**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 дисциплине «Анализ алгоритмов»

**Тема** Алгоритмы умножения матриц

**Студент** Алькина А.Р.

**Группа** ИУ7-54Б

**Оценка (баллы)** ______________________

**Преподаватели** Волкова Л. Л., Строганов Д. В.

Москва — 2023 г.

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

**Матрицами** называются массивы элементов, представленные в виде прямоугольных таблиц, для которых определены правила математических действий. Элементами матрицы могут являться числа, алгебраические символы или математические функции [1].

Матричная алгебра имеет обширные применения в различных отраслях знания – в математике, физике, информатике, экономике. Например, матрицы используется для решения систем алгебраических и дифференциальных уравнений, нахождения значений физических величин в квантовой теории, шифрования сообщений в Интернете [1].

Цель лабораторной работы — изучение, реализация и сравнение алгоритмов умножения матриц. В данной лабораторной работе рассматриваются стандартный алгоритм умножения матриц (с оптимизациями и без), алгоритм Винограда умножения матриц (с оптимизациями и без), алгоритм Штрассена умножения матриц (с оптимизациями и без).

Задачи лабораторной работы:

— исследовать алгоритмы умножения матриц;

— провести оптимизацию алгоритмов умножения матриц;

— применить методы динамического программирования для реализации алгоритмов умножения матриц;

— оценить и сравнить трудоёмкость алгоритмов;

— провести сравнительный анализ по времени и памяти на основе экспериментальных данных;

— подготовить отчет по лабораторной работе.

# 1 Аналитическая часть

Умножение матриц является основным инструментом линейной алгебры и имеет многочисленные применения в математике, физике, программировании [2]. Одним из самых эффективных по времени алгоритмов умножения матриц является алгоритм Винограда.

Пусть матрицы $A$ и $B$ заданы следующим образом:

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

$$B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1k} \\ b_{21} & b_{22} & \dots & b_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nk} \end{pmatrix}, \tag{1.2}$$

где:

— $m$ — количество строк в матрице $A$;

— $n$ — количество столбцов в матрице $A$ и количество строк в матрице $B$;

— $k$ — количество столбцов в матрице $B$.

Количество столбцов матрицы $A$ должно быть равно количеству строк матрицы $B$, чтобы можно было осуществить умножение. При этом результирующая матрица $A \times B$ будет иметь размер $[m \times k]$.

Каждый элемент результирующей матрицы (обозначим её $C$) вычисляется как:

$$c_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj}, \tag{1.3}$$

где:

— $i = 1 \dots m$;

— $j = 1 \dots k$;

— $r = 1 \ldots n$.

Алгоритмическая сложность классического алгоритма умножения матриц $O(n^3)$, так как необходимо использовать три цикла.

## 1.1. Алгоритм Винограда умножения матриц

Каждый элемент в результирующей матрице произведении матриц представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее, перед основными вложенными циклами умножения матриц.

Рассмотрим два вектора $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно:

$$V \cdot W = v_1w_1 + v_2w_2 + v_3w_3 + v_4w_4. \tag{1.4}$$

Это равенство можно переписать в виде:

$$V \cdot W = (v_1+w_2)(v_2+w_1)+(v_3+w_4)(v_4+w_3)-v_1v_2-v_3v_4-w_1w_2-w_3w_4. \tag{1.5}$$

Выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй.

Таким образом, количество умножений по сравнению с классически расписанных скалярным произведением векторов (4) уменьшено на два и составляет два умножения.

## 1.2. Алгоритм Штрассена умножения матриц

Алгоритм Штрассена – это разработанный Фолькером Штрассеном в 1969 году алгоритм, предназначенный для более быстрого умножения матриц. Он является обобщением метода умножения Карацубы на матрицы и предлагает более эффективный подход к умножению крупных матриц.

В отличие от классического алгоритма умножения, алгоритм Штрассена использует рекурсию и разделяет умножение матриц на подзадачи меньшего размера. Затем он комбинирует результаты рекурсивных вычислений, используя несколько предварительно вычисленных слагаемых. Это позволяет снизить число операций умножения и, в результате, улучшить время выполнения алгоритма.

Пусть $A, B$ — матрицы размера $2^k \times 2^k$. Их можно представить как блочные матрицы $2 \times 2$ из $2^{k-1} \times 2^{k-1}$ матриц:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \tag{1.6}$$

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \tag{1.7}$$

По принципу блочного умножения матрица $AB$ выражается следующим образом:

$$AB = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}. \tag{1.8}$$

Штрассен предложил модифицировать это выражение, чтобы получить семь умножений вместо восьми:

$$D = (A_{11} + A_{22})(B_{11} + B_{22}), \tag{1.9}$$

$$D_1 = (A_{12} - A_{22})(B_{21} + B_{22}), \tag{1.10}$$

$$D_2 = (A_{21} - A_{11})(B_{11} + B_{12}), \tag{1.11}$$

$$H_1 = (A_{11} + A_{12})B_{22}, \tag{1.12}$$

$$H_2 = (A_{21} + A_{22})B_{11}, \tag{1.13}$$

$$V_1 = A_{22}(B_{21} - B_{11}), \tag{1.14}$$

$$V_2 = A_{11}(B_{12} - B_{22}), \tag{1.15}$$

$$AB = \begin{pmatrix} D + D_1 + V_1 - H_1 & V_2 + H_1 \\ V_1 + H_2 & D + D_2 + V_2 - H_2 \end{pmatrix}. \tag{1.16}$$

# Вывод

Были рассмотрены алгоритмы умножения матриц: классическое умножение матриц, алгоритм Винограда, алгоритм Штрассена. Основное двух последних алгоритмов в отличие от классического алгоритма умножения матриц состоит в выполнении некоторой части вычислений предварительно и, как следствие, в уменьшении количества умножений, которые является более дорогостоящей операцией в смысле трудоёмкости алгоритмов.

# 2 Конструкторская часть

## 2.1. Разработка классического алгоритма умножения матриц

На рисунке 2.1 приведена схема классического алгоритма умножения матриц без оптимизаций.

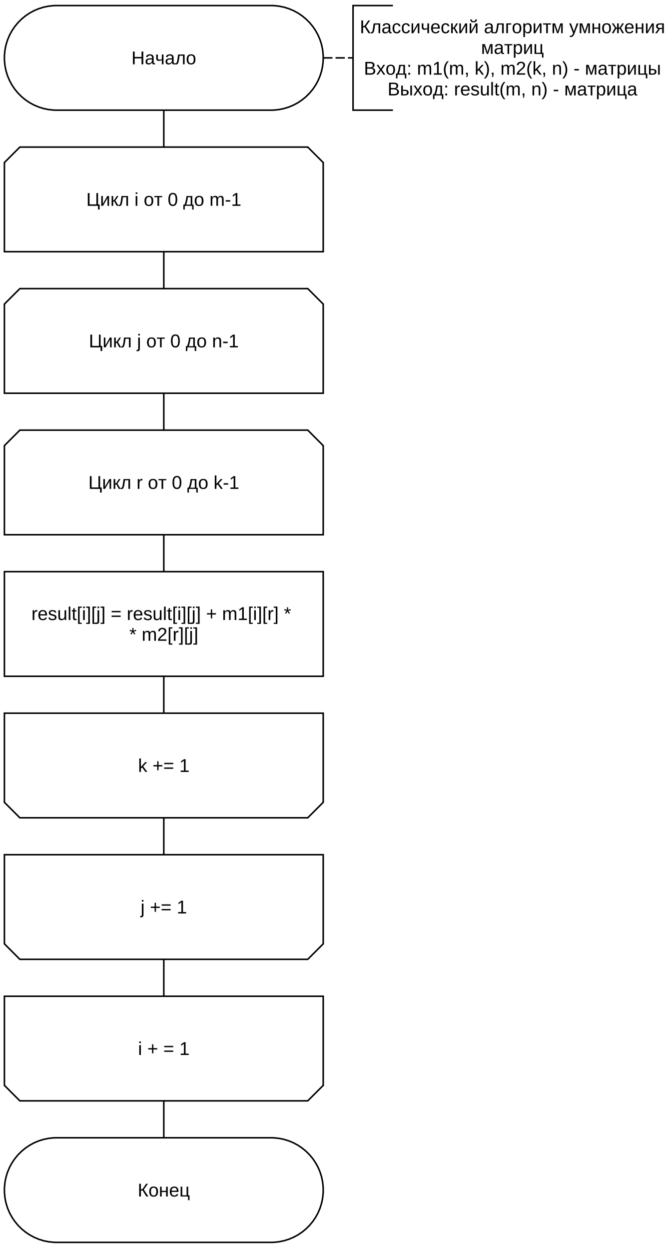


Рисунок 2.1 – Схема классического алгоритма умножения матриц без оптимизаций

## 2.2. Разработка классического алгоритма умножения матриц с оптимизацией

На рисунке 2.2 приведена схема классического алгоритма умножения матриц с заменой $a = a + b$ на $a+ = b$.

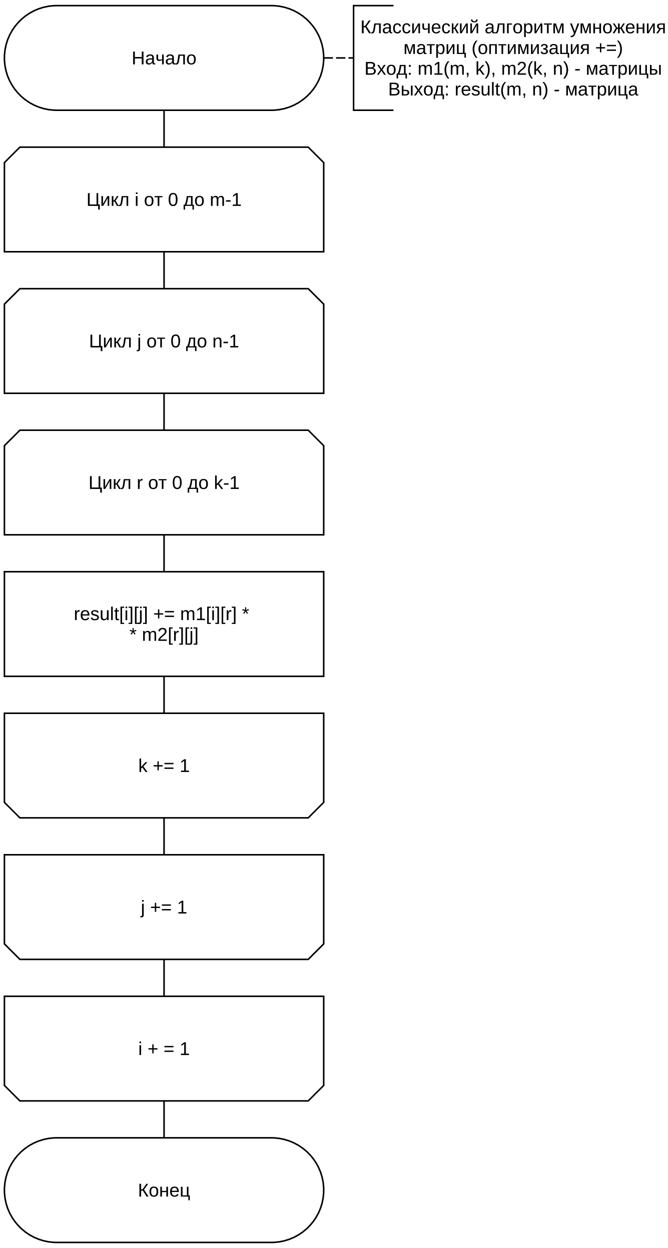


Рисунок 2.2 – Схема классического алгоритма умножения матриц с заменой $a = a + b$ на $a+ = b$

## 2.3. Разработка алгоритма Винограда умножения матриц без оптимизаций

На рисунках 2.3 — 2.7 приведена схема алгоритма Винограда умножения матриц без оптимизаций.

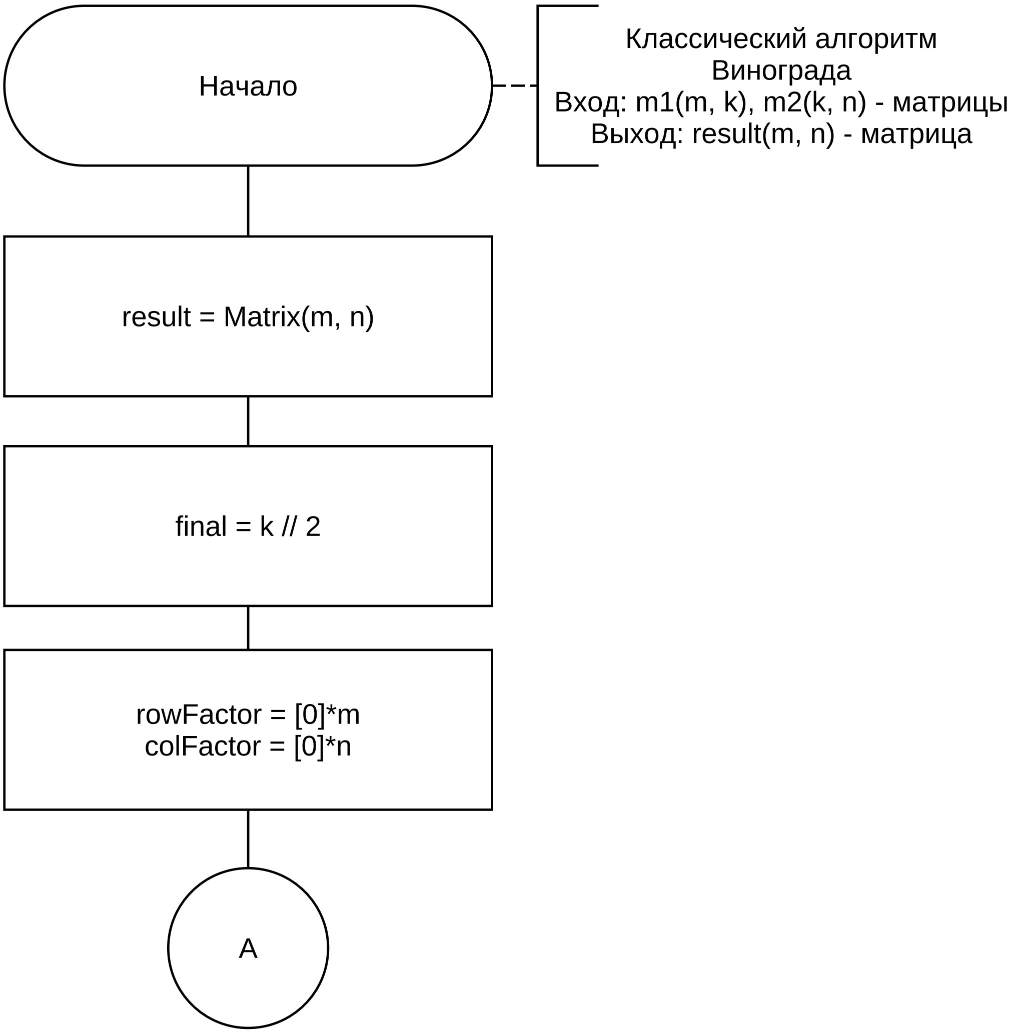


Рисунок 2.3 – Схема алгоритма Винограда умножения матриц без оптимизаций: начальная инициализация

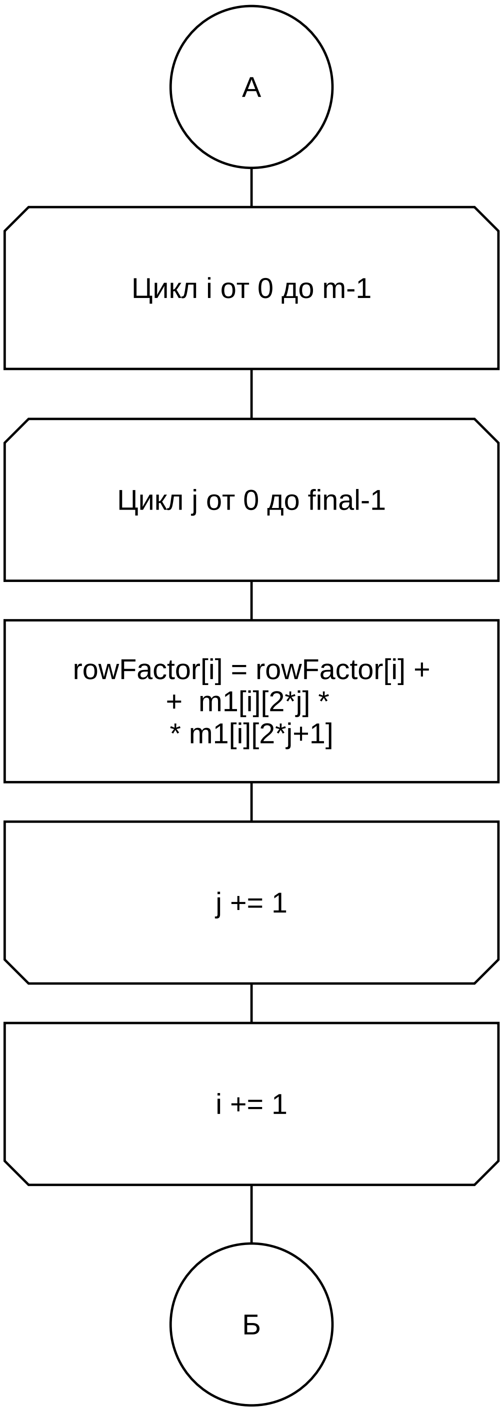


Рисунок 2.4 – Схема алгоритма Винограда умножения матриц без оптимизаций: предвычисление для строк матрицы

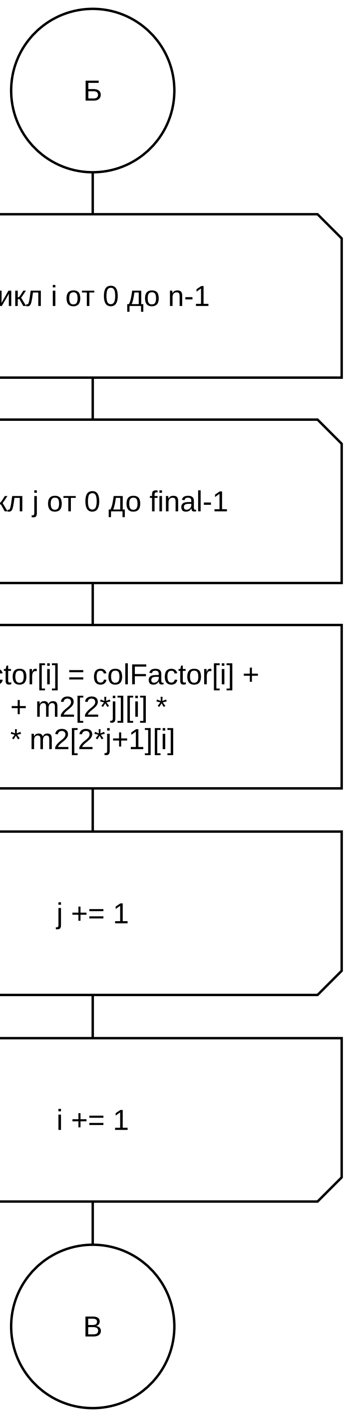


Рисунок 2.5 – Схема алгоритма Винограда умножения матриц без оптимизаций: предвычисление для столбцов матрицы

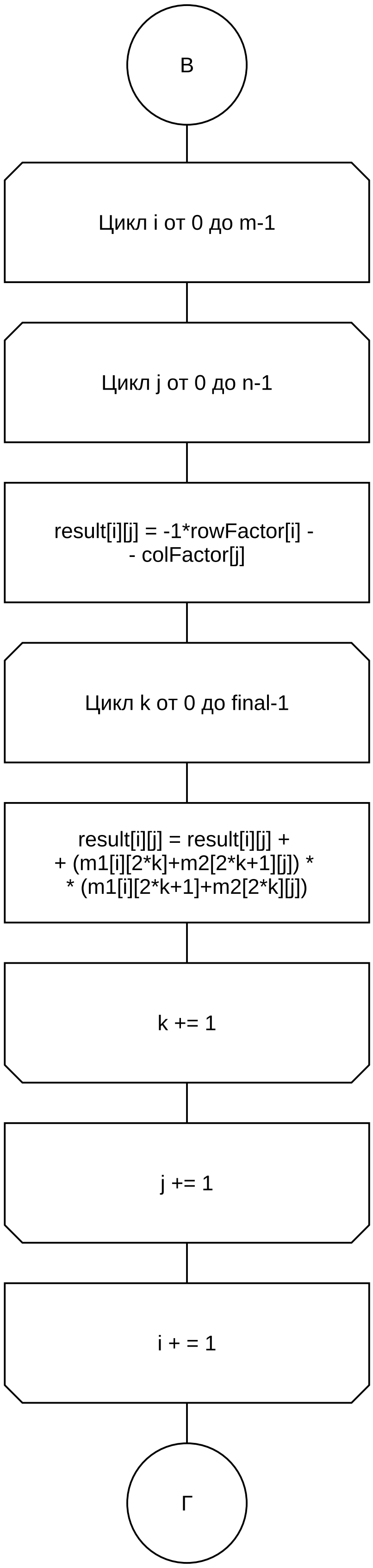


Рисунок 2.6 – Схема алгоритма Винограда умножения матриц без оптимизаций: основной цикл умножения

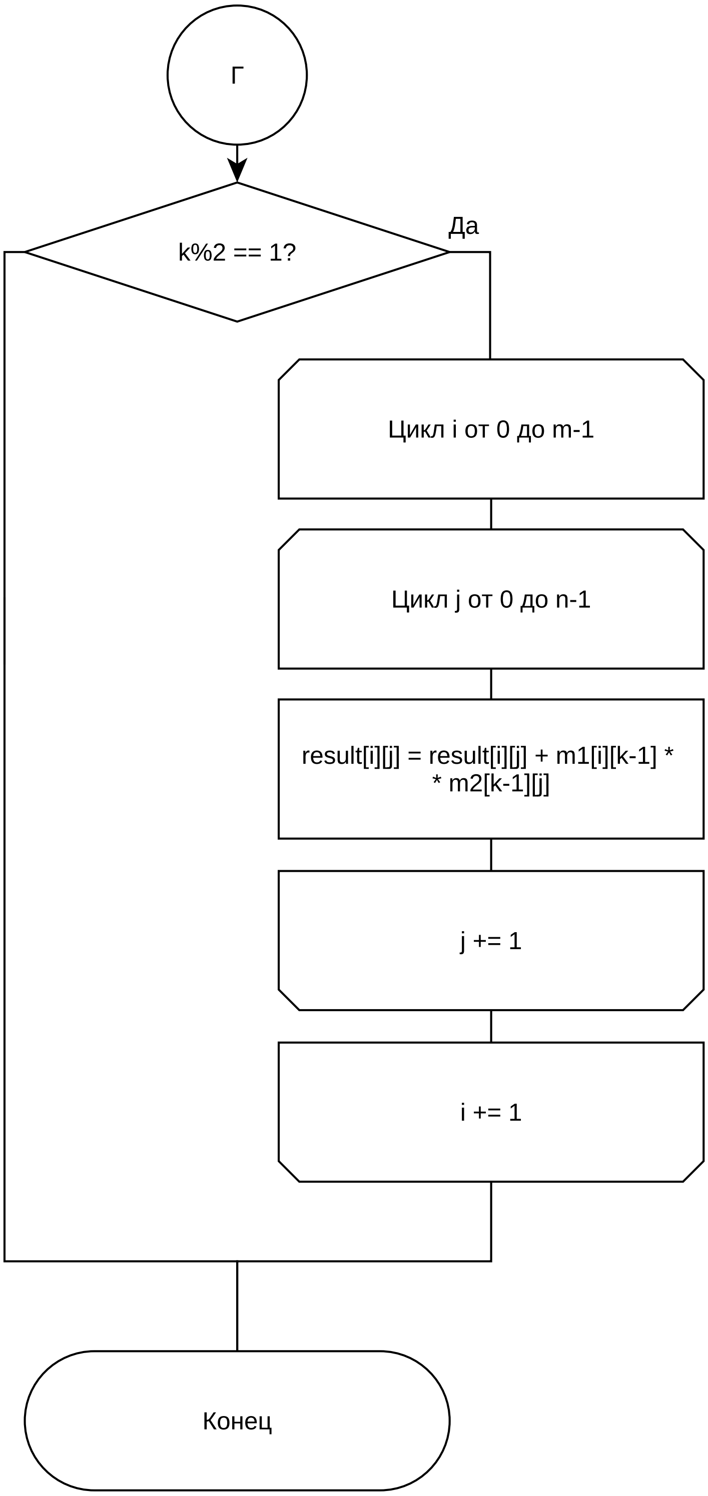


Рисунок 2.7 – Схема алгоритма Винограда умножения матриц без оптимизаций: обработка нечётной размерности

## 2.4. Разработка алгоритма Винограда умножения матриц с оптимизациями

На рисунках 2.8 — 2.12 приведена схема алгоритма Винограда умножения матриц с заменой умножения и деления на два на побитовый сдвиг, $a = a + b$ на $a+ = b$.

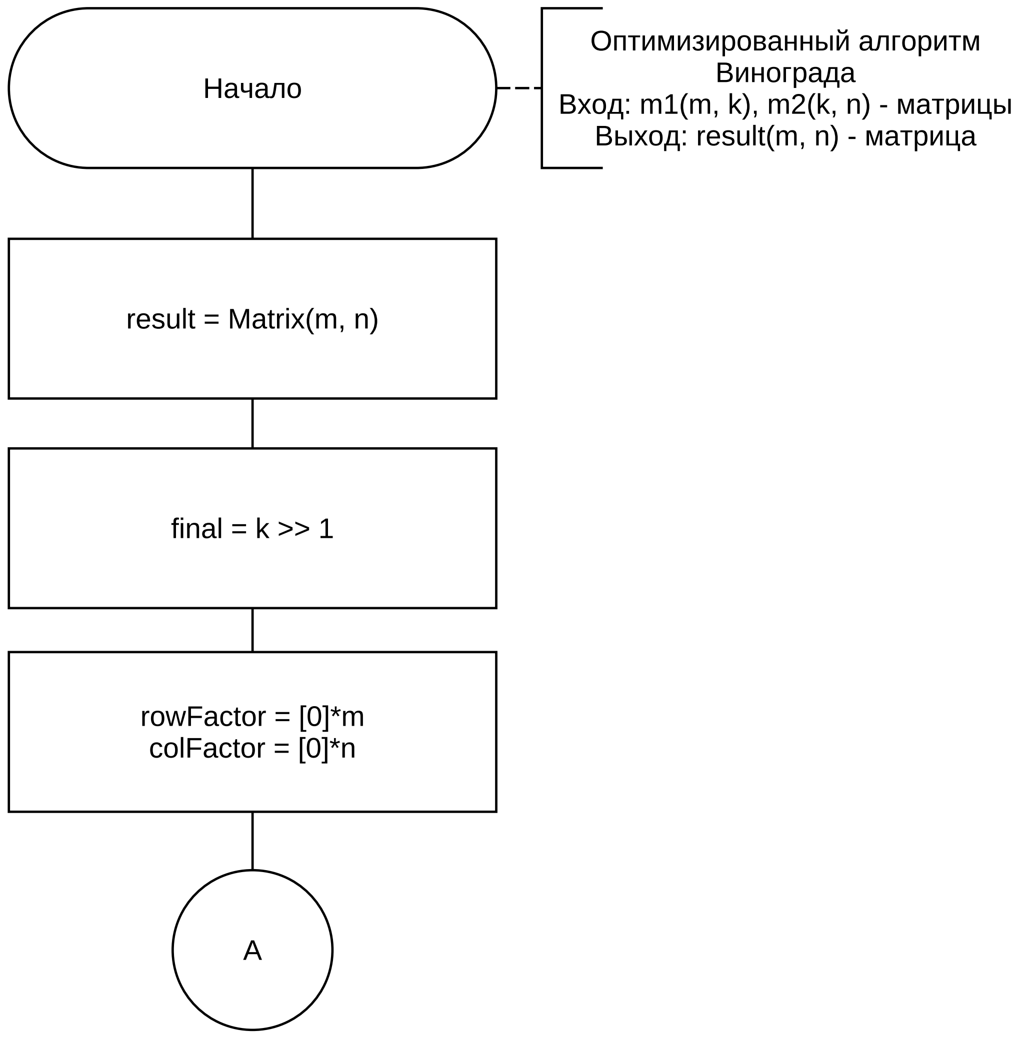


Рисунок 2.8 – Схема оптимизированного алгоритма Винограда умножения матриц: начальная инициализация

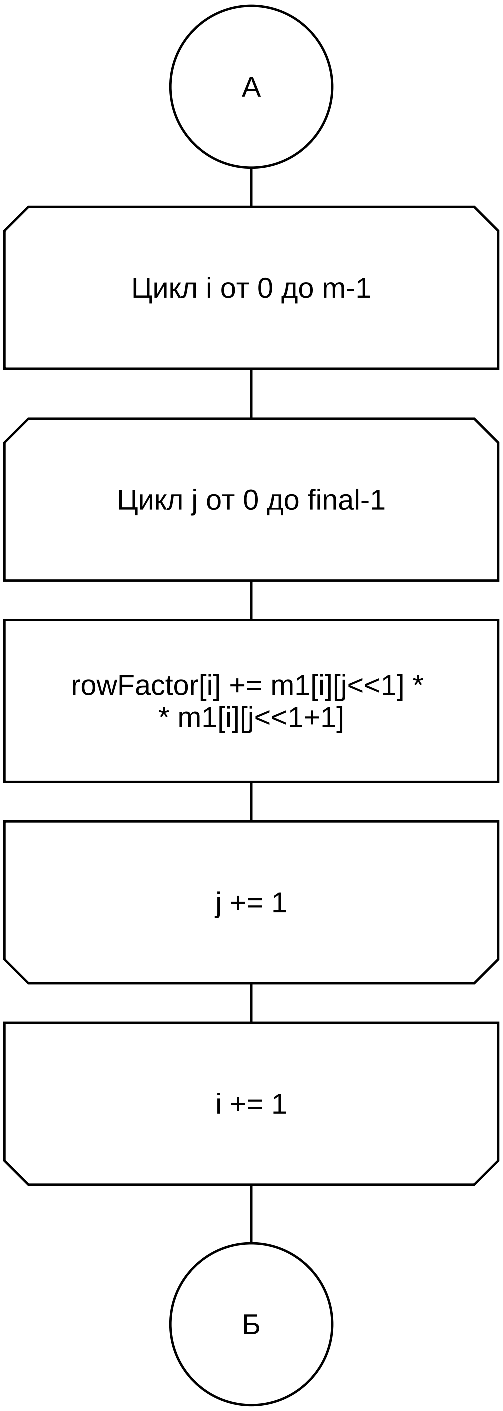


Рисунок 2.9 – Схема оптимизированного алгоритма Винограда умножения матриц: предвычисление для строк матрицы

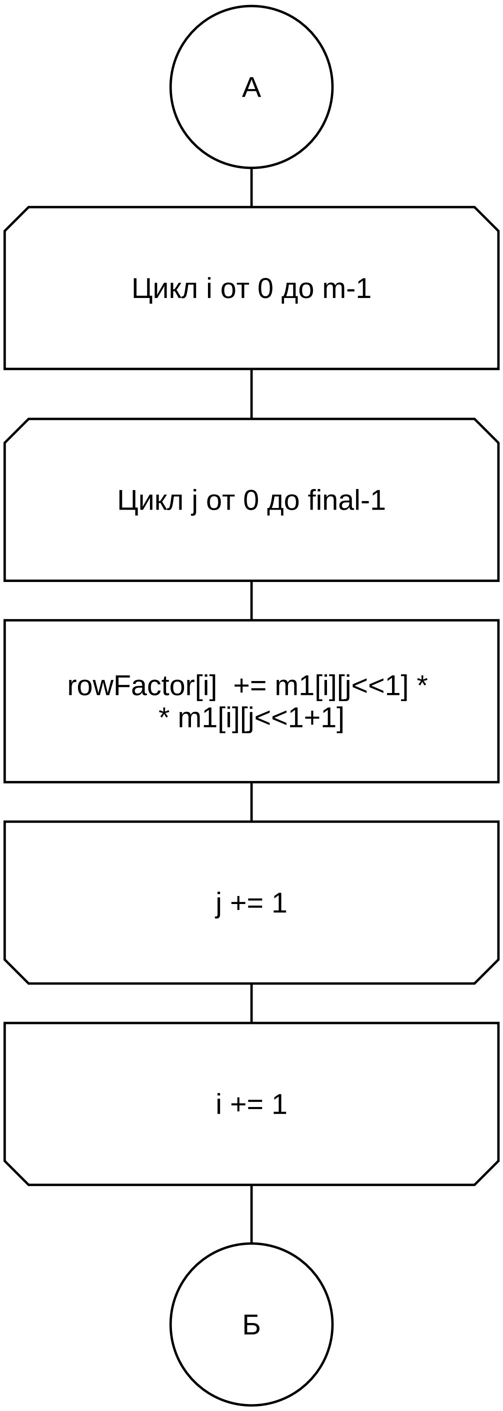


Рисунок 2.10 – Схема оптимизированного алгоритма Винограда умножения матриц: предвычисление для столбцов матрицы

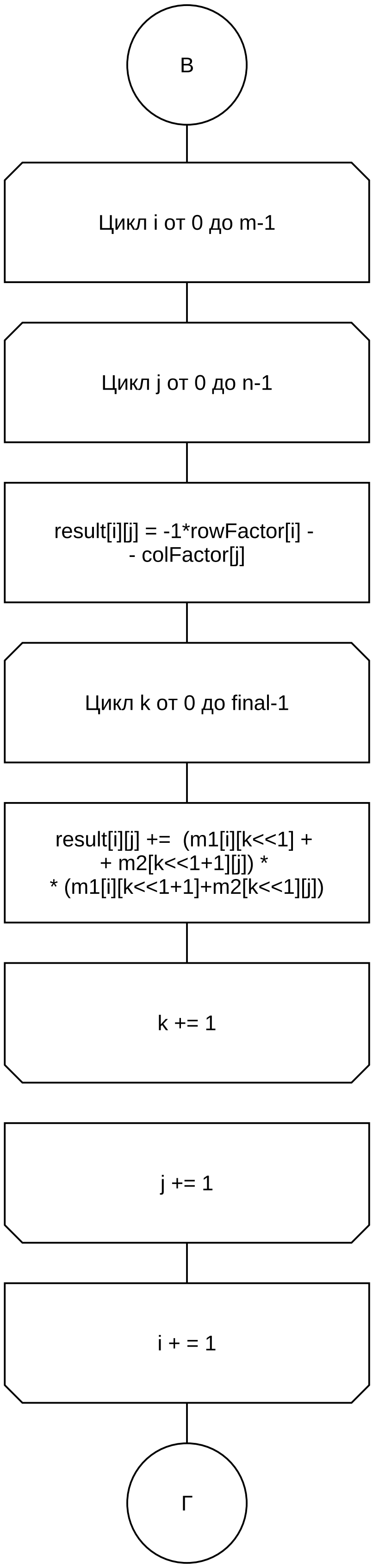


Рисунок 2.11 – Схема оптимизированного алгоритма Винограда умножения матриц: основной цикл умножения

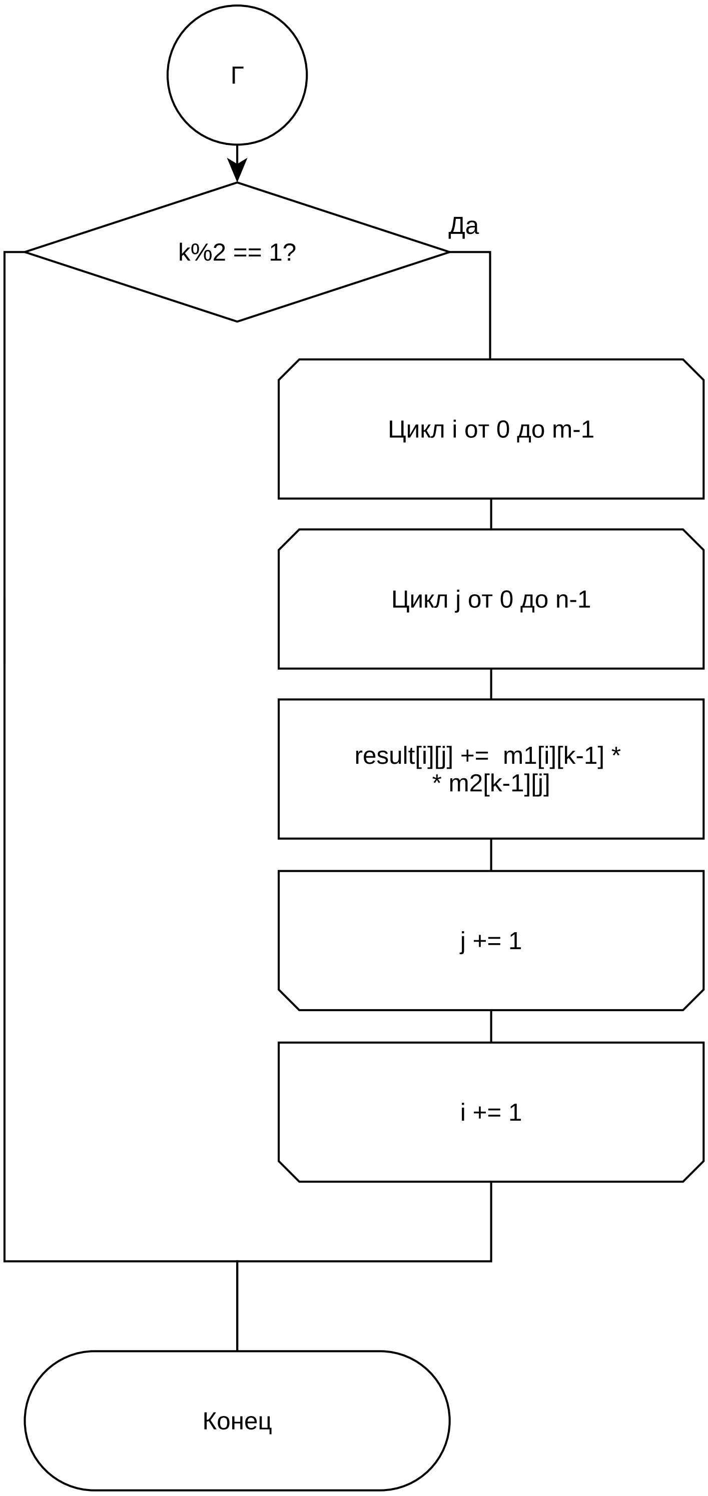


Рисунок 2.12 – Схема оптимизированного алгоритма Винограда умножения матриц: обработка нечётной размерности

## 2.5. Разработка алгоритма Штрассена умножения матриц без оптимизаций

На рисунках 2.13 — 2.14 приведена схема алгоритма Штрассена умножения матриц без оптимизаций.

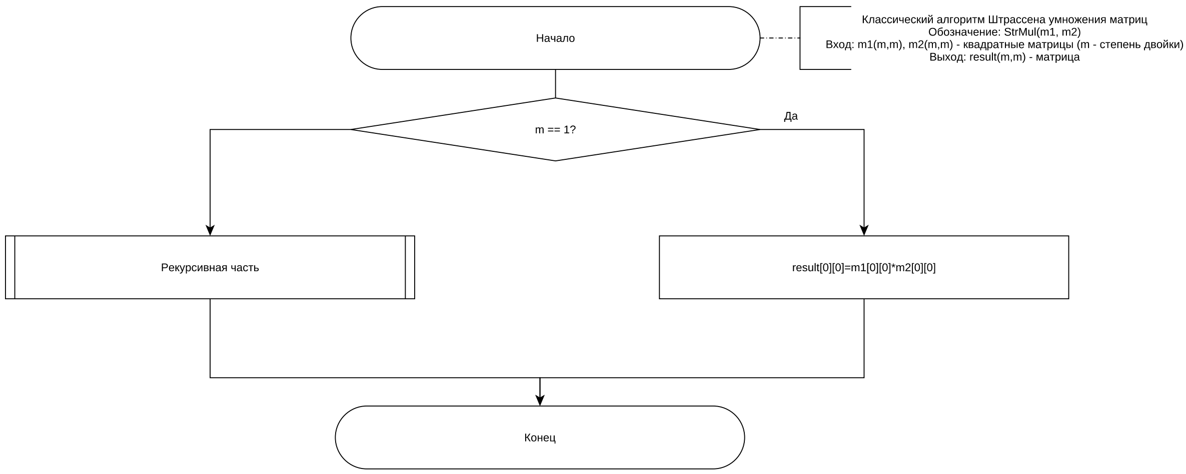


Рисунок 2.13 – Схема классического алгоритма Штрассена умножения матриц

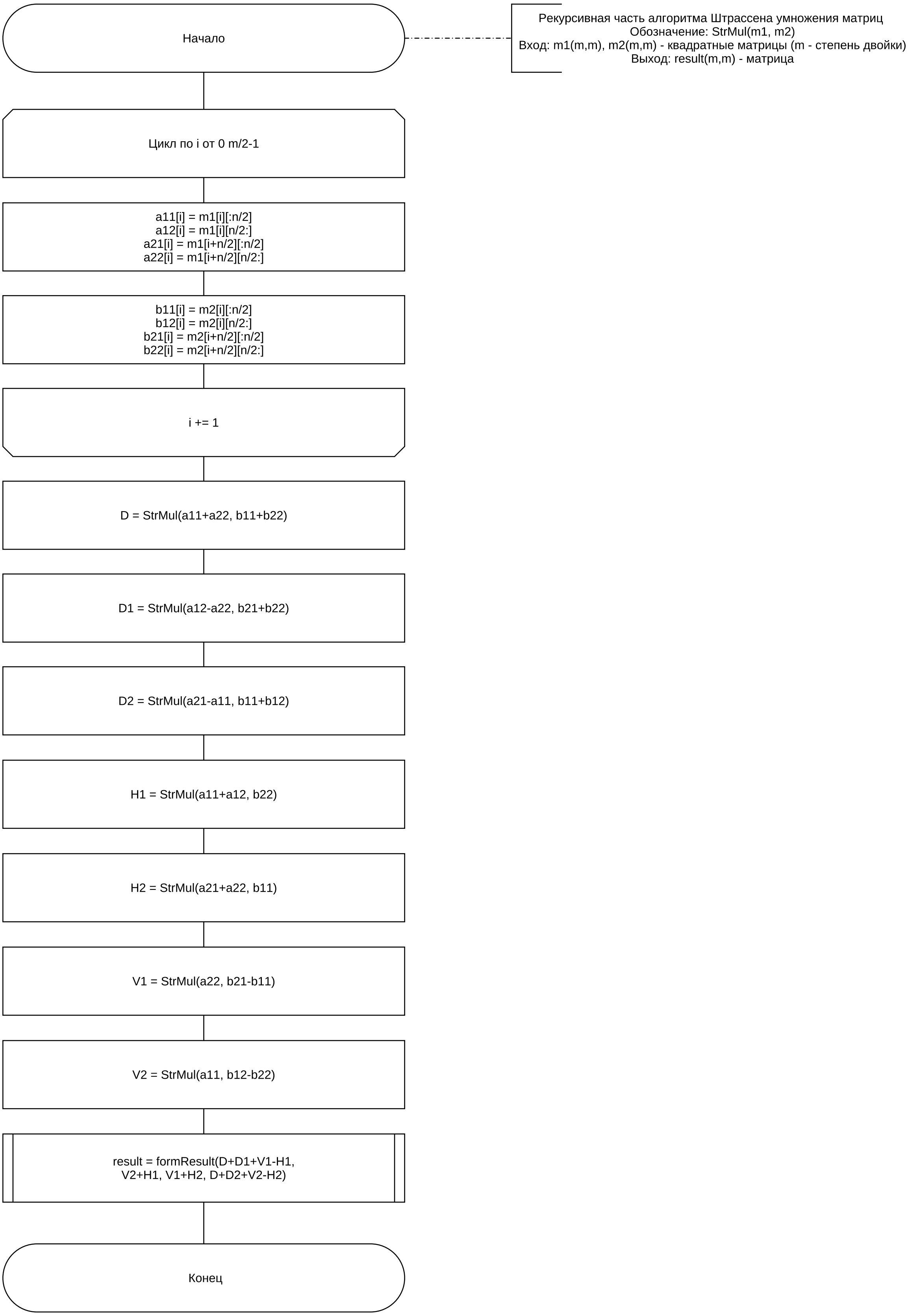


Рисунок 2.14 – Схема классического алгоритма Штрассена умножения матриц

## 2.6. Разработка алгоритма Штрассена умножения матриц с оптимизациями

На рисунках 2.15 — 2.16 приведена схема алгоритма Штрассена умножения матриц с оптимизациями: замена деления на два на побитовый сдвиг, предвычисление некоторых слагаемых.

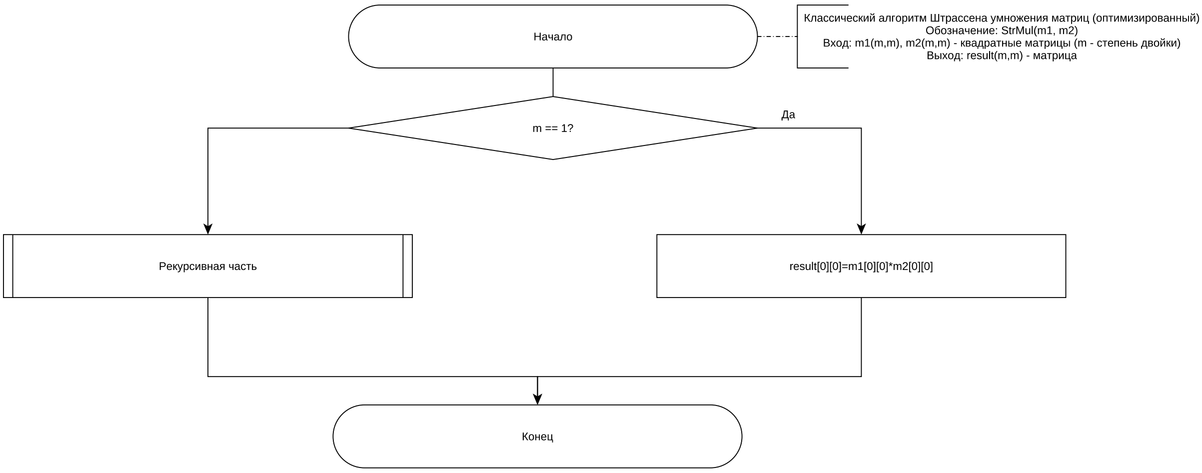


Рисунок 2.15 – Схема оптимизированного алгоритма Штрассена умножения матриц

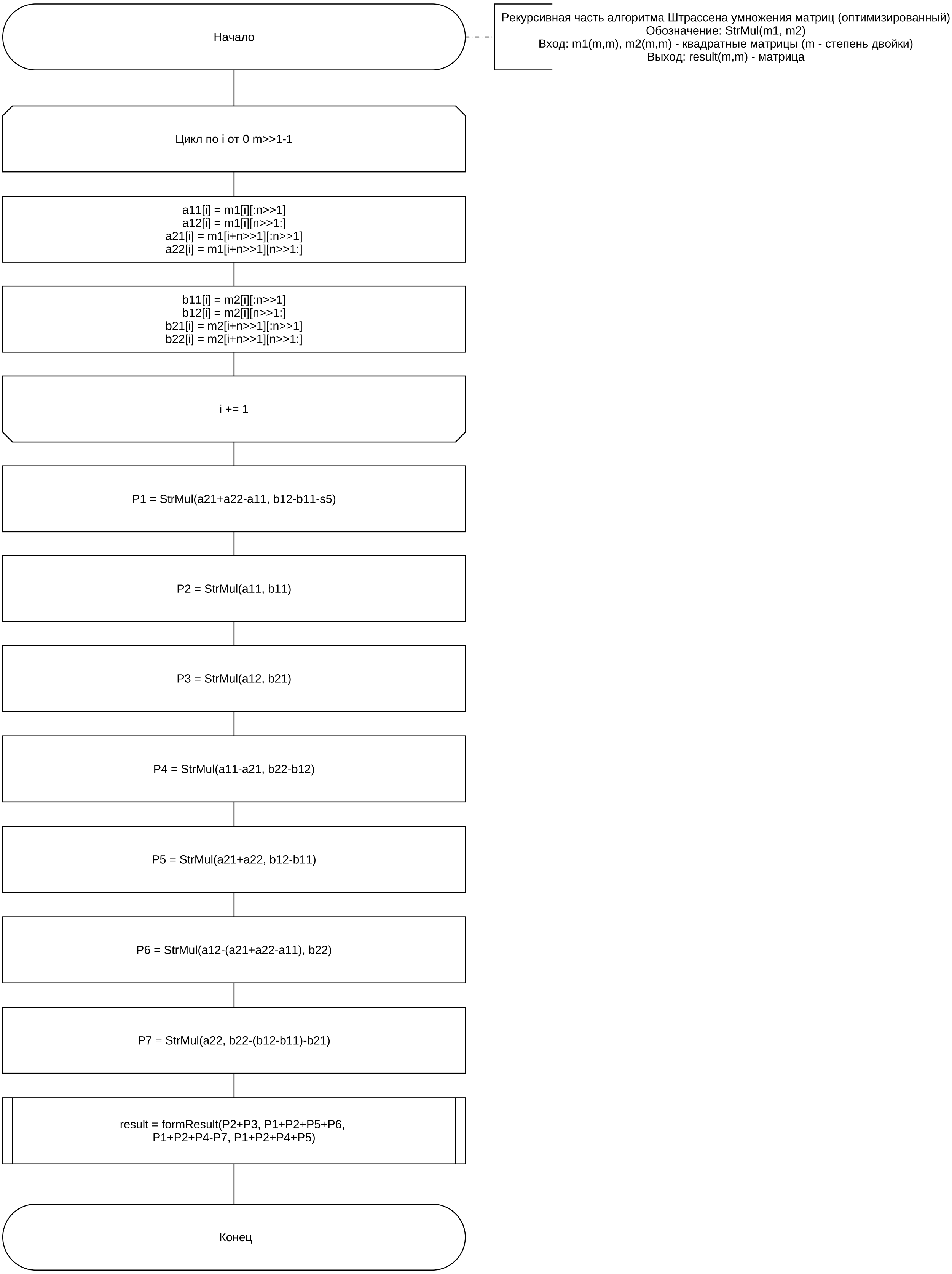


Рисунок 2.16 – Схема оптимизированного алгоритма Штрассена умножения матриц

## 2.7. Трудоемкость алгоритмов

### 2.7.1 Модель вычислений

Чтобы провести вычисление трудоемкости алгоритмов умножения матриц, введем модель вычислений:

1. операции из списка (2.1) имеют трудоемкость 1;

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

2. операции из списка (2.2) имеют трудоемкость 2;

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

3. трудоемкость оператора выбора `if условие then A else B` рассчитывается, как (2.3);

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.3}$$

4. трудоемкость цикла рассчитывается, как (2.4);

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \tag{2.4}$$

5. трудоемкость вызова функции равна 0.

### 2.7.2 Трудоёмкость стандартного алгоритма умножения матриц

Для стандартного алгоритма умножения матриц трудоемкость будет слагаться из:

— создания матрицы-результата, трудоёмкость которого: $f = 1 + 1 + (1 + 1 + M \cdot (1 + 1 + 2))$;

— внешнего цикла по $i \in [0..M - 1]$, трудоёмкость которого: $f = 1 + 1 + M \cdot (1 + 1 + f_{body})$;

— цикла по $j \in [0..N - 1]$, трудоёмкость которого: $f = 1 + 1 + N \cdot (1 + 1 + f_{body})$;

— цикла по $k \in [0..K - 1]$, трудоёмкость которого: $f = 1 + 1 + K \cdot (2 + 1 + 2 + 1 + 2 + 2 + 2 + 1 + 1) = 2 + 14K$.

При раскрытии скобок получается:

$f = 1 + 1 + (1 + 1 + M \cdot (1 + 1 + 2)) + 1 + 1 + M \cdot (1 + 1 + (1 + 1 + N \cdot (1 + 1 + 2 + 14K))) = 14MNK + 4MN + 8M + 6 \approx 14MNK$

Оценка трудоёмкости по самому быстрорастущему слагаемому из зависимости от входных данных: $O(n^3)$.

### 2.7.3 Трудоёмкость оптимизированного алгоритма умножения матриц

Для оптимизированного алгоритма умножения матриц трудоемкость будет слагаться из:

— создания матрицы-результата, трудоёмкость которого: $f = 1 + 1 + (1 + 1 + M \cdot (1 + 1 + 2))$;

— внешнего цикла по $i \in [0..M - 1]$, трудоёмкость которого: $f = 1 + 1 + M \cdot (1 + 1 + f_{body})$;

— цикла по $j \in [0..N - 1]$, трудоёмкость которого: $f = 1 + 1 + N \cdot (1 + 1 + f_{body})$;

— цикла по $k \in [0..K - 1]$, трудоёмкость которого: $f = 1 + 1 + K \cdot (2 + 1 + 2 + 2 + 2 + 1 + 1) = 2 + 11K$.

При раскрытии скобок получается:

$f = 1 + 1 + (1 + 1 + M \cdot (1 + 1 + 2)) + 1 + 1 + M \cdot (1 + 1 + (1 + 1 + N \cdot (1 + 1 + 2 + 11K))) = 11MNK + 4MN + 8M + 6 \approx 11MNK$

Оценка трудоёмкости по самому быстрорастущему слагаемому из зависимости от входных данных: $O(n^3)$.

### 2.7.4 Трудоёмкость стандартного алгоритма Винограда умножения матриц

Для стандартного алгоритма Винограда умножения матриц трудоемкость будет слагаться из:

— создания матрицы-результата, трудоёмкость которого: $f = 1 + 1 + (1 + 1 + M \cdot (1 + 1 + 2)) = 4M + 4$;

— начальной инициализации переменных и списков для вычисляемых значений для столбцов и строк, трудоёмкость которых: $f = 6 + 2 = 8$;

— заполнения массивов rowFactor и columnFactor, трудоёмкость которых одинакова и составляет: $f = 2 \cdot (1 + 1 + N \cdot (1 + 1 + \frac{K}{2} \cdot (1 + 1 + 4 + 4 + 2 + 5))) = 17NK + 4N + 4$;

— внешнего цикла по $i \in [0..M - 1]$, трудоёмкость которого: $f = 1 + 1 + M \cdot (1 + 1 + f_{body})$;

— цикла по $j \in [0..N - 1]$, трудоёмкость которого: $f = 1 + 1 + N \cdot (1 + 1 + 7 + f_{body})$;

— цикла по $k \in [0..K/2 - 1]$, трудоёмкость которого: $f = 1 + 1 + \frac{K}{2} \cdot 30 = 2 + 15K$;

— условия, трудоёмкость которого: $f = 2 + 1 + 0 = 3$, если условие не выполнилось; $f = 2 + 1 + 1 + 1 + M \cdot (1 + 1 + (1 + 1 + N \cdot (1 + 1 + 14))) = 16MN + 4M + 5$.

Итого, в случае невыполнения условия:

$f = 4M + 4 + 8 + 17NK + 4N + 4 + 15MNK + 11MN + 4M + 2 + 3 = 15MNK + 11MN + 17NK + 8M + 21 \approx 15MNK$

Итого, в случае выполнения условия:

$f = 4M + 4 + 8 + 17NK + 4N + 4 + 15MNK + 11MN + 4M + 2 + 16MN + 4M + 5 = 15MNK + 11MN + 12MN + 17NK + 12M + 23 \approx 15MNK$

Оценка трудоёмкости по самому быстрорастущему слагаемому из зависимости от входных данных: $O(n^3)$.

### 2.7.5 Трудоёмкость оптимизированного алгоритма Винограда умножения матриц

Для оптимизированного алгоритма Винограда умножения матриц трудоемкость будет слагаться из:

— создания матрицы-результата, трудоёмкость которого: $f = 1 + 1 + (1 + 1 + M \cdot (1 + 1 + 2)) = 4M + 4$;

— начальной инициализации переменных и списков для вычисляемых значений для столбцов и строк, трудоёмкость которых: $f = 2 + 2 = 4$;

— заполнения массивов rowFactor и columnFactor, трудоёмкость которых одинакова и составляет: $f = 2 \cdot (1 + 1 + N \cdot (1 + 1 + \frac{K}{2} \cdot (1 + 1 + 3 + 2 + 4))) = 11NK + 4N + 4$;

— внешнего цикла по $i \in [0..M - 1]$, трудоёмкость которого: $f = 1 + 1 + M \cdot (1 + 1 + f_{body})$;

— цикла по $j \in [0..N - 1]$, трудоёмкость которого: $f = 1 + 1 + N \cdot (1 + 1 + 7 + f_{body})$;

— цикла по $k \in [0..K/2 - 1]$, трудоёмкость которого: $f = 1 + 1 + \frac{K}{2} \cdot 22 = 2 + 11K$;

— условия, трудоёмкость которого: $f = 2 + 1 + 0 = 3$, если условие не выполнилось; $f = 2 + 1 + 1 + 1 + M \cdot (1 + 1 + (1 + 1 + N \cdot (1 + 1 + 11))) = 13MN + 4M + 5$.

Итого, в случае невыполнения условия:

$f = 4M + 4 + 4 + 11NK + 4N + 4 + 11MNK + 11MN + 4M + 2 + 3 = 11MNK + 11MN + 11NK + 8M + 4N + 17 \approx 11MNK$

Итого, в случае выполнения условия:

$f = 4M + 4 + 4 + 11NK + 4N + 4 + 11MNK + 11MN + 4M + 2 + 13MN + 4M + 5 = 11MNK + 24MN + 11NK + 12M + 4n + 19 \approx 11MNK$

Оценка трудоёмкости по самому быстрорастущему слагаемому из зависимости от входных данных: $O(n^3)$.

### 2.7.6 Трудоёмкость стандартного алгоритма Штрассена умножения матриц

Трудоёмкость сложения/вычитания матриц складывается из:

— создания матрицы-результата, трудоёмкость которого: $f = 1 + 1 + (1 + 1 + M \cdot (1 + 1 + 2)) = 4M + 4$;

— двух циклов, вложенных один в другой, трудоёмкость которых: $f = 1 + 1 + M \cdot (1 + 1 + (1 + 1 + N \cdot (1 + 1 + 8))) = 10MN + 4M + 2$.

В итоге:

$f = 10MN + 8M + 6$

Для стандартного алгоритма Штрассена умножения матриц трудоемкость будет слагаться из:

— начальной инициализации переменной n, трудоёмкость которой: $f = 1$;

— условия, трудоёмкость которого: $f = 1$, если условие не выполнилось; $f = 1+1+1+(1+1+1 \cdot (1+1+2))+9 = 18$, если условие выполнилось;

— создания 8 матриц-блоков, трудоёмкость которого: $f = 32 + 8 * (1 + 1 + (1 + 1 + \frac{M}{2} \cdot (1 + 1 + 2))) = 16M + 64$;

— инициализации 8 матриц-блоков, трудоёмкость которой: $f = 1 + 2 + \frac{M}{2} \cdot (2 + 1 + 60) = \frac{63}{2}M + 3$;

— инициализации переменных $p1 \ldots p10$, трудоёмкость которой: $f = 10 \cdot (10M^2 + 8M + 6 + 1) = 100M^2 + 80M + 70$;

— присвоения переменным $k1 \ldots k7$ результата вызова рекурсивной функции, трудоёмкость которого: $f = 7$;

— получения частей результирующей матрицы, трудоёмкость которого: $f = 8 + 8 * (10M^2 + 8M + 6) = 80M^2 + 64M + 56$;

— создания матрицы-результата, трудоёмкость которого: $f = 1 + 1 + (1 + 1 + M \cdot (1 + 1 + 2)) = 4M + 4$;

— двух циклов заполнения результирующей матрицы, трудоёмкость которых: $f = 1 + 3 + \frac{M}{2} \cdot (3 + 1 + 4) + 3 + 1 + \frac{M}{2} \cdot (1 + 1 + 10) = 10M + 8$.

Алгоритм Штрассена использует рекурсию. Чтобы подсчитать общую трудоёмкость, необходимо вычислить количество рекурсивных вызовов. Матрица размером 2 на 2 выполнит одно разбиение на четыре матрицы размером один на один. Матрицы размером 1 на 1 не вызывают дальнейших рекурсивных вызовов, так как являются основанием рекурсии. Соответственно, произвольная матрица размером $2^n$ на $2^n$ будет иметь $7^n$ рекурсивных вызовов до достижения основания, трудоёмкость каждого из которых равна общей трудоёмкости при невыполнении условия:

$f = 7^M \cdot (1 + 16M + 64 + \frac{63}{2}M + 3 + 100M^2 + 80M + 70 + 7 + 80M^2 + 64M + 56 + 4M + 4 + 10M + 8) = 213 \cdot 7^M + \frac{411 \cdot 7^M \cdot M}{2} + 180 \cdot 7^M \cdot M^2$.

По достижении основания рекурсии рекурсивных вызовов не будет выполнено:

$f = M^2 \cdot 18 = 18M^2$

Итого, трудоёмкость алгоритма Штрассена умножения матриц составляет:

$f = 213 \cdot 7^M + \frac{559 \cdot 7^M \cdot M}{2} + 100 \cdot 7^M \cdot M^2 + 18M^2$

Оценка трудоёмкости по самому быстрорастущему слагаемому из зависимости от входных данных: $O(n^2 \cdot 7^n)$.

### 2.7.7 Трудоёмкость оптимизированного алгоритма Штрассена умножения матриц

Трудоёмкость сложения/вычитания матриц складывается из:

- создания матрицы-результата, трудоёмкость которого: $f = 1 + 1 + (1 + 1 + M \cdot (1 + 1 + 2)) = 4M + 4$;
- двух циклов, вложенных один в другой, трудоёмкость которых: $f = 1 + 1 + M \cdot (1 + 1 + (1 + 1 + N \cdot (1 + 1 + 8))) = 10MN + 4M + 2$.

В итоге:

$f = 10MN + 8M + 6$

Для оптимизированного алгоритма Штрассена умножения матриц трудоемкость будет слагаться из:

- начальной инициализации переменной n, трудоёмкость которой: $f = 1$;
- условия, трудоёмкость которого: $f = 1$, если условие не выполнилось; $f = 1+1+1+(1+1+1 \cdot (1+1+2))+9 = 18$, если условие выполнилось;
- создания 8 матриц-блоков, трудоёмкость которого: $f = 32 + 8 * (1 + 1 + (1 + 1 + \frac{M}{2} \cdot (1 + 1 + 2))) = 16M + 56$;
- инициализации 8 матриц-блоков, трудоёмкость которой: $f = 1 + 2 + \frac{M}{2} \cdot (2 + 1 + 48) = \frac{51}{2}M + 3$;
- инициализации переменных $p1 \ldots p10]$, трудоёмкость которой: $f = 10 \cdot (10M^2 + 8M + 6 + 1) = 100M^2 + 80M + 70$;
- присвоения переменным $k1 \ldots k7$ результата вызова рекурсивной функции, трудоёмкость которого: $f = 7$;
- получения частей результирующей матрицы, трудоёмкость которого: $f = 8 + 8 * (10M^2 + 8M + 6) = 80M^2 + 64M + 56$;
- создания матрицы-результата, трудоёмкость которого: $f = 1 + 1 + (1 + 1 + M \cdot (1 + 1 + 2)) = 4M + 4$;

— двух циклов заполнения результирующей матрицы, трудоёмкость которых: $f = 1 + 2 + \frac{M}{2} \cdot (3 + 1 + 4) + 2 + 1 + \frac{M}{2} \cdot (1 + 1 + 8) = 9M + 6$.

Алгоритм Штрассена использует рекурсию. Чтобы подсчитать общую трудоёмкость, необходимо вычислить количество рекурсивных вызовов. Матрица размером 2 на 2 выполнит одно разбиение на четыре матрицы размером один на один. Матрицы размером 1 на 1 не вызывают дальнейших рекурсивных вызовов, так как являются основанием рекурсии. Соответственно, произвольная матрица размером $2^n$ на $2^n$ будет иметь $7^n$ рекурсивных вызовов до достижения основания, трудоёмкость каждого из которых равна общей трудоёмкости при невыполнении условия:

$f = 7^M \cdot (1 + 16M + 64 + \frac{51}{2}M + 3 + 100M^2 + 80M + 70 + 7 + 80M^2 + 64M + 56 + 4M + 4 + 9M + 6) = 211 \cdot 7^M + \frac{397 \cdot 7^M \cdot M}{2} + 180 \cdot 7^M \cdot M^2$.

По достижении основания рекурсии рекурсивных вызовов не будет выполнено:

$f = M^2 \cdot 18 = 18M^2$

Итого, трудоёмкость алгоритма Штрассена умножения матриц составляет:

$f = 203 \cdot 7^M + \frac{397 \cdot 7^M \cdot M}{2} + 100 \cdot 7^M \cdot M^2 + 18M^2$

Оценка трудоёмкости по самому быстрорастущему слагаемому из зависимости от входных данных: $O(n^2 \cdot 7^n)$.

# Вывод

В данном разделе представлены схемы алгоритмов умножения матриц: классического алгоритма умножения матриц без оптимизаций, оптимизированного алгоритма умножения матриц, классического алгоритма Винограда, оптимизированного алгоритма Винограда, классического алгоритма Штрассена, оптимизированного алгоритма Штрассена, также оценена их трудоёмкость.

По итогам расчётов самыми менее трудоёмкими реализациями алгоритмов являются оптимизированный классический алгоритм умножения матриц и оптимизированный алгоритм Винограда умножения матриц. Алгоритм Штрассена в данной реализации является значительно более трудоёмким, чем остальные алгоритмы. Применение оптимизаций в каждом случае показало уменьшение трудоёмкости реализации алгоритма.

# 3 Технологическая часть

В данном разделе представлены требования к программному обеспечению, а также рассматриваются средства реализации и приводятся листинги кода.

## 3.1. Требования к программному обеспечению

К программе предъявляется ряд требований: на вход подаются указатели на две матрицы, на выходе — матрица, являющаяся произведением исходных.

## 3.2. Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран многопоточный язык Go [3]. Выбор был сделан в пользу данного языка программирования, вследствие наличия пакетов для тестирования программного обеспечения.

## 3.3. Реализация

В листингах 3.1–3.2 приведены реализации алгоритмов классического алгоритма умножения матриц и оптимизированного классического алгоритма умножения матриц.

Листинг 3.1 – Классический алгоритм умножения матриц без оптимизаций

```
func BaseMulMatrixNoOpt(matrix1, matrix2 *matrix.Matrix) matrix.Matrix {
    result := matrix.CreateMatrix(matrix1.M, matrix2.N, false)

    for i := 0; i < matrix1.M; i++ {
        for j := 0; j < matrix2.N; j++ {
            for k := 0; k < matrix1.N; k++ {
                result.Matr[i][j] = result.Matr[i][j] +
                    matrix1.Matr[i][k]*matrix2.Matr[k][j]
            }
        }
    }

    return result
}
```

Листинг 3.2 – Классический алгоритм умножения матриц с оптимизациями

```
func BaseMulMatrixOpt1(matrix1, matrix2 *matrix.Matrix) matrix.Matrix {
    result := matrix.CreateMatrix(matrix1.M, matrix2.N, false)

    for i := 0; i < matrix1.M; i++ {
        for j := 0; j < matrix2.N; j++ {
            for k := 0; k < matrix1.N; k++ {
                result.Matr[i][j] += matrix1.Matr[i][k] * matrix2.Matr[k][j]
            }
        }
    }

    return result
}
```

В листингах 3.3–3.4 приведены реализации алгоритмов Винограда умножения матриц без оптимизаций и оптимизированного алгоритма Винограда умножения матриц.

Листинг 3.3 – Алгоритм Винограда умножения матриц без оптимизаций

```
func GrapeMulMatrixNoOpt(matrix1, matrix2 *matrix.Matrix) matrix.Matrix {
    result := matrix.CreateMatrix(matrix1.M, matrix2.N, false)
    finish_cols := matrix1.N / 2
    finish_rows := matrix2.M / 2

    rowFactor := make([]int, matrix1.M, matrix1.M)
    columnFactor := make([]int, matrix2.N, matrix2.N)

    for i := 0; i < matrix1.M; i++ {
        for j := 0; j < finish_cols; j++ {
            rowFactor[i] = rowFactor[i] + matrix1.Matr[i][2*j]*matrix1.Matr[i][2*j+1]
        }
    }

    for i := 0; i < matrix2.N; i++ {
        for j := 0; j < finish_rows; j++ {
            columnFactor[i] = columnFactor[i] +
                matrix2.Matr[2*j][i]*matrix2.Matr[2*j+1][i]
        }
    }

    for i := 0; i < matrix1.M; i++ {
        for j := 0; j < matrix2.N; j++ {
            result.Matr[i][j] = -1*rowFactor[i] - columnFactor[j]
            for k := 0; k < finish_cols; k++ {
                result.Matr[i][j] = result.Matr[i][j] + (matrix1.Matr[i][2*k]+
                    matrix2.Matr[2*k+1][j])*(matrix1.Matr[i][2*k+1]+matrix2.Matr[2*k][j])
            }
        }
    }

    if matrix1.N%2 == 1 {
        for i := 0; i < matrix1.M; i++ {
            for j := 0; j < matrix2.N; j++ {
                result.Matr[i][j] = result.Matr[i][j] +
                    matrix1.Matr[i][matrix1.N-1]*matrix2.Matr[matrix1.N-1][j]
            }
        }
    }

    return result
```

Листинг 3.4 – Оптимизированный алгоритм Винограда умножения матриц

```
func GrapeMulMatrixOpt1and2(matrix1, matrix2 *matrix.Matrix) matrix.Matrix {
    result := matrix.CreateMatrix(matrix1.M, matrix2.N, false)
    finish_cols := matrix1.N >> 1
    finish_rows := matrix2.M >> 1

    rowFactor := make([]int, matrix1.M, matrix1.M)
    columnFactor := make([]int, matrix2.N, matrix2.N)

    for i := 0; i < matrix1.M; i++ {
        for j := 0; j < finish_cols; j++ {
            rowFactor[i] += matrix1.Matr[i][j<<1] * matrix1.Matr[i][j<<1+1]
        }
    }

    for i := 0; i < matrix2.N; i++ {
        for j := 0; j < finish_rows; j++ {
            columnFactor[i] += matrix2.Matr[j<<1][i] * matrix2.Matr[j<<1+1][i]
        }
    }

    for i := 0; i < matrix1.M; i++ {
        for j := 0; j < matrix2.N; j++ {
            result.Matr[i][j] = -1*rowFactor[i] - columnFactor[j]
            for k := 0; k < finish_cols; k++ {
                result.Matr[i][j] += (matrix1.Matr[i][k<<1] + matrix2.Matr[k<<1+1][j]) *
                    (matrix1.Matr[i][k<<1+1] + matrix2.Matr[k<<1][j])
            }
        }
    }

    if matrix1.N%2 == 1 {
        for i := 0; i < matrix1.M; i++ {
            for j := 0; j < matrix2.N; j++ {
                result.Matr[i][j] += matrix1.Matr[i][matrix1.N-1] *
                    matrix2.Matr[matrix1.N-1][j]
            }
        }
    }

    return result
}
```

В листингах 3.5–3.9 приведены реализации алгоритмов Штрассена умножения матриц без оптимизаций и оптимизированного алгоритма Штрассена умножения матриц.

Листинг 3.5 – Алгоритм Штрассена умножения матриц без оптимизаций

```
func StrassenMulMatrixNoOpt(matrix1, matrix2 *matrix.Matrix) matrix.Matrix {
    n := matrix1.M

    if n == 1 {
        result := matrix.CreateMatrix(1, 1, false)
        result.Matr[0][0] = matrix1.Matr[0][0] * matrix2.Matr[0][0]
        return result
    }

    a11 := matrix.CreateMatrix(n/2, n/2, false)
    a12 := matrix.CreateMatrix(n/2, n/2, false)
    a21 := matrix.CreateMatrix(n/2, n/2, false)
    a22 := matrix.CreateMatrix(n/2, n/2, false)
    b11 := matrix.CreateMatrix(n/2, n/2, false)
    b12 := matrix.CreateMatrix(n/2, n/2, false)
    b21 := matrix.CreateMatrix(n/2, n/2, false)
    b22 := matrix.CreateMatrix(n/2, n/2, false)

    for i := 0; i < n/2; i++ {
        a11.Matr[i] = matrix1.Matr[i][:n/2]
        a12.Matr[i] = matrix1.Matr[i][n/2:]
        a21.Matr[i] = matrix1.Matr[i+n/2][:n/2]
        a22.Matr[i] = matrix1.Matr[i+n/2][n/2:]
        b11.Matr[i] = matrix2.Matr[i][:n/2]
        b12.Matr[i] = matrix2.Matr[i][n/2:]
        b21.Matr[i] = matrix2.Matr[i+n/2][:n/2]
        b22.Matr[i] = matrix2.Matr[i+n/2][n/2:]
    }

    p1 := matrix.AddMatrixes(&a11, &a22)
    p2 := matrix.AddMatrixes(&b11, &b22)
    p3 := matrix.SubMatrixes(&a12, &a22)
    p4 := matrix.AddMatrixes(&b11, &b12)
    p5 := matrix.AddMatrixes(&a11, &a12)
    p6 := matrix.AddMatrixes(&a21, &a22)
    p7 := matrix.SubMatrixes(&b21, &b11)
    p8 := matrix.SubMatrixes(&b12, &b22)
    p9 := matrix.SubMatrixes(&a21, &a11)
    p10 := matrix.AddMatrixes(&b21, &b22)

    k1 := StrassenMulMatrixNoOpt(&p1, &p2)
    k2 := StrassenMulMatrixNoOpt(&p3, &p10)
```

Листинг 3.6 – Продолжение листинга 3.5

```
    k3 := StrassenMulMatrixNoOpt(&p9, &p4)
    k7 := StrassenMulMatrixNoOpt(&p5, &b22)
    k4 := StrassenMulMatrixNoOpt(&p6, &b11)
    k5 := StrassenMulMatrixNoOpt(&a22, &p7)
    k6 := StrassenMulMatrixNoOpt(&a11, &p8)

    c11 := matrix.AddMatrixes(&k1, &k2)
    c12 := matrix.SubMatrixes(&k5, &k7)
    r11 := matrix.AddMatrixes(&c11, &c12)

    r12 := matrix.AddMatrixes(&k6, &k7)

    r21 := matrix.AddMatrixes(&k5, &k4)

    c22 := matrix.AddMatrixes(&k1, &k3)
    c23 := matrix.SubMatrixes(&k6, &k4)

    r22 := matrix.AddMatrixes(&c22, &c23)

    result := matrix.CreateMatrix(n, n, false)

    for i := 0; i < n/2; i++ {
        result.Matr[i] = append(r11.Matr[i], r12.Matr[i]...)
    }
    for i := n / 2; i < n; i++ {
        result.Matr[i] = append(r21.Matr[i-n/2], r22.Matr[i-n/2]...)
    }

    return result
}
```

Листинг 3.7 – Оптимизированный алгоритм Штрассена умножения матриц

```
func StrassenMulMatrixOpt1(matrix1, matrix2 *matrix.Matrix) matrix.Matrix {
    n := matrix1.M

    if n == 1 {
        result := matrix.CreateMatrix(1, 1, false)
        result.Matr[0][0] = matrix1.Matr[0][0] * matrix2.Matr[0][0]
        return result
    }

    a11 := matrix.CreateMatrix(n>>1, n>>1, false)
    a12 := matrix.CreateMatrix(n>>1, n>>1, false)
    a21 := matrix.CreateMatrix(n>>1, n>>1, false)
```

Листинг 3.8 – Продолжение листинга 3.7

```
    a22 := matrix.CreateMatrix(n>>1, n>>1, false)
    b11 := matrix.CreateMatrix(n>>1, n>>1, false)
    b12 := matrix.CreateMatrix(n>>1, n>>1, false)
    b21 := matrix.CreateMatrix(n>>1, n>>1, false)
    b22 := matrix.CreateMatrix(n>>1, n>>1, false)

    for i := 0; i < n>>1; i++ {
        a11.Matr[i] = matrix1.Matr[i][:n>>1]
        a12.Matr[i] = matrix1.Matr[i][n>>1:]
        a21.Matr[i] = matrix1.Matr[i+n>>1][:n>>1]
        a22.Matr[i] = matrix1.Matr[i+n>>1][n>>1:]
        b11.Matr[i] = matrix2.Matr[i][:n>>1]
        b12.Matr[i] = matrix2.Matr[i][n>>1:]
        b21.Matr[i] = matrix2.Matr[i+n>>1][:n>>1]
        b22.Matr[i] = matrix2.Matr[i+n>>1][n>>1:]
    }

    s1 := matrix.AddMatrixes(&a21, &a22)
    s2 := matrix.SubMatrixes(&s1, &a11)
    s3 := matrix.SubMatrixes(&a11, &a21)
    s4 := matrix.SubMatrixes(&a12, &s2)
    s5 := matrix.SubMatrixes(&b12, &b11)
    s6 := matrix.SubMatrixes(&b22, &s5)
    s7 := matrix.SubMatrixes(&b22, &b12)
    s8 := matrix.SubMatrixes(&s6, &b21)

    p1 := StrassenMulMatrixOpt1(&s2, &s6)
    p2 := StrassenMulMatrixOpt1(&a11, &b11)
    p3 := StrassenMulMatrixOpt1(&a12, &b21)
    p4 := StrassenMulMatrixOpt1(&s3, &s7)
    p5 := StrassenMulMatrixOpt1(&s1, &s5)
    p6 := StrassenMulMatrixOpt1(&s4, &b22)
    p7 := StrassenMulMatrixOpt1(&a22, &s8)

    t1 := matrix.AddMatrixes(&p1, &p2)
    t2 := matrix.AddMatrixes(&t1, &p4)

    r11 := matrix.AddMatrixes(&p2, &p3)
    r121 := matrix.AddMatrixes(&t1, &p5)
    r12 := matrix.AddMatrixes(&r121, &p6)
    r21 := matrix.SubMatrixes(&t2, &p7)
    r22 := matrix.AddMatrixes(&t2, &p5)

    result := matrix.CreateMatrix(n, n, false)

    for i := 0; i < n>>1; i++ {
        result.Matr[i] = append(r11.Matr[i], r12.Matr[i]...)
```

Листинг 3.9 – Продолжение листинга 3.8

```
    }
    for i := n >> 1; i < n; i++ {
        result.Matr[i] = append(r21.Matr[i-n>>1], r22.Matr[i-n>>1]...)
    }

    return result
}
```

Листинги со служебным кодом находятся в приложении.

## 3.4. Функциональное тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов умножения матриц: классического алгоритма умножения матриц, алгоритма Винограда умножения матриц, алгоритма Штрассена умножения матриц (неоптимизированная и оптимизированная версии). Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

| **Матрица 1** | **Матрица 2** | **Ожидаемый результат** |
|---|---|---|
| $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ | $\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$ | $\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$ |
| $\begin{pmatrix} 10 \end{pmatrix}$ | $\begin{pmatrix} 5 \end{pmatrix}$ | $\begin{pmatrix} 50 \end{pmatrix}$ |
| $\begin{pmatrix} 10 \\ 20 \\ 30 \end{pmatrix}$ | $\begin{pmatrix} 40 & 50 & 60 \end{pmatrix}$ | $\begin{pmatrix} 400 & 500 & 600 \\ 800 & 1000 & 1200 \\ 1200 & 1500 & 180 \end{pmatrix}$ |
| $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ | $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$ | $\begin{pmatrix} 38 & 44 & 50 & 56 \\ 83 & 98 & 113 & 128 \end{pmatrix}$ |

## 3.5. Пример работы

Демонстрация работы программы приведена на рисунке 3.1.

```
Добро пожаловать в главное меню!

        1. Ввести размерности и сгенерировать матрицы, заполненные случайными числами
        2. Ввести матрицы вручную
        3. Вывести матрицы

        Вычислить произведение матриц:
        4. классическим алгоритмом умножения матриц без оптимизаций
        5. классическим алгоритмом умножения матриц без оптимизаций с +=
        6. алгоритмом Винограда без оптимизаций
        7. алгоритмом Винограда с += и с << вместо *2
        8. алгоритмом Штрассена без оптимизаций
        9. алгоритмом Штрассена с >> вместо /2 и предвычислением некоторых слагаемых

        10. Замеры времени

        11. Вывести меню
        0. Выход

Выберите опцию: 1
Введите количество строк и столбцов для 1 матрицы и количество столбцов для 2й: 2 2 2

Выберите опцию: 3

Первая матрица:
   23    69
   45    33


Вторая матрица:
   24    22
   53    76


Выберите опцию: 4

Результат:
 4209  5750
 2829  3498


Выберите опцию: 0
```

Рисунок 3.1 – Демонстрация работы программы

# Вывод

В результате, были разработаны и протестированы следующие алгоритмы: классический алгоритм умножения матриц без оптимизаций, оптимизированный алгоритм умножения матриц, классический алгоритм Винограда, оптимизированный алгоритм Винограда, классический алгоритм Штрассена, оптимизированный алгоритма Штрассена

# 4 Исследовательская часть

## 4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

— операционная система: Ubuntu [4] 22.04.3 LTS;

— оперативная память: 15 ГБ;

— процессор: 12 ядер, AMD Ryzen 5 4600ГЦ with Radeon Graphics [5].

## 4.2. Время выполнения алгоритмов

Все замеры были проведены в одинаковых условиях.

Замеры проводились с помощью разработанной функции GetCPU, которая использует модуль syscall [6]. syscall.Rusage является структурой в пакете syscall, предназначенной для хранения информации о ресурсах, используемых процессом или потоком. Эта структура содержит различные поля, такие как Utime (время использования ЦПУ в пользовательском режиме), Stime (время использования ЦПУ в режиме ядра) и так далее.

Для получения времени использования ЦПУ, извлекаем атрибуты структуры Utime и Stime.

Выражение usage.Utime.Nano() + usage.Stime.Nano() возвращает сумму времени использования ЦПУ в наносекундах для процесса или потока. Это значение может быть использовано для измерения общего времени использования ЦПУ в пределах процесса или потока.

Результаты замеров приведены в таблице 4.1.

Обозначим:

— 1 — классический алгоритм умножения матриц без оптимизаций;

— 2 — оптимизированный классический алгоритм умножения матриц;

— 3 — алгоритм Винограда умножения матриц без оптимизаций;

— 4 — оптимизированный алгоритм Винограда умножения матриц;

— 5 — алгоритм Штрассена умножения матриц без оптимизаций;

— 6 — оптимизированный алгоритм Штрассена умножения матриц.

Таблица 4.1 – Замер времени для матриц, размером от 10x10 до 460x460

| **Размер матриц** | **Время, с** | | | | | |
|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** |
| 10 x 10 | 0.0000136 | 0.0000046 | 0.0000032 | 0.0000032 | 0.0012822 | 0.0024592 |
| 60 x 60 | 0.0015532 | 0.0005130 | 0.0004598 | 0.0004588 | 0.1342932 | 0.1240314 |
| 110 x 110 | 0.0088996 | 0.0032186 | 0.0031760 | 0.0028564 | 0.9753178 | 0.8676318 |
| 160 x 160 | 0.0136560 | 0.0102094 | 0.0090372 | 0.0093532 | 7.2003379 | 6.2366581 |
| 210 x 210 | 0.0363334 | 0.0223904 | 0.0202432 | 0.0314384 | 7.0362678 | 6.2163839 |
| 260 x 260 | 0.0433772 | 0.0679150 | 0.0389052 | 0.0386174 | 39.2253532 | 35.6911469 |
| 310 x 310 | 0.0974000 | 0.1092742 | 0.0978198 | 0.0988076 | 39.7415733 | 36.1835632 |
| 360 x 360 | 0.1376794 | 0.1306466 | 0.1113444 | 0.1105984 | 39.3739395 | 35.7239914 |
| 410 x 410 | 0.1801004 | 0.1845198 | 0.1632608 | 0.1742950 | 38.8734703 | 35.7225189 |
| 460 x 460 | 0.3638812 | 0.3688240 | 0.3200220 | 0.3442824 | 38.8343964 | 35.8125496 |

На следующем графике представлена зависимость времени работы алгоритмов умножения матриц от размеров матриц.

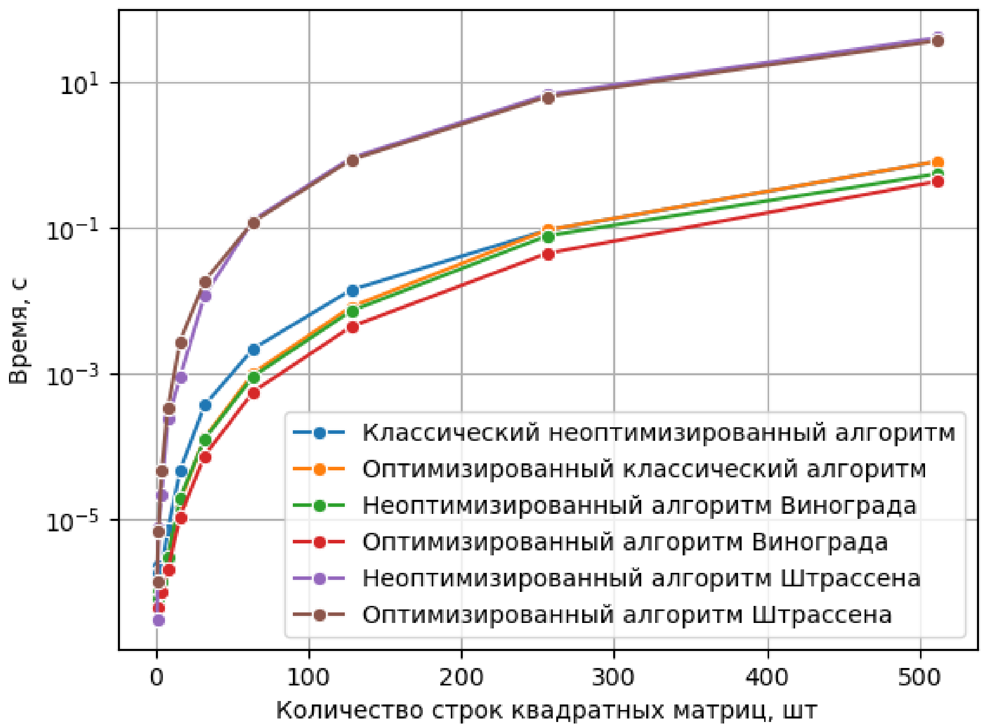


Рисунок 4.1 – Время работы алгоритмов умножения матриц

На следующем графике представлена зависимость времени работы алгоритмов умножения матриц: стандартного и алгоритма Винограда на чётных размерах матриц.

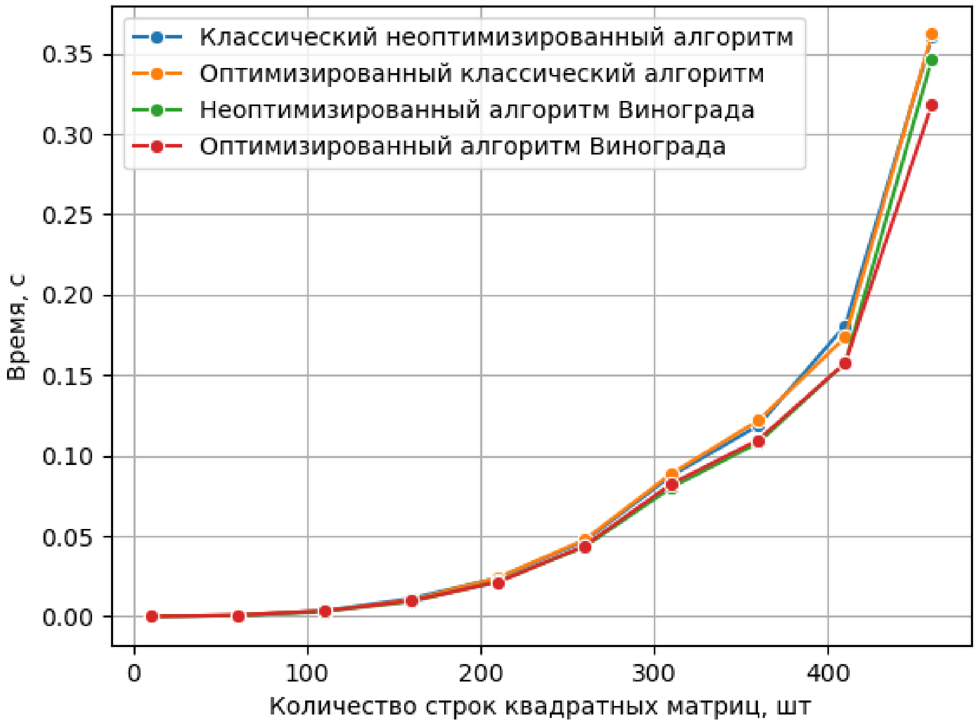


Рисунок 4.2 – Время работы алгоритмов умножения матриц при чётных размерах матриц

На графике 4.3 представлена зависимость времени работы алгоритмов умножения матриц: стандартного и алгоритма Винограда на нечётных размерах матриц.

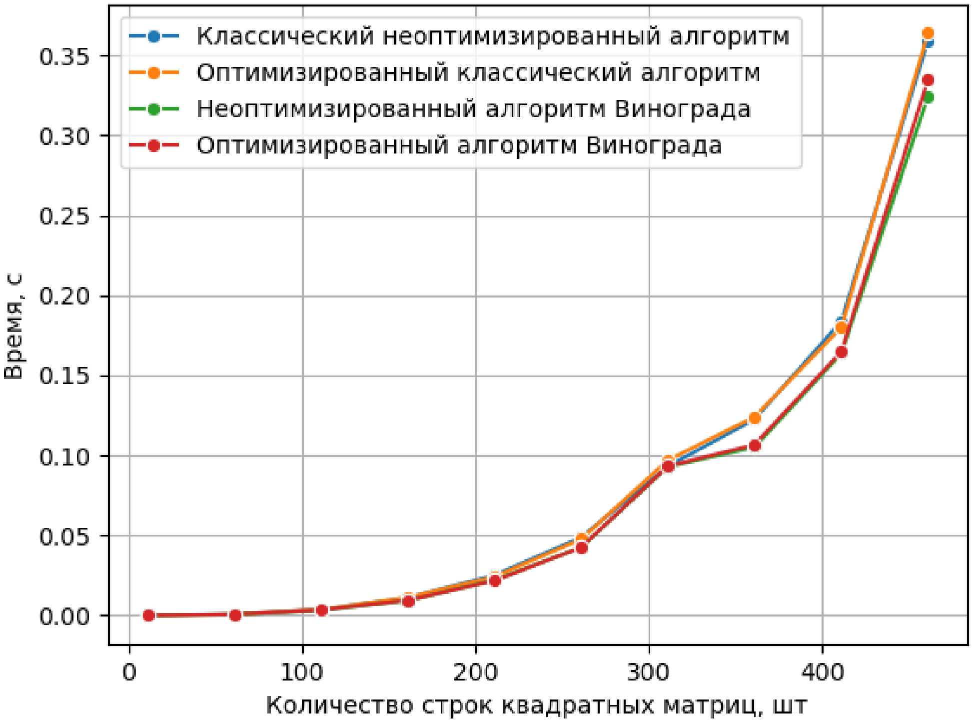


Рисунок 4.3 – Время работы алгоритмов умножения матриц при нечётных размерах матриц

График 4.4 представляет сравнение алгоритмов Винограда — оптимизированного и неоптимизированного — при чётных и нечётных размерах матриц.

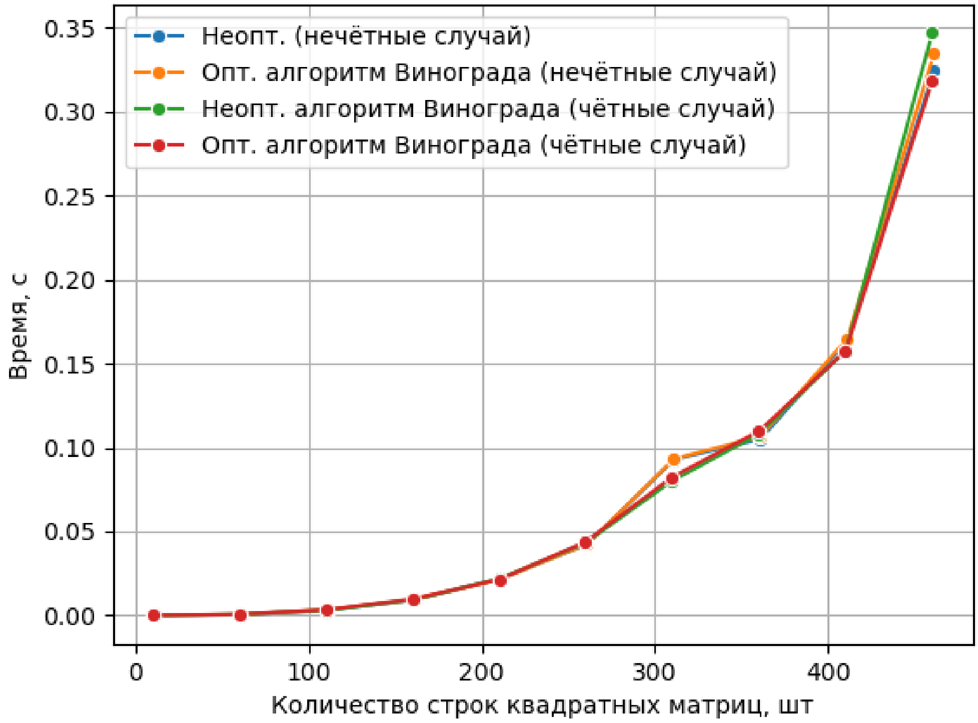


Рисунок 4.4 – Время работы алгоритмов умножения матриц при нечётных размерах матриц

## 4.3. Использование памяти

Выделение памяти при работе алгоритмов указано в листинге 4.1. Получено при помощи команды: `go test -bench . -benchmem`.

Листинг 4.1 – Использование памяти

```
goos: linux
goarch: amd64
pkg: lab02/algs
cpu: AMD Ryzen 5 4600H with Radeon Graphics
BenchmarkGrapeNoOpt10-12  391056 5882 ns/op 1200 B/op 13 allocs/op
BenchmarkGrapeNoOpt50-12  4334 313408 ns/op 22922 B/op 53 allocs/op
BenchmarkGrapeNoOpt100-12  466 2365682 ns/op 94476 B/op 103 allocs/op
BenchmarkGrapeNoOpt200-12  63 20472407 ns/op 378381 B/op 209 allocs/op
BenchmarkGrapeNoOpt500-12  3 431211485 ns/op 3442624 B/op 838 allocs/op
BenchmarkGrapeOpt1and2_10-12  188516 6185 ns/op 1200 B/op 13 allocs/op
BenchmarkGrapeOpt1and2_50-12  3373 397454 ns/op 22925 B/op 53 allocs/op
BenchmarkGrapeOpt1and2_100-12  468 2491422 ns/op 94474 B/op 103 allocs/op
BenchmarkGrapeOpt1and2_200-12  58 22430267 ns/op 379374 B/op 209 allocs/op
BenchmarkGrapeOpt1and2_500-12  3 413322829 ns/op 3442005 B/op 837 allocs/op
BenchmarkClassicNoOpt_10-12  184544 5892 ns/op 1040 B/op 11 allocs/op
BenchmarkClassicNoOpt_50-12  3822 391184 ns/op 22091 B/op 51 allocs/op
BenchmarkClassicNoOpt_100-12  464 2754122 ns/op 92685 B/op 101 allocs/op
BenchmarkClassicNoOpt_200-12  52 21414936 ns/op 377235 B/op 208 allocs/op
BenchmarkClassicNoOpt_500-12  2 530495077 ns/op 4120576 B/op 1002 allocs/op
BenchmarkClassicOpt_10-12  207182 5470 ns/op 1040 B/op 11 allocs/op
BenchmarkClassicOpt_50-12  3168 381204 ns/op 22093 B/op 51 allocs/op
BenchmarkClassicOpt_100-12  403 2541602 ns/op 92746 B/op 101 allocs/op
BenchmarkClassicOpt_200-12  58 21901074 ns/op 375790 B/op 207 allocs/op
BenchmarkClassicOpt_500-12  3 476170994 ns/op 3433813 B/op 835 allocs/op
BenchmarkStrassenNoOpt_10-12  504 2378451 ns/op 591878 B/op 29912 allocs/op
BenchmarkStrassenNoOpt_50-12  8 130821524 ns/op 29886534 B/op 1470256 allocs/op
BenchmarkStrassenNoOpt_100-12  2 927931622 ns/op 210566864 B/op 10293777 allocs/op
BenchmarkStrassenNoOpt_200-12  1 6504249701 ns/op 1478681112 B/op 72059515 allocs/op
BenchmarkStrassenNoOpt_500-12  1 38492639772 ns/op 10360946760 B/op 504419085 allocs/op
BenchmarkStrassenOpt_10-12  547 2408159 ns/op 540373 B/op 27281 allocs/op
BenchmarkStrassenOpt_50-12  10 116502307 ns/op 27280347 B/op 1340875 allocs/op
BenchmarkStrassenOpt_100-12  2 791231776 ns/op 192251576 B/op 9387957 allocs/op
BenchmarkStrassenOpt_200-12  1 5547761377 ns/op 1350076848 B/op 65718384 allocs/op
BenchmarkStrassenOpt_500-12  1 32803397495 ns/op 9459129344 B/op 460030419 allocs/op
PASS
ok  lab02/algs 91.158s
```

На рисунке 4.5 представлено сравнение потребления памяти алгоритмами умножения матриц

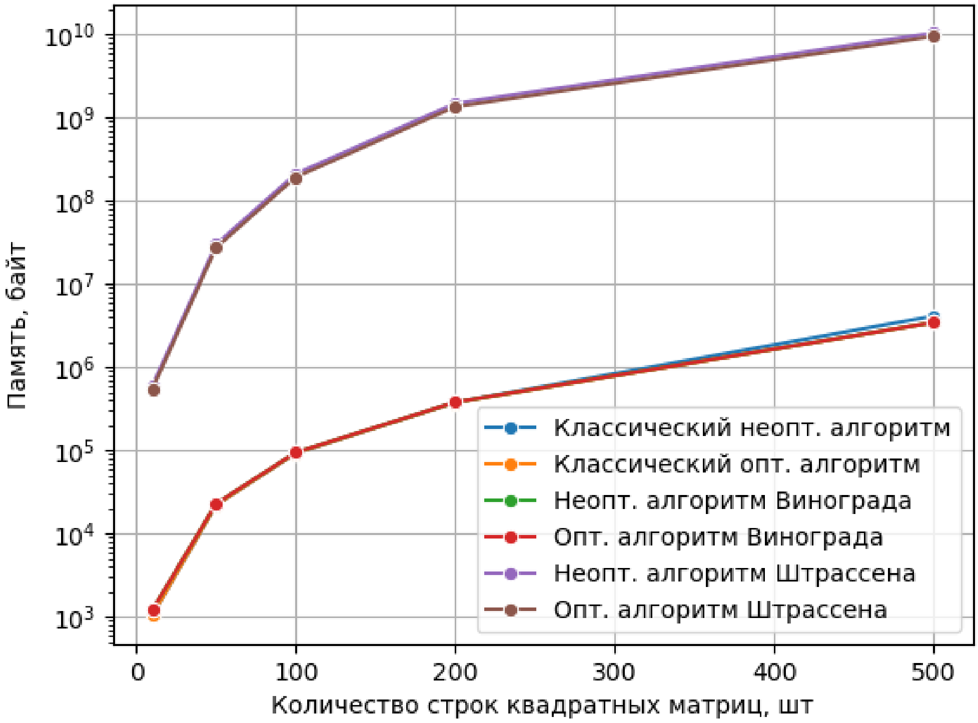


Рисунок 4.5 – Память, потребляемая алгоритмами умножения матриц

Алгоритм Штрассена потребляет больше всего памяти, а стандартный алгоритм умножения матриц и алгоритм Винограда умножения матриц потребляют соизмеримое друг с другом количество памяти.

# Вывод

Рекурсивный алгоритм Штрассена умножения матриц теоретически эффективнее, классического алгоритма умножения матриц и алгоритма Винограда умножения матриц, однако тестирование показало, что реализация этого алгоритма работает медленнее всех остальных реализаций других алгоритмов (в 100 раз медленнее). Время выполнения рекурсивного алгоритма Штрассена увеличивается с увеличением размерности матриц.

Алгоритм Винограда умножения матриц оказался самым эффективным из реализованных алгоритмов. Его преимущество начинает выделяться уже на матрицах размером 10 на 10 (в 4 раза быстрее, чем классический алгоритм умножения матриц.

Оптимизации каждой из реализаций алгоритмов показывают улучшение производительности всех рассматриваемых алгоримов.

Рекурсивный алгоритм Штрассена также потребляет самое большое количество памяти среди всех представленных алгоритмов (в примерно 500 раз больше памяти). Стандартный алгоритм умножения матриц и алгоритм Винограда потребляют примерно одинаковое количество памяти. Стоит отметить, что оптимизации оказывают положительное влияние на количество потребляемой памяти.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были выполнены все поставленные задачи:

— были исследованы алгоритмы умножения матриц;

— была проведена оптимизация алгоритмов умножения матриц;

— были применены методы динамического программирования для реализации алгоритмов умножения матриц;

— были проведены оценка и сравнение трудоёмкости алгоритмов;

— был проведён сравнительный анализ по времени и памяти на основе экспериментальных данных;

— подготовлен отчет по лабораторной работе.

Исследование позволило выявить различия в производительности различных алгоритмов умножения матриц. В частности, алгоритм Винограда оказался самым эффективным среди реализованных (в 1.125 раз быстрее стандартного алгоритма умножения матриц, в 243 раза быстрее алгоритма Штрассена умножения матриц).

Также была проведена оценка трудоёмкости данных алгоритмов на заданной модели вычислений. В результате наиболее трудоёмким оказался алгоритм Штрассена умножения матриц в неоптимизированной версии. Стандартный алгоритм и алгоритм Винограда умножения матриц имеют соизмеримые трудоёмкости. Оптимизации оказали положительное влияние на трудоёмкость реализаций алгритмов.

По потребляемой памяти также наиболее выигрышными оказались стандартный алгоримт умножения матриц и алгоритм Винограда умножения матриц. Алгоритм Штрассена потребляет значительно больше памяти, чем остальные алгоритмы за счёт своей рекурсивной реализации и использования дополнительных блочных подматриц.

В целом, исследование позволило получить практические и теоретические результаты, подтверждающие различия в производительности и использовании памяти различных алгоритмов умножения матриц. Эти результаты могут служить основой для выбора наиболее эффективного алгоритма в зависимости от конкретных требований и ограничений проекта.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] В. Конев В. Линейная алгебра. Учебное пособие. — Томск. Изд. ТПУ., 2008. Т. 65. С. 845–848.

[2] Анисимов Н. С. Строганов Ю. В. Реализация алгоритма умножения матриц по Винограду на языке Haskell. — МГТУ им. Н.Э.Баумана, 2018. Т. 6.

[3] The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/` (дата обращения: 17.10.2023).

[4] Linux — Официальная документация [Электронный ресурс]. Режим доступа: `https://www.linux.org/` (дата обращения: 17.10.2023).

[5] AMD Processors [Электронный ресурс]. Режим доступа: `https://www.amd.com/en.html` (дата обращения: 17.10.2023).

[6] Пакет syscall [Электронный ресурс]. Режим доступа: `https://pkg.go.dev/syscall` (дата обращения: 17.10.2023).

# ПРИЛОЖЕНИЕ А

В листинге А.1 приведена структура Matrix.

Листинг А.1 — Структура Matrix

```go
package matrix

import (
    "fmt"
    "math"
    "math/rand"
    "time"
)

type Matrix struct {
    Matr [][]int
    M int
    N int
}

func NilMatrix() Matrix {
    var m Matrix
    m.Matr = nil
    m.M = 0
    m.N = 0
    return m
}

func (m *Matrix) MakeMatrix() {
    m.Matr = make([][]int, m.M)
    for i := range m.Matr {
        m.Matr[i] = make([]int, m.N)
    }
}

func (m *Matrix) FillMatrix() {
    rand.Seed(time.Now().UnixNano())

    for i := 0; i < m.M; i++ {
        for j := 0; j < m.N; j++ {
            m.Matr[i][j] = rand.Intn(100)
        }
    }
}

func CreateMatrix(m int, n int, fill bool) Matrix {
    matrix := Matrix{M: m, N: n}
    matrix.MakeMatrix()
```

```
    if fill {
        matrix.FillMatrix()
    }

    return matrix
}

func (m *Matrix) OutputMatrix() {
    for i := 0; i < m.M; i++ {
        for j := 0; j < m.N; j++ {
            fmt.Printf("%5d ", m.Matr[i][j])
        }
        fmt.Println()
    }
    fmt.Println()
}

func AddMatrixes(m1, m2 *Matrix) Matrix {
    m := CreateMatrix(m1.M, m1.N, false)

    for i := 0; i < m1.M; i++ {
        for j := 0; j < m1.N; j++ {
            m.Matr[i][j] = m1.Matr[i][j] + m2.Matr[i][j]
        }
    }

    return m
}

func SubMatrixes(m1, m2 *Matrix) Matrix {
    m := CreateMatrix(m1.M, m1.N, false)

    for i := 0; i < m1.M; i++ {
        for j := 0; j < m1.N; j++ {
            m.Matr[i][j] = m1.Matr[i][j] - m2.Matr[i][j]
        }
    }

    return m
}

func (m *Matrix) completeMatrixToSq() {
    if m.M > m.N {
        for j := 0; j < m.M-m.N; j++ {
            for i := range m.Matr {
                m.Matr[i] = append(m.Matr[i], 0)
            }
        }
```

```
        m.N = m.M
    } else if m.N > m.M {
        for j := 0; j < m.N-m.M; j++ {
            m.Matr = append(m.Matr, make([]int, m.N))
        }
        m.M = m.N
    }
}

func (m *Matrix) completeMatrixToDeg(deg int) {
    for j := 0; j < deg-m.M; j++ {
        m.Matr = append(m.Matr, make([]int, deg))
    }

    for j := 0; j < deg-m.N; j++ {
        for i := range m.Matr {
            m.Matr[i] = append(m.Matr[i], 0)
        }
    }

    m.M = deg
    m.N = deg
}

func getDegSize(m, n int) int {
    k := int(math.Max(float64(m), float64(n)))
    if k == 0 {
        return 0
    }

    var i int
    for i = 1; i < k; i *= 2 {
    }

    return i
}

func CopyMatrix(matr *Matrix) Matrix {
    cm := CreateMatrix(matr.M, matr.N, false)

    for i := 0; i < matr.M; i++ {
        for j := 0; j < matr.N; j++ {
            cm.Matr[i][j] = matr.Matr[i][j]
        }
    }

    return cm
}
```

```

func CompleteMatrixes(m1, m2 *Matrix) {
    m1.completeMatrixToSq()
    m2.completeMatrixToSq()

    n := getDegSize(m1.M, m2.M)

    m1.completeMatrixToDeg(n)
    m2.completeMatrixToDeg(n)
}

func (m *Matrix) CorrectMatrix(k, n int) Matrix {
    res := CreateMatrix(k, n, false)

    for i := 0; i < k; i++ {
        for j := 0; j < n; j++ {
            res.Matr[i][j] = m.Matr[i][j]
        }
    }

    return res
}
```

В листинге А.2 приведен модуль для замеров процессорного времени выполнения программы.

Листинг А.2 — Измерение времени

```
package time_measure

import (
    "fmt"
    "lab02/algs"
    "lab02/inp_out"
    "lab02/matrix"

    "math"
    "syscall"
)

const N int = 500
const nAlgs int = 6

func GetCPU() int64 {
    usage := new(syscall.Rusage)
    syscall.Getrusage(syscall.RUSAGE_SELF, usage)
    return usage.Utime.Nano() + usage.Stime.Nano()
}

```

```
func grapeMulMatrixNoOptTimeMeasurement(matrix1, matrix2 *matrix.Matrix) (float32,
    matrix.Matrix) {
    var sum float32

    var startTime, finishTime int64
    var mtr matrix.Matrix

    for i := 0; i < N; i++ {
        startTime = GetCPU()
        mtr = algs.GrapeMulMatrixNoOpt(matrix1, matrix2)
        finishTime = GetCPU()
        sum += float32(finishTime - startTime)
    }

    mtr.Matr[0][0] *= 1

    return (sum / float32(N)) / 1e+9, mtr
}

func grapeMulMatrixOpt1TimeMeasurement(matrix1, matrix2 *matrix.Matrix) (float32,
    matrix.Matrix) {
    var sum float32

    var startTime, finishTime int64
    var mtr matrix.Matrix

    for i := 0; i < N; i++ {
        startTime = GetCPU()
        mtr = algs.GrapeMulMatrixOpt1(matrix1, matrix2)
        finishTime = GetCPU()
        sum += float32(finishTime - startTime)
    }

    mtr.Matr[0][0] *= 1

    return (sum / float32(N)) / 1e+9, mtr
}

func grapeMulMatrixOpt2TimeMeasurement(matrix1, matrix2 *matrix.Matrix) (float32,
    matrix.Matrix) {
    var sum float32

    var startTime, finishTime int64
    var mtr matrix.Matrix

    for i := 0; i < N; i++ {
        startTime = GetCPU()
        mtr = algs.GrapeMulMatrixOpt2(matrix1, matrix2)
```

```
        finishTime = GetCPU()
        sum += float32(finishTime - startTime)
    }

    mtr.Matr[0][0] *= 1

    return (sum / float32(N)) / 1e+9, mtr
}

func grapeMulMatrixOpt3TimeMeasurement(matrix1, matrix2 *matrix.Matrix) (float32,
    matrix.Matrix) {
    var sum float32

    var startTime, finishTime int64
    var mtr matrix.Matrix

    for i := 0; i < N; i++ {
        startTime = GetCPU()
        mtr = algs.GrapeMulMatrixOpt3(matrix1, matrix2)
        finishTime = GetCPU()
        sum += float32(finishTime - startTime)
    }

    mtr.Matr[0][0] *= 1

    return (sum / float32(N)) / 1e+9, mtr
}

func grapeMulMatrixOpt1and2TimeMeasurement(matrix1, matrix2 *matrix.Matrix) (float32,
    matrix.Matrix) {
    var sum float32

    var startTime, finishTime int64
    var mtr matrix.Matrix

    for i := 0; i < N; i++ {
        startTime = GetCPU()
        mtr = algs.GrapeMulMatrixOpt1and2(matrix1, matrix2)
        finishTime = GetCPU()
        sum += float32(finishTime - startTime)
    }

    mtr.Matr[0][0] *= 1

    return (sum / float32(N)) / 1e+9, mtr
}

func grapeMulMatrixOpt1and3TimeMeasurement(matrix1, matrix2 *matrix.Matrix) (float32,
```

```
    matrix.Matrix) {
    var sum float32

    var startTime, finishTime int64
    var mtr matrix.Matrix

    for i := 0; i < N; i++ {
        startTime = GetCPU()
        mtr = algs.GrapeMulMatrixOpt1and3(matrix1, matrix2)
        finishTime = GetCPU()
        sum += float32(finishTime - startTime)
    }

    mtr.Matr[0][0] *= 1

    return (sum / float32(N)) / 1e+9, mtr
}

func grapeMulMatrixOpt1and3BuffTimeMeasurement(matrix1, matrix2 *matrix.Matrix)
    (float32, matrix.Matrix) {
    var sum float32

    var startTime, finishTime int64
    var mtr matrix.Matrix

    for i := 0; i < N; i++ {
        startTime = GetCPU()
        mtr = algs.GrapeMulMatrixOpt1and3WithBuffer(matrix1, matrix2)
        finishTime = GetCPU()
        sum += float32(finishTime - startTime)
    }

    mtr.Matr[0][0] *= 1

    return (sum / float32(N)) / 1e+9, mtr
}

func baseMulMatrixNoOptTimeMeasurement(matrix1, matrix2 *matrix.Matrix) (float32,
    matrix.Matrix) {
    var sum float32

    var startTime, finishTime int64
    var mtr matrix.Matrix

    for i := 0; i < N; i++ {
        startTime = GetCPU()
        mtr = algs.BaseMulMatrixNoOpt(matrix1, matrix2)
        finishTime = GetCPU()
```

```
        sum += float32(finishTime - startTime)
    }

    mtr.Matr[0][0] *= 1

    return (sum / float32(N)) / 1e+9, mtr
}

func baseMulMatrixOpt1TimeMeasurement(matrix1, matrix2 *matrix.Matrix) (float32,
    matrix.Matrix) {
    var sum float32

    var startTime, finishTime int64
    var mtr matrix.Matrix

    for i := 0; i < N; i++ {
        startTime = GetCPU()
        mtr = algs.BaseMulMatrixOpt1(matrix1, matrix2)
        finishTime = GetCPU()
        sum += float32(finishTime - startTime)
    }

    mtr.Matr[0][0] *= 1

    return (sum / float32(N)) / 1e+9, mtr
}

func strassenMulMatrixNoOptTimeMeasurement(matrix1, matrix2 *matrix.Matrix) (float32,
    matrix.Matrix) {
    var sum float32

    var startTime, finishTime int64
    var mtr matrix.Matrix

    matrix.CompleteMatrixes(matrix1, matrix2)

    for i := 0; i < N; i++ {
        startTime = GetCPU()
        mtr = algs.StrassenMulMatrixNoOpt(matrix1, matrix2)
        finishTime = GetCPU()
        sum += float32(finishTime - startTime)
    }

    mtr.Matr[0][0] *= 1

    return (sum / float32(N)) / 1e+9, mtr
}

```

```go
func strassenMulMatrixOpt1TimeMeasurement(matrix1, matrix2 *matrix.Matrix) (float32,
    matrix.Matrix) {
    var sum float32

    var startTime, finishTime int64
    var mtr matrix.Matrix

    matrix.CompleteMatrixes(matrix1, matrix2)

    for i := 0; i < N; i++ {
        startTime = GetCPU()
        mtr = algs.StrassenMulMatrixOpt1(matrix1, matrix2)
        finishTime = GetCPU()
        sum += float32(finishTime - startTime)
    }

    mtr.Matr[0][0] *= 1

    return (sum / float32(N)) / 1e+9, mtr
}

func MeasureTime_v2(begin, end int) ([][]float32, []int) {
    seconds := make([][]float32, 6)

    number := 0
    for n := end; n >= begin; n /= 2 {
        number += 1
    }

    sizes := make([]int, number)
    var matr1, matr2, res matrix.Matrix

    func_pointers := [6]func(*matrix.Matrix, *matrix.Matrix) (float32, matrix.Matrix){
        baseMulMatrixNoOptTimeMeasurement,
        baseMulMatrixOpt1TimeMeasurement,
        grapeMulMatrixNoOptTimeMeasurement,
        grapeMulMatrixOpt1and2TimeMeasurement,
        strassenMulMatrixNoOptTimeMeasurement,
        strassenMulMatrixOpt1TimeMeasurement,
    }

    for i := 0; i < 6; i++ {
        seconds[i] = make([]float32, number)
    }

    for i := 0; i < 6; i++ {
        for j := 0; j < number; j++ {
            matr1, matr2 = inp_out.GenerateSquareMatrixes(begin * int(math.Pow(2,
```

```
                float64(j))))
            seconds[i][j], res = func_pointers[i](&matr1, &matr2)
            sizes[j] = begin * int(math.Pow(2, float64(j)))
        }
    }

    res.Matr[0][0] *= 1

    return seconds, sizes
}

func MeasureTime(begin, end, step int) ([][]float32, []int) {
    seconds := make([][]float32, nAlgs)

    number := (end-begin)/step + 1

    sizes := make([]int, number)
    var matr1, matr2, res matrix.Matrix

    func_pointers := [nAlgs]func(*matrix.Matrix, *matrix.Matrix) (float32,
        matrix.Matrix){
        baseMulMatrixNoOptTimeMeasurement,
        baseMulMatrixOpt1TimeMeasurement,
        grapeMulMatrixNoOptTimeMeasurement,
        grapeMulMatrixOpt1and2TimeMeasurement,
        strassenMulMatrixNoOptTimeMeasurement,
        strassenMulMatrixOpt1TimeMeasurement,
    }

    for i := 0; i < nAlgs; i++ {
        seconds[i] = make([]float32, number)
    }

    for i := 0; i < nAlgs; i++ {
        for j := 0; j < number; j++ {
            matr1, matr2 = inp_out.GenerateSquareMatrixes(begin + step*j)
            seconds[i][j], res = func_pointers[i](&matr1, &matr2)
            fmt.Println(i)
            sizes[j] = begin + step*j
        }
    }

    res.Matr[0][0] *= 1

    return seconds, sizes
}
```