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 №2

Название: Умножение матриц. Формула Винограда

Дисциплина: Анализ алгоритмов

Студент	ИУ7-53Б		И. О. Артемьев
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Л. Л. Волкова
		Подпись, дата	И. О. Фамилия

Москва, 2021 г.

Содержание

Страница

Введение

В настоящее время компьютеры оперируют множеством типов данных. Одним из них являются матрицы. Для некоторых алгоритмов необходимо выполнять их перемножение. Данная операция является затратной по времени, имея сложность порядка $O(N^3)$. Поэтому Ш. Виноград создал свой алгоритм умножения матриц, который являлся ассимптотически самым быстрым из всех. Однако преимущества появляются только на матрицах большого размера, настолько что современная вычислительная техника не способна оперировать таким объемом данных.

Целью данной работы является реализация и изучение следующих алгоритмов:

- перемножение матриц алгоритмом Винограда.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить выбранные алгоритмы умножения матриц и способы оптимизации;
- составить схемы рассмотренных алгоритмов;
- реализовать разработанные алгоритмы умножения матриц;
- провести сравнительный анализ алгоритмов по затрачиваемым ресурсам (время и память);
- описать и обосновать полученные результаты.

1. Аналитический раздел

В данном разделе будут представлены описания алгоритмов умножения матриц стандартным способом, методом Винограда и его оптимизации.

1.1 Стандартный алгоритм

Пусть даны две прямоугольные матрицы:

$$A_{lm} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \dots & a_{lm} \end{pmatrix}, \quad B_{mn} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}, \tag{1.1}$$

Индексы у матриц обозначают их размерность - матрица A размером $l \cdot m$, матрица B - $m \cdot n$ соответственно. Перемножать матрицы можно только когда вторая размерность A и первая размерность B совпадают. Тогда результатом умножения данных матриц будет являться матрица C размером $l \cdot n$:

$$C_{ln} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \dots & c_{ln} \end{pmatrix}, \tag{1.2}$$

где каждый элемент матрицы вычисляется следующим образом:

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = \overline{1, l}; j = \overline{1, n}) \tag{1.3}$$

1.2 Алгоритм Копперсмита - Винограда

Ассимптотика данного алгоритма является лучшей среди существующих, она составляет $O(N^{2,3755})$.

Рассмотрим два вектора: $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Здесь веткор V играет роль строки первой матрицы-множителя, вектор W - столбца второй матрицы-множителя. Их скалярное произведение равно:

$$V \cdot W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1 v_2 - v_3 v_4 - w_1 w_2 - w_3 w_4. \quad (1.4)$$

Это выражение эквивалентно:

$$V \cdot W = (u_1 + w_2)(u_2 + w_1) + (u_3 + w_4)(u_4 + w_3) - u_1 u_2 - u_3 u_4 - w_1 w_2 - w_3 w_4 \quad (1.5)$$

В случае, если размерность умножаемых векторов нечетна (например, равна 5), добавляется следующее слагаемое:

$$V \cdot W += v_5 w_5 \quad (1.6)$$

Преимуществом данного метода вычисления является то, что слагаемые $v_1 v_2, v_3 v_4, w_1 w_2, w_3 w_4$ можно рассчитать для каждых строки/столбца заранее, тем самым сократить количество операций умножения и привести к менее затратной операции сложения.

Также как варианты оптимизации можно использовать побитовые сдвиги, += , использовать буферы и другое.

1.3 Вывод

В данном разделе были рассмотренны принципы работы алгоритмов умножения матриц. Полученных знаний достаточно для разработки выбранных алгоритмов.

В качестве входных данных в программу будут подаваться две матрицы

элементов, на выходных данных будет результирующая матрица. Ограничениями для работы ПО является то, что элементы входных матриц являются целыми числами.

Реализуемое ПО будет работать в пользовательском режиме (вывод результирующей матрицы перемножения двух других), а также в экспериментальном (проведение замеров времени выполнения алгоритмов).

2. Конструкторский раздел

В данном разделе будут спроектированы схемы алгоритмов, произведена оценка трудоемкости алгоритмов, описаны используемые типы данных, а также произведена оценка памяти и описана структура ПО.

2.1 Схемы алгоритмов

На рисунках 2.1 - 2.2 представлены схемы рассматриваемых алгоритмов.

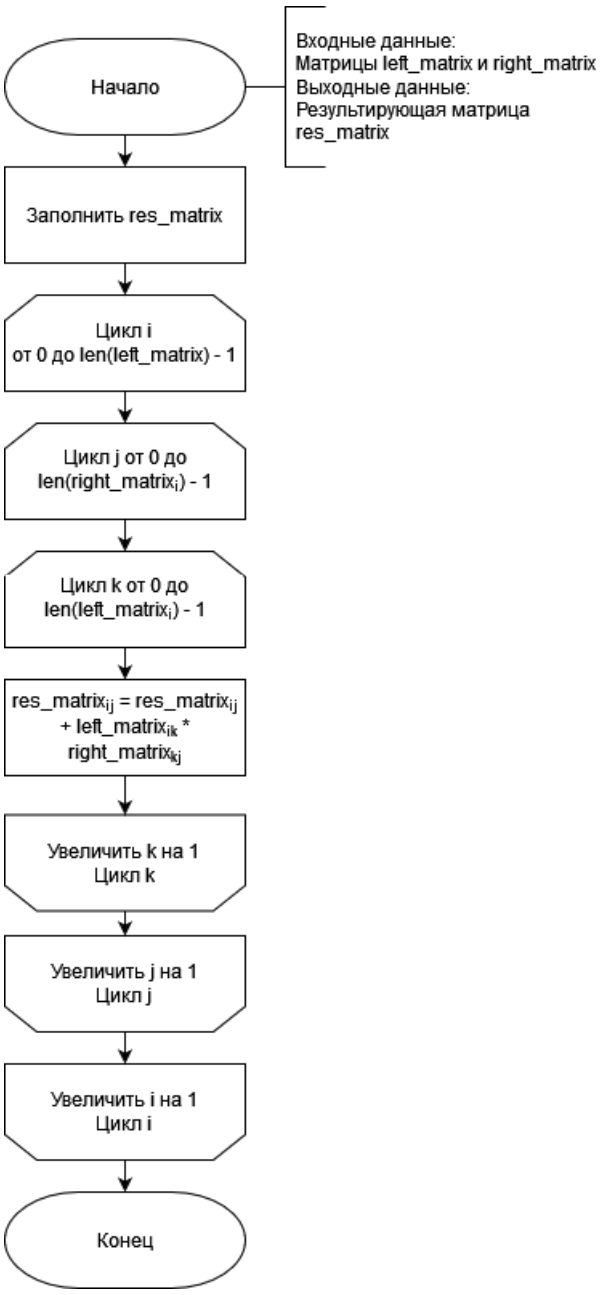

Рисунок 2.1 – Схема стандартного алгоритма умножения матриц

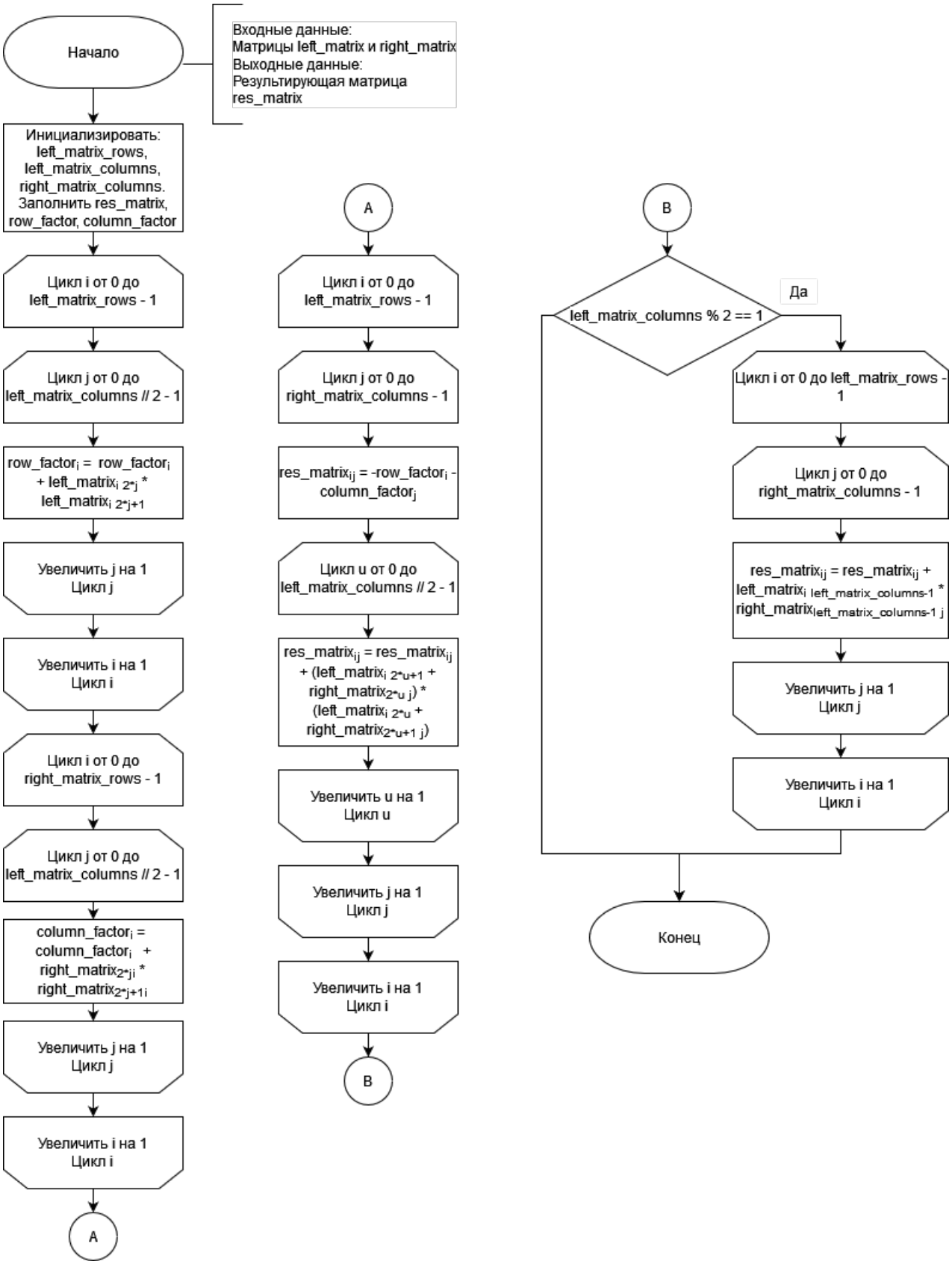

Рисунок 2.2 – Схема алгоритма умножения матриц Винограда

2.2 Модель оценки трудоемкости алгоритмов

Трудоемкость алгоритмов

Введем модель оценки трудоемкости.

1. Трудоемкость базовых операций
   - Примем трудоемкость следующих операций равной 1:

$$=, +, -, + =, - =, ==, ! =, <, >, \leq, \geq, !, \&\&, ||, [] \tag{2.1}$$

   - Примем трудоемкость следующих операций равной 2:

$$*, /, \%, * =, / =, \tag{2.2}$$

2. Трудоемкость циклов

$$f = f_{init} + f_{compare} + N_{iter} * (f_{body} + f_{inc} + f_{compare}) \tag{2.3}$$

3. Трудоемкость условного оператора Пусть трудоемкость самого условного перехода равна 0. Тогда

$$f_{if} = f_{condition} + \begin{bmatrix} min(f_{true}, f_{false}), \text{в лучшем случае} \\ max(f_{true}, f_{false}), \text{в худшем случае} \end{bmatrix} \tag{2.4}$$

где

f_{init} - трудоемкость инициализации,

$f_{compare}$ - трудоемкость сравнения,

N_{iter} - количество итераций,

f_{body} - трудоемкость тела цикла,

f_{inc} - трудоемкость инкрементации,

$f_{condition}$ - трудоемкость условия,

f_{true}, f_{false} - трудоемкость веток условного оператора.

Трудоемкость стандартного алгоритма

Обозначу псевдонимы к переменным для краткого обращения:

- L = len(left_matrix);
- N = len(right_matrix[i];
- M = len(left_matrix[i]).

Трудоемкость стандартного алгоритма состоит из:

- трудоемкость цикла по i от 1 до L: $f = 2 + L(2 + f_{body})$;
- трудоемкость цикла по j от 1 до N: $f = 2 + 3 + N(2 + f_{body})$;
- трудоемкость цикла по k от 1 до M: $f = 2 + M(2 + 12) = 2 + 14M$.

Итого, трудоемкость стандартного алгоритма равна:

$$f = 2+L(2+2+N(2+3+2+14M)) = 2+4L+7LN+14LNM \approx 14LNM \approx O(N^3) \tag{2.5}$$

Трудоемкость алгоритма Винограда

Обозначу псевдонимы к переменным для краткого обращения:

- L = len(left_matrix);
- N = len(right_matrix[i];
- M = len(left_matrix[i]).
- R = row_factor.
- C = column_factor.

Трудоемкость алгоритма Винограда состоит из:

- аллокация и инициализация векторов R и C: $f_{RC} = L+N+2+L(2+3+\frac{1}{2}M(3+14))+2+N(2+3+\frac{1}{2}M(3+14)) = 2+6L+6N+\frac{17}{2}LM+\frac{17}{2}NM$;

- цикл по i от 1 до L: $f_L = 2 + L(2 + f_{body})$;
- цикл по j от 1 до N: $f_N = 2 + N(2 + 7 + f_{body})$;
- цикл по u от 1 до M/2: $f_{M/2} = 3 + \frac{1}{2}M(3 + 28) = 3 + \frac{31}{2}M$;
- проверка размеров на нечетность:

$$f_{if} = 3 + \begin{cases} 0, & \text{л.с.} \\ 2 + L(2 + 2 + N(2 + 11)) & \text{х.с.} \end{cases} \tag{2.6}$$

Итого, трудоемкость алгоритма Винограда для лучшего случая равна:

$$f = 4+10L+6N+\frac{17}{2}LM+\frac{17}{2}NM+12LN+\frac{31}{2}LNM \approx 15,5LNM \approx O(N^3) \tag{2.7}$$

Для худшего случая равна:

$$f = 6+14L+6N+\frac{17}{2}LM+\frac{17}{2}NM+25LN+\frac{31}{2}LNM \approx 15,5LNM \approx O(N^3) \tag{2.8}$$

Трудоемкость оптимизированного алгоритма Винограда

Обозначу псевдонимы к переменным для краткого обращения:

- L = len(left_matrix);
- N = len(right_matrix[i];
- M = len(left_matrix[i]).
- R = row_factor.
- C = column_factor.

Трудоемкость оптимизированного алгоритма Винограда состоит из:

- аллокация и инициализация векторов R и C: $f_{RC} = L+N+2+L(2+3+\frac{1}{2}M(3+13))+2+N(2+3+\frac{1}{2}M(3+13)) = 2+6L+6N+8LM+8NM$;
- цикл по i от 1 до L: $f_L = 2 + L(2 + f_{body})$;
- цикл по j от 1 до N: $f_N = 2 + N(2 + 5 + f_{body} + f_{if} + 3)$;
- цикл по k от 1 до M/2: $f_{M/2} = 3 + \frac{1}{2}M(3 + 23) = 3 + 13M$;
- проверка размеров на нечетность:

$$f_{if} = 3 + \begin{cases} 0, & \text{л.с.} \\ 10 & \text{х.с.} \end{cases} \tag{2.9}$$

Итого, трудоемкость оптимизированного алгоритма Винограда для лучшего случая равна:

$$f = 4 + 10L + 6N + 8LM + 8NM + 16LN + 13LNM \approx 13LNM \approx O(N^3) \tag{2.10}$$

Для худшего случая равна:

$$f = 4 + 10L + 6N + 8LM + 8NM + 26LN + 13LNM \approx 13LNM \approx O(N^3) \tag{2.11}$$

2.3 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- матрица типа int заданного размера;
- размеры матрицы - целое число int;
- вспомогательные массивы типа int заданного размера.

2.4 Описание выделенных классов эквивалентности

Для дальнейшего тестирования работы программы были выделены следующие классы эквивалентности:

- квадратная матрица четных размеров;
- квадратная матрица нечетных размеров.

2.5 Структура ПО

ПО будет состоять из следующих модулей:

- main.py - модуль, вызывающий загрузку меню (является модулем запуска);
- menu.py - модуль, содержащий меню пользователя;
- myio.py - модуль, содержащий функции формирования матриц;
- muls.py - модуль, содержащий функции перемножения матриц;
- plot.py - модуль, содержащий функции для построения графиков сложности перемножения матриц.

2.6 Вывод

На основе полученных в аналитическом разделе знаний об алгоритмах были спроектированы схемы алгоритмов, произведена оценка их трудоемкости, выбраны используемые типы данных, проведена оценка затрачиваемого объема памяти, а также описана структура ПО.

3. Технологический раздел

В данном разделе будут приведены требования к ПО, средства его реализации и листинга кода алгоритмов, а также рассмотрены тестовые случаи.

3.1 Средства реализации

Для реализации программ я выбрал язык программирования Python, так как я очень хорошо знаком с этим языком и пишу на нем давно, также этот язык является удобным, безопасным и в нем присутствуют инструменты замера времени.

3.2 Листинги кода

Листинги 3.1 - 3.3 демонстрируют реализацию алгоритмов умножения матриц.

Листинг 3.1 – Стандартный алгоритм умножения матриц

```
def defaultMatrixMul(left_matrix: list, right_matrix: list) -> list:
    if len(left_matrix) == 0 or len(right_matrix) == 0:
        print("Matrices are empty")
    elif len(left_matrix[0]) != len(right_matrix):
        print("Matrices cannot be multiplied")
    else:
        res_matrix = [
            [0 for _ in range(len(right_matrix[i]))] for i in \
            range(len(left_matrix))
        ]

        for i in range(len(left_matrix)):
            for j in range(len(right_matrix[i])):
                for k in range(len(left_matrix[i])):
                    res_matrix[i][j] += left_matrix[i][k] * \
                                        right_matrix[k][j]

        return res_matrix
    return
```

Листинг 3.2 – Алгоритм умножения матриц Винограда

```
def vinMatrixMul(left_matrix: list, right_matrix: list) -> list:
    if len(left_matrix) == 0 or len(right_matrix) == 0:
        print("Matrices are empty")
    elif len(left_matrix[0]) != len(right_matrix):
        print("Matrices cannot be multiplied")
    else:
        left_matrix_rows = len(left_matrix)
        left_matrix_columns = len(left_matrix[0])
        right_matrix_columns = len(right_matrix[0])

        res_matrix = [
            [0 for _ in range(right_matrix_columns)] for _ in \
                                        range(left_matrix_rows)
        ]

        row_factor = [0 for _ in range(left_matrix_rows)]
        for i in range(left_matrix_rows):
            for j in range(left_matrix_columns // 2):
                row_factor[i] += left_matrix[i][2 * j] * \
                                 left_matrix[i][2 * j + 1]

        column_factor = [0 for _ in range(right_matrix_columns)]
        for i in range(right_matrix_columns):
            for j in range(left_matrix_columns // 2):
                column_factor[i] += right_matrix[2 * j][i] * \
                                    right_matrix[2 * j + 1][i]

        for i in range(left_matrix_rows):
            for j in range(right_matrix_columns):
                res_matrix[i][j] = -row_factor[i] - column_factor[j]
                for u in range(left_matrix_columns // 2):
                    res_matrix[i][j] += (
                        left_matrix[i][2 * u + 1] + right_matrix[2 * u][j]
                    ) * (left_matrix[i][2 * u] + \
                        right_matrix[2 * u + 1][j])

        if left_matrix_columns % 2 == 1:
            for i in range(left_matrix_rows):
                for j in range(right_matrix_columns):
                    res_matrix[i][j] += (
                        left_matrix[i][left_matrix_columns - 1]
                        * right_matrix[left_matrix_columns - 1][j]
                    )

        return res_matrix
    return
```

Листинг 3.3 – Алгоритм умножения матриц Винограда (оптимизированный)

```
def optimizedVinMatrixMul(left_matrix: list, right_matrix: list) -> list:
    if len(left_matrix) == 0 or len(right_matrix) == 0:
        print("Matrices are empty")
    elif len(left_matrix[0]) != len(right_matrix):
        print("Matrices cannot be multiplied")
    else:
        left_matrix_rows = len(left_matrix)
        left_matrix_columns = len(left_matrix[0])
        right_matrix_columns = len(right_matrix[0])

        res_matrix = [
            [0 for _ in range(right_matrix_columns)] for _ in \
                                        range(left_matrix_rows)
        ]

        row_factor = [0 for _ in range(left_matrix_rows)]
        for i in range(left_matrix_rows):
            for j in range(1, left_matrix_columns, 2):
                row_factor[i] += left_matrix[i][j] * left_matrix[i][j - 1]

        column_factor = [0 for _ in range(right_matrix_columns)]
        for i in range(right_matrix_columns):
            for j in range(1, left_matrix_columns, 2):
                column_factor[i] += right_matrix[j][i] * \
                                    right_matrix[j - 1][i]

        flag = left_matrix_rows % 2
        for i in range(left_matrix_rows):
            for j in range(right_matrix_columns):
                res_matrix[i][j] = -(row_factor[i] + column_factor[j])
                for u in range(1, left_matrix_columns, 2):
                    res_matrix[i][j] += (left_matrix[i][u - 1] + \
                                         right_matrix[u][j]) * (
                        left_matrix[i][u] + right_matrix[u - 1][j]
                    )
                if flag:
                    res_matrix[i][j] += (
                        left_matrix[i][left_matrix_columns - 1]
                        * right_matrix[left_matrix_columns - 1][j]
                    )

        return res_matrix
    return
```

3.3 Тестирование ПО

В таблице 3.1 приведены тестовые случаи для алгоритмов умножения матриц. Случаи 1 - 2 - умножение однострочных/одностолбцовых матриц, 3 - 5 - матрицы одного размера, 6 - некорректный размер.

Таблица 3.1 – Тестовые случаи

N	Матрица 1	Матрица 2	Результат
1	$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 3 \\ 6 \\ 9 \end{pmatrix}$
2	$\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 3 \end{pmatrix}$
3	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$
4	$\begin{pmatrix} 3 \end{pmatrix}$	$\begin{pmatrix} 3 \end{pmatrix}$	$\begin{pmatrix} 9 \end{pmatrix}$
5	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 18 & 24 & 30 \\ 54 & 69 & 84 \\ 90 & 114 & 138 \end{pmatrix}$
6	$\begin{pmatrix} 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 4 & 5 \end{pmatrix}$	Некорректный размер

3.4 Вывод

В данном разделе были представлены выбор языка программирования, листинги реализаций алгоритмов и результаты тестирования.

4. Исследовательский раздел

В данном разделе будут представлены замеры времени работы реализаций алгоритмов.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее.

- Операционная система: macOS Catalina версия 10.15.6;
- Память: 8 GB 1600 MHz DDR3;
- Процессор: 1,8 GHz 2-ядерный процессор Intel Core i5.

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения рабочего стола.

4.2 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов с помощью функции process_time(), которая находится в модуле time языка Python. Для сравнения алгоритмов перемножения матриц используются квадратные матрицы четных и нечетных размерностей. Замеры времени усреднялись, для каждого алгоритма проводилось по 3 итерации.

Примем за лучший случай четную размерность, а за худший - нечетную.

Таблица 4.1 – Временные замеры работы алгоритмов для лучшего случая

Количество	Простой	Виноград	Виноград с оптимизациями
100	0.42	0.34	0.28
200	2.82	2.81	2.28
350	9.26	9.81	7.81
400	22.67	24.25	19.05
500	42.26	48.95	38.01

Таблица 4.2 – Временные замеры работы алгоритмов для худшего случая

Количество	Простой	Виноград	Виноград с оптимизациями
101	0.30	0.36	0.29
201	2.44	2.88	2.39
301	8.74	10.07	7.93
401	21.16	24.43	19.51
501	41.33	49.17	38.61

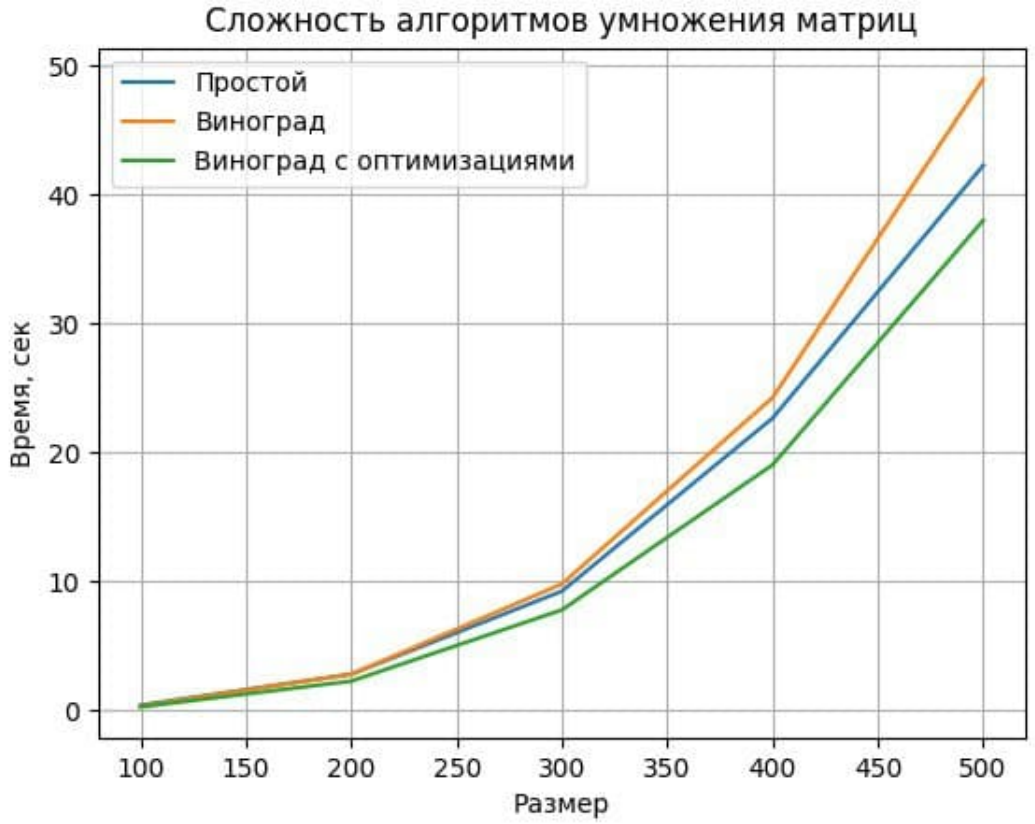

Рисунок 4.1 – Время работы алгоритмов для лучшего случая

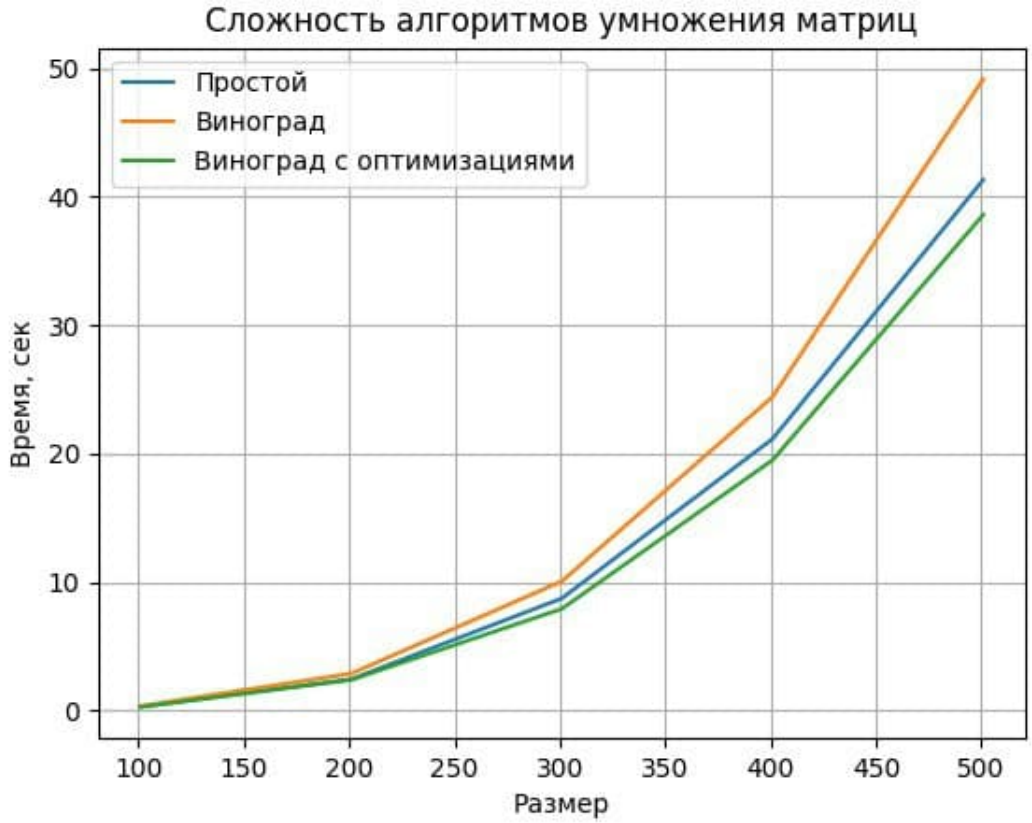

Рисунок 4.2 – Время работы алгоритмов для худшего случая

4.3 Вывод

В данном разделе было произведено сравнение количества затраченного времени вышеизложенных алгоритмов.

Исходя из полученных результатов, можно сделать вывод, что простой алгоритм и алгоритм винограда (без оптимизаций) работают приблизительно одинаково, но на графиках мы получили, что простой алгоритм работает чуть быстрее (± 7 сек), возможно, это произошло из-за особенностей языка Python, но в любом случае алгоритм винограда следует применять только при очень больших размерностях матриц. Так же мы видим, что виноград с оптимизациями работает быстрее всех алгоритмов на лучших и худших случаях ± 5 сек.

Заключение

В ходе выполнения данной лабораторной работы были выполнены следующие задачи:

- Изучены выбранные алгоритмы умножения матриц и способы оптимизации;
- Составлены схемы рассмотренных алгоритмов;
- Реализованны разработанные алгоритмы умножения матриц;
- Проведен сравнительный анализ алгоритмов по затрачиваемым ресурсам (время и память);
- Описаны и обоснованы полученные результаты.

В результате исследований можно прийти к выводу, что стандартный алгоритм и алгоритм винограда (без оптимизаций) работают приблизительно одинаково. Сложность всех алгоритмов равняется $O(N^3)$, оптимизированный алгоритм Винограда работает быстрее в среднем на 20% при любых данных. По используемой памяти алгоритмы практически не отличаются, за исключением использования двух дополнительных массивов в алгоритме Винограда.

Список литературы

1. Welcome to Python [Электронный ресурс] Режим доступа: https://www.python.org/, (дата обращения: 13.10.2021)

2. time - Time access and conversions [Электронный ресурс] Режим доступа: https://docs.python.org/3/library/time.html, (дата обращения: 01.11.2021)

3. Умножение матриц [Электронный ресурс] Режим доступа: https://zaochnik.com/spravochnik/matematika/matritsy/umnozhenie-matrits/, (дата обращения: 01.11.2021)

4. Алгоритм Винограда [Электронный ресурс] Режим доступа: http://algolib.narod.ru/Math/Matrix.html, (дата обращения: 01.11.2021)

5. Алгоритм Винограда с оптимизацией [Электронный ресурс] Режим доступа: https://www.gyrnal.ru/statyi/ru/1153/, (дата обращения: 01.11.2021)