Государственное образовательное учреждение высшего профессионального образования
“Московский государственный технический университет имени Н.Э.Баумана”

Дисциплина: Анализ алгоритмов

Лабораторная работа №2

Исследование сложности алгоритмов умножения матриц

Студент группы ИУ7-55Б,
Руднев К. К.,

Преподаватель,
Волкова Л. Л.,
Строганов Ю. В.

2019 г.

Оглавление

Введение

Цель работы: изучение алгоритмов умножения матриц. В данной лабораторной работе рассматривается стандартный алгоритм умножения матриц, алгоритм Винограда и оптимизированный алгоритм Винограда. Также требуется изучить расчет сложности алгоритмов, получить навыки в улучшении алгоритмов.

Задачи:

- изучить алгоритмы умножения матриц: стандартный и алгоритм Винограда;
- оптимизировать алгоритм Винограда;
- дать теоритическую оценку базового алгоритма матриц, алгоритма Винограда и оптимизированного алгоритма Винограда;
- реализовать алгоритмы умножения матриц на одном из языков программирования;
- сравнить время работы алгоритмов.

Глава 1

Аналитическая часть

В рамках раздела будет дано аналитическое описание алгоритма для умножения матриц стандартным методом и по Винограду.

1.1 Описание алгоритмов

Пусть даны две прямоугольные матрицы A[MxQ] и B[QxN]:

$$A = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1Q} \\ a_{21} & a_{22} & ... & a_{1Q} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ a_{M1} & a_{M2} & ... & a_{MQ} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} & ... & b_{1N} \\ b_{21} & b_{22} & ... & b_{1N} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ b_{Q1} & b_{Q2} & ... & b_{QN} \end{bmatrix}.$$

Тогда матрица C[MxN]:

$$C = \begin{bmatrix} c_{11} & c_{12} & ... & c_{1N} \\ c_{21} & c_{22} & ... & c_{1N} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ c_{M1} & c_{M2} & ... & c_{MN} \end{bmatrix}, \text{ в которой:}$$

$$c_{ij} = \sum_{r=1}^{m} a_{ir} * b_{rj} \quad (i = 1, 2, ...M; j = 1, 2, ...N). \quad (1.1)$$

называется их произведением [1].

Алгоритм Винограда

Пусть даны две прямоугольные матрицы A[1xQ] и B[Qx1]:

$$U = \begin{bmatrix} u_1 & u_2 & ... & u_Q \end{bmatrix}, V = \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_Q \end{bmatrix}.$$

Тогда матрица C[1x1] = A*B по формуле 1.2 [2]:

$$C_{ij} = u_1 * v_1 + u_1 * v_1 + ... + u_Q * v_Q = \\ (u_1 + v_2) \cdot (u_2 + v_1) + ... + (u_{Q-2} + v_{Q-2}) * (u_Q + v_Q) - \\ u_1 * u_2 - u_{Q-3} * u_{Q-2} - u_{Q-1} * u_Q - v_1 * v_2 - \\ v_{Q-3} * v_{Q-2} - v_{Q-1} * v_Q \quad (1.2)$$

В случае, если матрица имеет нечетную размерность, необходимо отдельно обработать последний вектор (формула 1.3):

$$C_{ij} = u_{in-1} * v_{n-1j} \quad (1.3)$$

1.2 Вывод

Были рассмотрены алгоритмы классического умножения матриц и алгоритм Винограда, основное отличие которых - наличие предварительной обработки, что позволяет снизить количество операций умножения.

Глава 2

Конструкторская часть

В рамках раздела на рисунках 2.1-2.7 будут представлены схемы алгоритмов стандартного умножения и алгоритма умножения матриц по Винограду.

2.1 Разработка алгоритмов

Схема стандартного алгоритма умножения матриц представлена на рисунках 2.1-2.2.
Схема алгоритма умножения матриц по Винограду представлена на рисунках 2.3-2.7.

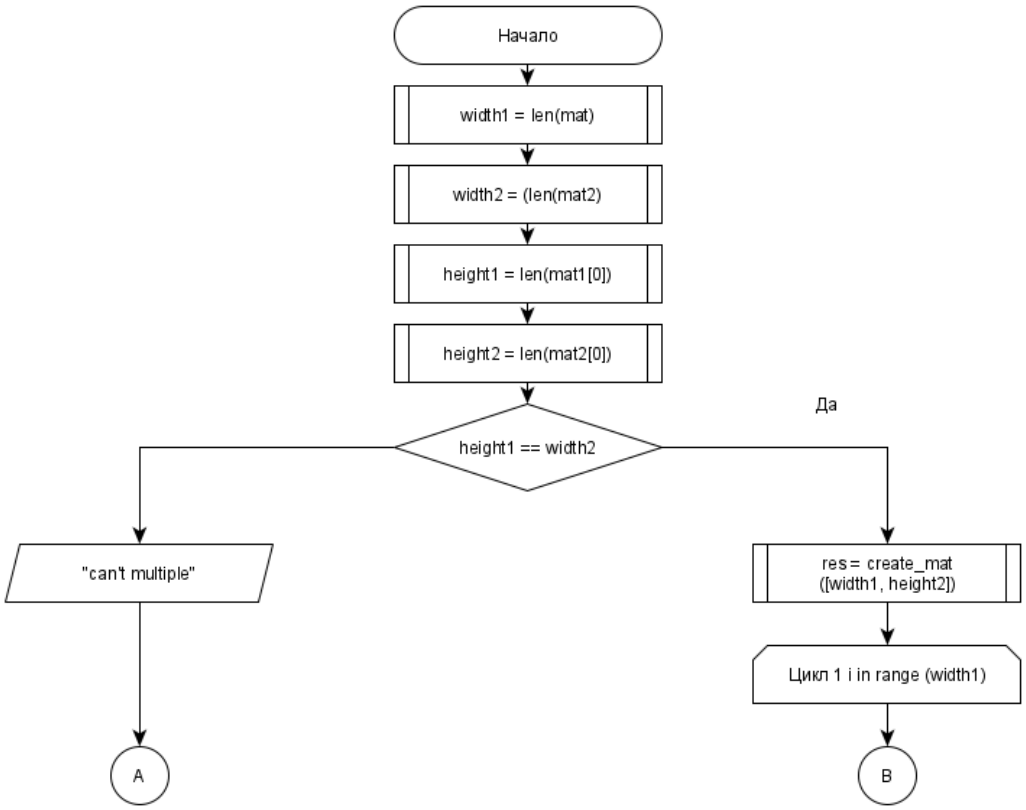

Рис. 2.1: Стандартный алгоритм умножения матриц. Часть 1

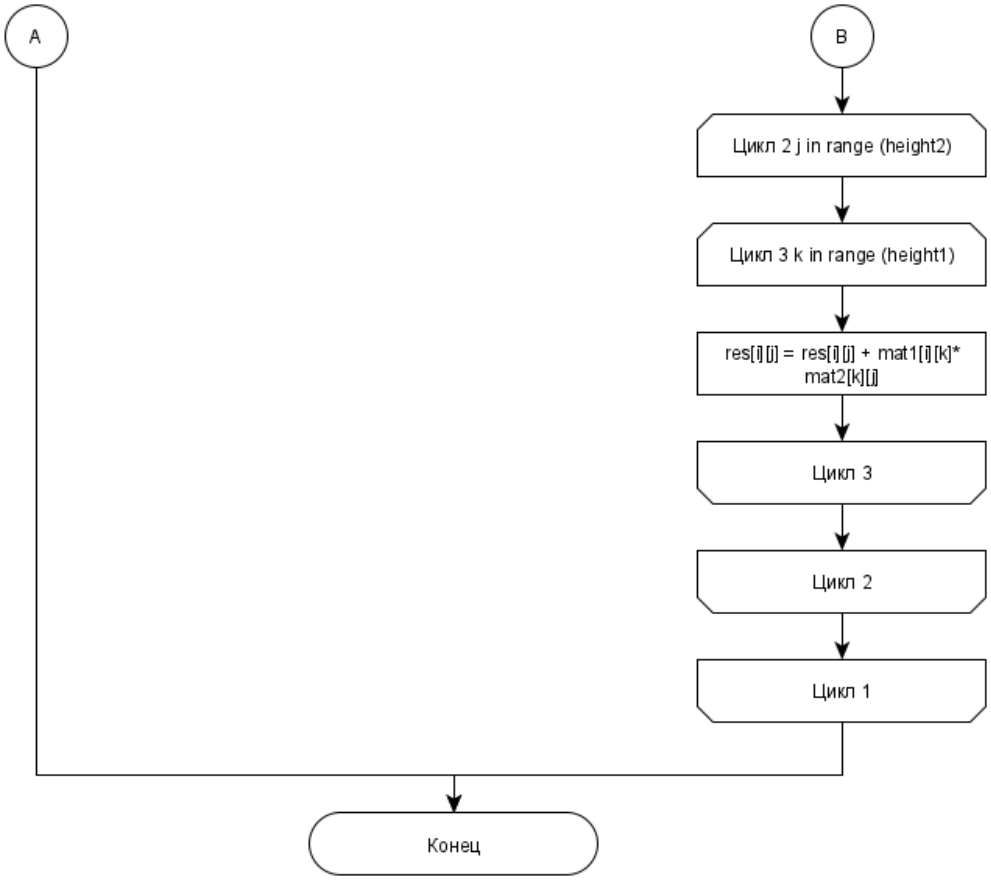

Рис. 2.2: Стандартный алгоритм умножения матриц. Часть 2

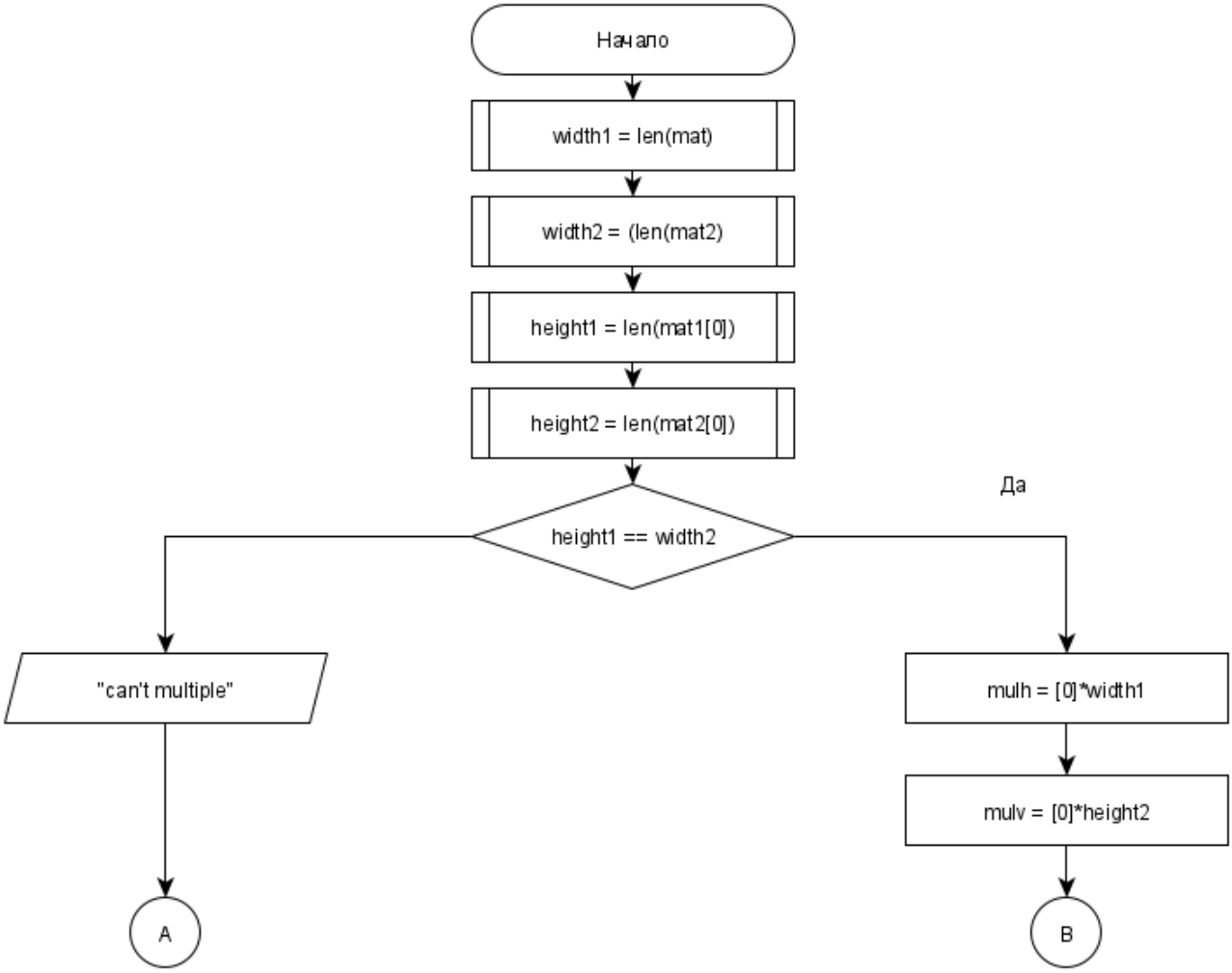

Рис. 2.3: Алгоритм Винограда умножения матриц. Часть 1

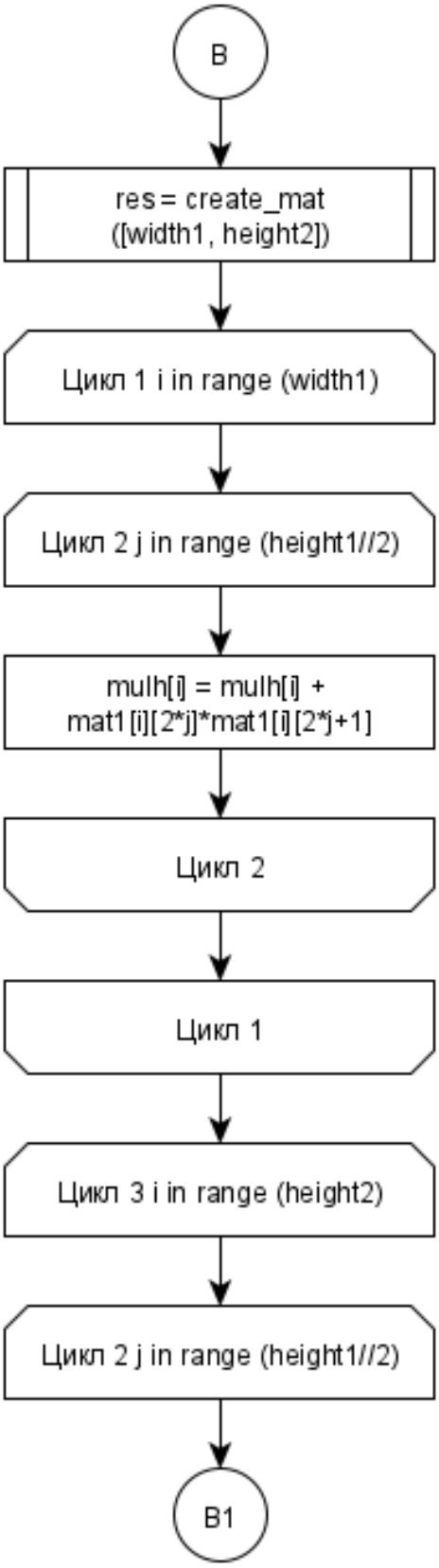

Рис. 2.4: Алгоритм Винограда умножения матриц. Часть 2

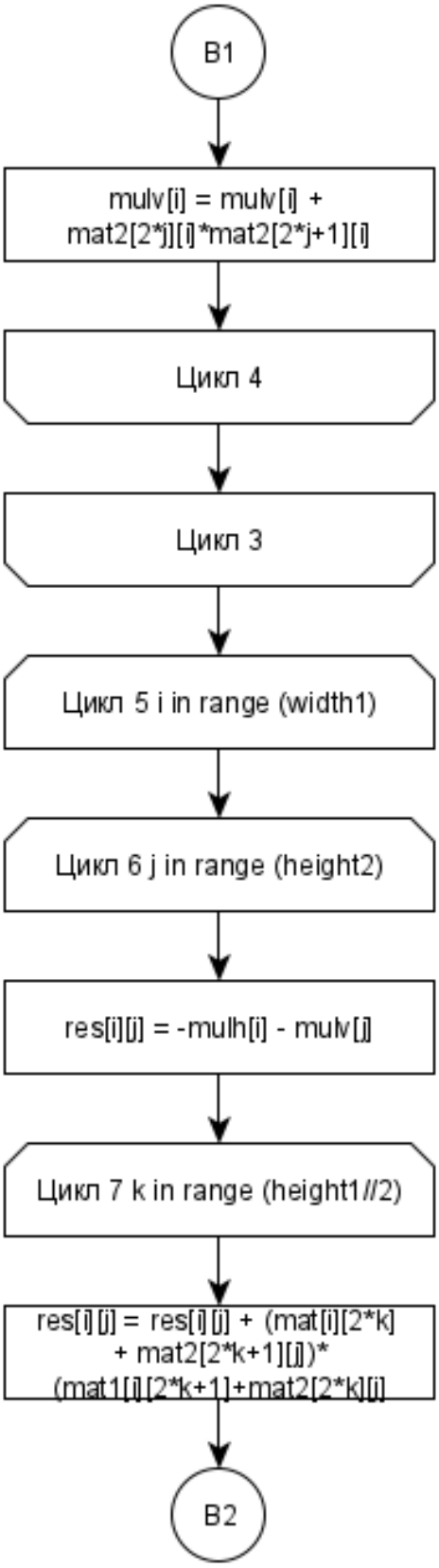

Рис. 2.5: Алгоритм Винограда умножения матриц. Часть 3

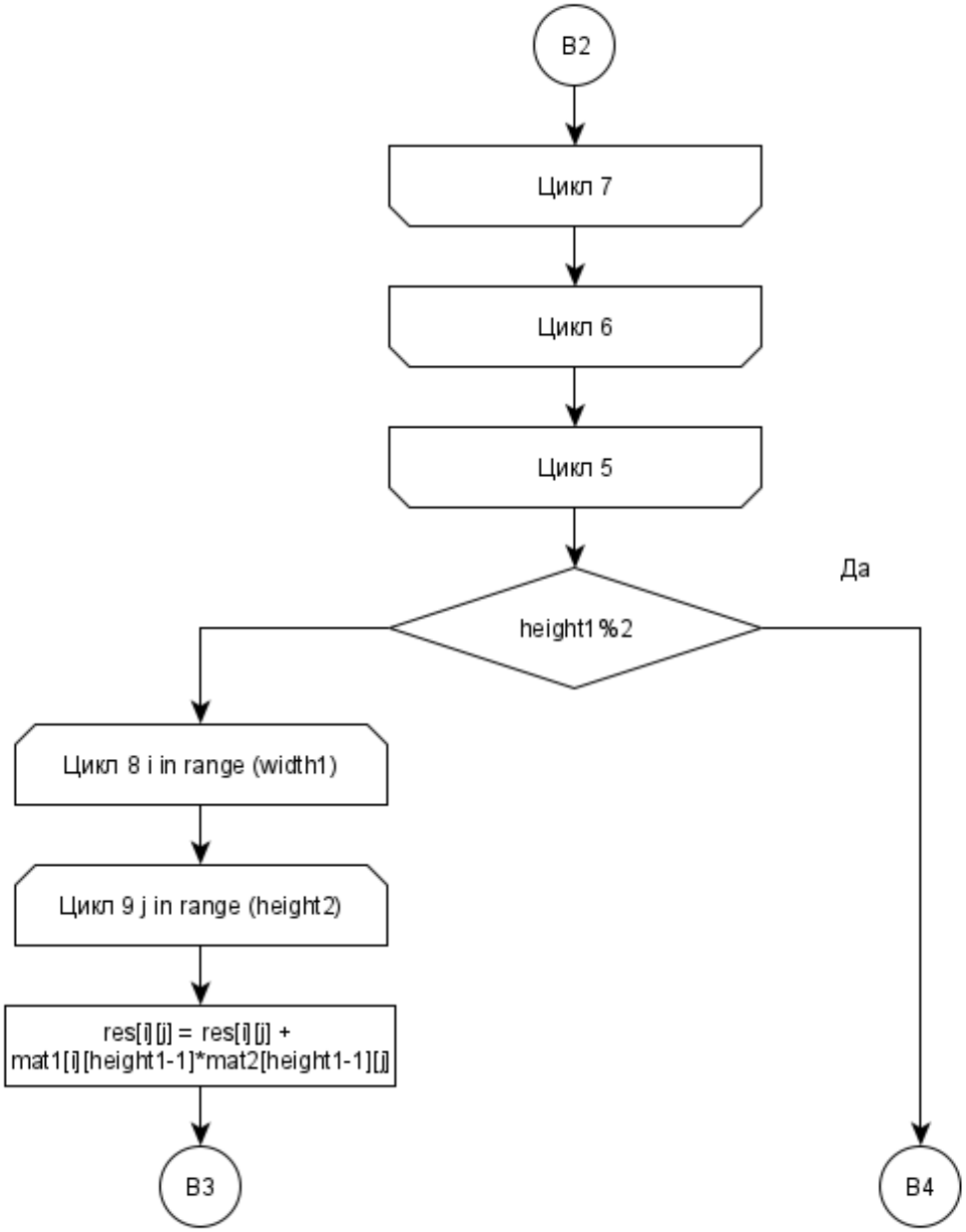

Рис. 2.6: Алгоритм Винограда умножения матриц. Часть 4

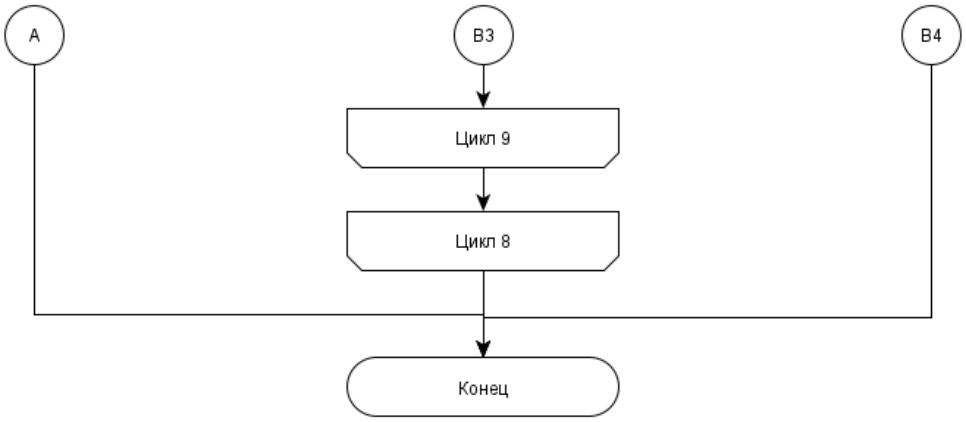

Рис. 2.7: Алгоритм Винограда умножения матриц. Часть 5

2.2 Вывод

В рамках раздела были рассмотрены схемы алгоритм умножения классически и по Винограду с целью их дальнейшего переноса в программу.

Глава 3

Технологическая часть

В рамках раздела будут описаны инструментарии разработки, выбор среды, требования к ПО. Также будут предоставлены листинги конкретных реализаций алгоритмов.

Замеры времени были произведены на: Intel(R) Core(TM) i3-6006U, 2 ядра, 4 логических процессоров.

3.1 Средства реализации

Для реализации алгоритмов использовался язык программирования Python 3.8.0 и среда разработки PyCharm Community Edition 2019.3.1 by JetBrains. У меня есть определенный опыт работы с данным языком, которого будет достаточно для реализации текущей лабораторной работы, а среда разработки имеет бесплатную комьюнити версию и удобный интерфейс, упрощающий разработку приложения/скрипта.

Замер времени реализован с помощью функции process_time() библиотеки time. Измеряется время исполнения кода чистого алгоритма (без учета времени на создание матриц, генерацию данных и т.п.).

3.2 Требования к программному обеспечению

На вход программа должна получать две матрицы, для которых вычисляется их произведение тремя алгоритмами (стандартный алгоритм умножения матриц, алгоритм умножения матриц по Винограду, а также оптимизированный алгоритм умножения матриц по Винограду).

На выход программа должна выдавать результирующую матрицу всеми тремя алгоритмами, должна корректно обрабатывать исключительные ситуации вроде невозможности произведения данных матриц.

3.3 Листинг кода

На листинге 3.1 представлены дополнительные функции и объявления, необходимые для реализации алгоритмов.

На листинге 3.2 представлена реализация алгоритма Винограда.

На листинге 3.3 представлена реализация оптимизированного алгоритма Винограда.

На листинге 3.4 представлена реализация стандартного алгоритма.

Листинг 3.1: Вспомогательные функции и объявления

```
import random
import time
```

```
def print_matrix(mat):
        if mat:
                print("")
        for i in range (len(mat)):
                print(mat[i])

def create_mat(size):
        width = size[0]
        height = size[1]

        res = []
        for i in range (width):
                res.append([0]*(height))

        return res

def generate_mat(size):
        mat = create_mat(size)
        random.seed()

        for i in range (size[0]):
                for j in range (size[1]):
                        mat[i][j] = random.randint(0, 10)

        return mat
```

Листинг 3.2: Алгоритм Винограда

```
def vinograd(mat1, mat2):
        width1 = len(mat1)
        width2 = len(mat2)

        height1 = len(mat1[0])
        height2 = len(mat2[0])

        if height1 != width2:
                print("\nCan't multiplex given matrixes")
                return

        mulh = [0] * (width1)
        mulv = [0] * (height2)

        res = create_mat([width1, height2])
        for i in range (width1):
                for j in range (height1//2):
                        mulh[i] = mulh[i] + mat1[i][2*j] * mat1[i][2*j+1]

        for i in range (height2):
                for j in range (height1//2):
                        mulv[i] = mulv[i] + mat2[2*j][i] * mat2[2*j+1][i]

        for i in range (width1):
```

```
                for j in range (height2):
                        res[i][j] = -mulh[i] -mulv[j]
                        for k in range (height1//2):
                                res[i][j] = res[i][j] + (mat1[i][2*k]+
                                        mat2[2*k+1][j])*(mat1[i][2*k+1]+
                                        mat2[2*k][j])

        if height1%2:
                for i in range (width1):
                        for j in range (height2):
                                res[i][j] = res[i][j] + mat1[i][height1-1] *
                                        mat2[height1-1][j]

        return res
```

Листинг 3.3: Оптимизированный алгоритм Винограда

```
def vinograd_optimized(mat1, mat2):
        width1 = len(mat1)
        width2 = len(mat2)

        height1 = len(mat1[0])
        height2 = len(mat2[0])

        if height1 != width2:
                print("\nCan't multiplex given matrixes")
                return

        mulh = [0] * (width1)
        mulv = [0] * (height2)
        high_value = (height1//2)*2

        res = create_mat([width1, height2])
        for i in range (width1):
                for j in range (0, high_value, 2):
                        mulh[i] -= mat1[i][j]*mat1[i][j+1]

        for i in range (height2):
                for j in range (0, high_value, 2):
                        mulv[i] -= mat2[j][i]*mat2[j+1][i]

        for i in range (width1):
                for j in range (height2):
                        res[i][j] = mulh[i] + mulv[j]

                        buffer = 0
                        for k in range (0, high_value, 2):
                                buffer += (mat1[i][k]+mat2[k+1][j])*
                                        (mat1[i][k+1]+mat2[k][j])
                        res[i][j] += buffer

                        if height1 % 2:
                                res[i][j] += mat1[i][height1-1]*
```

```
                                                mat2[height1-1][j]

        return res
```

Листинг 3.4: Стандартный алгоритм умножения матриц

```
def based(mat1, mat2):
        width1 = len(mat1)
        width2 = len(mat2)

        height1 = len(mat1[0])
        height2 = len(mat2[0])

        if height1 != width2:
                print("\nCan't multiplex given matrixes")
                return

        res = create_mat([width1, height2])
        for i in range (width1):
                for j in range (height2):
                        for k in range (height1):
                                res[i][j] = res[i][j] + mat1[i][k]*mat2[k][j]

        return res
```

3.4 Вывод

В рамках раздела были предъявлены требования к программному обеспечению. На основании их были разработаны и представлены конкретные реализации всех трёх алгоритмов умножения матриц.

Глава 4

Экспериментальная часть

В рамках раздела на рисунках 4.1-4.2 будут представлены результаты работы программы. Будут проведены эксперименты по вычислению времени выполнения алгоритмов, результаты которых представлены на рисунке 4.3.

4.1 Примеры работы

```
rows1: 3
columns1: 4
rows2: 4
columns2: 8

[9, 10, 4, 7]
[4, 2, 5, 0]
[7, 4, 10, 5]
first matrix

[5, 10, 5, 8, 1, 7, 0, 7]
[1, 10, 6, 10, 2, 4, 7, 10]
[5, 7, 7, 5, 2, 3, 5, 7]
[8, 6, 9, 6, 10, 1, 7, 10]
second matrix

[131, 260, 196, 234, 107, 122, 139, 261]
[47, 95, 67, 77, 18, 51, 39, 83]
[129, 210, 174, 176, 85, 100, 113, 209]
based result

[131, 260, 196, 234, 107, 122, 139, 261]
[47, 95, 67, 77, 18, 51, 39, 83]
[129, 210, 174, 176, 85, 100, 113, 209]
vinograd result

[131, 260, 196, 234, 107, 122, 139, 261]
[47, 95, 67, 77, 18, 51, 39, 83]
[129, 210, 174, 176, 85, 100, 113, 209]
vinograd optimized result
```

Рис. 4.1: Результат работы программы при корректных исходных данных

```
rows1: 7
columns1: 4
rows2: 2
columns2: 9

[3, 9, 10, 4]
[2, 7, 10, 10]
[8, 4, 3, 10]
[1, 7, 6, 8]
[3, 0, 6, 5]
[9, 0, 0, 0]
[3, 6, 9, 2]
first matrix

[2, 6, 10, 10, 7, 7, 8, 3, 3]
[4, 3, 0, 1, 5, 2, 9, 6, 6]
second matrix

Can't multiplex given matrixes
based result

Can't multiplex given matrixes
vinograd result

Can't multiplex given matrixes
vinograd optimized result
```

Рис. 4.2: Результат работы программы при некорректных исходных данных

4.2 Сравнительный анализ на основе экспериментальных данных

В ходе эксперимента был проведен замер времени для всех реализаций алгоритмов на размерностях матриц 100x100,...,1000x1000 и 101x101,...,1001x1001.

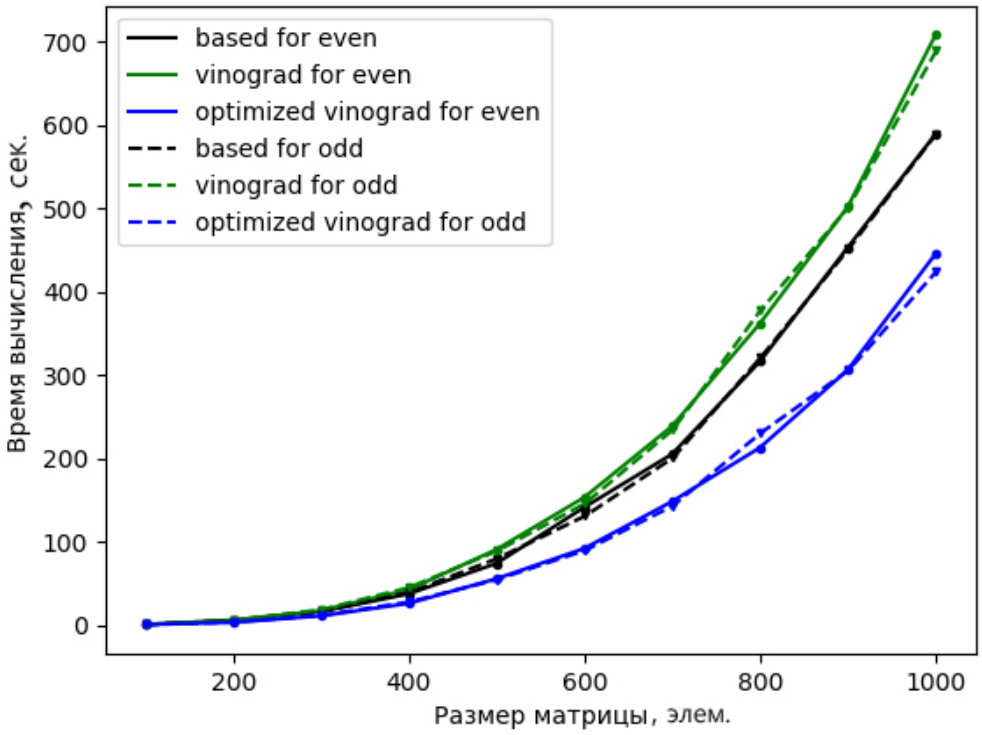

Рис. 4.3: Сравнение времени выполнения алгоритмов на разных входных данных

4.3 Оценка трудоёмкости

Считаем, что перемножаются матрицы размерами (width1, height1) и (width2,height2), height1 == width2 в случае выполнения произведения.

4.3.1 Модель оценки трудоёмкости

Введём систему оценки трудоемкости.

1. Объявление переменной/массива без определения имеет трудоёмкость 0
2. Операторы +,-,*,/,=, а также +=,-=,// имеют трудоёмкость 1
3. Оператор доступа по индексу [] имеет трудоёмкость 1
4. Логические операции имеют трудоёмкость 1
5. Цикл имеет трудоёмкость 2+n(2+t), где n - кол-во итераций, t-трудоёмкость тела цикла

4.3.2 Стандартный алгоритм

Расчёт трудоёмкости для стандартного алгоритма.

$$F = 2 + width1 * (2 + 2 + height2 * (2 + 2 + height1 * (2 + 8 + 1 + 1 + 1))) =$$
$$2 + width1 * (4 + height2 * (4 + 13 * height1)) =$$
$$2 + width1 * (4 + 4 * height2 + 13 * height2 * height1) =$$
$$2 + 4 * width1 + 4 * width1 * height2 + 13 * width1 * height1 * height2 \sim O(n^3)$$

4.3.3 Алгоритм Винограда

Расчёт трудоёмкости для алгоритма Винограда.

$$F_1 = 2 + width1 * (2 + 3 + height1//2 * (3 + 6 + 1 + 2 + 3)) =$$
$$2 + width1 * (5 + height1//2 * (15)) =$$
$$2 + width1 * (5 + 15 * height1//2) =$$
$$2 + 5 * width1 + 15 * width1 * height1//2$$

$$F_2 = 2 + height2 * (2 + 3 + height1//2 * (3 + 6 + 1 + 2 + 3)) =$$
$$2 + height2 * (5 + height1//2 * (15)) =$$
$$2 + height2 * (5 + 15 * height1//2) =$$
$$2 + 5 * height2 + 15 * height2 * height1//2$$

$$F_3 = 2 + width1 * (2 + 2 + height2 * (2 + 4 + 1 + 2 + 3 + height1//2 * (3 + 12 + 1 + 5 + 5))) =$$
$$2 + width1 * (4 + height2 * (12 + 26 * height1//2)) =$$
$$2 + width1 * (4 + 12 * height2 + 26 * height2 * height1//2) =$$
$$2 + 4 * width1 + 12 * width1 * height2 + 26 * width1 * height2 * height1//2$$

$$F_4 = \begin{bmatrix} 1 \\ 3 + width1 * (2 + 2 + height2 * (2 + 8 + 1 + 3 + 1)) \end{bmatrix} =$$
$$\begin{bmatrix} 1 \\ 3 + width1 * (4 + height2 * (15)) \end{bmatrix} =$$
$$\begin{bmatrix} 1 \\ 3 + 4 * width1 + 15 * width1 * height2 \end{bmatrix}$$

$$F = 2 + 5 * width1 + 15 * width1 * height1//2 +$$
$$2 + 5 * height2 + 15 * height2 * height1//2 +$$
$$2 + 4 * width1 + 12 * width1 * height2 + 26 * width1 * height2 * height1//2 +$$
$$\begin{bmatrix} 1 \\ 3 + 4 * width1 + 15 * width1 * height2 \end{bmatrix} \sim O(n^3)$$

4.3.4 Оптимизированный алгоритм Винограда

Расчёт трудоёмкости для оптимизированного алгоритма Винограда.

$$F_1 = 2 + width1 * (2 + 2 + height1//2 * (2 + 5 + 1 + 1 + 1)) = \\ 2 + width1 * (4 + height1//2 * (10)) = \\ 2 + width1 * (4 + 10 * height1//2) = \\ 2 + 4 * width1 + 10 * width1 * height1//2$$

$$F_2 = 2 + height2 * (2 + 2 + height1//2 * (2 + 5 + 1 + 1 + 1)) = \\ 2 + height2 * (4 + height1//2 * (10)) = \\ 2 + height2 * (4 + 10 * height1//2) = \\ 2 + 4 * height2 + 10 * height2 * height1//2$$

$$F_3 = 2+width1*(2+2+height2*(2+4+1+1+1+\begin{bmatrix} 1 \\ 6+1+2+1 \end{bmatrix}+height1//2*(2+8+1+4+1))) = \\ 2 + width1 * (4 + height2 * (14 + \begin{bmatrix} 1 \\ 10 \end{bmatrix} + 16 * height1//2)) = \\ 2 + width1 * (4 + height2 + \begin{bmatrix} height2 \\ 10 * height2 \end{bmatrix} + 16 * height2 * height1//2 = \\ 2+4*width1+14*width1*height2+\begin{bmatrix} width1 * height2 \\ 10 * width1 * height2 \end{bmatrix}+16*width1*height2*height1//2$$

$$F = 2 + 4 * width1 + 10 * width1 * height1//2+ \\ 2 + 4 * height2 + 10 * height2 * height1//2+ \\ 2+4*width1+14*width1*height2+\begin{bmatrix} width1 * height2 \\ 10 * width1 * height2 \end{bmatrix}+16*width1*height2*height1//2 \sim O(n^3)$$

4.4 Выполненная оптимизация

1. height1//2 заменено на height1//2*2, в связанных циклах последовали соответствующие изменения:
   - в циклах заполнения вспомогательных массивов mulh и mulv 2*j заменено на j
   - в основном цикле умножения матриц 2*k заменено на k
2. Работа со значениями матрицы производится через буферную переменную
3. Условие на проверку нечетной размерности результирующей матрицы перенесено из отдельного вложенного цикла в основной вложенный цикл

4.5 Вывод

Алгоритм умножения матриц по Винограду работает быстрее, чем его аналог в лице стандартного алгоритма. Выигрыш во времени достигается путем введения двух дополнительных массивов и тем самым снижением числа умножений, причем разница составляет до 45% в пользу оптимизированного алгоритма Винограда. На нечетных размерностях матриц может происходить нивелирование преимущества за счет дополнительных проверок и траты ресурсов на обработку последней строки/столбца, причем разница составляет не более 5%.

Заключение

В результате выполнения данной работы был реализован классический алгоритм умножения матриц. Был изучен и реализован алгоритм Винограда. Был разработан и реализован оптимизированный вариант алгоритма Винограда. Была выбрана модель оценки трудоёмкости и по ней были даны оценки трудоёмкости классическому алгоритму умножения матриц, алгоритму Винограда (для лучшего и худшего случая) и оптимизированному алгоритму Винограда (для лучшего и худшего случая). Проведены замеры времени для алгоритмов.

В ходе экспериментов выяснилось, что алгоритм Винограда работает до 45% быстрее по сравнению со стандартным алгоритмом.

Разница между лучшим и худшим случаями алгоритма Винограда составляет до 5% на тестовых размерностях до 1001x1001.

Литература

[1] И. В. Белоусов(2006), Матрицы и определители, учебное пособие по линейной алгебре, с. 1 - 16

[2] Le Gall, F. (2012), "Faster algorithms for rectangular matrix multiplication Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2012), pp. 514–523