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"

Тема Алгоритмы умножения матриц

Студент Пронина Л. Ю.

Группа ИУ7-54Б

Оценка (баллы) _____

Преподаватель Волкова Л. Л.

Москва — 2023 г.

Содержание

ВВЕДЕНИЕ

Лабораторная работа посвящена алгоритмам умножения матриц, важной операции в линейной алгебре и вычислительной математике. Умножение матриц играет важную роль во многих областях, таких как машинное обучение, обработка изображений, криптография и другие.

Целью данной работы является изучение, реализация и исследование алгоритмов умножения матриц - классический алгоритм, алгоритм Винограда, оптимизированный алгоритм Винограда и алгоритм Штрассена. Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить и реализовать алгоритмы умножения матриц: классический, Винограда, его оптимизацию и Штрассена;
- провести тестирование, чтобы измерить время выполнения и использование помяти для каждого алгоритма;
- провести сравнительный анализ процессорного времени приведенных алгоритмов;
- описать и обосновать полученные результаты в отчете о выполненной лабораторной работе.

1 Аналитическая часть

В этом разделе будут представлены алгоритмы умножения матриц: классический, Винограда и Штрассена.

1.1 Матрица

Матрица - математический объект, который представляет собой двумерный массив, в котором элементы располагаются по строкам и столбцам[1].

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

Можно выделить следующие операции над матрицами:

- матрицы одинакового размера можно складывать и вычитать;
- количество столбцов одной матрицы равно количеству строк другой матрицы - их можно перемножить, причем количество строк будет, как у первой матрицы, а столбцов - как у второй.

Замечание: операция умножения матриц не коммутативна - если A и B - квадратные матрицы, а C - результат их перемножения, то произведение AB и BA дадут разный результат C.

1.2 Стандартный алгоритм

Пусть даны две матрицы

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

тогда матрица C

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

где

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

будет называться произведением матриц A и B.

Стандартный алгоритм реализует данную формулу.

1.3 Алгоритм Винограда

Алгоритм Винограда — алгоритм умножения квадратных матриц. Начальная версия имела асимптотическую сложность алгоритма примерно $O(n^{2,3755})$, где n - размер стороны матрицы, но после доработки он стал обладать лучшей асимптотикой среди всех алгоритмов умножения матриц[2].

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: $V \cdot W = v_1 w_1 + v_2 w_2 + v_3 w_3 + v_4 w_4$, что эквивалентно (1.4):

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

Пусть матрицы A, B, C ранее определенных размеров. Упомянутое скалярное произведение, по замыслу Винограда, можно произвести иначе в формуле 1.5:

$$C_{ij} = \sum_{k=1}^{q/2} (a_{i,2k-1} + b_{2k,j})(a_{i,2k} + b_{2k-1,j}) - \sum_{k=1}^{q/2} a_{i,2k-1} a_{i,2k} - \sum_{k=1}^{q/2} b_{2k-1,j} b_{2k,j} \tag{1.5}$$

Казалось бы, это только увеличит количество арифметических операций по сравнению с классическим методом, однако Виноград предложил находить второе и третье слагаемые в Формуле 1.5 предварительном этапе вы-

числений, заранее для каждой строки матрицы A и столбца B соответственно. Так, вычислив единожды для строки i матрицы A значение выражения $\sum_{k=1}^{q/2} a_{i,2k-1}a_{i,2k}$ его можно далее использовать m раз при нахождении элементов i-ой строчки матрицы C. Аналогично, вычислив единожды для столбца j матрицы B значение выражения $\sum_{k=1}^{q/2} b_{2k-1,j}b_{2k,j}$ его можно далее использовать n раз при нахождении элементов j-ого столбца матрицы C[3].

За счёт предварительной обработки данных можно получить прирост производительности: несмотря на то, что полученное выражение требует большего количества операций, чем стандартное умножение матриц, выражение в правой части равенства можно вычислить заранее и запомнить для каждой строки первой матрицы и каждого столбца второй матрицы. Это позволит выполнить лишь два умножения и пять сложений, при учёте, что потом будет сложено только с двумя предварительно посчитанными суммами соседних элементов текущих строк и столбцов. Операция сложения выполняется быстрее, поэтому на практике алгоритм должен работать быстрее обычного алгоритма перемножения матриц.

Стоит упомянуть, что при нечётном значении размера матрицы нужно дополнительно добавить произведения крайних элементов соответствующих строк и столбцов.

1.4 Оптимизированный алгоритм Винограда

При программной реализации рассмотренного выше алгоритма Винограда можно сделать следующие оптимизации:

1. предвычислять некоторые слагаемые для алгоритма;
2. операцию умножения на 2 программно эффективнее реализовывать как побитовый сдвиг влево на 1;
3. операции сложения и вычитания с присваиванием следует реализовывать при помощи соответствующего оператора $+=$ или $-=$ (при наличии данных операторов в выбранном языке программирования).

1.5 Алгоритм Штрассена

Алгоритм Штрассена является эффективным методом умножения матриц, который основан на принципе "разделяй и властвуй". Он позволяет уменьшить время выполнения умножения матриц за счет рекурсивного разделения матриц на более маленькие подматрицы[4].

Алгоритм Штрассена следует следующим шагам:

1. Разделить исходные матрицы A и B на равные подматрицы размером $n/2$.

2. Вычислить семь промежуточных матриц путем рекурсивного умножения.

3. Сложить и вычесть эти промежуточные матрицы, чтобы получить результат умножения.

Алгоритм Штрассена имеет следующую рекурсивную формулу (1.6).

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{1.6}$$

$C_{11} = M_1 + M_4 - M_5 + M_7$, $C_{12} = M_3 + M_5$,
$C_{21} = M_2 + M_4$, $C_{22} = M_1 - M_2 + M_3 + M_6$,
где
$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$, $M_2 = (A_{21} + A_{22})B_{11}$, $M_3 = A_{11}(B_{12} - B_{22})$,
$M_4 = A_{22}(B_{21} - B_{11})$, $M_5 = (A_{11} + A_{12})B_{22}$, $M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$,
$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$.

Алгоритм Штрассена имеет лучшую асимптотическую сложность, чем стандартный алгоритм умножения матриц. Однако он имеет более высокую константу, поэтому может быть эффективным только для очень больших матриц. Реализация алгоритма Штрассена требует дополнительной памяти для хранения промежуточных матриц, поэтому также чувствителен к ограниченным ресурсам памяти компьютера.

1.6 Вывод

В данном разделе были рассмотрены алгоритмы умножения матриц - стандартного, Винограда, Винограда с оптимизацией и Штрассена. Алгоритм стандартного умножения матриц является простым и прямолинейным, но имеет временную сложность $O(n^3)$, где n - размерность матрицы.

Алгоритм умножения матриц Винограда вводит дополнительные расчеты для оптимизации операций умножения. Это позволяет сократить количество операций умножения и тем самым улучшить производительность. Однако, он оказывается выигрышным только для больших матриц.

Алгоритм Штрассена базируется на использовании деления матрицы на подматрицы и рекурсивном умножении этих подматриц. Это позволяет значительно сократить количество операций умножения, но требует дополнительных операций сложения матриц. В определенных случаях (зависит от размера матрицы) алгоритм Штрассена оказывается эффективнее остальных алгоритмов, но на маленьких матрицах может быть менее производительным.

2 Конструкторская часть

В этом разделе будут представлены схемы алгоритмов перемножения матриц - стандартного, Винограда, оптимизации алгоритма Винограда и Штрассена.

2.1 Сведения о модулях программы

Программа состоит из двух модулей:

- *main.py* - файл, содержащий весь служебный код;
- *algorythms.py* - файл, содержащий код всех алгоритмов перемножения матриц.

2.2 Разработка алгоритмов

На рисунке 2.1 представлена схема алгоритма для стандартного умножения матриц. На рисунках 2.2-2.3 схема алгоритма Винограда умножения матриц, на 2.4-2.5 схема оптимизированного алгоритма Винограда, а на рисунках 2.6-2.7 схема алгоритма Штрассена.

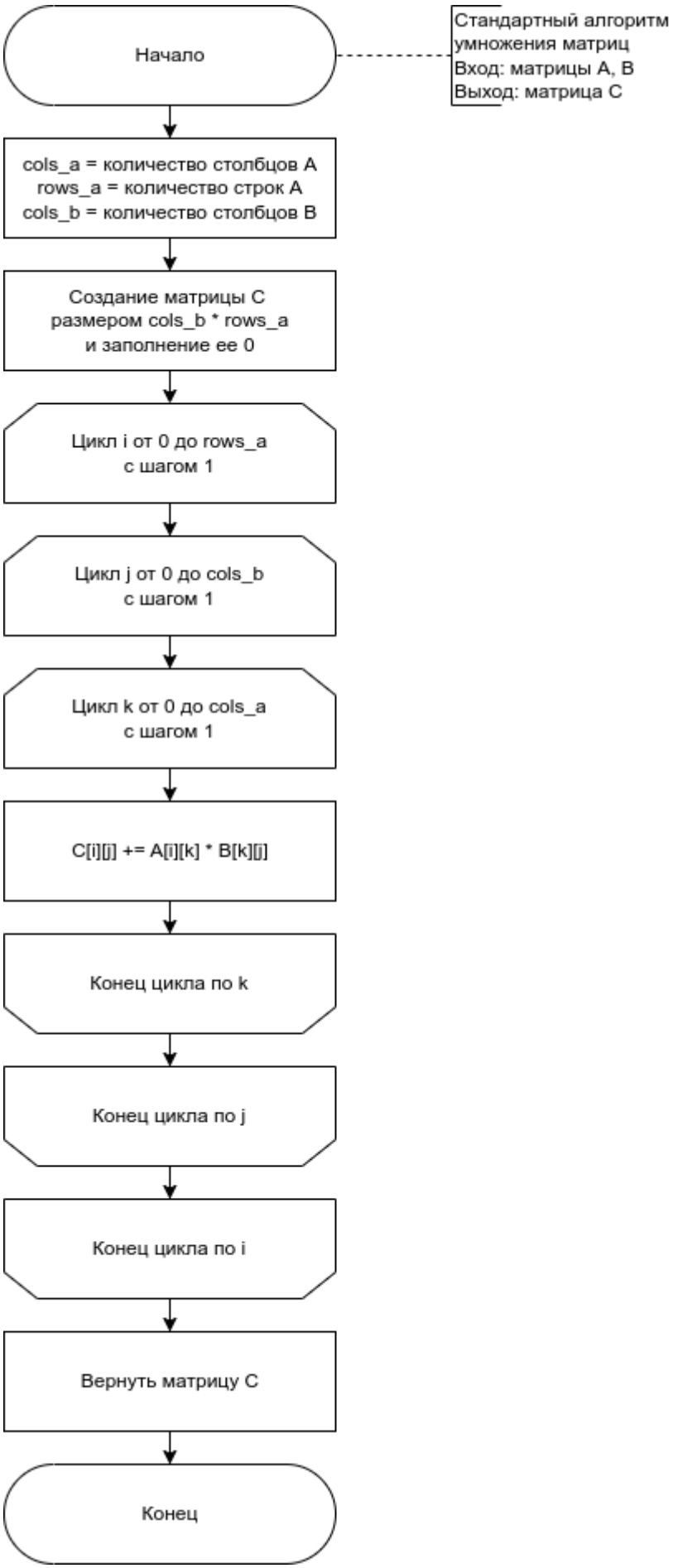

Рисунок 2.1 – Схема стандартного алгоритма умножения матриц

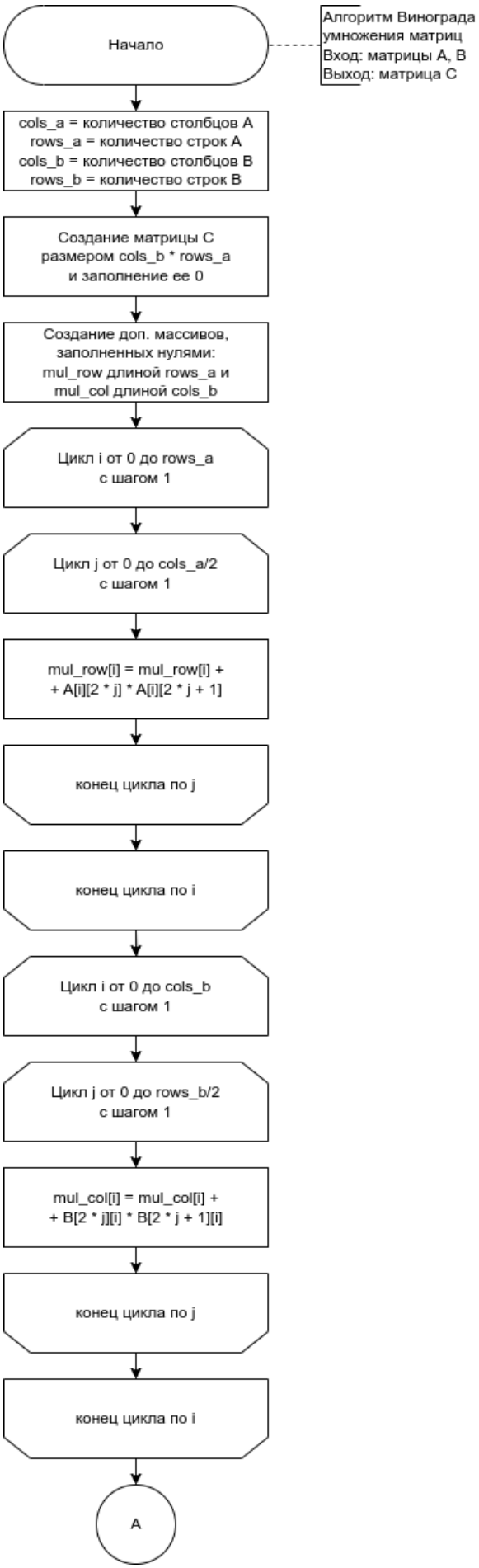

Рисунок 2.2 – Схема алгоритма Винограда (часть 1)

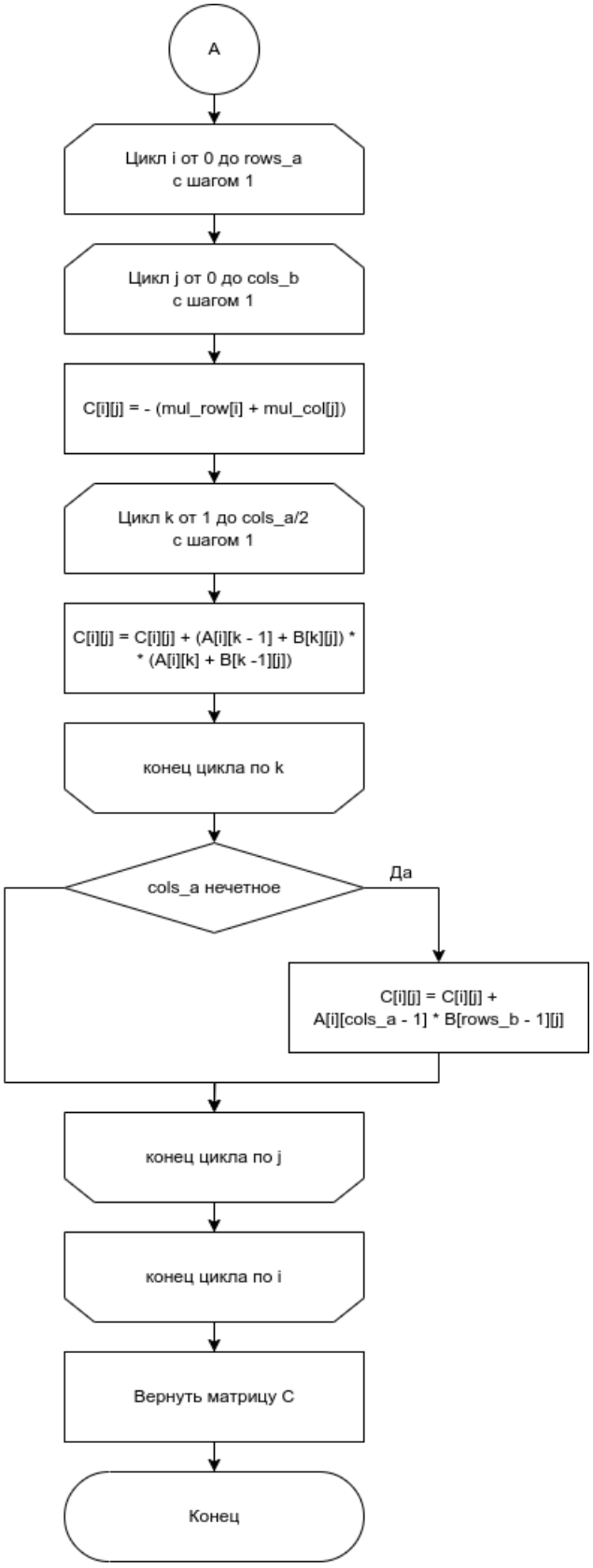

Рисунок 2.3 – Схема алгоритма Винограда (часть 2)

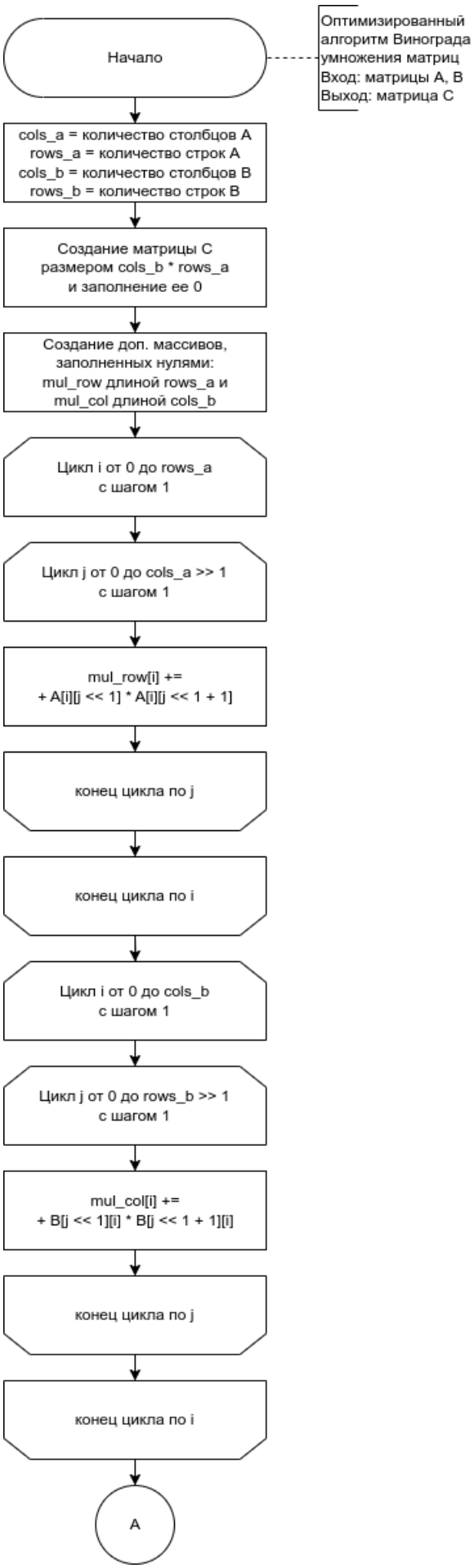

Рисунок 2.4 – Схема оптимизированного алгоритма Винограда (часть 1)

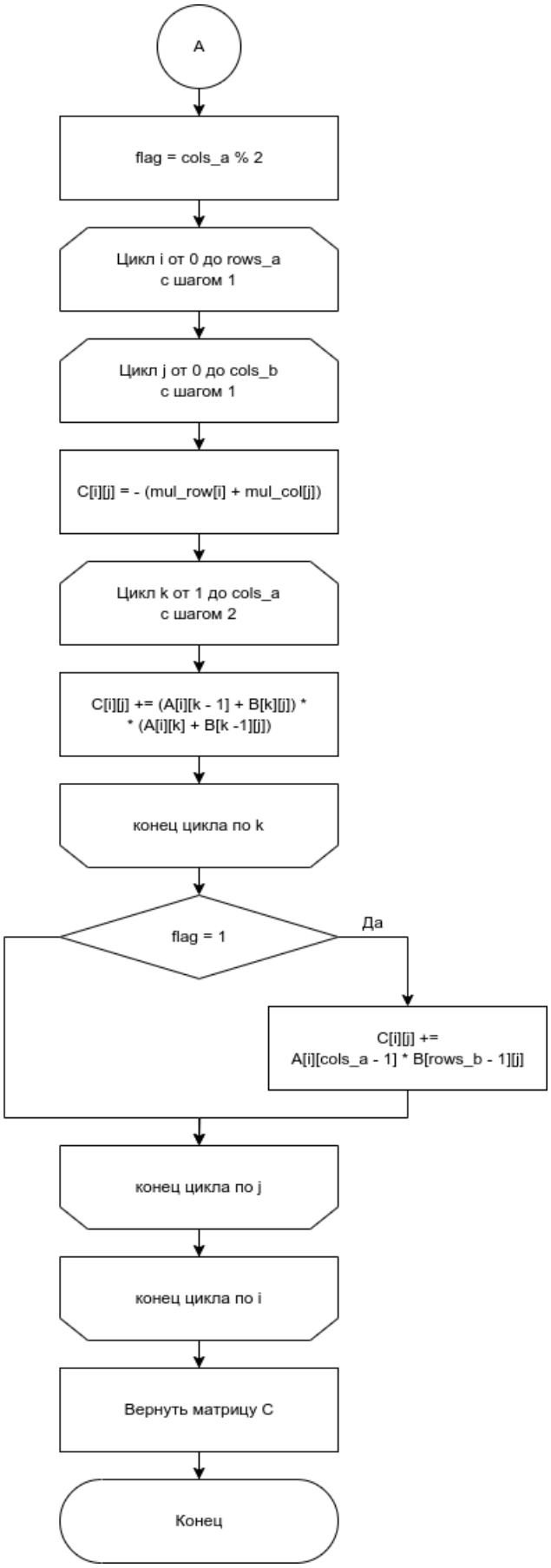

Рисунок 2.5 – Схема оптимизированного алгоритма Винограда (часть 2)

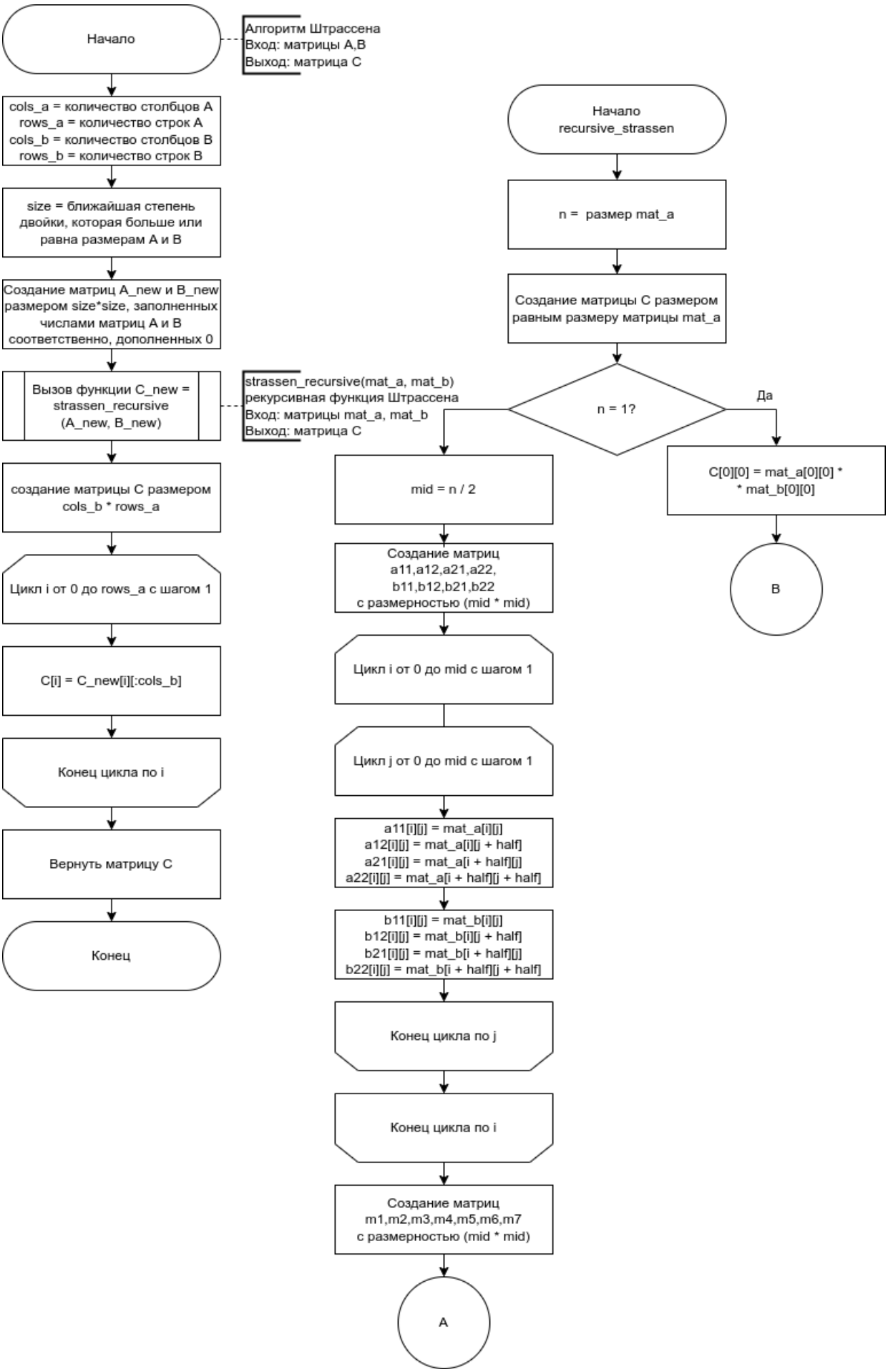

Рисунок 2.6 – Схема алгоритма Штрассена (часть 1)

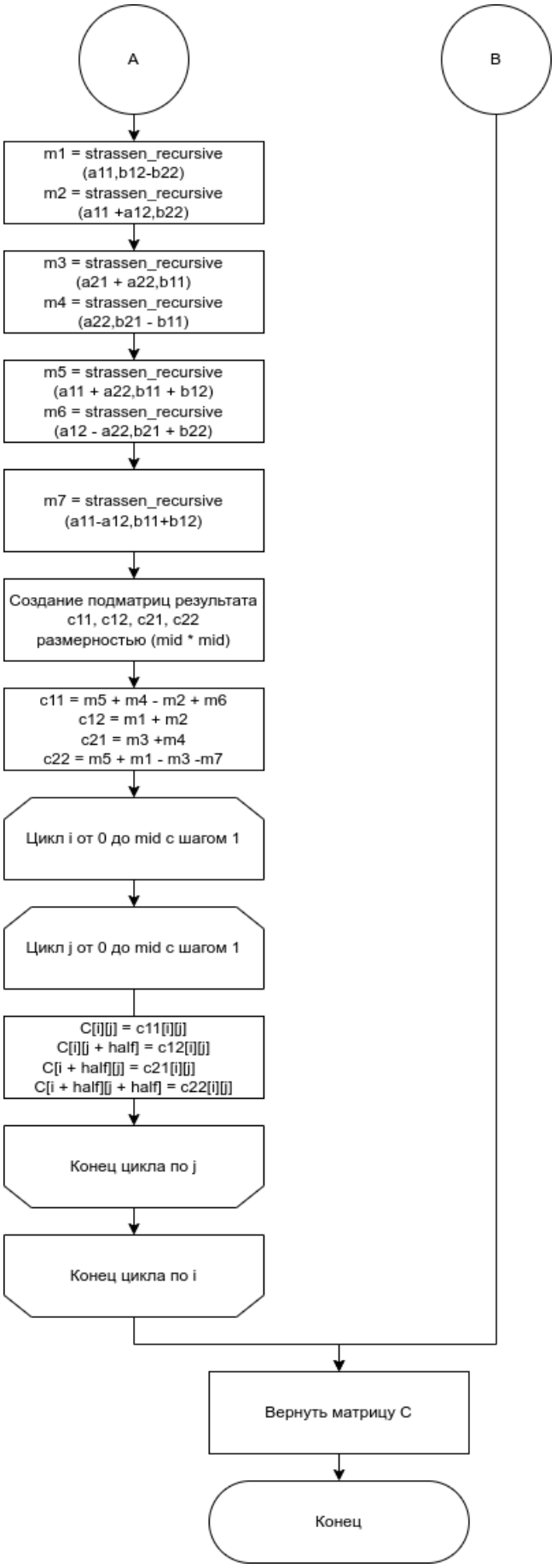

Рисунок 2.7 – Схема алгоритма Штрассена (часть 2)

2.3 Модель вычислений

Чтобы провести вычисление трудоемкости алгоритмов умножения матриц, введем модель вычислений [5]:

1. Трудоемкость следующих базовых операций единична: +, -, =, +=, -=, ==, !=, <, >, <=, >=, [], ++, –, «, ». Операции *, %, / имеют трудоемкость 2.

2. трудоемкость оператора выбора `if условие then A else B` рассчитывается, как (2.1);

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.1}$$

3. трудоемкость цикла рассчитывается, как (2.2);

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \tag{2.2}$$

4. трудоемкость передачи параметров в функцию и возврат из нее равны 0.

2.4 Трудоемкость алгоритмов

Рассчитаем трудоемкость алгоритмов умножения матриц. Введем обозначения: $M =$ количество строк матрицы , $N-$ количество столбцов B, $K-$ количество столбцов или количество строк B.

2.4.1 Стандартный алгоритм умножения матриц

Для стандартного алгоритма умножения матриц трудоемкость будет складываться из:

- внешнего цикла по $i \in [1..M]$, трудоёмкость которого: $f = 2 + M \cdot (2 + f_{body})$;
- цикла по $j \in [1..N]$, трудоёмкость которого: $f = 2 + N \cdot (2 + f_{body})$;
- цикла по $k \in [1..K]$, трудоёмкость которого: $f = 2 + 9K$.

Поскольку трудоемкость стандартного алгоритма равна трудоемкости внешнего цикла, то:

$$f_{standard} = 2+M\cdot(4+N\cdot(4+9K)) = 2+4M+4MN+9MNK \approx 9MNK \quad (2.3)$$

2.4.2 Алгоритм Винограда

Чтобы вычислить трудоемкость алгоритма Винограда, нужно учесть следующее:

- создание и инициализация массивов a_tmp и b_tmp, трудоёмкость которых (2.4):
$$f_{init} = M + N; \quad (2.4)$$
- заполнение массива a_tmp, трудоёмкость которого (2.5):
$$f_{a\_tmp} = 2 + M(4 + \frac{K}{2} \cdot 15); \quad (2.5)$$
- заполнение массива b_tmp, трудоёмкость которого (2.6):
$$f_{b\_tmp} = 2 + N(4 + \frac{K}{2} \cdot 15); \quad (2.6)$$
- цикл заполнения для чётных размеров, трудоёмкость которого (2.7):
$$f_{cycle} = 2 + M(4 + N \cdot (14 + \frac{K}{2} \cdot 28)); \quad (2.7)$$

- цикл заполнения для нечётных размеров, трудоемкость которого (2.8):

$$f_{cycle} = 2 + M(4 + N \cdot (28 + \frac{K}{2} \cdot 28)). \quad (2.8)$$

Тогда для худшего случая (нечётный общий размер матриц) имеем (2.9):

$$f_{worst} = f_{a\_tmp} + f_{b\_tmp} + f_{cycle_odd} \approx 14 \cdot MNK \quad (2.9)$$

Для лучшего случая (чётный общий размер матриц) имеем (2.10):

$$f_{best} = f_{a\_tmp} + f_{a\_tmp} + f_{cycle} \approx 14 \cdot MNK \quad (2.10)$$

2.4.3 Оптимизированный алгоритм Винограда

Оптимизация заключается в:

- использовании побитового сдвига вместо деления на 2;
- операции сложения и вычитания заменены на операции $+=$ и $-=$ соответственно;
- вычисление четности матрицы вынесено из цикла.

Тогда трудоемкость оптимизированного алгоритма Винограда состоит из:

- создания и инициализации массивов a_tmp и b_tmp (2.4);
- заполнения массива a_tmp, трудоёмкость которого (2.5);
- заполнения массива b_tmp, трудоёмкость которого (2.6);
- цикла заполнения для чётных размеров, трудоёмкость которого (2.11):

$$f_{cycle} = 2 + M(4 + N \cdot (11 + \frac{K}{2} \cdot 17)); \quad (2.11)$$

- цикла заполнения для чётных размеров, трудоёмкость которого (2.12):

$$f_{cycle} = 2 + M(4 + N \cdot (22 + \frac{K}{2} \cdot 17)). \tag{2.12}$$

Тогда для худшего случая (нечётный общий размер матриц) имеем (2.13):

$$f_{worst} = f_{a\_tmp} + f_{a\_tmp} + f_{impr_{cycle_odd}} \approx 8.5MNK \tag{2.13}$$

Для лучшего случая (чётный общий размер матриц) имеем (2.14):

$$f_{best} = f_{a\_tmp} + f_{a\_tmp} + f_{impr_{cycle}} \approx 8.5MNK \tag{2.14}$$

2.4.4 Алгоритм Штрассена

Введем размер S - ближайшая степень двойки, которая больше или равна размерам матриц A и B.

Для алгоритма Штрассена умножения матриц трудоемкость будет складываться из:

- создания и инициализации матриц a_new и b_new, трудоемкость которого (2.15):

$$f_{init} = f_{a\_new} + f_{b\_new} = 2 * S; \tag{2.15}$$

- заполнения матрицы a_new, трудоемкость которого (2.16):

$$f_{fill_a} = 2 + M(4 + K \cdot 5); \tag{2.16}$$

- заполнения матрицы b_new, трудоемкость которого (2.17):

$$f_{fill_b} = 2 + K(4 + N \cdot 5); \tag{2.17}$$

- вызова функции $strassen\_recursive$, трудоемкость которого 0 и трудоемкости этой функции, которую посчитаем отдельно;

- заполнения матрицы C, трудоемкость которого (2.18):

$$f_{fill_c} = 2 + M \cdot 3. \tag{2.18}$$

Вычислим трудоемкость функции *strassen_recursive*. Пусть n - размер матриц, которые передаются в эту функцию, тогда трудоемкость состоит из:

- создания и инициализации матриц a11, a12, a21, a22, b11, b12, b21, b22, трудоемкость которого (2.19):

$$f_{init} = 8 \cdot (\frac{n}{2} + \frac{n}{2}) = 8n; \tag{2.19}$$

- заполнения этих матриц, трудоемкость которого (2.20):

$$f_{fill} = 16 + 20n; \tag{2.20}$$

- вызова *strassen_recursive* 7 раз, трудоемкость которых 0, но трудоемкость вычисления передаваемых параметров (2.21):

$$f_{calc\_param} = 10 \cdot (2 + n + 2n^2); \tag{2.21}$$

- вычисления подматриц результата, трудоемкость которого (2.22):

$$f_{calc\_res} = 8 \cdot (2 + n + 2n^2); \tag{2.22}$$

- составления результирующей матрицы, трудоемкость которого (2.23):

$$f_{res} = 4 + 4n. \tag{2.23}$$

Таким образом, общая трудоемкость алгоритма Штрассена (2.24):

$$f_{sht} = 6 + 2S + 7M + 5MK + 5NK + \sum_{n=1}^{\frac{n}{2}} 52 + 28n + 36n^2 \tag{2.24}$$

Сумма происходит для n по степеням двойки от 1 до $\frac{n}{2}$.

2.5 Классы эквивалентности при тестировании

Для тестирования выделены классы эквивалентности, представленные ниже.

1. Одна из матриц - пустая;
2. Количество столбцов одной матрицы не равно количеству строк второй матрицы;
3. Перемножение квадратных матриц;
4. Перемножение матриц разных размеров (при этом количество столбцов одной матрицы не равно количеству строк второй матрицы).

2.6 Вывод

В данном разделе были построены схемы алгоритмов умножения матриц рассматриваемых в лабораторной работе, были описаны классы эквивалентности для тестирования, модули программы, а также проведена теоретическая оценка трудоемкости алгоритмов.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, используемые типы данных, а также представлены листинги алгоритмов умножения матриц.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования *Python*[6]. В текущей лабораторной работе требуется замерить процессорное время для выполняемой программы, а также построить графики. Все эти инструменты присутствуют в выбранном языке программирования.

Время работы было замерено с помощью функции *process_time(...)* из библиотеки *time*[7].

3.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие типы данных:

- количество строк - целое число типа *int*;
- количество столбцов - целое число типа *int*;
- матрица - двумерный список типа *int*.

3.3 Реализация алгоритмов

В листингах 3.1-3.3 представлены реализации алгоритмов умножения матриц - стандартного, Винограда, оптимизированного алгоритма Винограда. Реализация алгоритма Штрассена находится в приложении.

Листинг 3.1 – Стандартный алгоритм умножения матриц

```
def standart_alg(mat_a, mat_b):
    rows_a = len(mat_a)
    cols_a = len(mat_a[0])
    cols_b = len(mat_b[0])
    if cols_a != len(mat_b):
        return []
    result = [[0] * cols_b for _ in range(rows_a)]
    for i in range(rows_a):
        for j in range(cols_b):
            for k in range(cols_a):
                result[i][j] += mat_a[i][k] * mat_b[k][j]
    return result
```

Листинг 3.2 – Оптимизированный алгоритм Винограда

```
def optimized_vinograd_alg(mat_a, mat_b):
    rows_a = len(mat_a)
    cols_a = len(mat_a[0])
    rows_b = len(mat_b)
    cols_b = len(mat_b[0])
    if cols_a != rows_b:
        return []
    result = [[0] * cols_b for _ in range(rows_a)]
    mul_row = [0] * rows_a
    mul_col = [0] * cols_b
    for i in range(rows_a):
        for j in range(cols_a >> 1):
            mul_row[i] += mat_a[i][j << 1] * mat_a[i][(j << 1) + 1]
    for i in range(cols_b):
        for j in range(rows_b >> 1):
            mul_col[i] += mat_b[j << 1][i] * mat_b[(j << 1) + 1][i]
    flag = cols_a % 2
    for i in range(rows_a):
        for j in range(cols_b):
            result[i][j] = -mul_row[i] - mul_col[j]
    for k in range(1, cols_a, 2):
        result[i][j] += (mat_a[i][k - 1] + mat_b[k][j]) *
           (mat_a[i][k] + mat_b[k - 1][j])
    if flag:
        result[i][j] += mat_a[i][cols_a - 1] * mat_b[rows_b - 1][j]
    return result
```

Листинг 3.3 – Алгоритм Винограда

```
def vinograd_alg(mat_a, mat_b):
    rows_a = len(mat_a)
    cols_a = len(mat_a[0])
    rows_b = len(mat_b)
    cols_b = len(mat_b[0])
    if cols_a != rows_b:
        return []
    result = [[0] * cols_b for _ in range(rows_a)]
    mul_row = [0] * rows_a
    mul_col = [0] * cols_b
    for i in range(rows_a):
        for j in range(cols_a // 2):
            mul_row[i] = mul_row[i] + mat_a[i][2 * j] * mat_a[i][2
                * j + 1]
    for i in range(cols_b):
        for j in range(rows_b // 2):
            mul_col[i] = mul_col[i] + mat_b[2 * j][i] * mat_b[2 * j
                + 1][i]
    for i in range(rows_a):
        for j in range(cols_b):
            result[i][j] = -mul_row[i] - mul_col[j]
            for k in range(cols_a // 2):
                result[i][j] = result[i][j] + (mat_a[i][2 * k] +
                    mat_b[2 * k + 1][j]) * (mat_a[i][2 * k + 1] +
                    mat_b[2 * k][j])
            if cols_a % 2 == 1:
                result[i][j] = result[i][j] + mat_a[i][cols_a - 1]
                    * mat_b[rows_b - 1][j]
    return result
```

3.4 Функциональные тесты

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы умножения матриц, рассматриваемых в данной лабораторной работе. Тесты *для всех алгоритмов* пройдены успешно.

Таблица 3.1 – Функциональные тесты

Матрица 1	Матрица 2	Ожидаемый результат
$\begin{pmatrix} 1 & 5 & 7 \\ 2 & 6 & 8 \\ 3 & 7 & 9 \end{pmatrix}$	$(\ \ )$	Сообщение об ошибке
$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	Сообщение об ошибке
$\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$
(2)	(2)	(4)

3.5 Вывод

Были представлены листинги всех алгоритмов умножения матриц - стандартного, Винограда, оптимизированного алгоритма Винограда и Штрассена. Также в данном разделе была приведена информация о выбранных средствах для разработки алгоритмов и используемых типов данных.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программы, а также проведен сравнительный анализ алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее:

- операционная система: Ubuntu 22.04.3 [8] Linux [9] x86_64;
- память: 16 Гб;
- процессор: Intel® Core™ i5-1135G7 @ 2.40Гц.

При тестировании ноутбук не был включен в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

```
      Меню

     1. Станадартное умножение матриц
     2. Алгоритм Винограда
     3. Оптимизированный алгоритм Винограда
     4. Алгоритм Штрассена
     5. Все алгоритмы
     6. Замеры времени
     0. Выход

         Выбор:     5

Введите количество строк:       2
Введите количество столбцов:    3

Введите матрицу по строчно (в одной строке - все числа для данной строки матрицы):
2 3 4
3 4 5

Введите количество строк:       3
Введите количество столбцов:    4

Введите матрицу по строчно (в одной строке - все числа для данной строки матрицы):
3 4 5 6
5 4 3 2
2 3 6 7


Результат стандартаного алгоритма:

29  32  43  46
39  43  57  61


Результат алгоритма Винограда:

29  32  43  46
39  43  57  61


Результат оптимизированного алгоритма Винограда:

29  32  43  46
39  43  57  61


Результат алгоритма Штрассена:

29  32  43  46
39  43  57  61
```

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Как было сказано выше, используется функция замера процессорного времени process_time(...) из библиотеки time на Python. Функция возвращает пользовательское процессорное время типа float.

Использовать функцию приходится дважды, затем из конечного времени нужно вычесть начальное, чтобы получить результат.

Замеры проводились для матриц размером от 2 до 75 по 100 раз на различных входных матрицах.

Результаты замеров приведены в таблице 4.1(время в мкс).

Таблица 4.1 – Результаты замеров времени

Размер	Стандартный	Виноград	Виноград (опт)	Штрассен
2	8218.84	12583.32	13206.95	44853.92
4	12207.12	16312.57	15636.80	140016.41
8	61371.23	80948.35	74771.01	964163.95
10	114888.44	147519.98	126668.54	6695137.08
15	392224.91	447195.45	367630.54	6730375.88
16	451583.92	517977.17	446052.49	6794859.96
20	884854.95	998077.34	864776.02	50403825.46
25	1733791.32	1929171.57	1599970.97	50669717.92
30	2992857.34	3191459.81	2661066.53	49970742.91
32	3410088.33	3680717.81	3153943.64	49520702.65
35	4684638.14	5025734.80	4324064.92	350374351.26
40	7043207.35	7541179.09	6344228.52	361474685.14
45	9936251.45	10389906.65	8707965.21	357458368.76
50	13718930.74	14088592.20	11780340.67	393463534.63
55	20068585.48	21020754.80	18465829.43	412183255.14
60	25700641.25	26571696.68	22441666.79	399229563.90
64	31508333.87	33101892.75	28246602.11	403796660.20
65	32977113.41	34376070.33	28565568.59	2675449374.09
70	40812887.65	41532495.83	34562454.85	2780235588.35
75	49781888.03	51604794.53	44260030.02	2595885483.53

Также на рисунках 4.2–4.3 приведены графические результаты замеров.

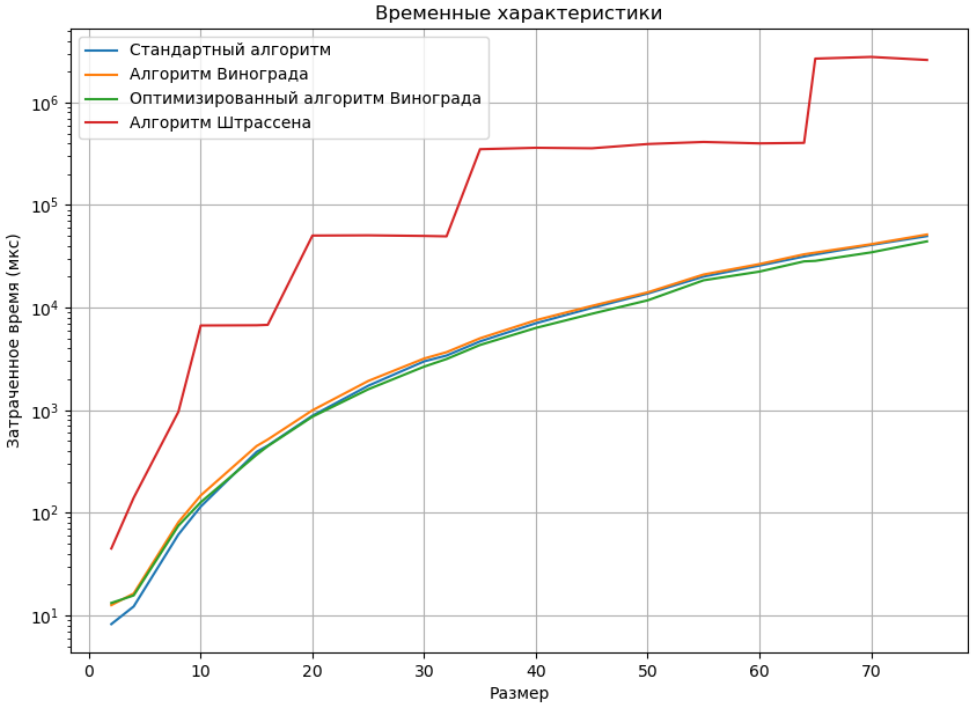

Рисунок 4.2 – Сравнение по времени алгоритмов: стандартного, Винограда, оптимизированного Винограда и Штрассена

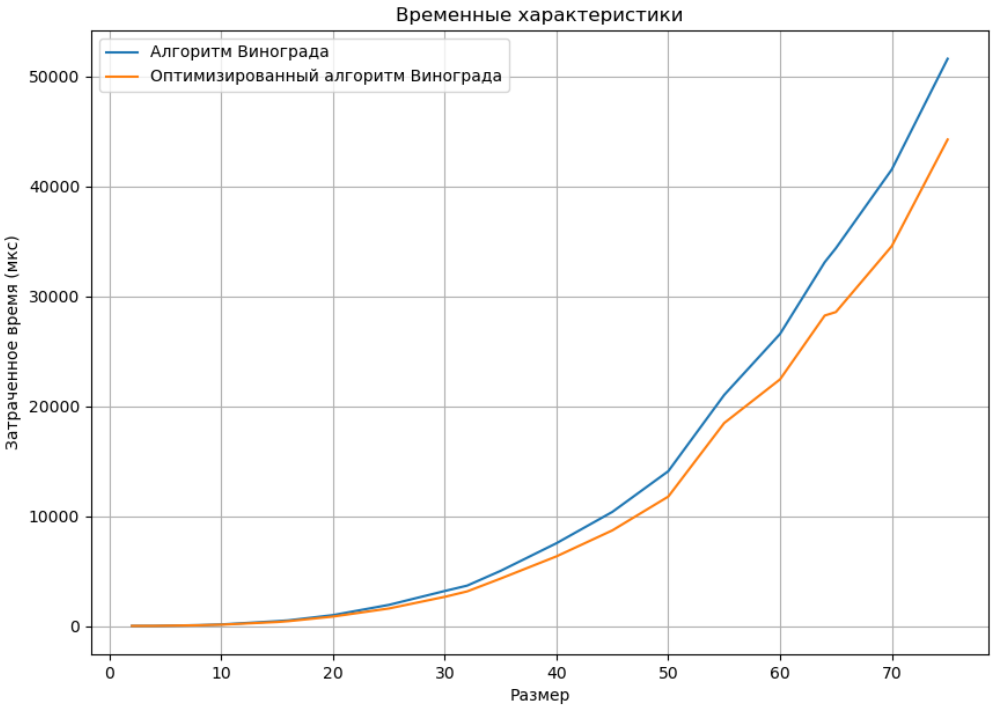

Рисунок 4.3 – Сравнение по времени алгоритма Винограда и оптимизированного алгоритма Винограда

4.4 Вывод

В результате эксперимента было получено, что при размерах матрицы свыше 10, оптимизированный алгоритм Винограда быстрее обычного алгоритма Винограда более, чем 1.2 раза, быстрее алгоритма Штрассена примерно в 20 раз, а так же быстрее стандартного алгоритма в 1.1 раз. В итоге, можно сказать, что при таких данных следует использовать оптимизированный алгоритм Винограда.

Также при проведении эксперимента было выявлено, что на четных размерах реализация алгоритма Винограда в 1.2 раза быстрее, чем на нечетных размерах матриц, что обусловлено необходимостью проводить дополнительные вычисления для крайних строк и столбцов. Следовательно, стоит использовать алгоритм Винограда для матриц, которые имеют четные размеры.

ЗАКЛЮЧЕНИЕ

В результате эксперимента было получено, что при размерах матрицы свыше 10, оптимизированный алгоритм Винограда быстрее обычного алгоритма Винограда более, чем 1.2 раза, быстрее алгоритма Штрассена примерно в 20 раз, а так же быстрее стандартного алгоритма в 1.1 раз. В итоге, можно сказать, что при таких данных следует использовать оптимизированный алгоритм Винограда.

Также при проведении эксперимента было выявлено, что на четных размерах реализация алгоритма Винограда в 1.2 раза быстрее, чем на нечетных размерах матриц, что обусловлено необходимостью проводить дополнительные вычисления для крайних строк и столбцов. Следовательно, стоит использовать алгоритм Винограда для матриц, которые имеют четные размеры.

При рассмотрении трудоемкости алгоритмов были получены следующие результаты: алгоритм Штрассена имеет наименьшую трудоемкость при матрицах, размер которых более 100 и является степенью двойки, в остальных случаях выигрывает оптимизированный алгоритм Винограда, после которого идет стандартный алгоритм, а самым трудоемким оказался обычный алгоритм Винограда.

В ходе выполнения лабораторной работы были решены следующие задачи:

- были изучены и реализованы алгоритмы умножения матриц: классический, Винограда, его оптимизацию и Штрассена;
- проведено тестирование для измерения времени выполнения и использования помяти для каждого алгоритма;
- проведен сравнительный анализ процессорного времени приведенных алгоритмов;
- подготовлен отчет о лабораторной работе.

Поставленная цель лабораторной работы была достигнута.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

[1] Матрица [Электронный ресурс]. Режим доступа: https://terme.ru/termin/matrica.html (дата обращения: 17.10.2023).

[2] Умножение матриц [Электронный ресурс]. Режим доступа: http://algolib.narod.ru/Math/Matrix.html (дата обращения: 17.10.2023).

[3] Головашкин Д. Л. Векторные алгоритмы вычислительной линейной алгебры: учеб. пособие. — Самара: Изд-во Самарского университета, 2019.

[4] Кормен Т.Х. Алгоритмы: построение и анализ. М: Пер. с анг. Изд-во «Вильямс», 2009. с. 893.

[5] М. В. Ульянов Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ. 2007.

[6] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 17.10.2023).

[7] time — Time access and conversions [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html#functions (дата обращения: 17.10.2023).

[8] Ubuntu 20.04.3 LTS (Focal Fossa) [Электронный ресурс]. Режим доступа: https://releases.ubuntu.com/20.04/ (дата обращения: 17.10.2023).

[9] Linux [Электронный ресурс]. Режим доступа: https://www.linux.org/forums/#linux-tutorials.122 (дата обращения: 17.10.2023).

ПРИЛОЖЕНИЕ 1

В листинге 1 представлена реализация алгоритма Штрассена.

Листинг 1 – Алгоритм Штрассена

```
def strassen_alg(mat_a, mat_b):
    rows_a = len(mat_a)
    cols_a = len(mat_a[0])
    rows_b = len(mat_b)
    cols_b = len(mat_b[0])
    if cols_a != rows_b:
        return []
    size = max(rows_a, cols_a, rows_b, cols_b)
    size = 2 ** (size - 1).bit_length()
    A_padded = [[0] * size for _ in range(size)]
    B_padded = [[0] * size for _ in range(size)]
    for i in range(rows_a):
        for j in range(cols_a):
            A_padded[i][j] = mat_a[i][j]
    for i in range(rows_b):
        for j in range(cols_b):
            B_padded[i][j] = mat_b[i][j]
    C_padded = strassen_recursive(A_padded, B_padded)
    C = []
    for i in range(rows_a):
        C.append(C_padded[i][:cols_b])
    return C

def strassen_recursive(mat_a, mat_b):
    n = len(mat_a)
    if n == 1:
        return [[mat_a[0][0] * mat_b[0][0]]]
    mid = n // 2
    A11 = [mat_a[i][:mid] for i in range(mid)]
    A12 = [mat_a[i][mid:] for i in range(mid)]
    A21 = [mat_a[i][:mid] for i in range(mid, n)]
    A22 = [mat_a[i][mid:] for i in range(mid, n)]
    B11 = [mat_b[i][:mid] for i in range(mid)]
    B12 = [mat_b[i][mid:] for i in range(mid)]
    B21 = [mat_b[i][:mid] for i in range(mid, n)]
    B22 = [mat_b[i][mid:] for i in range(mid, n)]
```

```
37      M1 = strassen_recursive(matrix_add(A11, A22), matrix_add(B11,
            B22))
38      M2 = strassen_recursive(matrix_add(A21, A22), B11)
39      M3 = strassen_recursive(A11, matrix_sub(B12, B22))
40      M4 = strassen_recursive(A22, matrix_sub(B21, B11))
41      M5 = strassen_recursive(matrix_add(A11, A12), B22)
42      M6 = strassen_recursive(matrix_sub(A11, A21), matrix_add(B11,
            B12))
43      M7 = strassen_recursive(matrix_sub(A12, A22), matrix_add(B21,
            B22))
44      C11 = matrix_add(matrix_sub(matrix_add(M1, M4), M5), M7)
45      C12 = matrix_add(M3, M5)
46      C21 = matrix_add(M2, M4)
47      C22 = matrix_sub(matrix_sub(matrix_add(M1, M3), M2), M6)
48      C = []
49      for i in range(mid):
50          C.append(C11[i] + C12[i])
51      for i in range(mid, n):
52          C.append(C21[i-mid] + C22[i-mid])
53      return C
54
55  def matrix_add(mat_a, mat_b):
56      return [[mat_a[i][j] + mat_b[i][j] for j in
            range(len(mat_a[i]))] for i in range(len(mat_a))]
57
58  def matrix_sub(mat_a, mat_b):
59      return [[mat_a[i][j] - mat_b[i][j] for j in
            range(len(mat_a[i]))] for i in range(len(mat_a))]
```