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 [**knut**].

Алгоритмы сортировки находят применение в следующих сферах:

— базы данных;

— анализ данных и статистика;

— алгоритмы машинного обучения;

— криптография.

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

— описать алгоритмы блочной, быстрой сортировок и сортировку выбором;

— разработать программное обеспечение, реализующее алгоритмы сортировок;

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

— проанализировать затраты реализаций алгоритмов по времени.

1 Аналитический раздел

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

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 или два элемента, поэтому возможно отсортировать элементы за конечное число шагов [**article_sorts**].

1.2 Алгоритм быстрой сортировки

Данный алгоритм можно разделить на следующие шаги [**quicksort**]:

— разбить массива относительно опорного элемента;

— рекурсивно отсортировать каждую часть массива:

1.3 Алгоритм сортировки выбором

Данный алгоритм можно разделить на следующие шаги [**selectionsort**]:

— 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.

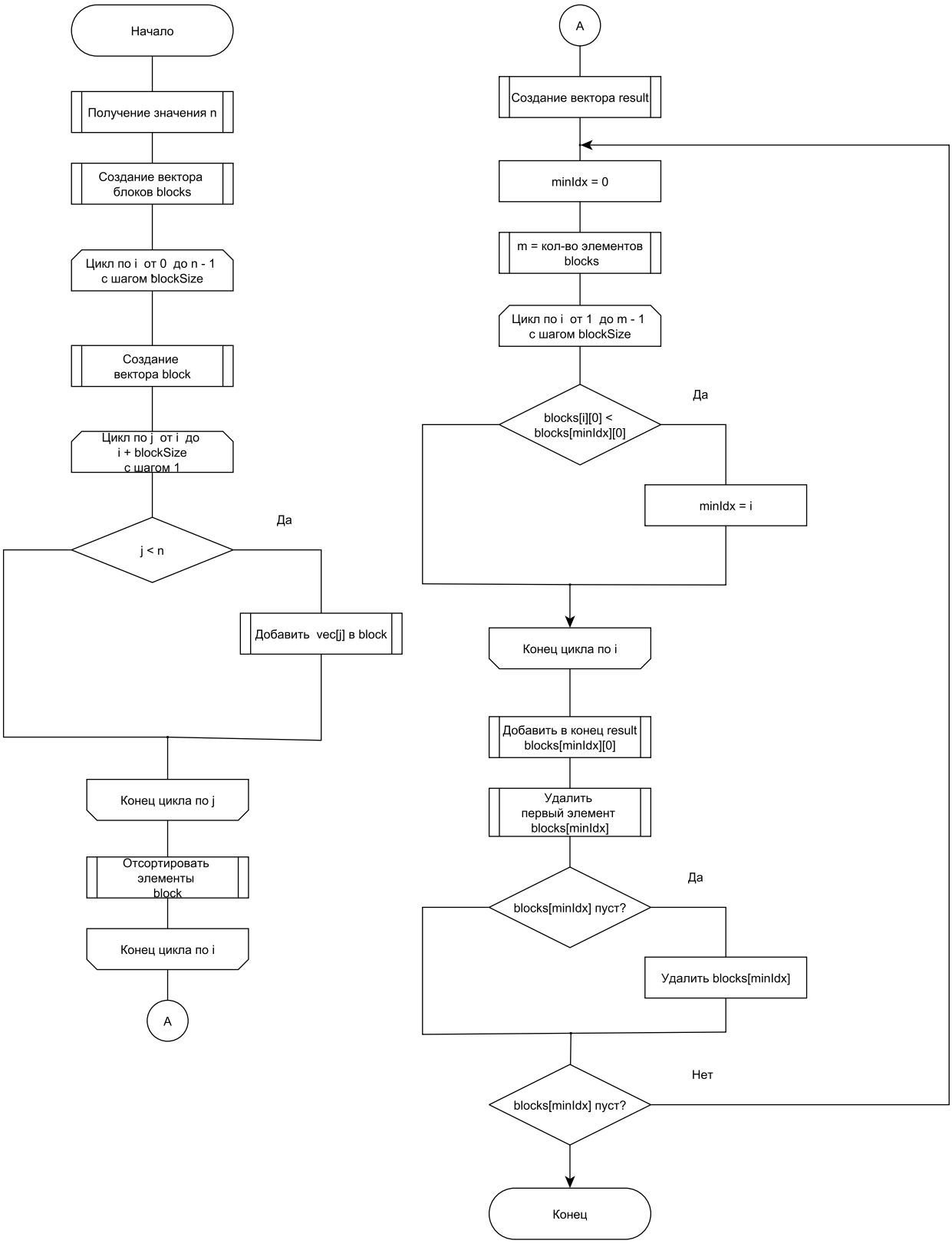

Рисунок 2.1 – Схема алгоритма блочной сортировки

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.

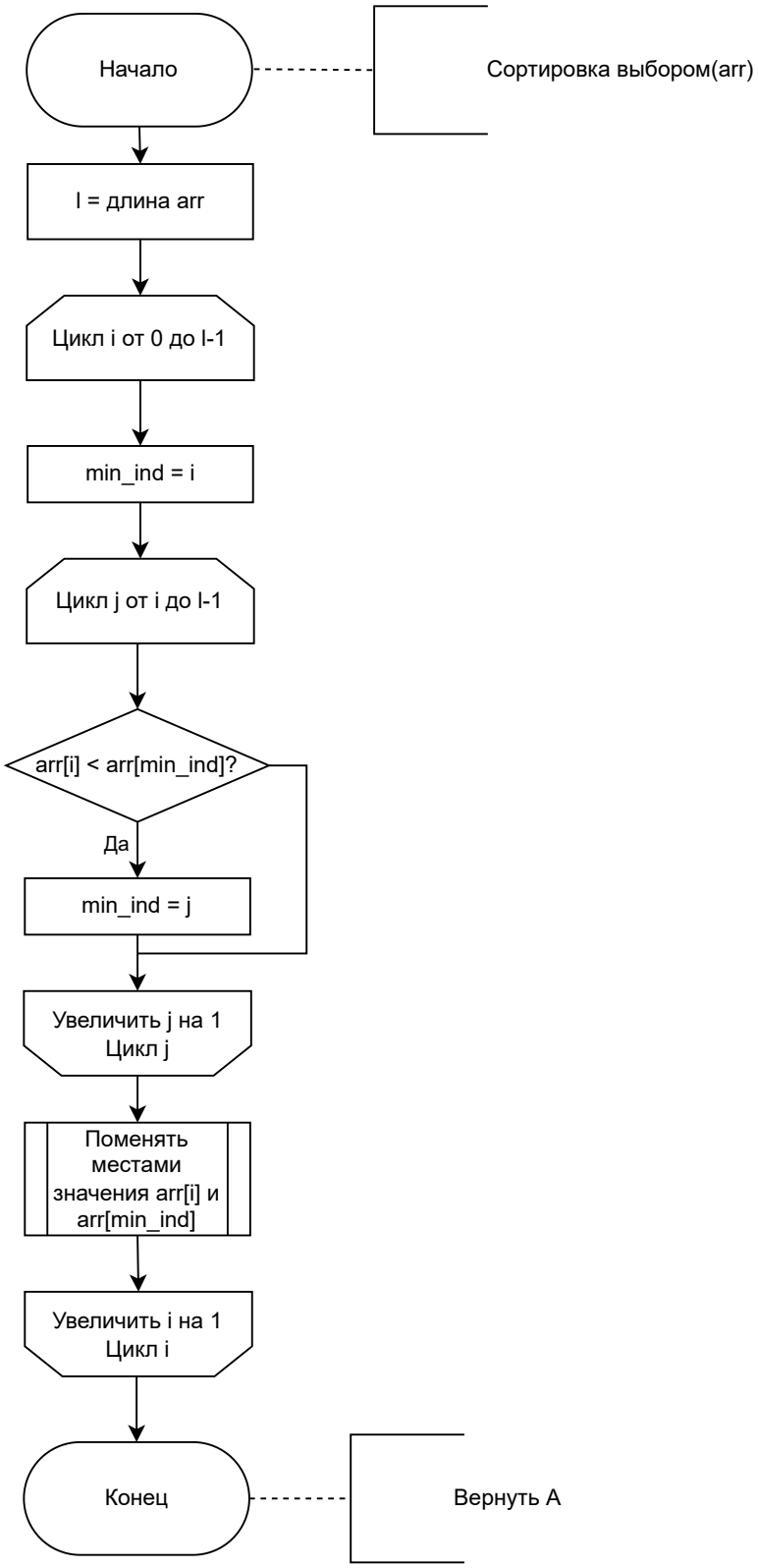

Рисунок 2.3 – Схема алгоритма сортировки выбором

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 формуле (2.1),

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

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 формуле (2.2),

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

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 (2.3).

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

Худший случай: большое количество пустых блоков, массив отсортирован в обратном порядке (худший случай сортировки перемешиванием, которая используется в блочной сортировке), расчет трудоемкости приведен в выражении (2.4).

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

Вывод о трудоемкости блочной сортировки

Данная реализация зависит от выбранного размера блока, а также от количества сортируемых элементов. В соответствии с выражениями (2.4, 2.3) операнды $\frac{n \cdot k}{2}$, $\frac{n^2}{k}$ и $\frac{n}{k}$ зависят от значений n и k, первый приведенный операнд возрастает при увеличении k, другие убывают.

2.4.2 Трудоемкость алгоритма гномьей сортировки

Для данной сортировки лучшим случаем является отсортированный массив, тогда трудоемкость алгоритма будет рассчитываться по формуле (2.5).

$$
f_{best} = 1 + 1 + (N - 1)(1 + 1 + 4 + 1) = 7N - 5 \approx O(N) \tag{2.5}
$$

Для данного алгоритма худшим является случай, когда массив отсорти-

рован в обратном порядке, тогда трудоемкость рассчитывается по (2.6).

$$\begin{aligned} f_{worst} = 1 + 1 + \frac{(N+1)N}{2}(1+1+4+9+1) + N + \\ + \frac{(N+1)N}{2}(1+1+4+1) = \\ = \frac{23}{2}N^2 + \frac{25}{2}N + 2 \approx O(N^2) \end{aligned} \tag{2.6}$$

2.4.3 Трудоемкость алгоритма сортировки выбором

Трудоемкость сортировки выбором в худшем случае $O(N^2)$
Трудоемкость сортировки выбором в лучшем случае $O(N^2)$

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 в худшем случае обладает пирамидальная сортировка $O(N \log_2 N)$. Алгоритм гномьей сортировки оказывается лучше всех прочих сортировок в лучшем случае (т. е. для отсортированных массивов), обладая асимптотической сложностью $O(N)$, но при худшем случае проигрывает по константе алгоритму Шелла, т. о. асимптотическая сложность в худшем случае у гномьей сортировки $O(\frac{23}{2}N^2)$, а у Шелла $O(\frac{32}{3}N^2)$. В лучшем случае асимптотическая сложность алгоритма Шелла $O(10N \log_2 N)$ меньше, чем асимптотическая сложность алгоритма пирамидальной сортировки $O(29N \log_2 N)$.

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* [**python**]. 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* [**python-time**].

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) [**intel**];

— 16 Гб оперативная память DDR4;

— операционная система Windows 10 Pro [**windows**].

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	0.078125	0.125	0.203125
200	0.203125	0.265625	0.90625
300	0.40625	0.5	1.78125
400	0.765625	0.546875	3.484375
500	1.078125	0.734375	5.625
600	1.390625	0.890625	7.8125
700	1.8125	1.15625	10.609375
800	2.234375	1.25	14.25
900	2.984375	1.296875	18.0625
1000	3.734375	1.5	22.515625

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

Размер массива	Блочная	Быстрая	Выбором
100	0.078125	0.125	0.203125
200	0.203125	0.25	0.765625
300	0.40625	0.390625	1.9375
400	0.640625	0.546875	3.3475
500	0.96875	0.6875	5.375
600	1.328125	0.875	7.609375
700	1.75	1.046875	10.640625
800	2.609375	1.140625	14.046875
900	2.9375	1.25	17.828125
1000	3.71875	1.4375	22.265625

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

Размер массива	Блочная	Быстрая	Выбором
100	0.0625	0.109375	0.203125
200	0.203125	0.3125	0.828125
300	0.390625	0.4375	1.859375
400	0.640625	0.578125	3.453125
500	1	0.6875	5.3125
600	1.3125	0.859375	7.90625
700	1.890625	1.03125	10.46875
800	2.28125	1.125	13.8125
900	2.9375	1.3125	18.109375
1000	3.4375	1.4375	22.421875

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

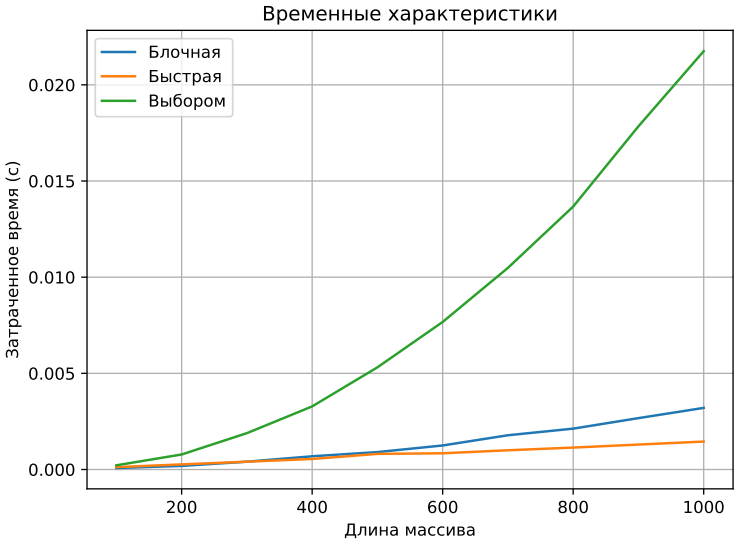

Рисунок 4.2 – Сравнение реализаций алгоритмов по времени выполнения на неотсортированных массивах

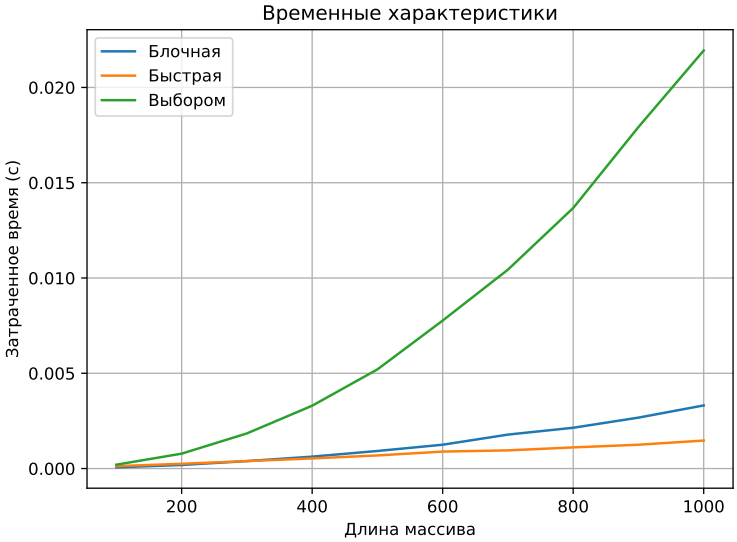

Рисунок 4.3 – Сравнение реализаций алгоритмов по времени выполнения на отсортированных в обратном порядке массивах

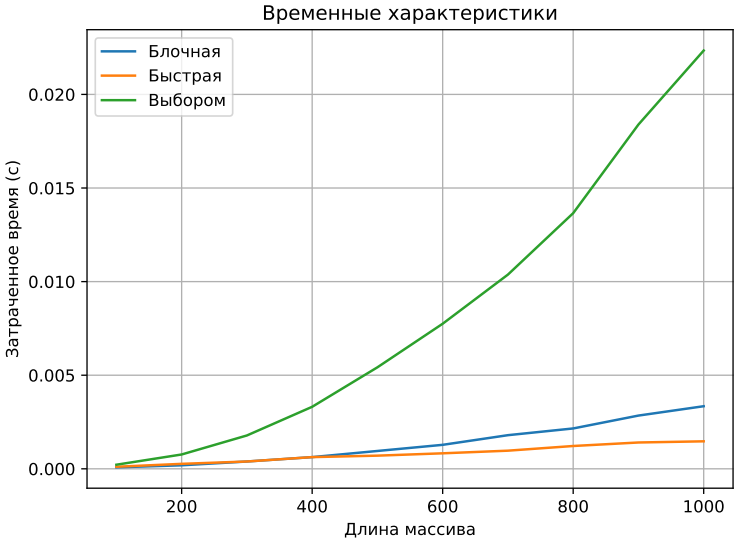

Рисунок 4.4 – Сравнение реализаций алгоритмов по времени выполнения на отсортированных массивах

4.4 Характеристики по памяти

Введем следующие обозначения:

— n — длина массива, который необходимо отсортировать arr;

— $size()$ — функция, вычисляющая размер в байтах;

— int — целочисленный тип данных;

— $float$ — вещественный тип данных.

Максимальное требование по памяти реализации алгоритма Шелла складывается из 5 локальных переменных типа int, адреса возврата int, возвращаемого значения (ссылки) и рассчитывается по формуле (4.1).

$$f_{memshell} = 6 \cdot size(int) + size(float*). \tag{4.1}$$

Аналогично, реализация алгоритма гномьей сортировки максимально требует памяти под 2 локальные переменные типа int, адрес возврата int, возвращаемое значение (ссылку), т. о. требования по памяти рассчитываются по формуле (4.2).

$$f_{memgnome} = 3 \cdot size(int) + size(float*). \tag{4.2}$$

Максимальное требование реализации алгоритма пирамидальной сортировки по памяти формируется из 3 локальных переменных типа int, адреса возврата int, возвращаемого по ссылке значения, при этом максимальная глубина рекурсии подпрограммы *heapify* равна $\log_2 N$. В подпрограмме *heapify* используются 3 локальных переменных типа int, а для вызова в нее необходимо передать массив по ссылке и 2 переменные типа int. Память требуемая реализацией алгоритма пирамидальной сортировки рассчитывается по формуле (4.3).

$$f_{heap} = 3 \cdot size(int) + size(float*) + \log_2 N(6 \cdot size(int) + size(float*)). \tag{4.3}$$

Вывод

В результате замеров времени выполнения реализаций различных алгоритмов было выявлено, что для массивов длины 9000, отсортированных в обратном порядке, реализация алгоритма Шелла по времени оказалась в 2.6 раза лучше, чем реализация гномьей сортировки, и в 4.5 раза лучше реализации пирамидальной сортировки. В свою очередь, реализация гномьей сортировки оказалась лучше в 1.7 раз по времени выполнения, чем реализация пирамидальной сортировки. Что соответствует теоретической оценке трудоемкости. Поскольку алгоритм Шелла по теоретической оценке трудоемкости в худшем случае выигрывает по константе гномью сортировку, $O(\frac{32}{3}N^2)$ и $O(\frac{23}{2}N^2)$ соответственно, при этом алгоритм пирамидальной сортировки, обладая теоретической оценкой трудоемкости $O(\frac{87}{2}N\log_2 N)$, из-за большой константы проигрывает остальным, хоть и обладает меньшей скоростью роста.

Для отсортированных массивов длинной 9000 реализация гномьей сортировки оказалась лучше по времени в 9 раза, чем реализация алгоритма Шелла, и в 42 раза лучше, чем реализация пирамидальной сортировки. В свою очередь, реализация пирамидальной сортировки на отсортированных массивах, оказалась хуже в 4.5 раза, чем реализации алгоритма Шелла по времени выполнения. Что соответствует теоретической оценке трудоемкости. Поскольку в лучшем случае алгоритм гномьей сортировки обладает наименьшей асимптотической оценкой и малой константой $O(7N)$. А алгоритм Шелла выигрывает алгоритм пирамидальной сортировки по константе, $O(10N\log_2 N)$ и $O(29N\log_2 N)$ соответственно.

Для случайно упорядоченных массивов длинной 9000 реализация гномьей сортировки оказалась лучше по времени в 1.4 раза, чем реализация алгоритма Шелла, и в 5.3 раза лучше, чем реализация пирамидальной сортировки. В свою очередь, реализация пирамидальной сортировки на случайно упорядоченных массивах, оказалась хуже в 3.9 раз, чем реализации алгоритма Шелла по времени выполнения.

При этом меньше всего памяти требует реализация гномьей сортировки, а больше всего — реализация пирамидальной сортировки.

Стоит заметить, что для обратно упорядоченных массивов длинной менее 2500, реализация гномьей сортировки показывала лучшие результаты. На случайно упорядоченных массивах, гномья сортировка хоть и оказалась

эффективней, но обладая большей скоростью роста, при больших длинах массивов она окажется менее эффективной, чем сортировка Шелла и пирамидальная. То же касается и пирамидальной сортировки, за счет большой константы, она показала худший результат во всех случаях, но поскольку асимптотическая оценка этого алгоритма меньше остальных для случайно упорядоченных и обратно упорядоченных массивов, данная реализация покажет лучшую эффективность по времени при гораздо больших длинах массивов.

ЗАКЛЮЧЕНИЕ

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

Для достижения поставленной цели были выполнены следующие задачи.

— описаны алгоритмы гномьей, пирамидальной сортировок и сортировка по алгоритму Шелла;

— разработано программное обеспечение, реализующее алгоритмы сортировок;

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

— проведен анализ затрат реализаций алгоритмов по времени.

В результате исследования реализаций различных алгоритмов было получено, что для массивов длины 9000, отсортированных в обратном порядке, реализация алгоритма Шелла по времени оказалась в 2.6 раза лучше, чем реализация гномьей сортировки, и в 4.5 раза лучше реализации пирамидальной сортировки. В свою очередь, реализация гномьей сортировки оказалась лучше в 1.7 раз по времени выполнения, чем реализация пирамидальной сортировки. Для отсортированных массивов длинной 9000 реализация гномьей сортировки оказалась лучше по времени в 9 раза, чем реализация алгоритма Шелла, и в 42 раза лучше, чем реализация пирамидальной сортировки. В свою очередь, реализация пирамидальной сортировки на отсортированных массивах, оказалась хуже в 4.5 раза, чем реализации алгоритма Шелла по времени выполнения. Для случайно упорядоченных массивов длинной 9000 реализация гномьей сортировки оказалась лучше по времени в 1.4 раза, чем реализация алгоритма Шелла, и в 5.3 раза лучше, чем реализация пирамидальной сортировки. В свою очередь, реализация пирамидальной сортировки на случайно упорядоченных массивах, оказалась хуже в 3.9 раз, чем реализации алгоритма Шелла по времени выполнения.

Для обратно упорядоченных массивов длинной менее 2500, реализация гномьей сортировки показывала лучшие результаты. На случайно упорядоченных массивах, гномья сортировка хоть и оказалась эффективней, но обладая

большей скоростью роста, при больших длинах массивов она окажется менее эффективной, чем сортировка Шелла и пирамидальная. То же касается и пирамидальной сортировки, за счет большой константы, она показала худший результат во всех случаях, но поскольку асимптотическая оценка этого алгоритма меньше остальных для случайно упорядоченных и обратно упорядоченных массивов, данная реализация покажет лучшую эффективность по времени при гораздо больших длинах массивов.