лучше в 6.52 раза по времени выполнения, чем реализация сортировки выбором.

Для случайно упорядоченных массивов длиной 1000 реализация быстрой сортировки оказалась лучше по времени в 2.49 раз, чем реализация блочной сортировки, и в 15.01 раза лучше, чем реализация сортировки выбором. В свою очередь, реализация блочной сортировки на случайно упорядоченных массивах оказалась лучше в 6.03 раза по времени выполнения, чем реализация сортировки выбором.

Для массивов длиной менее 300, реализация блочной сортировки была лучше или такой же по времени выполнения по сравнению с быстрой сортировкой, но на массивах длиной более 300 быстрая сортировка становилась лучше по времени выполнению.

В результате теоретической оценки алгоритмов по памяти можно сделать вывод о том, что алгоритм сортировки выбором является наименее ресурсозатратным. Алгоритм быстрой сортировки, напротив, требует больше всего памяти, что объясняется тем, что алгоритм рекурсивный, и на каждый вызов функции требуется выделение памяти на стеке для сохранения информации, связанной с этим вызовом.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Э. К. Д.* Искусство программирования, том 3. Сортировка и поиск, 2-е изд. //. Т. 832. — Пер. с англ. М.: ООО 'И. Д. Вильямс', 2007.
2. Быстрая сортировка: [Электронный ресурс]. — Режим доступа: `https://kvodo.ru/quicksort.html` (дата обращения: 01.12.2023).
3. Сортировка выбором: [Электронный ресурс]. — Режим доступа: `https://kvodo.ru/sortirovka-vyiborom-2.html` (дата обращения: 01.12.2023).
4. The official home of the Python Programming Language [Электронный ресурс]. — Режим доступа: `https://www.python.org/` (дата обращения: 01.12.2023).
5. time — Time access and conversions [Электронный ресурс]. — Режим доступа: `https://docs.python.org/3/library/time.html` (дата обращения: 01.12.2023).
6. Intel Core i5-10400F. — Режим доступа: `https://openbenchmarking.org/s/Intel+Core+i5-10400F` (дата обращения 30.11.2023).
7. Windows 10 Pro 22h2 64-bit [Электронный ресурс]. — Режим доступа: `https://www.microsoft.com/ru-ru/software-download/windows10` (дата обращения: 01.12.2023).