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 № 2

по курсу «Анализ алгоритмов»

на тему: «Умножение матриц»

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

Матрицы являются одним из основных инструментов линейной алгебры, они позволяют описывать и анализировать линейные отношения между различными объектами и явлениями. В настоящее время матрицы широко используются в науке, технике, экономике и других сферах человеческой деятельности.

Размеры матриц могут отличаться в зависимости от конкретной задачи, поэтому оптимизация алгоритмов обработки матриц является важной задачей программирования. Основной акцент будет сделан на оптимизации алгоритма умножения матриц.

Цель данной лабораторной работы — исследовать алгоритмы умножения матриц и их оптимизации. Для достижения поставленной цели необходимо выполнить следующие задачи.

1) Описать алгоритмы умножения матриц:
   - — классический алгоритм;
   - — алгоритм Винограда;
   - — алгоритм Штрассена.
2) Оптимизировать алгоритм Винограда.
3) Разработать программное обеспечение, реализующее алгоритмы умножения.
4) Выбрать инструменты для реализации и замера процессорного времени выполнения алгоритмов.
5) Проанализировать затраты реализаций алгоритмов по времени.

# 1 Аналитический раздел

Матрицей называется прямоугольная таблица чисел, вида (1.1), состоящая из $m$ строк и $n$ столбцов [1].

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \tag{1.1}$$

Пусть $A$ — матрица, тогда $_{ij}$ — элемент этой матрицы, который находится на *i-ой* строке и *j-ом* столбце.

Если количество столбцов первой матрицы совпадает с количеством строк второй матрицы, то возможно выполнить их матричное умножение. В результате умножения получится матрица-произведение, количество строк в которой равно количеству строк первой матрицы, а количество столбцов равно количеству столбцов второй матрицы.

## 1.1 Классический алгоритм

Пусть даны две прямоугольные матрицы $A$ и $B$ размеров $[m \times n]$ и $[n \times k]$ соответственно. В результате произведение матриц $A$ и $B$ получим матрицу $C$ размера $[m \times k]$, элементы которой вычисляются по следующей формуле

$$c_{ij} = \sum_{l=1}^{n} a_{il} b_{lj} \tag{1.2}$$

Классический алгоритм умножения матриц реализует формулу (1.2).

## 1.2 Алгоритм Винограда

Произведение матриц $A$ и $B$ можно связать со скалярным произведением строки матрицы $A$ на столбец матрицы $B$ [2].

Рассмотрим два вектора $V = (v1, v2, v3, v4)$ и $W = (w1, w2, w3, w4)$.

Их скалярное произведение равно

$$V \cdot W = v_1 \cdot w_1 + v_2 \cdot w_2 + v_3 \cdot w_3 + v_4 \cdot w_4 \tag{1.3}$$

Равенство (1.3) можно переписать в виде

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

Теперь допустим, что у нас есть две матрицы $A$ и $B$ размерности $m \times n$ и $n \times p$ соответственно, и мы хотим найти их произведение $C = A \cdot B$. Тогда алгоритм будет состоять из следующих шагов.

1) Подготовительные вычисления. Сначала создаются два вспомогательных массива

$$\text{rowFactor}[i] = \sum_{j=0}^{n/2-1} A[i][2j+1] \cdot A[i][2j] \quad (1.5)$$

для $0 \le i < m$

$$\text{colFactor}[j] = \sum_{i=0}^{n/2-1} B[2i+1][j] \cdot B[2i][j] \quad (1.6)$$

для $0 \le j < p$.

2) Умножение матриц. Вычисляем результирующую матрицу $C$ по формуле

$$\begin{aligned} C[i][j] = &\sum_{k=0}^{n/2-1} (A[i][2k+1] + B[2k][j]) \cdot (A[i][2k] + B[2k+1][j]) \\ &- \text{rowFactor}[i] - \text{colFactor}[j] \end{aligned} \quad (1.7)$$

для $0 \le i < m$ и $0 \le j < p$.

3) Коррекция. Если $n$ нечетно, добавляем коррекцию, в соответствии со следующей формулой

$$C[i][j] += A[i][n] \cdot B[n][j] \quad (1.8)$$

для $0 \le i < m$ и $0 \le j < p$.

## 1.3 Алгоритм Штрассена

Алгоритм Штрассена — это алгоритм умножения квадратных матриц, который является более эффективным для больших матриц, чем классический метод умножения [3].

Если добавить к матрицам $A$ и $B$ одинаковые нулевые строки и столбцы, их произведение станет равно матрице $C$ с теми же добавленными строками и столбцами. Поэтому в данном алгоритме рассматриваются матрицы порядка $2^{k+1}$, где $k \in \mathbb{N}$, а все остальные матрицы, сводятся к этому размеру добавлением нулевых строк и столбцов.

Алгоритм состоит из следующих шагов.

1) Разбиение матриц

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \tag{1.9}$$

где $A_{ij}$ и $B_{ij}$ – матрицы порядка $2^k$

2) Вычисление вспомогательных матриц

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned} \tag{1.10}$$

3) Вычисление результирующих подматриц

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned} \tag{1.11}$$

Результирующая матрица состоит из $C_{ij}$

$$AB = C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \tag{1.12}$$

## Вывод

В данном разделе были описаны алгоритмы классического умножения матриц, алгоритм Штрассена и алгоритм Винограда.

# 2 Конструкторский раздел

В этом разделе будет представлено описание используемых типов данных, а также схематические изображения алгоритмов матричного умножения: стандартного, Штрассена, алгоритма Винограда и оптимизированного алгоритма Винограда.

## 2.1 Требования к программному обеспечению

Программа должна поддерживать два режима работы: режим массового замера времени и режим умножения матриц.

Режим массового замера времени должен обладать следующей функциональностью:

— генерировать матрицы различного размер для проведения замеров;

— осуществлять массовый замер, используя сгенерированные данные;

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

К режиму умножения матриц выдвигается ряд требований:

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

— наличие интерфейса для выбора действий;

— проверять возможность умножения матриц.

## 2.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры и типы данных:

— целые числа представляют количество строк и столбцов;

— матрица — двумерный список вещественных чисел.

## 2.3 Разработка алгоритмов

На рисунке 2.1 представлена схема классического алгоритма, выполняющего умножение двух матриц. На рисунках 2.2 – 2.4 изображены схемы алгоритма Винограда без оптимизаций. На рисунках 2.5 и 2.6 изображены схемы алгоритмов, которые реализуют алгоритм умножения матриц Штрассена. На рисунках 2.7 – 2.9 изображены схемы алгоритма Винограда с оптимизациями.

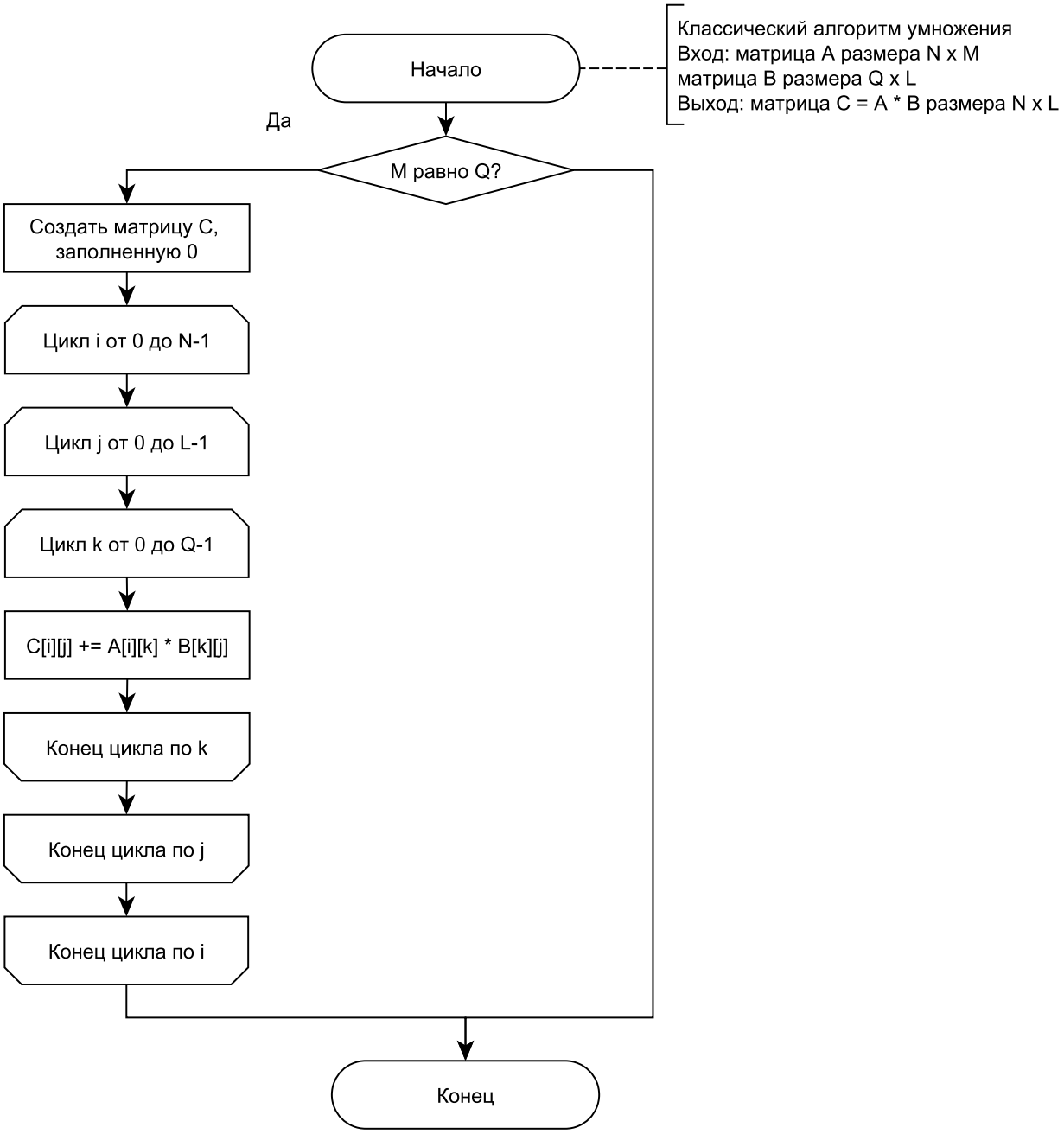


Рисунок 2.1 – Классический алгоритм умножения матриц

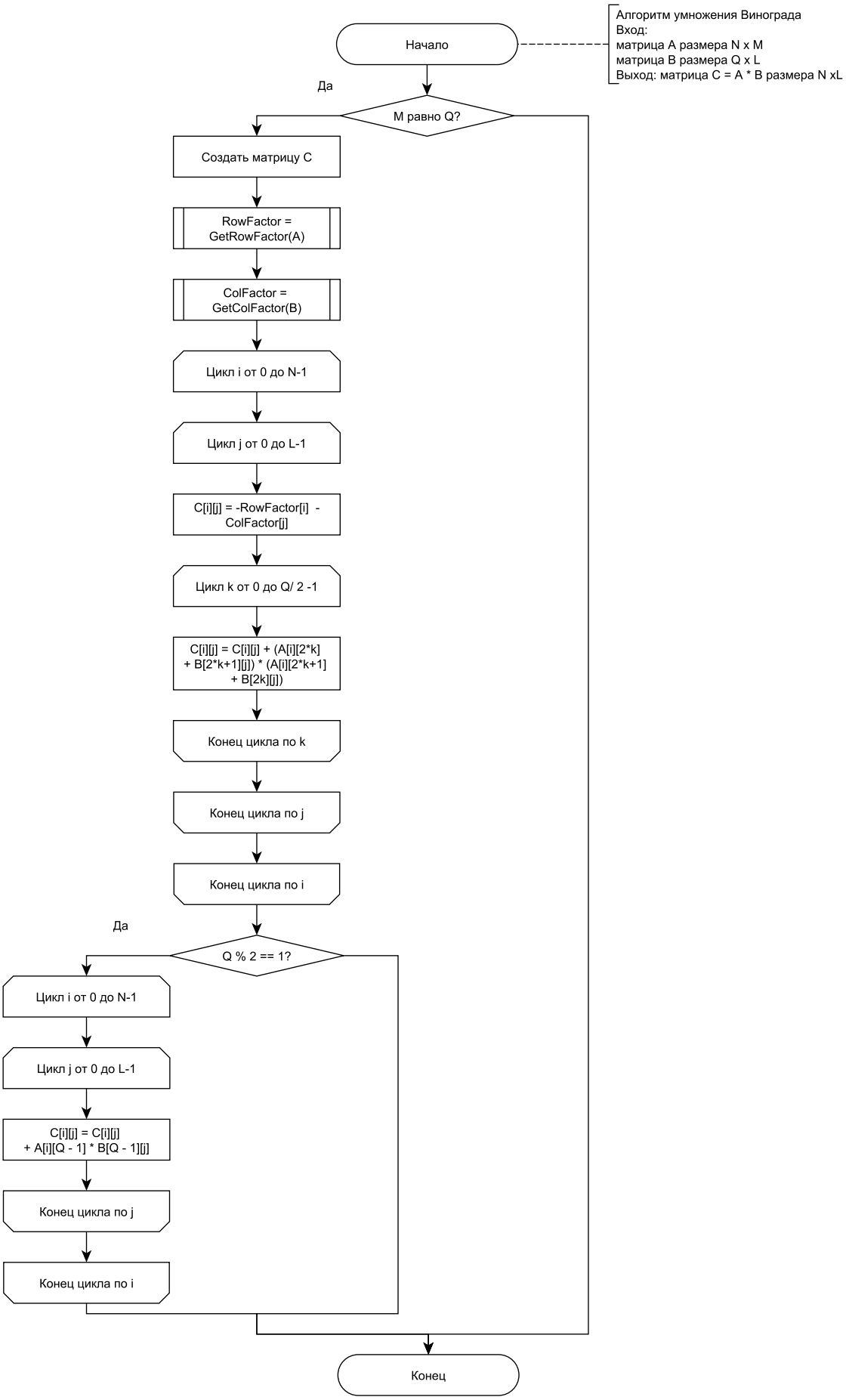


Рисунок 2.2 – Алгоритм умножения матриц Винограда

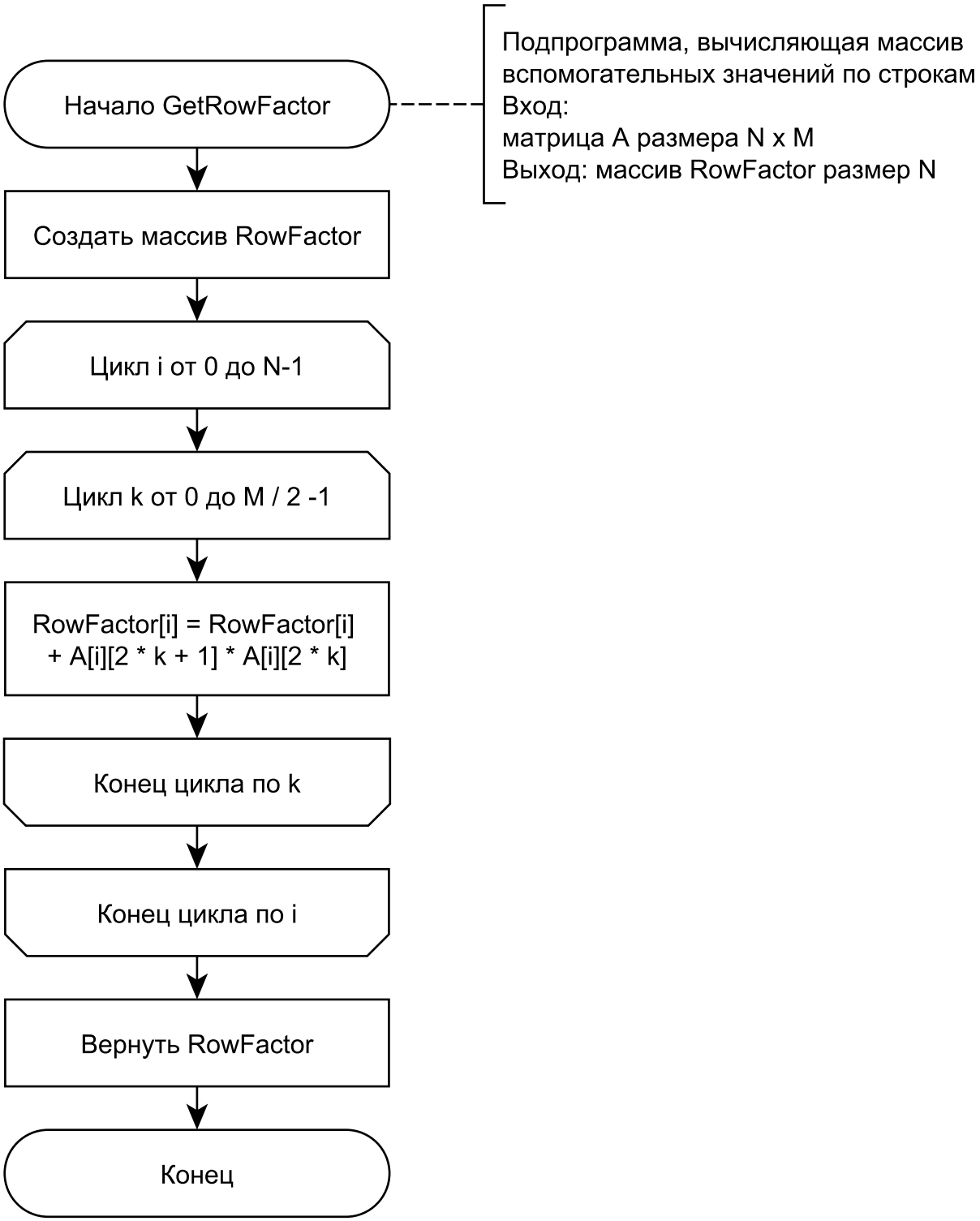


Рисунок 2.3 – Схема алгоритма для вспомогательной подпрограммы, вычисляющей массив вспомогательных значений по строкам

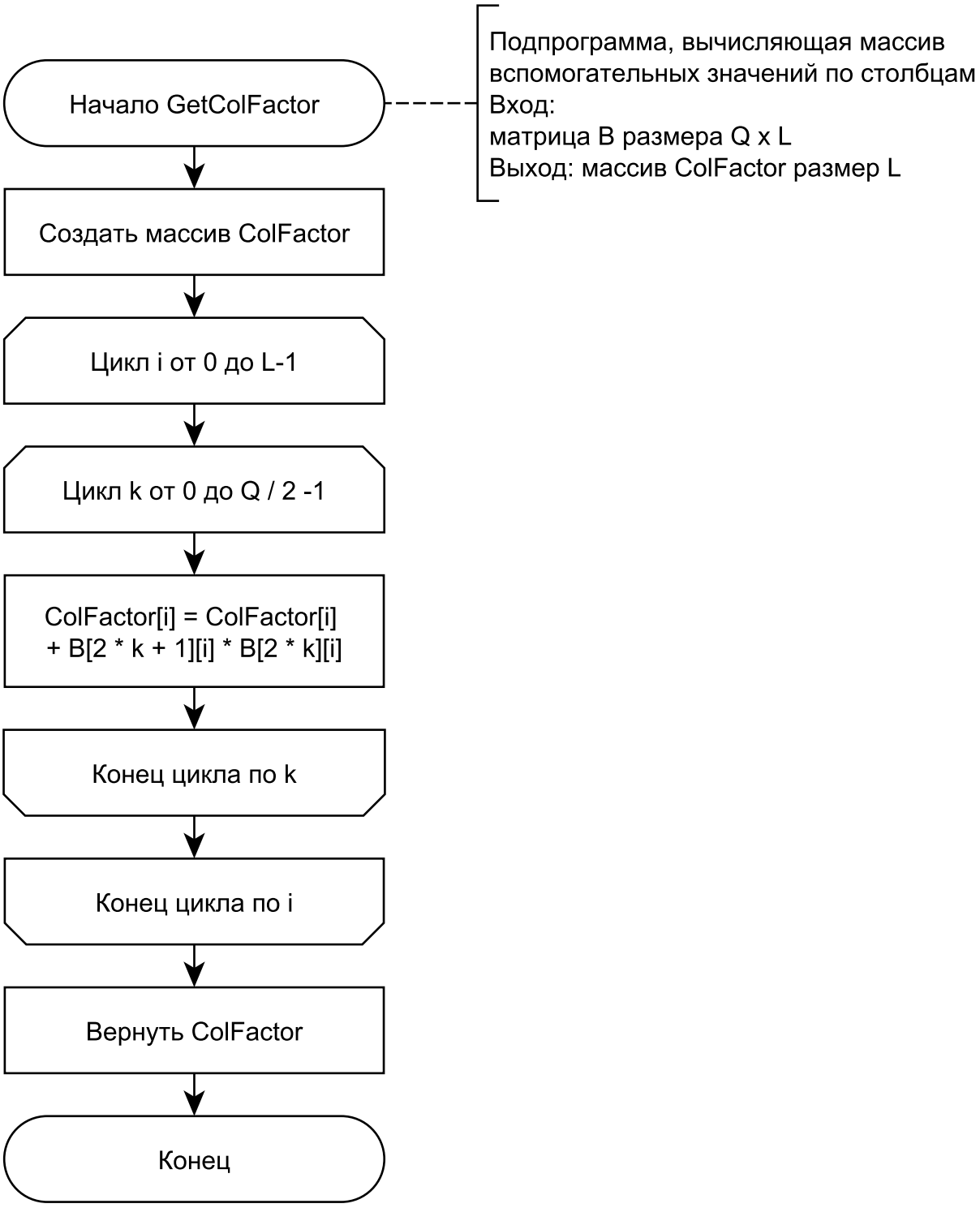


Рисунок 2.4 – Схема алгоритма для вспомогательной подпрограммы, вычисляющей массив вспомогательных значений по столбцам

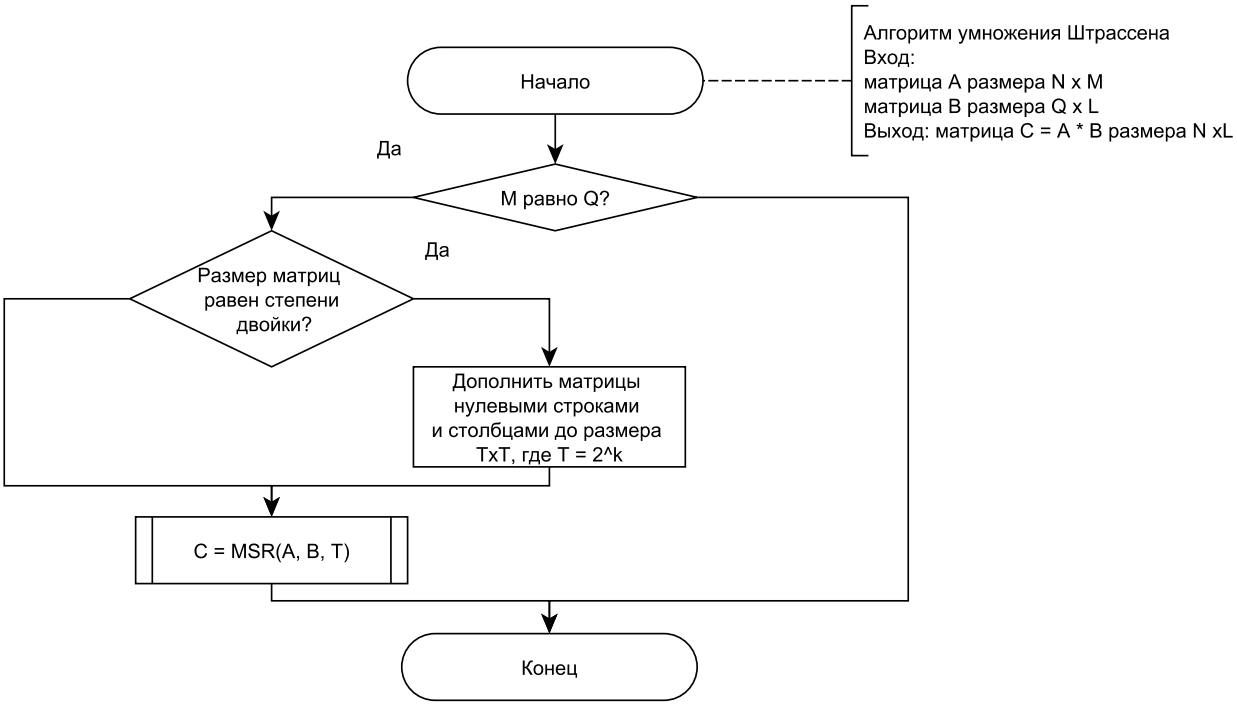


Рисунок 2.5 – Алгоритм умножения матриц Штрассена

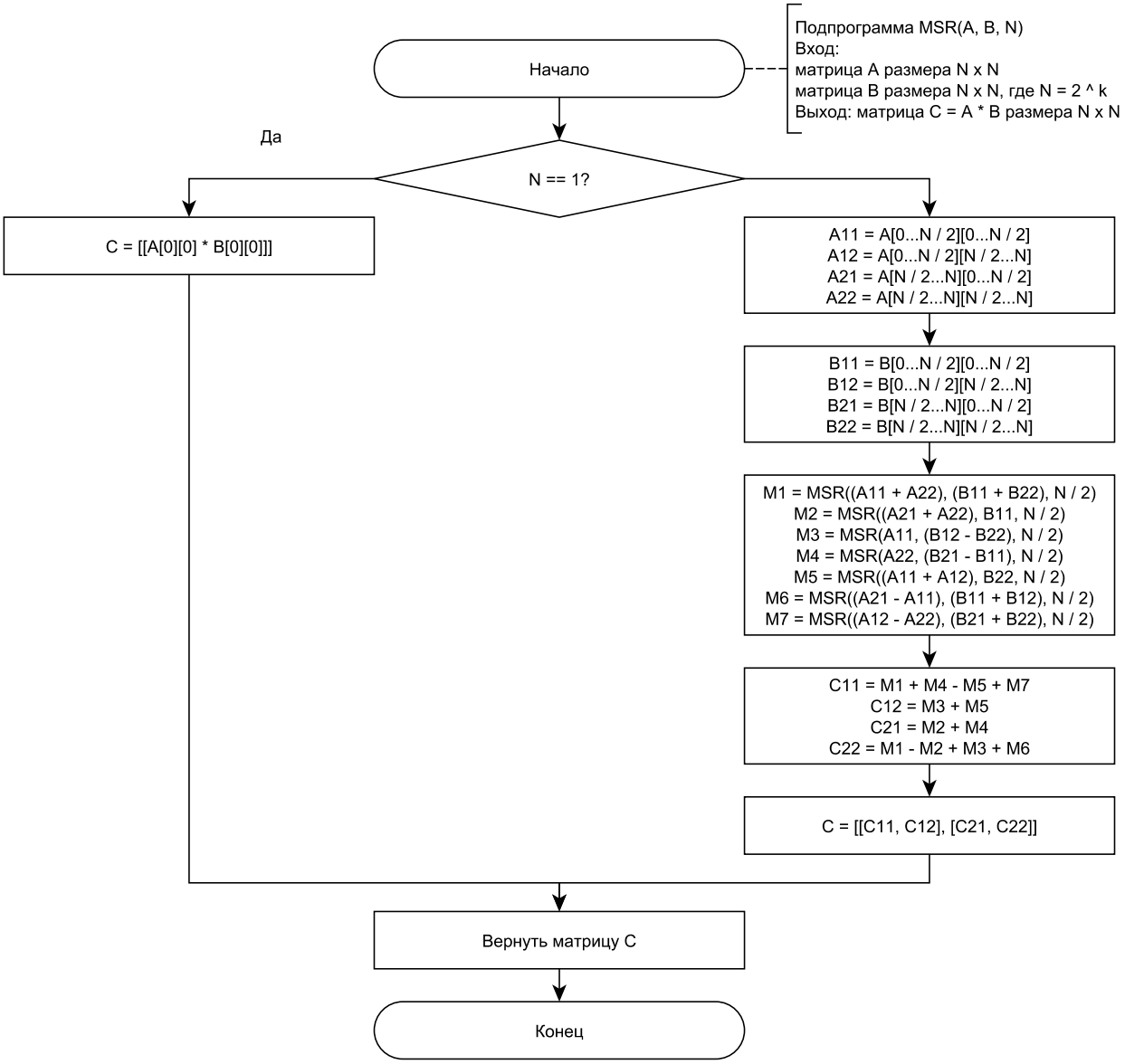


Рисунок 2.6 – Схема алгоритма подпрограммы MSR, вычисляющей результат умножения матриц по алгоритму Штрассена

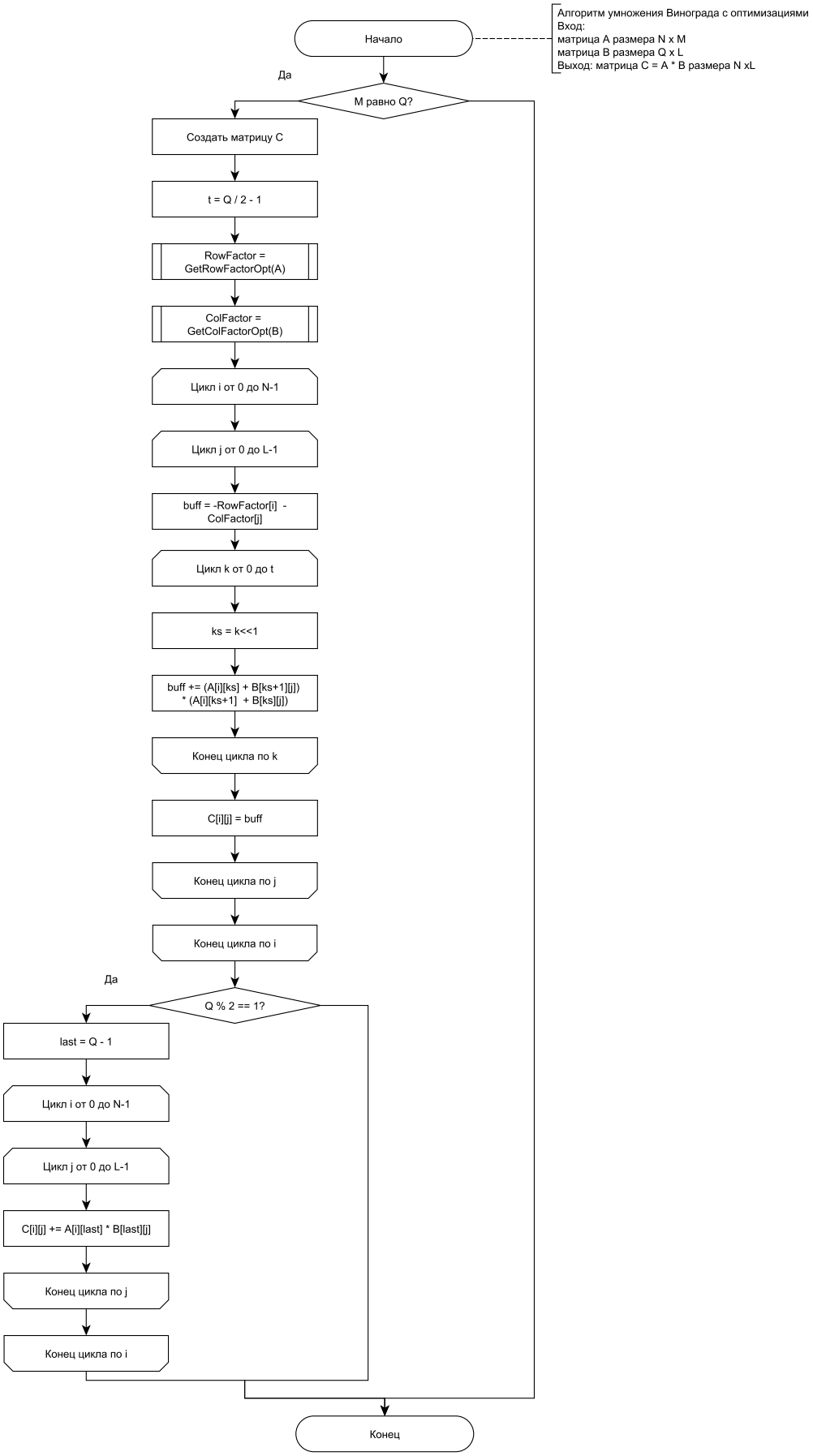


Рисунок 2.7 – Алгоритм умножения матриц Винограда с оптимизациями

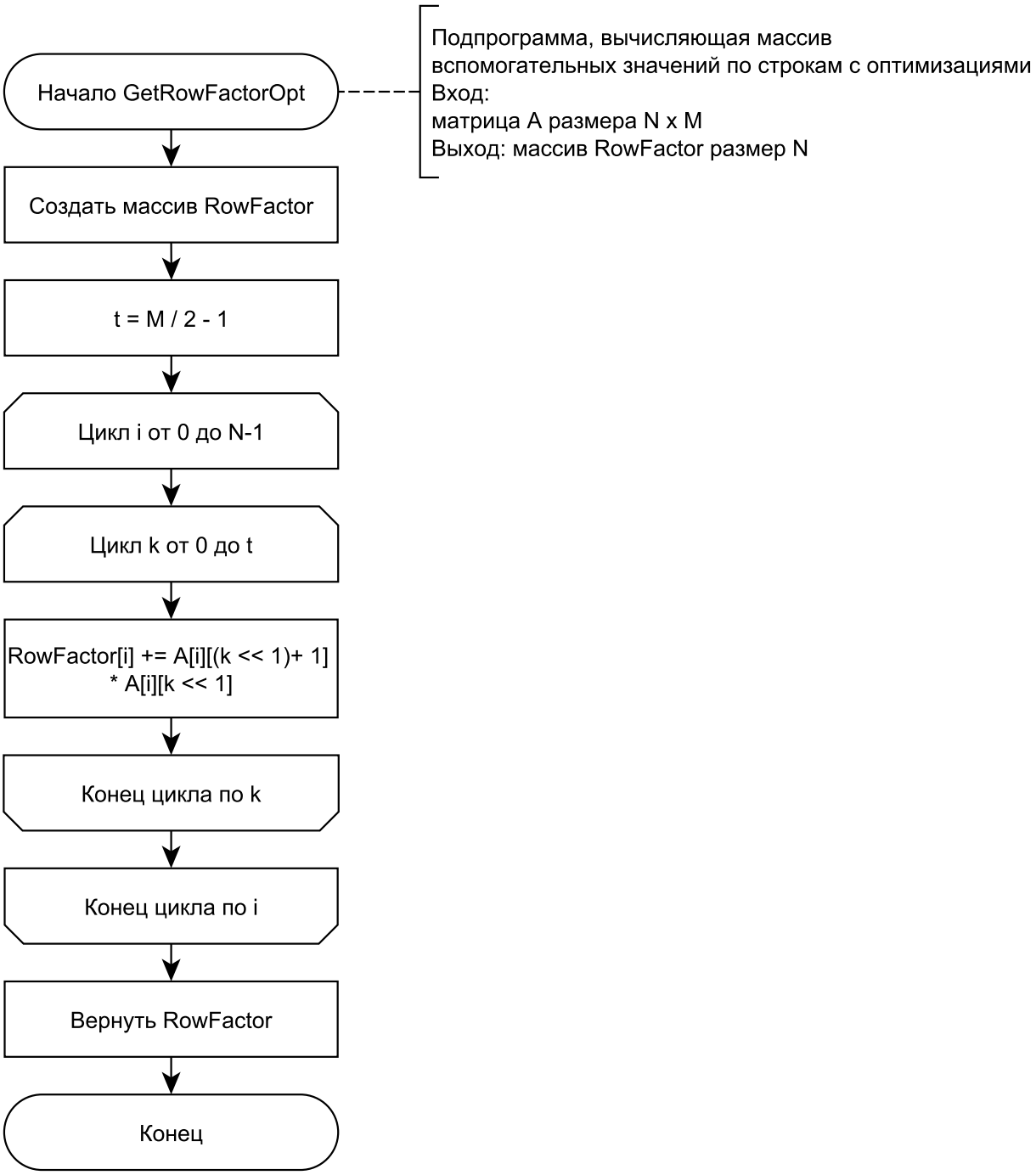


Рисунок 2.8 – Схема алгоритма для вспомогательной подпрограммы, вычисляющей массив вспомогательных значений по строкам с оптимизациями

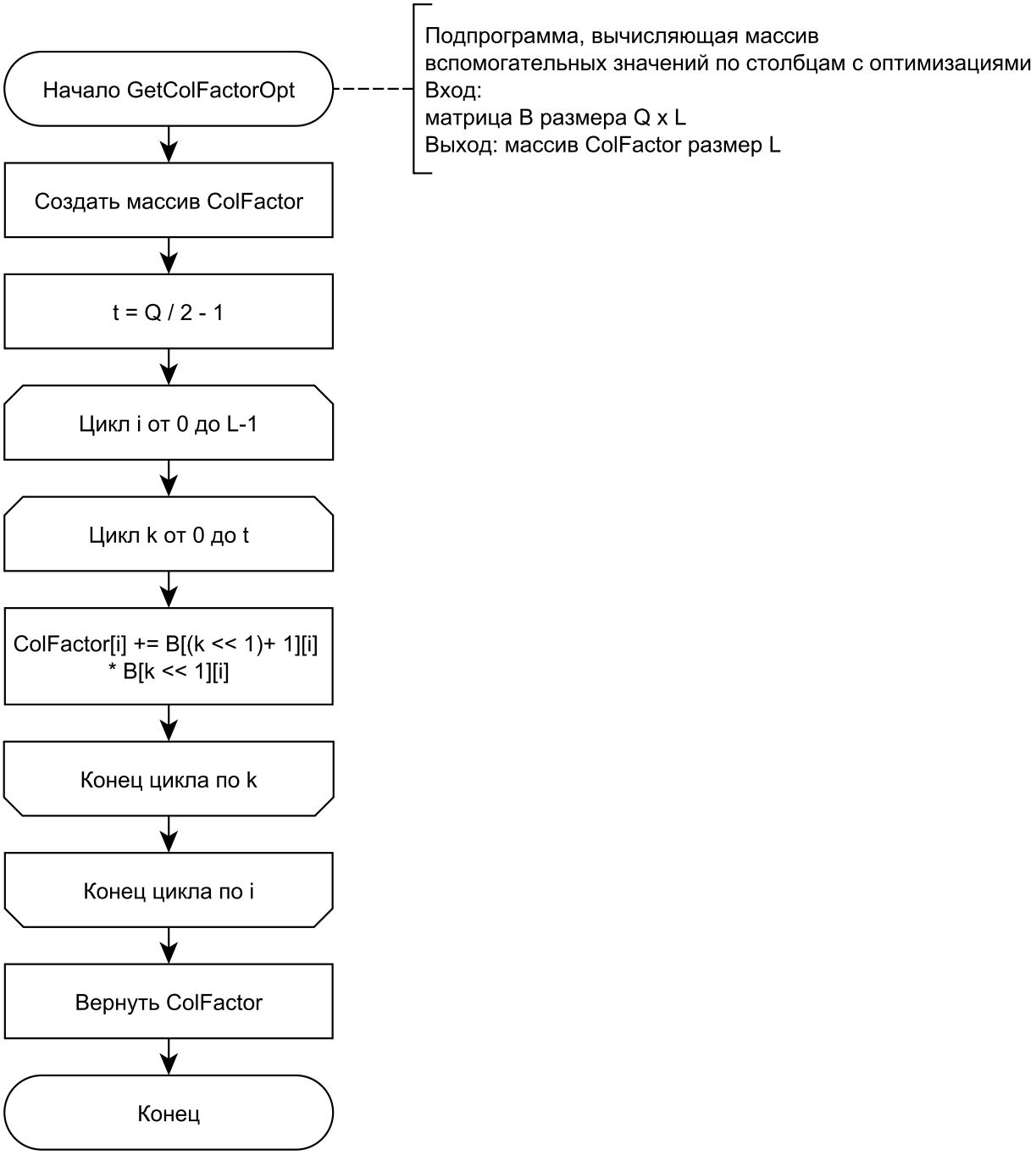


Рисунок 2.9 – Схема алгоритма для вспомогательной подпрограммы, вычисляющей массив вспомогательных значений по столбцам с оптимизациями

## 2.4 Оценка трудоемкости алгоритмов

Введем модель для оценки трудоемкости алгоритмов.

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

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

3) Трудоемкость условного перехода примем 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 формуле

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

6) Вызов подпрограмм и передачу параметров примем за 0.

### 2.4.1 Трудоемкость классического алгоритма

Трудоемкость классического алгоритма складывается из следующих слагаемых.

1) Внешний цикл по $i \in [0 \dots N)$ , трудоёмкость которого: $f_i = 2 + N \cdot (2 + f_j)$.

2) Цикла по $j \in [0 \dots L)$, трудоёмкость которого: $f_j = 2 + L \cdot (2 + f_k)$.

3) Цикла по $k \in [0 \dots Q)$, трудоёмкость которого: $f_k = 2 + Q \cdot (2 + 2 + 1 + 2 + 2 + 2) = 2 + 11Q$.

Тогда в итоге, трудоемкость классического алгоритма рассчитывается по формуле

$$f_{classic} = 2 + N \cdot (2 + 2 + L \cdot (2 + 2 + 11Q)) = 11NLQ + 4NL + 4N + 2 \quad (2.3)$$

### 2.4.2 Трудоемкость алгоритма Винограда

Трудоемкость алгоритма Винограда формируется из следующих частей.

1) Трудоемкости вычисления массива $RowFactor$, которая рассчитывается по формуле (2.4).

$$f_r = 2 + N \cdot (2 + 4 + \frac{M}{2} \cdot (3 + 1 + 15)) = \frac{19}{2}NM + 6N + 2 \quad (2.4)$$

2) Трудоемкости вычисления массива $ColFactor$, которая рассчитывается, аналогично пункту 1, по формуле

$$f_c = \frac{19}{2}LQ + 6L + 2 \quad (2.5)$$

3) Трудоемкости основной части алгоритма, которая вычисляется по формуле

$$\begin{gathered} f_{main} = 2 + N \cdot (2 + 2 + L \cdot (2 + 7 + 4 + \frac{Q}{2} \cdot (4 + 28))) = \\ = \frac{32}{2}NLQ + 13NL + 4N + 2 \end{gathered} \quad (2.6)$$

4) Трудоемкости поправки, вносимой для нечетного $Q$, рассчитываемой по следующей формуле

$$f_{err} = 3 + \begin{cases} 0, & \text{лучший случай} \\ 16LN + 4N + 2, & \text{худший случай} \end{cases} \quad (2.7)$$

В итоге, трудоемкость алгоритма Винограда, вычисляется по формуле

$$f_{vinograd} = f_r + f_c + f_{main} + f_{err} = \frac{32}{2}NLQ + \frac{19}{2}LQ + \frac{19}{2}NM + \\ +13NL + 10N + 6L + 9 + \begin{cases} 0, & \text{л. с.} \\ 16LN + 4N + 2, & \text{х. с.} \end{cases} \tag{2.8}$$

### 2.4.3 Трудоемкость оптимизированного алгоритма Винограда

Произведены следующие оптимизации алгоритма Винограда:

— заменим операцию $x = x + k$ на $x+ = k$;

— заменим умножение на 2 на побитовый сдвиг;

— будем предвычислять некоторые слагаемые для алгоритма.

Трудоемкость оптимизированного алгоритма Винограда формируется из следующих частей.

1) Трудоемкости вычисления массива $RowFactor$, которая рассчитывается по формуле

$$f_r = 3 + 2 + N \cdot (2 + 2 + \frac{M}{2} \cdot (2 + 11)) = \frac{13}{2}NM + 4N + 5 \tag{2.9}$$

2) Трудоемкости вычисления массива $ColFactor$, которая рассчитывается, аналогично пункту 1, по формуле

$$f_c = \frac{13}{2}LQ + 4L + 5 \tag{2.10}$$

3) Трудоемкости основной части алгоритма, которая вычисляется по формуле

$$f_{main} = 3 + 2 + N \cdot (2 + 2 + 3 + L \cdot (2 + 5 + 2 + \frac{Q}{2} \cdot (2 + 17))) = \\ = \frac{21}{2}NLQ + 9NL + 7N + 5 \tag{2.11}$$

4) Трудоемкости поправки, вносимой для нечетного $Q$, рассчитываемой по формуле

$$f_{err} = 3 + \begin{cases} 0, & \text{лучший случай} \\ 11LN + 4N + 4, & \text{худший случай} \end{cases} \tag{2.12}$$

В итоге, трудоемкость оптимизированного алгоритма Винограда, вычисляется по формуле

$$\begin{gathered} f_{vinograd} = f_r + f_c + f_{main} + f_{err} = \frac{21}{2}NLQ + \frac{13}{2}LQ + \frac{13}{2}NM + \\ +9NL + 11N + 4L + 15 + \begin{cases} 0, & \text{л. с.} \\ 11LN + 4N + 4, & \text{х. с.} \end{cases} \end{gathered} \tag{2.13}$$

### 2.4.4 Трудоемкость алгоритма Штрассена

Рассчитаем трудоемкость подпрограммы MSR, изображенной на рисунке 2.6. Будем считать затраты на расширение размера матрицы незначительными. Также будем считать, что стоимость срезов равна 1.

Трудоемкость сложения или вычитания двух матриц размера $N \times N$, рассчитывается по формуле

$$f_{sum}(N) = 2 + N \cdot (2 + 2 + N \cdot (2 + 8)) = 10N^2 + 4N + 2 \tag{2.14}$$

Тогда трудоемкость описывается рекуррентной формулой

$$T(n) = \begin{cases} 7, & n == 1 \\ 7T(\frac{n}{2}) + 18f_{sum}(\frac{n}{2}) + 65, & n > 1 \end{cases} \tag{2.15}$$

Тогда возможно рассчитать трудоемкость для матриц порядка $n$ по формуле

$$T(n) = 7^{\log_2 n}T(1) + \sum_{i=0}^{(\log_2 n)-1} (7^i(18f_{sum}(\frac{n}{2^{i+1}}) + 65)) \tag{2.16}$$

## 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 самой маленькой асимптотической сложностью обладает алгоритм Штрассена $O(n^{\log_2 7})$.

Трудоемкость алгоритмов Винограда без оптимизаций, с оптимизациями и классического алгоритма оценивается как $O(NLQ)$. Но у этих алгоритмов различные мультипликаторы: 16, 21/2 и 11 соответственно. Значит, с учетом мультипликаторов оценка трудоемкости алгоритма Винограда без оптимизаций больше, чем оценка трудоемкости классического алгоритма. Но при этом с учетом мультипликаторов оценка трудоемкости алгоритма Винограда с оптимизациями меньше, чем оценка трудоемкости классического алгоритма.

# 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]. Такой выбор обусловлен опытом работы с этим языком программирования. Также данный язык позволяет замерять процессорное время с помощью модуля *time* и в нем присутствует библиотека **numpy** для удобной работы с матрицами.

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*;

— *algorithms.py* — файл, содержащий код реализаций всех алгоритмов умножения матриц;

— *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 классического алгоритма, алгоритмов Винограда с оптимизацией и без. Реализация алгоритма Штрассена приведена на листинге в приложении А.1.

Листинг 3.1 – Функция умножения матриц по классическому алгоритму

```
import numpy as np

def classical_mult(A, B):
    if len(A[0]) != len(B):
        raise ValueError("Matrix shapes doesnt match")

    c = np.zeros((len(A), len(B[0])))

    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                c[i][j] += A[i][k] * B[k][j]

    return c
```

Листинг 3.2 – Функция умножения матриц по алгоритму Винограда

```
import numpy as np

def get_row_factor(A):
    row_factor = [0] * len(A)
    for i in range(len(A)):
        for k in range(len(A[0]) // 2):
            row_factor[i] = row_factor[i] + A[i][2 * k + 1] *
               A[i][2 * k]
    return row_factor


def get_col_factor(B):
    col_factor = [0] * len(B[0])
    for i in range(len(B[0])):
        for k in range(len(B) // 2):
            col_factor[i] = col_factor[i] + B[2 * k + 1][i] *
               B[2 * k][i]
    return col_factor


def vinograd(A, B):
    if len(A[0]) != len(B):
        raise ValueError("Matrix shapes doesnt match")
    c = np.zeros((len(A), len(B[0])))
    row_factor = get_row_factor(A)
    col_factor = get_col_factor(B)
    for i in range(len(A)):
        for j in range(len(B[0])):
            c[i][j] = -row_factor[i] - col_factor[j]
            for k in range(len(B) // 2):
                c[i][j] = c[i][j] + (A[i][2 * k] + B[2 * k +
                   1][j]) * (
                    A[i][2 * k + 1] + B[2 * k][j]
                )
    if len(B) % 2 == 1:
        for i in range(len(A)):
            for j in range(len(B[0])):
                c[i][j] = c[i][j] + A[i][len(B) - 1] * B[len(B)
                   - 1][j]
    return c
```

Листинг 3.3 – Функция умножения матриц по алгоритму Винограда с оптимизацией

```
def get_row_factor_opt(A):
    row_factor = [0] * len(A)
    temp = len(A[0]) // 2
    for i in range(len(A)):
        for k in range(temp):
            row_factor[i] += A[i][(k << 1) + 1] * A[i][k << 1]
    return row_factor


def get_col_factor_opt(B):
    col_factor = [0] * len(B[0])
    temp = len(B) // 2
    for i in range(len(B[0])):
        for k in range(temp):
            col_factor[i] += B[(k << 1) + 1][i] * B[k << 1][i]
    return col_factor


def vinograd_opt(A, B):
    if len(A[0]) != len(B):
        raise ValueError("Matrix shapes doesnt match")
    c = np.zeros((len(A), len(B[0])))
    row_factor = get_row_factor_opt(A)
    col_factor = get_col_factor_opt(B)
    temp = range(len(B) // 2)
    for i in range(len(A)):
        for j in range(len(B[0])):
            buff = -row_factor[i] - col_factor[j]
            for k in temp:
                ks = k << 1
                buff += (A[i][ks] + B[ks + 1][j]) * (A[i][ks +
                   1] + B[ks][j])
            c[i][j] = buff
    if len(B) % 2 == 1:
        for i in range(len(A)):
            for j in range(len(B[0])):
                c[i][j] += A[i][-1] * B[-1][j]
    return c
```

## 3.4 Функциональные тесты

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

Таблица 3.1 – Функциональные тесты для алгоритмов умножения матриц

| **Входные данные** | | **Результат для всех алгоритмов** | |
|---|---|---|---|
| **Матрица 1** | **Матрица 2** | **Ожидаемый результат** | **Фактический результат** |
| $\begin{pmatrix} 1 & 5 & 7 \\ 2 & 6 & 8 \\ 3 & 7 & 9 \end{pmatrix}$ | $\begin{pmatrix} \end{pmatrix}$ | Сообщение об ошибке | Сообщение об ошибке |
| $\begin{pmatrix} 1 & 5 & 7 \end{pmatrix}$ | $\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$ | Сообщение об ошибке | Сообщение об ошибке |
| $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ | $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ | $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ | $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ |
| $\begin{pmatrix} 3 & 5 \\ 2 & 1 \\ 9 & 7 \end{pmatrix}$ | $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ | $\begin{pmatrix} 23 & 31 & 39 \\ 6 & 9 & 12 \\ 37 & 53 & 69 \end{pmatrix}$ | $\begin{pmatrix} 23 & 31 & 39 \\ 6 & 9 & 12 \\ 37 & 53 & 69 \end{pmatrix}$ |
| $\begin{pmatrix} 10 \end{pmatrix}$ | $\begin{pmatrix} 35 \end{pmatrix}$ | $\begin{pmatrix} 350 \end{pmatrix}$ | $\begin{pmatrix} 350 \end{pmatrix}$ |
| $\begin{pmatrix} 1 & 5 & 7 \\ 2 & 6 & 8 \\ 3 & 7 & 9 \end{pmatrix}$ | $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ | $\begin{pmatrix} 70 & 83 & 96 \\ 82 & 98 & 114 \\ 94 & 113 & 132 \end{pmatrix}$ | $\begin{pmatrix} 70 & 83 & 96 \\ 82 & 98 & 114 \\ 94 & 113 & 132 \end{pmatrix}$ |

## Вывод

Были разработаны и протестированы спроектированные алгоритмы: классического умножения матриц, алгоритм Винограда, оптимизированный алгоритм Винограда, а также алгоритм Штрассена.

# 4 Исследовательский раздел

В данном разделе будут приведены примеры работы программ, 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 умножения матриц $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ и $B = \begin{pmatrix} 3 & 6 & 7 \\ 2 & 5 & 8 \\ 1 & 4 & 9 \end{pmatrix}$.

```
МЕНЮ:

1 - Умножить две матрицы всеми способами
2 - Провести замерный эксперимент
3 - Вывести функциональные тесты
0 - Выйти

Выберите пункт меню:

1
Введите n, m матрицы A через пробел: 3 3
Введите матрицу:
Строка 1: 1 2 3
Строка 2: 4 5 6
Строка 3: 7 8 9
Введите n, m матрицы B через пробел: 3 3
Введите матрицу:
Строка 1: 3 6 7
Строка 2: 2 5 8
Строка 3: 1 4 9
Результат умножения по классическому алгоритму:
[[ 10.  28.  50.]
 [ 28.  73. 122.]
 [ 46. 118. 194.]]
Результат умножения по алгоритму Винограда:
[[ 10.  28.  50.]
 [ 28.  73. 122.]
 [ 46. 118. 194.]]
Результат умножения по алгоритму Винограда с оптимизацией:
[[ 10.  28.  50.]
 [ 28.  73. 122.]
 [ 46. 118. 194.]]
Результат умножения по алгоритму Штрассена:
[[ 10.  28.  50.   0.]
 [ 28.  73. 122.   0.]
 [ 46. 118. 194.   0.]
 [  0.   0.   0.   0.]]
```

Рисунок 4.1 – Демонстрация работы программы при умножении двух матриц

На рисунке 4.2 представлена демонстрация работы разработанного программного обеспечения, а именно показано, что невозможно умножить матрицы $A = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$ и $B = \begin{pmatrix} 4 & 5 & 2 \\ 6 & 3 & 1 \end{pmatrix}$.

```
МЕНЮ:

1 - Умножить две матрицы всеми способами
2 - Провести замерный эксперимент
3 - Вывести функциональные тесты
0 - Выйти

Выберите пункт меню:

1
Введите n, m матрицы A через пробел: 3 1
Введите матрицу:
Строка 1: 1
Строка 2: 2
Строка 3: 3
Введите n, m матрицы B через пробел: 2 3
Введите матрицу:
Строка 1: 4 5 2
Строка 2: 6 3 1
Матрицы нельзя умножить

МЕНЮ:
```

Рисунок 4.2 – Демонстрация работы программы при умножении двух матриц

## 4.2 Технические характеристики

Технические характеристики компьютера, на котором выполнялись замеры по времени.

— 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 умножения матриц приведены в таблицах 4.1 – 4.3. Замеры времени проводились на квадратных матрицах одного порядка и усреднялись для каждого набора одинаковых экспериментов. В таблицах 4.1 – 4.3 используются следующие обозначения:

— К — реализация классического алгоритма умножения матриц;

— В — реализация алгоритма Винограда умножения матриц;

— ВО — реализация алгоритма Винограда с оптимизацией умножения матриц;

— Ш — реализация алгоритма Штрассена умножения матриц.

Таблица 4.1 – Время работы реализации алгоритмов (в с)

| Порядок матриц | К | В | ВО | Ш |
|---|---|---|---|---|
| 6 | $0.19 \cdot 10^{-3}$ | $0.19 \cdot 10^{-3}$ | $0.16 \cdot 10^{-3}$ | $1.31 \cdot 10^{-3}$ |
| 11 | $0.91 \cdot 10^{-3}$ | $1.12 \cdot 10^{-3}$ | $0.94 \cdot 10^{-3}$ | $9.16 \cdot 10^{-3}$ |
| 16 | $2.78 \cdot 10^{-3}$ | $3.28 \cdot 10^{-3}$ | $2.47 \cdot 10^{-3}$ | $9.28 \cdot 10^{-3}$ |
| 21 | $6.47 \cdot 10^{-3}$ | $6.84 \cdot 10^{-3}$ | $5.25 \cdot 10^{-3}$ | $64.82 \cdot 10^{-3}$ |
| 26 | $11.42 \cdot 10^{-3}$ | $12.72 \cdot 10^{-3}$ | $10.31 \cdot 10^{-3}$ | $65.63 \cdot 10^{-3}$ |
| 31 | $20.05 \cdot 10^{-3}$ | $22.06 \cdot 10^{-3}$ | $17.15 \cdot 10^{-3}$ | $65.40 \cdot 10^{-3}$ |
| 36 | $30.94 \cdot 10^{-3}$ | $34.08 \cdot 10^{-3}$ | $25.62 \cdot 10^{-3}$ | $460.05 \cdot 10^{-3}$ |
| 41 | $46.22 \cdot 10^{-3}$ | $51.39 \cdot 10^{-3}$ | $37.77 \cdot 10^{-3}$ | $453.52 \cdot 10^{-3}$ |
| 46 | $62.93 \cdot 10^{-3}$ | $70.60 \cdot 10^{-3}$ | $54.02 \cdot 10^{-3}$ | $477.28 \cdot 10^{-3}$ |
| 51 | $87.80 \cdot 10^{-3}$ | $94.71 \cdot 10^{-3}$ | $73.72 \cdot 10^{-3}$ | $454.81 \cdot 10^{-3}$ |

Таблица 4.2 – Время работы реализации алгоритмов для четных порядков матриц (в с)

| Порядок матриц | К | В | ВО | Ш |
|---|---|---|---|---|
| 6 | $0.19 \cdot 10^{-3}$ | $0.19 \cdot 10^{-3}$ | $0.16 \cdot 10^{-3}$ | $1.37 \cdot 10^{-3}$ |
| 10 | $0.72 \cdot 10^{-3}$ | $0.81 \cdot 10^{-3}$ | $0.63 \cdot 10^{-3}$ | $9.31 \cdot 10^{-3}$ |
| 14 | $1.84 \cdot 10^{-3}$ | $2.16 \cdot 10^{-3}$ | $1.75 \cdot 10^{-3}$ | $9.16 \cdot 10^{-3}$ |
| 18 | $3.87 \cdot 10^{-3}$ | $4.38 \cdot 10^{-3}$ | $3.34 \cdot 10^{-3}$ | $65.92 \cdot 10^{-3}$ |
| 22 | $7.16 \cdot 10^{-3}$ | $7.78 \cdot 10^{-3}$ | $6.09 \cdot 10^{-3}$ | $66.31 \cdot 10^{-3}$ |
| 26 | $11.42 \cdot 10^{-3}$ | $12.93 \cdot 10^{-3}$ | $10.44 \cdot 10^{-3}$ | $66.22 \cdot 10^{-3}$ |
| 30 | $18.37 \cdot 10^{-3}$ | $19.62 \cdot 10^{-3}$ | $15.30 \cdot 10^{-3}$ | $67.03 \cdot 10^{-3}$ |
| 34 | $25.84 \cdot 10^{-3}$ | $28.27 \cdot 10^{-3}$ | $21.37 \cdot 10^{-3}$ | $453.68 \cdot 10^{-3}$ |
| 38 | $35.18 \cdot 10^{-3}$ | $39.05 \cdot 10^{-3}$ | $30.06 \cdot 10^{-3}$ | $451.90 \cdot 10^{-3}$ |
| 42 | $47.32 \cdot 10^{-3}$ | $52.01 \cdot 10^{-3}$ | $41.03 \cdot 10^{-3}$ | $477.52 \cdot 10^{-3}$ |
| 46 | $64.52 \cdot 10^{-3}$ | $69.98 \cdot 10^{-3}$ | $52.35 \cdot 10^{-3}$ | $471.24 \cdot 10^{-3}$ |
| 50 | $79.81 \cdot 10^{-3}$ | $87.32 \cdot 10^{-3}$ | $68.66 \cdot 10^{-3}$ | $478.76 \cdot 10^{-3}$ |

Таблица 4.3 – Время работы реализации алгоритмов для нечетных порядков матриц (в с)

| Порядок матриц | К | В | ВО | Ш |
|---|---|---|---|---|
| 7 | $0.25 \cdot 10^{-3}$ | $0.31 \cdot 10^{-3}$ | $0.25 \cdot 10^{-3}$ | $1.34 \cdot 10^{-3}$ |
| 11 | $0.91 \cdot 10^{-3}$ | $1.12 \cdot 10^{-3}$ | $0.91 \cdot 10^{-3}$ | $9.78 \cdot 10^{-3}$ |
| 15 | $2.37 \cdot 10^{-3}$ | $2.66 \cdot 10^{-3}$ | $2.06 \cdot 10^{-3}$ | $9.81 \cdot 10^{-3}$ |
| 19 | $4.75 \cdot 10^{-3}$ | $5.22 \cdot 10^{-3}$ | $4.09 \cdot 10^{-3}$ | $65.44 \cdot 10^{-3}$ |
| 23 | $7.94 \cdot 10^{-3}$ | $9.19 \cdot 10^{-3}$ | $7.16 \cdot 10^{-3}$ | $64.98 \cdot 10^{-3}$ |
| 27 | $13.42 \cdot 10^{-3}$ | $14.93 \cdot 10^{-3}$ | $10.83 \cdot 10^{-3}$ | $65.31 \cdot 10^{-3}$ |
| 31 | $19.23 \cdot 10^{-3}$ | $21.42 \cdot 10^{-3}$ | $16.72 \cdot 10^{-3}$ | $65.05 \cdot 10^{-3}$ |
| 35 | $27.95 \cdot 10^{-3}$ | $30.75 \cdot 10^{-3}$ | $23.66 \cdot 10^{-3}$ | $459.08 \cdot 10^{-3}$ |
| 39 | $38.71 \cdot 10^{-3}$ | $43.27 \cdot 10^{-3}$ | $33.24 \cdot 10^{-3}$ | $457.94 \cdot 10^{-3}$ |
| 43 | $52.67 \cdot 10^{-3}$ | $57.76 \cdot 10^{-3}$ | $44.79 \cdot 10^{-3}$ | $479.53 \cdot 10^{-3}$ |
| 47 | $68.56 \cdot 10^{-3}$ | $75.01 \cdot 10^{-3}$ | $57.13 \cdot 10^{-3}$ | $450.26 \cdot 10^{-3}$ |
| 51 | $88.12 \cdot 10^{-3}$ | $96.09 \cdot 10^{-3}$ | $73.70 \cdot 10^{-3}$ | $481.89 \cdot 10^{-3}$ |

На рисунках 4.3 – 4.5 изображены графики зависимостей времени выполнения реализаций алгоритмов от порядка умножаемых матриц.

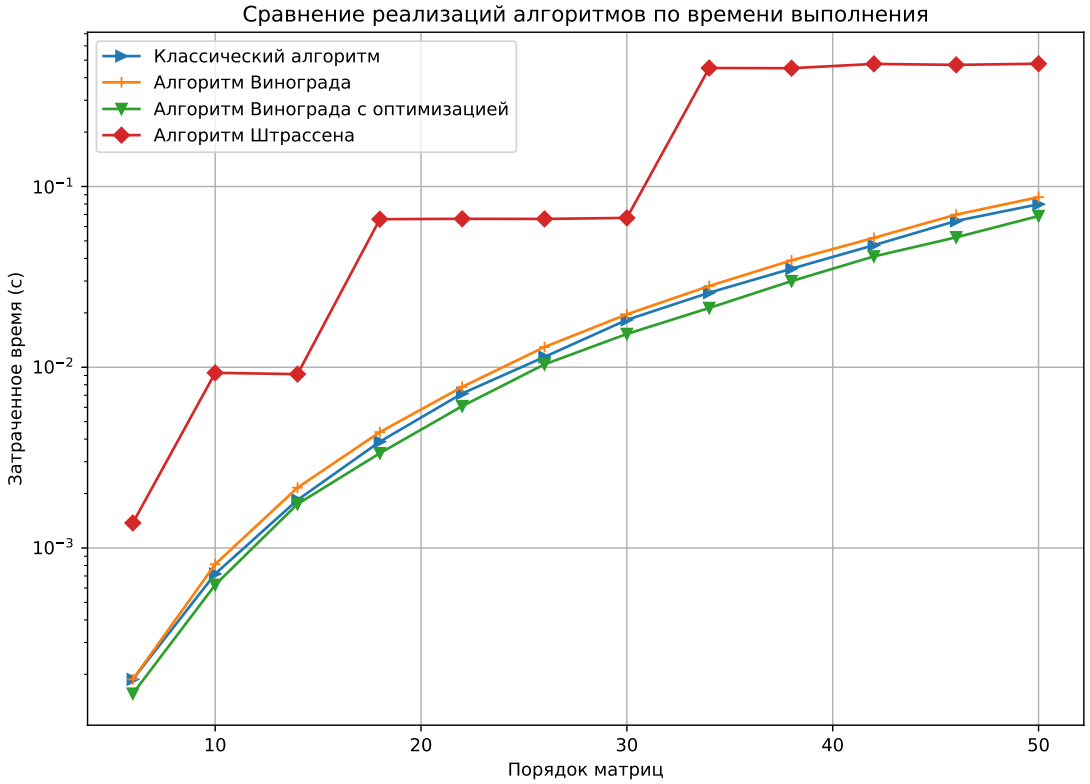


Рисунок 4.3 – Сравнение реализаций алгоритмов по времени выполнения

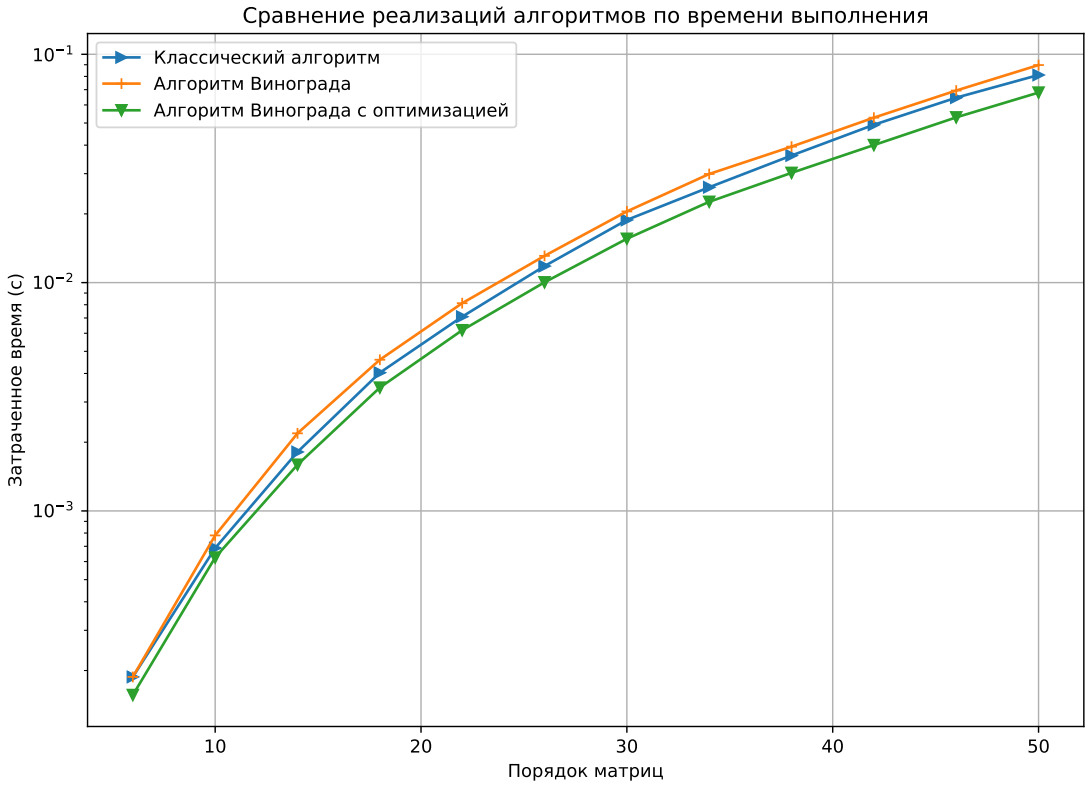


Рисунок 4.4 – Сравнение реализаций алгоритмов по времени выполнения для четных порядков матриц

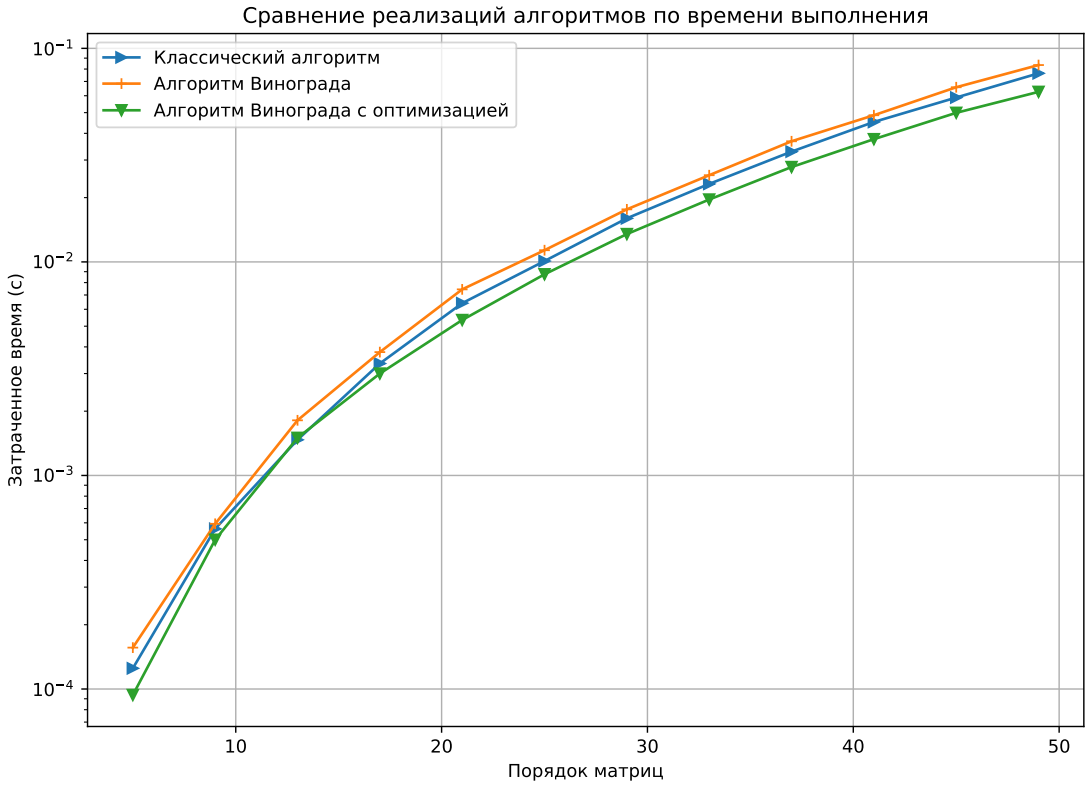


Рисунок 4.5 – Сравнение реализаций алгоритмов по времени выполнения для нечетных порядков матриц

## Вывод

В результате замеров времени выполнения реализаций различных алгоритмов было получено, что для матриц с порядком 51, реализация алгоритма Винограда оказалась в 1.08 раз хуже реализации классического алгоритма по времени выполнения, при этом реализация алгоритма Винограда с оптимизациями оказалась лучше в 1.19 раз реализации классического алгоритма и в 1.28 раз лучше реализации без оптимизаций по времени выполнения. Для матриц с четным порядком 50, реализация алгоритма Винограда хуже реализации классического алгоритма в 1.09 раз по времени выполнения, а реализация алгоритма Винограда с оптимизацией оказалась лучше классического алгоритма в 1.16 раз.

Для матриц с порядком 51, реализация алгоритма Штрассена оказалась хуже по времени выполнения в 5.17 раз, в 4.79 раз и в 6.16 раз, чем реализации классического алгоритма, алгоритма Винограда и алгоритма Винограда с оптимизациями соответственно. Такой результат обусловлен тем, что в данной реализации много времени тратится на выделение подматриц, их сложение и вычитание, а также на рекурсивные вызовы. По теоретической оценке трудоемкости, можно заметить насколько велика константа перед $n^{\log_2 7}$.

Стоит заметить, что на графике зависимости времени выполнения реализаций алгоритмов от порядка матриц 4.3, у реализации алгоритма Штрассена присутствуют отчетливые ступеньки. Это обусловлено особенностью алгоритма, который может работать только с квадратными матрицами порядка $2^k$, а матрицы других размеров приводятся к данному размеру дополнением нулевыми строками и столбцами. Таким образом, разные размеры $n$ могу приводится к одному новому порядку $2^k$, что в свою очередь приводит к почти идентичному времени обработки.

Полученные на практике результаты примерно соответствуют теоретическим оценкам.

# ЗАКЛЮЧЕНИЕ

В результате исследования реализаций алгоритмов было выявлено, что для матриц с порядком 51, реализация алгоритма Винограда оказалась в 1.08 раз хуже реализации классического алгоритма по времени выполнения, при этом реализация алгоритма Винограда с оптимизациями оказалась лучше в 1.19 раз реализации классического алгоритма и в 1.28 раз лучше реализации без оптимизаций по времени выполнения. Для матриц с четным порядком 50, реализация алгоритма Винограда хуже реализации классического алгоритма в 1.09 раз по времени выполнения, а реализация алгоритма Винограда с оптимизацией оказалась лучше классического алгоритма в 1.16 раз.

Для матриц с порядком 51, реализация алгоритма Штрассена оказалась хуже по времени выполнения в 5.17 раз, в 4.79 раз и в 6.16 раз, чем реализации классического алгоритма, алгоритма Винограда и алгоритма Винограда с оптимизациями соответственно.

Цель данной лабораторной работы была достигнута: были исследованы алгоритмы умножения матриц.

Были выполнены следующие задачи.

1) Описаны следующие алгоритмы умножения матриц:
   - классический алгоритм;
   - алгоритм Винограда;
   - алгоритм Штрассена.
2) Проведена оптимизация алгоритма Винограда.
3) Разработано программное обеспечение, реализующее алгоритмы умножения.
4) Выбраны инструменты для реализации алгоритмов и замера процессорного времени их выполнения.
5) Проведен анализ затрат реализаций алгоритмов по времени.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. Конспект лекций, читаемых в курсе «Высшая математика» Южного федерального университета. — Режим доступа: http://mmtb.uginfo.sfedu.ru/algebra/Print/print_I-1.pdf (дата обращения: 07.12.2023).
2. *Головашкин Д. Л.* Векторные алгоритмы вычислительной линейной алгебры: учеб. пособие. //. — — Самара: Изд-во Самарского университета, 2019. — С. 28—35.
3. Gaussian Elimination is not Optimal. — Режим доступа: https://www2.math.uconn.edu/~leykekhman/courses/MATH_5510/fa_2016/papers/fast_matrix.pdf (дата обращения: 07.12.2023).
4. The official home of the Python Programming Language [Электронный ресурс]. — Режим доступа: https://www.python.org/ (дата обращения: 07.12.2023).
5. time — Time access and conversions [Электронный ресурс]. — Режим доступа: https://docs.python.org/3/library/time.html (дата обращения: 07.12.2023).
6. Intel Core i5-10400F. — Режим доступа: https://openbenchmarking.org/s/Intel+Core+i5-10400F (дата обращения 07.12.2023).
7. Windows 10 Pro 22h2 64-bit [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/ru-ru/software-download/windows10 (дата обращения: 07.12.2023).

# ПРИЛОЖЕНИЕ А
## Реализация алгоритма Штрассена

Листинг А.1 – Функция умножения матриц по алгоритму Штрассена

```
import numpy as np

def pad_to_power_of_two(A, B):
    n1, m1 = A.shape
    n2, m2 = B.shape

    n = max(n1, n2)
    m = max(m1, m2)

    target_size = 2 ** int(np.ceil(np.log2(max(m, n))))

    temp_A = np.zeros((target_size, target_size))
    temp_B = np.zeros((target_size, target_size))

    temp_A[:n1, :m1] = A
    temp_B[:n2, :m2] = B

    return temp_A, temp_B


def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0


def strassen(A, B):
    if len(A[0]) != len(B):
        raise ValueError("Matrix shapes doesnt match")

    temp_A, temp_B = A, B
    if A.shape != B.shape or not is_power_of_two(len(A)):
        temp_A, temp_B = pad_to_power_of_two(A, B)

    def MSR(A, B):
        if len(A) == 1:
            return A @ B

        N = len(A)
```

```
38
39          A11 = A[: N // 2, : N // 2]
40          A12 = A[: N // 2, N // 2 :]
41          A21 = A[N // 2 :, : N // 2]
42          A22 = A[N // 2 :, N // 2 :]
43
44          B11 = B[: N // 2, : N // 2]
45          B12 = B[: N // 2, N // 2 :]
46          B21 = B[N // 2 :, : N // 2]
47          B22 = B[N // 2 :, N // 2 :]
48
49          M1 = MSR(A11 + A22, B11 + B22)
50          M2 = MSR(A21 + A22, B11)
51          M3 = MSR(A11, B12 - B22)
52          M4 = MSR(A22, B21 - B11)
53          M5 = MSR(A11 + A12, B22)
54          M6 = MSR(A21 - A11, B11 + B12)
55          M7 = MSR(A12 - A22, B21 + B22)
56
57          C11 = M1 + M4 - M5 + M7
58          C12 = M3 + M5
59          C21 = M2 + M4
60          C22 = M1 - M2 + M3 + M6
61
62          C = np.zeros((N, N))
63          C[: N // 2, : N // 2] = C11
64          C[: N // 2, N // 2 :] = C12
65          C[N // 2 :, : N // 2] = C21
66          C[N // 2 :, N // 2 :] = C22
67
68          return C
69
70      return MSR(temp_A, temp_B)
```