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»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

Название: Расстояния Левенштейна и Дамерау – Левенштейна

Дисциплина: Анализ алгоритмов

Студент	ИУ7-56Б		Ковель А.Д.
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Волкова Л.Л.
Преподаватель			Строганов Ю.В.
		Подпись, дата	И. О. Фамилия

Москва, 2022 г.

Оглавление

Введение

Одной из важнейших процедур обработки структурированной информации является сортировка.

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

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

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

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

Цель лабораторной работы — реализация и исследование сортировок: бинарным деревом, поразрядной, подсчетом.

Задачи данной лабораторной:

- изучить и реализовать три алгоритма сортировки: бинарным деревом, поразрядной, подсчетом;
- провести сравнительный анализ трудоемкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
- провести сравнительный анализ алгоритмов на основе экспериментальных данных, а именно по времени;
- описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов сортировки бинарным деревом, подсчетом и поразрядная.

1.1 Сортировка бинарным деревом

Сортировка бинарным деревом [1] — деревом назовем упорядоченную структуру данных, в которой каждому элементу — предшественнику или корню (под)дерева — поставлены в соответствие по крайней мере два других элемента (преемника). Причем для каждого предшественника выполнено следующее правило: левый преемник всегда меньше, а правый преемник всегда больше или равен предшественнику. Вместо 'предшественник' и 'преемник' также употребляют термины 'родитель' и 'сын'. Все элементы дерева также называют 'узлами'.

При добавлении в дерево нового элемента его последовательно сравнивают с нижестоящими узлами, таким образом вставляя на место. Если элемент >= корня — он идет в правое поддерево, сравниваем его уже с правым сыном, иначе — он идет в левое поддерево, сравниваем с левым, и так далее, пока есть сыновья, с которыми можно сравнить.

1.2 Сортировка подсчетом

Сортировка подсчетом [2] — Сортировка подсчетом — это алгоритм сортировки на основе целых чисел для сортировки массива, ключи которого лежат в определенном диапазоне. Он подсчитывает общее количество элементов с каждым уникальным значением ключа, а затем использует эти подсчеты для определения позиций каждого значения ключа в выходных данных.

1.3 Сортировка поразрядная

Сортировка поразрядная [3]. Массив несколько раз перебирается и элементы перегруппировываются в зависимости от того, какая цифра находится в определённом разряде. После обработки разрядов (всех или почти всех) массив оказывается упорядоченным.При этом разряды могут обрабатываться в противоположных направлениях - от младших к старшим или наоборот.

Вывод

В данной работе стоит задача реализации 3 алгоритмов сортировки, а именно: бинарным деревом, подсчетом и поразрядная. Необходимо оценить теоретическую оценку алгоритмов и проверить ее экспериментально.

2 Конструкторский раздел

В данном разделе представлены схемы реализуемых алгоритмов и их модификации.

2.1 Трудоемкость алгоритмов

Для получения функции трудоемкости алгоритма необходимо ввести модель оценки трудоемкости. Трудоемкость "элементарных"операций оценивается следующим образом:

1. Трудоемкость 1 имеют операции:

$$+, -, =, <, >, <=, >=, ==, + =, - =,$$
$$++, --, [], \&\&, ||, >>, <<$$

2. Трудоемкость 2 имеют операции:

$$*, /, \backslash, \%$$

3. Трудоемкость конструкции ветвления определяется согласно формуле 2.1

$$f_{if} = f_{condition} + \begin{cases} min\left(f_{true}, f_{false}\right) & \text{в лучшем случае,} \\ max\left(f_{true}, f_{false}\right) & \text{в худшем случае.} \end{cases} \quad (2.1)$$

4. Трудоемкость цикла расчитывается по формуле 2.2

$$f_{loop} = f_{init} + f_{cmp} + N\left(f_{body} + f_{inc} + f_{cmp}\right), \quad (2.2)$$

где
f_{init} – трудоемкость инициализации,
f_{body} – трудоемкость тела цикла,
f_{iter} – трудоемкость инкремента,
f_{cmp} – трудоемкость сравнения,
N – количество повторов.

5. Трудоемкость вызова функции равна 0.

2.2 Трудоемкость алгоритмов

2.2.1 Алгоритм сортировки бинарным деревом

Трудоёмкость данного алгоритма посчитаем следующим образом: сортировка – преобразование массива в бинарное дерево поиска посредством операции вставки нового элемента в бинарное дерево. Операция вставки в бинарное дерево имеет сложность $\log_2(size)$, где $size$ - количество элементов в дереве. Для преобразования массива или списка размером $size$ потребуется использовать операцию вставки в биннарное дерево $size$ раз, таким образом, итоговая трудоёмкость данной сортировки будет равна (2.3):

$$f_{radix} = size \cdot \log_2(size) \tag{2.3}$$

2.2.2 Алгоритм сортировки подсчетом

Трудоёмкость алгоритма сортировки подсчётом, где size — количество элементов в массиве (2.4):

$$\begin{aligned} f_{count} &= size + 10 + 2 + size \cdot 8 + 2 + 10 \cdot 6 + 3 \\ &\quad + size \cdot 14 + 1 + size \cdot 5 \\ &= 78 + 28 \cdot size \end{aligned} \tag{2.4}$$

2.2.3 Алгоритм поразрядной сортировки

Трудоёмкость алгоритма поразрядной сортировки равна

$$f_{radix} = 1 + 2 + 5 \cdot size + 1 + 2 + m * (f_{count} + 1 + 2), \tag{2.5}$$

где m - количество разрядов в максимальном элементе, f_{count} - трудоёмкость алгоритма сортировки подсчётом.

Итоговая трудоёмкость порязрядной сортировки, использующей сортировку подсчётом в рамках одного разряда:

2.3 Схемы алгоритмов

На рисунке 2.3 приведена схема алгоритма сортировки бинарным деревом. На рисунке 2.1 приведена схема алгоритма сортировки подсчетом. Рисунок 2.2 демонстрируют схему алгоритма поразряндной сортировки.

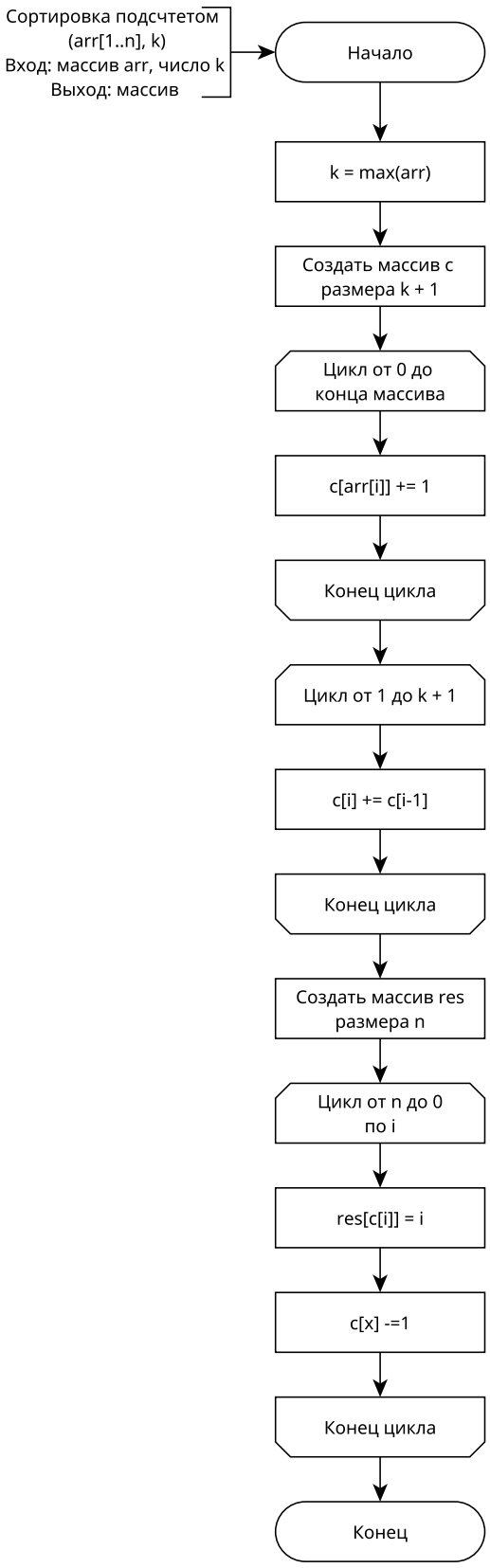

Рисунок 2.1 – Схема алгоритма сортировки подсчетом

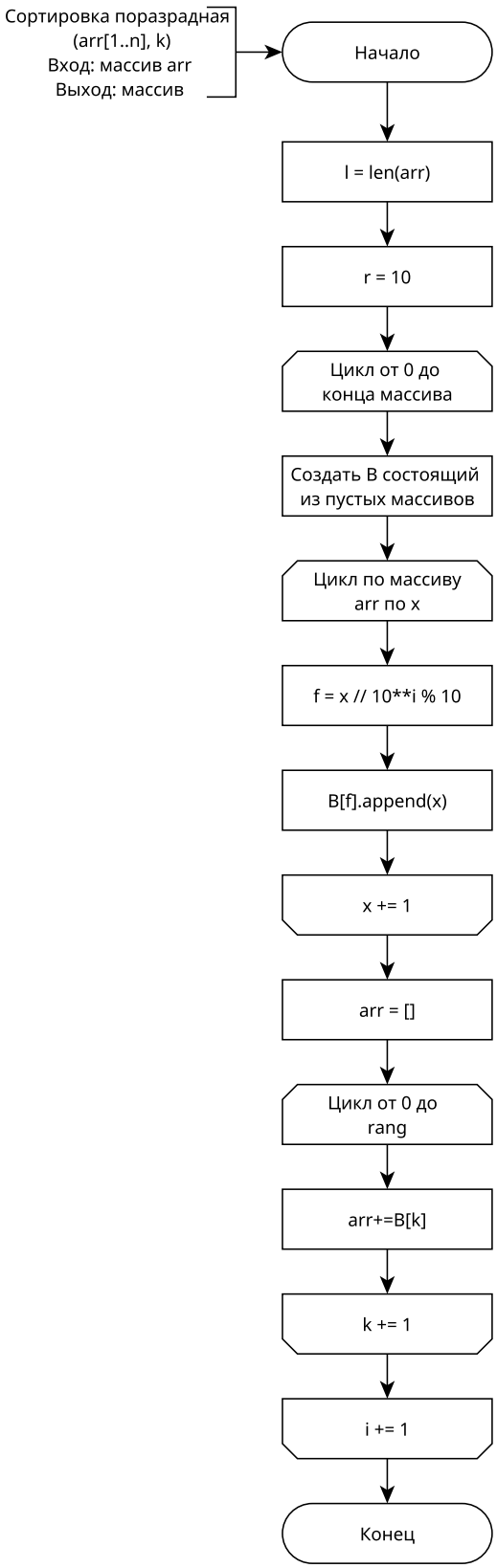

Рисунок 2.2 – Схема алгоритма поразряндной сортировки

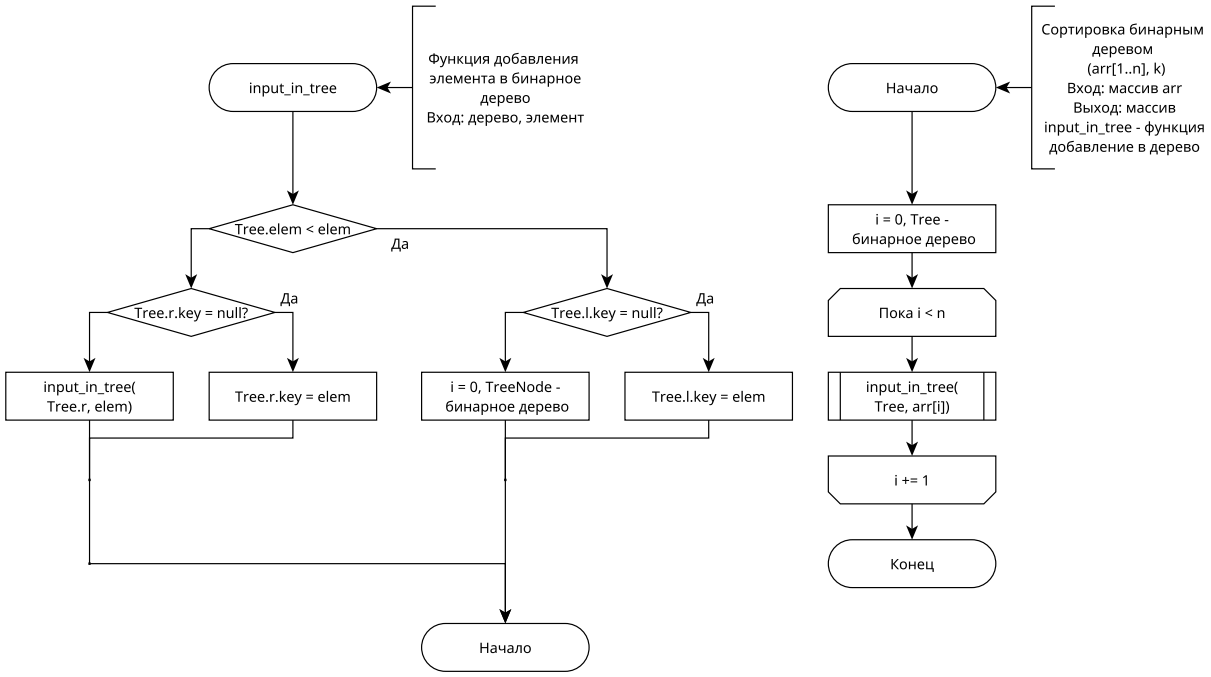

Рисунок 2.3 – Схема алгоритма сортировки бинарным деревом

2.3.1 Вывод

Были разработаны схемы всех трех алгоритмов сортировки. Также для каждого из них были рассчитаны трудоёмкости по введённой модели вычислений с учётом лучших и худших случаев.

3 Технологический раздел

В данном разделе будут приведены требования к программному обеспечению, средства реализации и листинга кода.

3.1 Требования к ПО

Программное обеспечение должно удовлетворять следующим требованиям:

- программа получает на вход с клавиатуры две матрицы размеров в пределах 10000×10000 либо получает два числа – размерность матрицы в пределах 10000;
- программа выдает матрицу - произведение двух полученных матриц;
- в программе возможно измерение процессорного времени.

3.2 Средства реализации

Для реализации ПО был выбран язык программирования Python[?].

В данном языке есть все требующиеся инструменты для данной лабораторной работы.

В качестве среды разработки была выбрана среда VS Code[?], запуск происходил через команду python main.py.

3.3 Средства замера времени

Алгоритмы тестировались при помощи функции process_time библиотеки time 3.1. Данная команда возвращает значения процессорного времени типа int в наносекундах.

Замеры времени для каждого алгоритма проводились 100 раз.

Листинг 3.1 – Пример теста эффективности

```
def test_simple_mult(A, B):
# Start the stopwatch / counter
t1_start = process_time()
for i in range(N_TEST):
simple_mult(A, M, B, N, M)
# Stop the stopwatch / counter
t1_stop = process_time()
```

3.4 Листинги кода

Листинг 3.2 демонстрирует классический алгоритм умножения.

Листинг 3.2 – Классический алгоритм умножения

```
def simple_mult(mat1, m, mat2, n, q):
res = np.zeros([m, q])
for i in range(m):
for j in range(q):
for k in range(n):
res[i][j] = res[i][j] + mat1[i][k]  mat2[k][j]return res
```

Листинг 3.3 – умножение матриц алгоритмом Винограда.

Листинг 3.3 – Алгоритм умнложения Виноградом

```
1   def precompile\_rows\_win(mat, n, m):
2   mh = np.zeros([n])
3   for i in range(n):
4   for j in range(m // 2):
5   mh[i] = mh[i] + mat[i][j  2] *mat[i][j *2 + 1]return mhdef
        precompile_cols_win(mat, n, m):mv = np.zeros([m])for i in range(m):for j in range(n
        // 2):mv[i] = mv[i] + mat[j *2][i] *mat[j *2 + 1][i]return mvdef winograd_mult(A, m,
        B, n, q):res = np.zeros([m, q])mh = precompile_rows_win(A, m, n)mv =
        precompile_cols_win(B, n, q)for i in range(m):for j in range(q):res[i][j] = -mh[i] -
        mv[j]for k in range(n // 2):res[i][j] = res[i][j] + (A[i][k*2] + B[k*2+1][j])*(A[i][k*2+1]
        + B[k*2][j])if n % 2 != 0:for i in range(n):for j in range(m):res[i][j] = res[i][j] +
        A[i][n-1]*B[n-1][j]return res
```

Листинг 3.4 – умножение оптимизированным алгоритмом Винограда.

Листинг 3.4 – Оптимизированный алгоритм умножения Виноградом

```
1  def precompile_rows_win_opt(mat, n, m):
2  mh = np.zeros([n])
3  opt = m // 2
4  for i in range(n):
5  for j in range(opt):
6  t = j << 1
7  mh[i] += mat[i][t] mat[i][t + 1]return mhdef precompile_colswin(mat, n, m) : mv =
       np.zeros([m])opt = n//2foriinrange(m) : forjinrange(opt) : t = j << 1mv[i]+ =
       mat[t][i] * mat[t + 1][i]returnmvdefwinograd_mult_opt(A, m, B, n, q) : res =
       np.zeros([m, q])mh = precompile_rows_win(A, m, n)mv =
       precompile_colswin(B, n, q)opt = n//2foriinrange(m) : forjinrange(q) : res[i][j] =
       −mh[i] − mv[j]forkinrange(n//2) : t = k << 1res[i][j]+ =
       (A[i][t] + B[t + 1][j]) * (A[i][t + 1] + B[t][j])ifnforiinrange(n) : forjinrange(m) :
       res[i][j]+ = A[i][n − 1] * B[n − 1][j]returnres
```

3.5 Тестовые данные

Таблица 3.1 – Тестовые случаи

<table>
<tr><th rowspan="2">№</th><th rowspan="2">Вход. матрица №1</th><th rowspan="2">Вход. матрица №2</th><th colspan="3">Результат</th></tr>
<tr><th>Классический</th><th>Виноград</th><th>Виноград оптимизированный</th></tr>
<tr><td>1</td><td>[0.6]
−3 5 −1 7
−8 2 −2 1
0 −3 −4 0
−6 0 5 1</td><td>[0.6]
−3 5 −1 7
−8 2 −2 1
0 −3 −4 0
−6 0 5 1</td><td>[0.6]
−73 −2 32 −9
2 −30 17 −53
24 6 22 −3
12 −45 −9 −41</td><td>[0.6]
−73 −2 32 −9
2 −30 17 −53
24 6 22 −3
12 −45 −9 −41</td><td>[0.6]
−73 −2 32 −9
2 −30 17 −53
24 6 22 −3
12 −45 −9 −41</td></tr>
<tr><td>2</td><td>[0.6]
0 0 −2
2 3 2
−4 0 −1</td><td>[0.6]
0 0 −2
2 3 2
−4 0 −1</td><td>[0.6]
8 0 2
−2 9 0
4 0 9</td><td>[0.6]
8 0 2
−2 9 0
4 0 9</td><td>[0.6]
8 0 2
−2 9 0
4 0 9</td></tr>
<tr><td>3</td><td>[0.6]
5 7
7 0
3 1</td><td>[0.6]
5 7 7 0
3 1 −7 5</td><td>[0.6]
46 42 −14 35
35 49 49 0
18 22 14 5</td><td>[0.6]
46 42 −14 35
35 49 49 0
18 22 14 5</td><td>[0.6]
46 42 −14 35
35 49 49 0
18 22 14 5</td></tr>
<tr><td>4</td><td>[0.6]
−4 9
−4 −1
−1 5
5 3</td><td>[0.6]
−4 9 −4
−1 −1 5</td><td>[0.6]
7 −45 61
17 −35 11
−1 −14 29
−23 42 −5</td><td>[0.6]
7 −45 61
17 −35 11
−1 −14 29
−23 42 −5</td><td>[0.6]
7 −45 61
17 −35 11
−1 −14 29
−23 42 −5</td></tr>
</table>

3.6 Вывод

Было написано и протестировано программное обеспечение для решения поставленной задачи.

4 Исследовательская часть

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система Pop!_OS 22.04 LTS [**?**] Linux [**?**];
- Оперативная память 16 Гб;
- Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [**?**].

Во время тестирования устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений окружения, окружением и системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

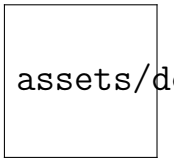

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Результаты профилирования алгоритмов приведены в таблице 4.1. Результаты тестирования приведены в таблице 3.1.

4.4 Графики функций

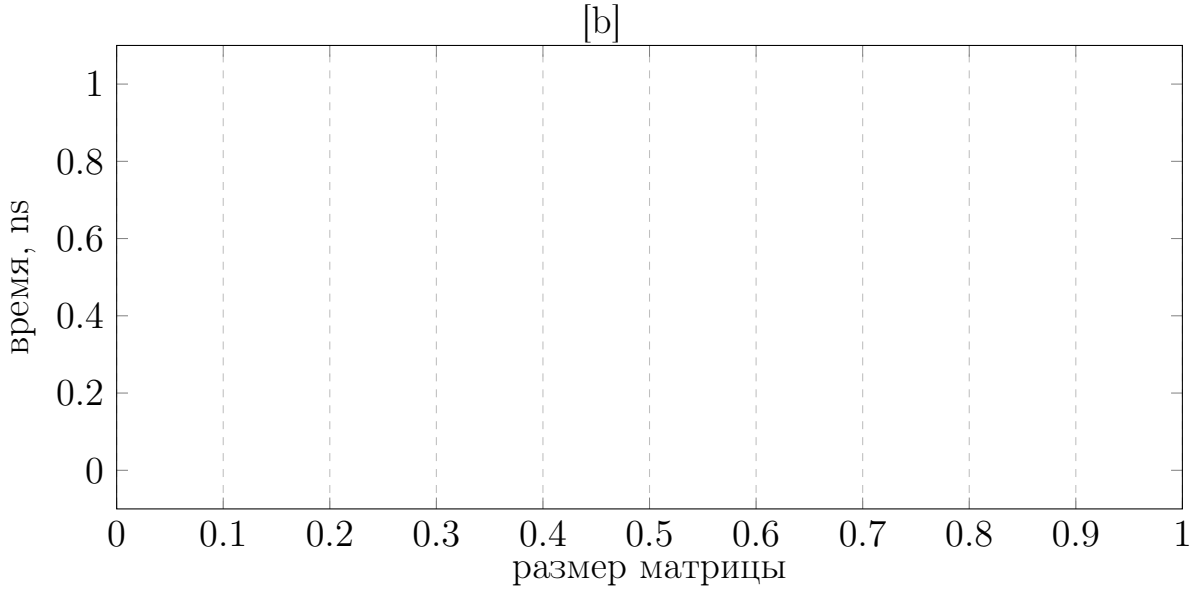

Рисунок 4.2 – Четная размерность матрицы

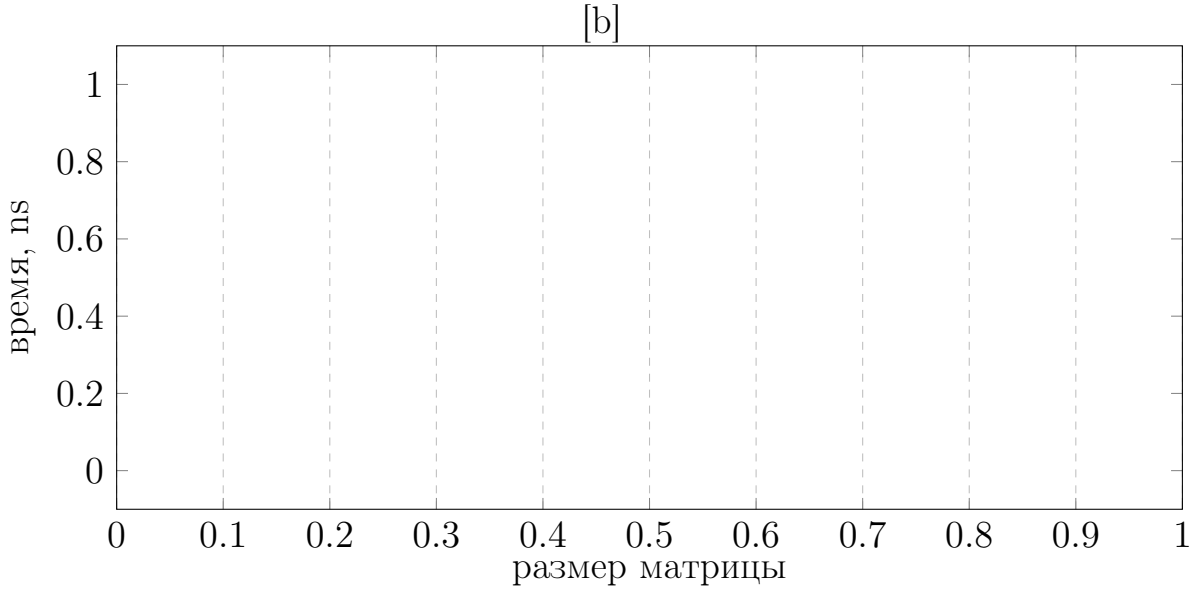

Рисунок 4.3 – Нечетная размерность матрицы

Таблица 4.1 – Время выполнения алгоритмов

[ht!]0.45

Таблица 4.2 – Четная размерность матриц

n	Время, ns		
	Класс.	Виноград	Вин. опт.
10	4.49e-06	3.91e-06	4.01e-06
20	2.74e-05	2.24e-05	2.30e-05
30	8.84e-05	6.72e-05	7.01e-05
40	2.09e-04	1.49e-04	1.58e-04
50	3.96e-04	2.98e-04	3.14e-04
60	6.58e-04	4.95e-04	5.15e-04
70	1.07e-03	7.95e-04	8.43e-04
80	1.60e-03	1.20e-03	1.22e-03
90	2.26e-03	1.68e-03	1.77e-03
100	3.14e-03	2.25e-03	2.38e-03
150	1.06e-02	7.88e-03	8.33e-03
200	2.84e-02	2.20e-02	2.34e-02
250	6.73e-02	5.34e-02	5.74e-02
300	1.14e-01	8.97e-02	9.60e-02
350	1.80e-01	1.39e-01	1.50e-01
400	2.67e-01	2.05e-01	2.21e-01
450	4.17e-01	3.23e-01	3.48e-01
500	5.71e-01	4.42e-01	4.75e-01

[ht!]0.45

Таблица 4.3 – Нечетная размерность матриц

n	Время, ns		
	Класс.	Виноград	Вин. опт.
11	5.17e-06	4.56e-06	5.05e-06
21	3.25e-05	2.58e-05	2.59e-05
31	9.63e-05	7.23e-05	7.69e-05
41	2.14e-04	1.64e-04	1.73e-04
51	4.10e-04	3.15e-04	3.38e-04
61	7.13e-04	5.37e-04	5.70e-04

Вывод

В данном разделе были сравнены алгоритмы по времени. Оптимизированный алгоритм Винограда является самым быстрым, за счет проведенных изменений в стандартном алгоритме Винограда.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- были изучены и реализованы 3 алгоритма перемножения матриц: обычный, Копперсмита-Винограда, модифицированный Копперсмита-Винограда;
- был произведен анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
- был сделан сравнительный анализ алгоритмов на основе экспериментальных данных;
- подготовлен отчет о лабораторной работе.

Оптимизированный алгоритм Винограда быстрее обычного на 5 (на 0.1 наносекунду) процентов при размерах матрицы 500 на 500.

Поставленная цель достигнута.

Литература

[1] Сортировка бинарным деревом [Электронный ресурс]. Режим доступа: http://algolist.ru/sort/faq/q7.php (дата обращения: 24.10.2022).

[2] Сортировка подсчетом [Электронный ресурс]. Режим доступа: https://www.techiedelight.com/ru/counting-sort-algorithm-implementation/ (дата обращения: 24.10.2022).

[3] Сортировка по разрядам [Электронный ресурс]. Режим доступа: http://algolab.valemak.com/radix (дата обращения: 24.10.2022).