

МГТУ им. БАУМАНА

ЛАБОРАТОРНАЯ РАБОТА №2

По курсу: "АНАЛИЗ АЛГОРИТМОВ"

Алгоритмы умножения матриц

Работу выполнил: Луговой Дмитрий, ИУ7-51Б

Преподаватель: Волкова Л.Л.

Москва, 2019

Оглавление

Введение	3
1 Аналитическая часть	5
1.1 Стандартный алгоритм	5
1.2 Алгоритм Копперсмита-Винограда	6
2 Конструкторская часть	7
2.1 Схемы алгоритмов	8
2.2 Расчет трудоемкости	18
3 Технологическая часть	20
3.1 Требования к ПО	20
3.2 Средства реализации	20
3.3 Листинги кода	20
4 Экспериментальная часть	23
4.1 Примеры работы	23
4.2 Функциональное тестирование	24
4.3 Сравнение алгоритмов по времени	24
Заключение	26

Введение

Целью данной лабораторной работы является исследование существующих алгоритмов умножения матриц и их трудоемкости.

Примем следующую модель вычислений:

1. Трудоемкость базовых операций
Операции $+$, $-$, $*$, $/$, $\%$, $=$, $>$, $<$, \leq , \geq , $==$, \neq , $[]$, $+=$, $- =$ - имеют стоимость 1.
2. Трудоемкость условного перехода
Условный переход имеет стоимость 0, при этом оцениваем расчет условия:

```
if (n % 2 == 1)
{
    // Тело 1
}
else
{
    // Тело 2
}
```

$$f_{if} = f_{условия} + \begin{cases} f_{\text{тело 1}}, & \text{при нечетном } N \\ f_{\text{тело 2}}, & \text{при четном } N \end{cases}$$

3. Трудоемкость цикла *for*

$$f_{\text{цикла}} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}),$$

где N - число повторений цикла.

Задачи работы

Задачами данной лабораторной являются:

1. Реализовать следующие алгоритмы умножения матриц:
 - Стандартный алгоритм
 - Алгоритм Винограда
 - Оптимизированный алгоритм Винограда
2. Проанализировать трудоемкость данных алгоритмов
3. Провести эксперименты с замерами времени

1 | Аналитическая часть

В этом разделе содержатся описания умножения матриц.

1.1 Стандартный алгоритм

Пусть даны две прямоугольные матрицы A и B размерностей $m \times n$, $n \times q$ соответственно:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix},$$
$$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1q} \\ b_{21} & b_{22} & \cdots & b_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nq} \end{bmatrix}.$$

Тогда матрица C размерностью $m \times q$

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1q} \\ c_{21} & c_{22} & \cdots & c_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mq} \end{bmatrix},$$

в которой:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj} \quad (i = 1, 2, \dots, m; j = 1, 2, \dots, q)$$

называется их произведением. Операция умножения двух матриц выполняема только в том случае, если число столбцов в первом сомножителе равно числу строк во втором; в этом случае говорят, что матрицы согласованы. В частности, умножение всегда выполнимо, если оба сомножителя — [[Квадратная матрица|квадратная матрица]] одного и того же порядка.

Таким образом, из существования произведения AB вовсе не следует существование произведения BA

1.2 Алгоритм Копперсмита-Винограда

Алгоритм Копперсмита—Винограда — алгоритм умножения матриц, предложенный в 1987 году Д. Копперсмитом и Ш. Виноградом. В исходной версии асимптотическая сложность алгоритма составляла $O(n^{2,3755})$, где n — размер стороны матрицы. Алгоритм Копперсмита—Винограда, с учетом серии улучшений и доработок в последующие годы, обладает лучшей асимптотикой среди известных алгоритмов умножения матриц.

Если посмотреть на результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее. Рассмотрим два вектора $V = (v1, v2, v3, v4)$ и $W = (w1, w2, w3, w4)$. Их скалярное произведение равно: $V * W = v1w1 + v2w2 + v3w3 + v4w4$. Это равенство можно переписать в виде: $V * W = (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3) - v1v2 - v3v4 - w1w2 - w3w4$.

Кажется, что второе выражение задает больше работы, чем первое: вместо четырех умножений мы насчитываем их шесть, а вместо трех сложений — десять. Однако 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. На практике это означает, что над предварительно обработанными элементами нам придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

2 | Конструкторская часть

В этом разделе содержатся схемы алгоритмов умножения матриц и подсчет трудоемкости.

2.1 Схемы алгоритмов

На Рис.2.1 и 2.2 представлена схема стандартного алгоритма умножения матриц.

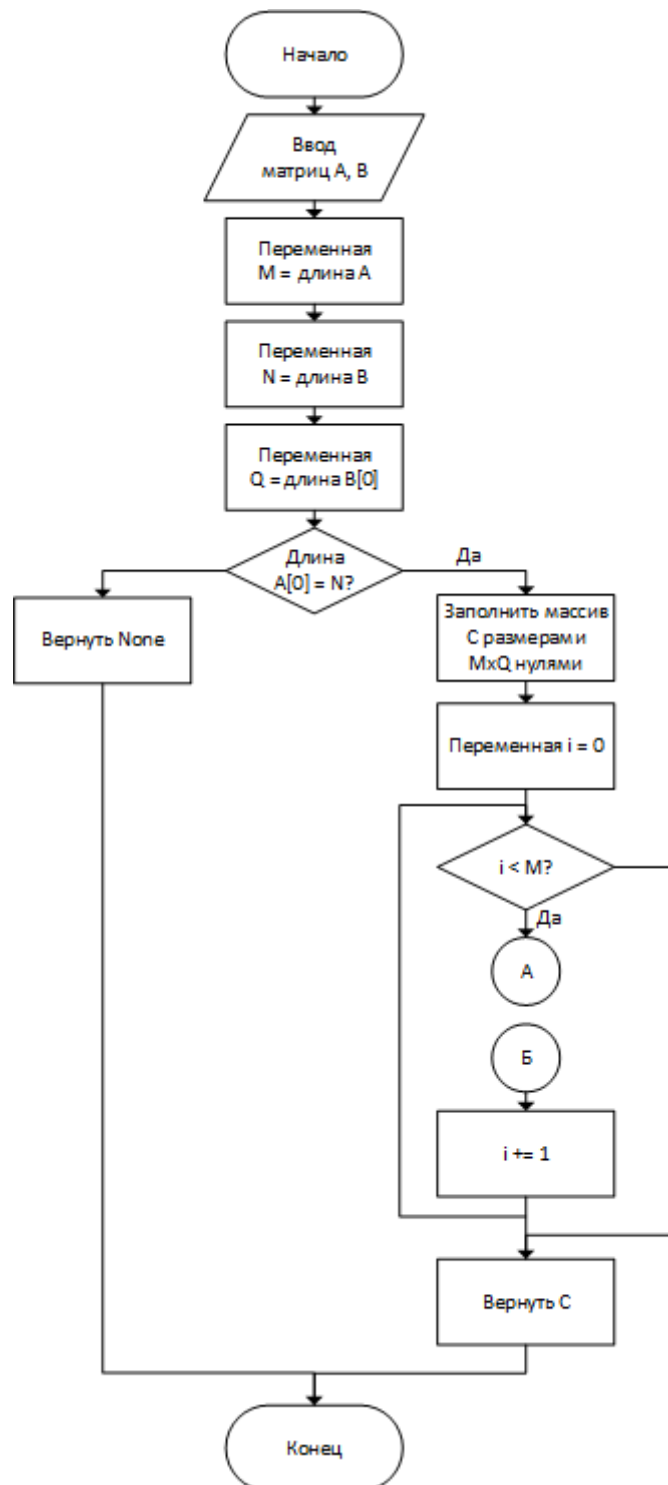


Рис. 2.1: Стандартный алгоритм умножения матриц

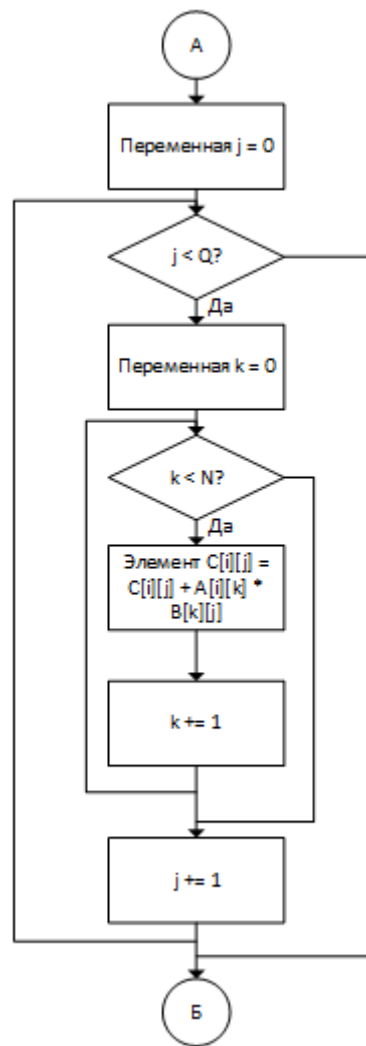


Рис. 2.2: Стандартный алгоритм умножения матриц(продолжение)

На Рис.2.3, 2.4, 2.5 и 2.6 представлена схема алгоритма Винограда умножения матриц.

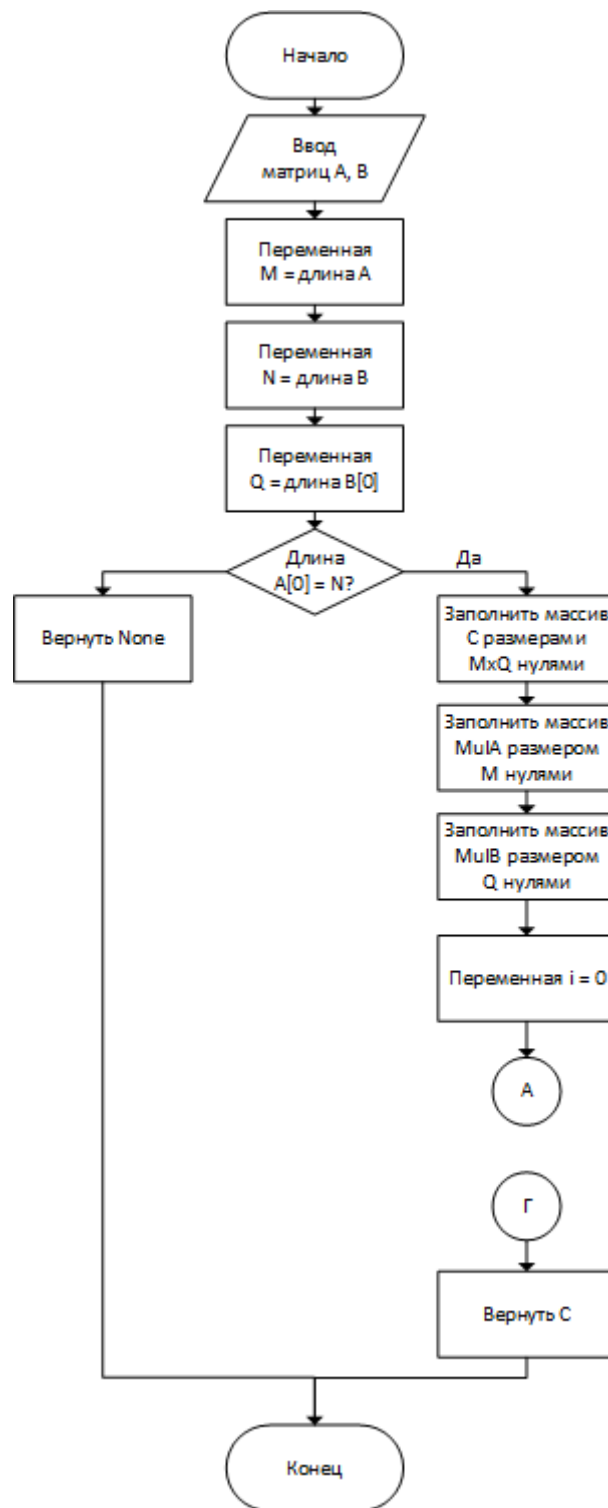


Рис. 2.3: Алгоритм Винограда умножения матриц

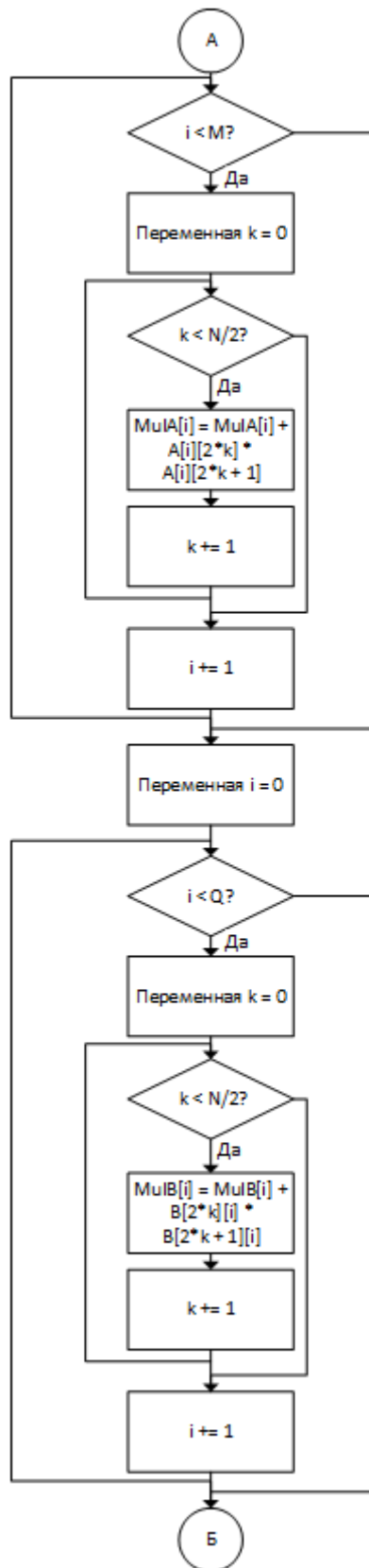


Рис. 2.4: Алгоритм Винограда умножения матриц(продолжение 1)

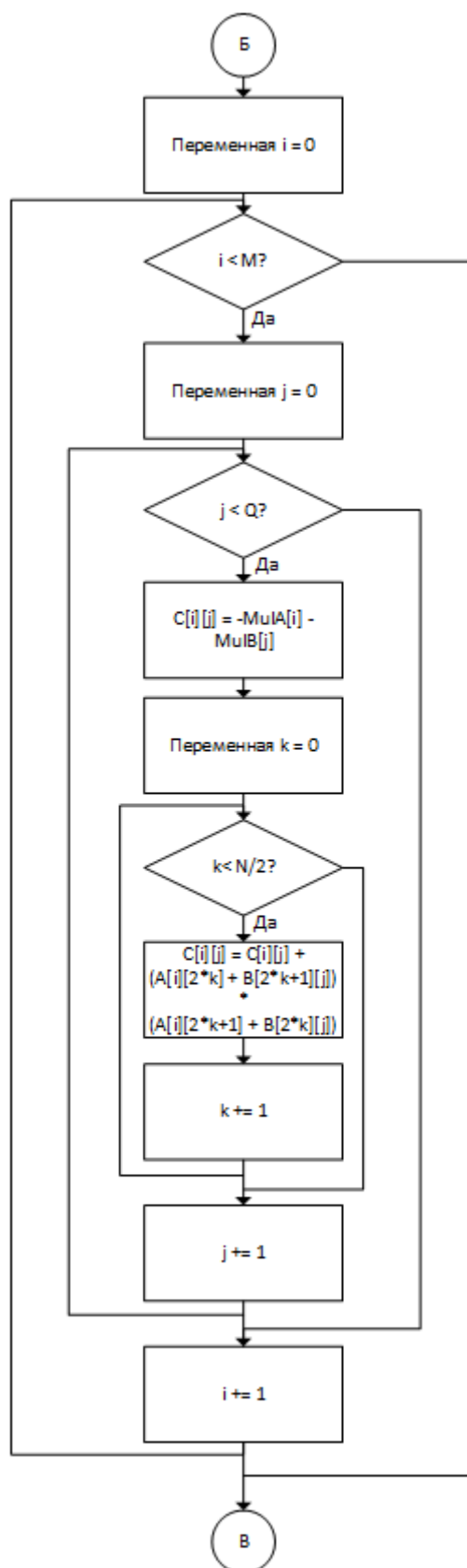


Рис. 2.5: Алгоритм Винограда умножения матриц(продолжение 2)

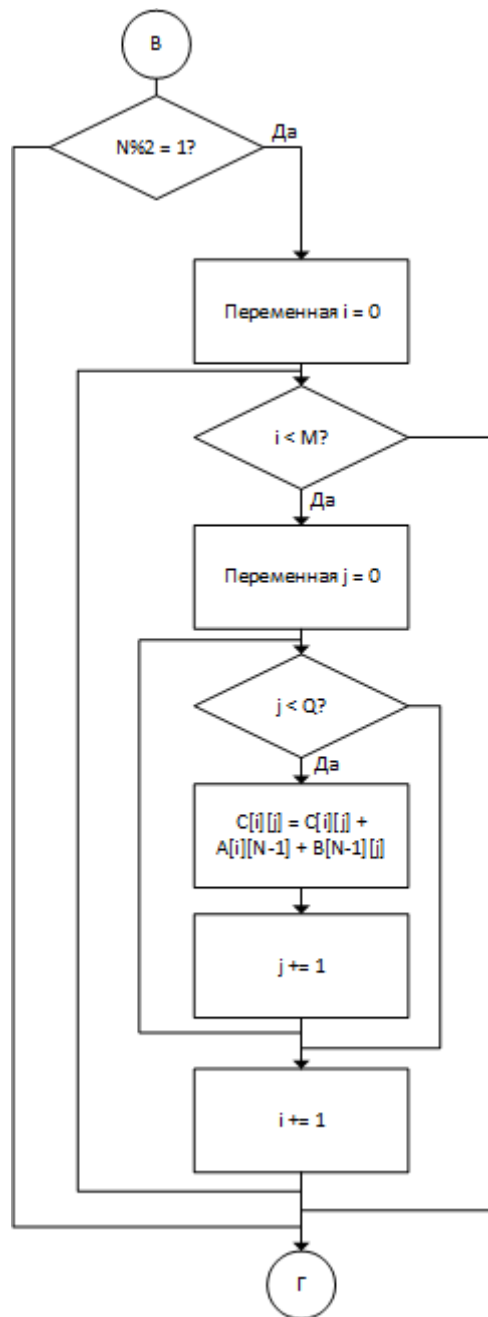


Рис. 2.6: Алгоритм Винограда умножения матриц(продолжение 3)

На Рис.2.7, 2.8, 2.9 и 2.10 представлена схема оптимизированного алгоритма Винограда умножения матриц.

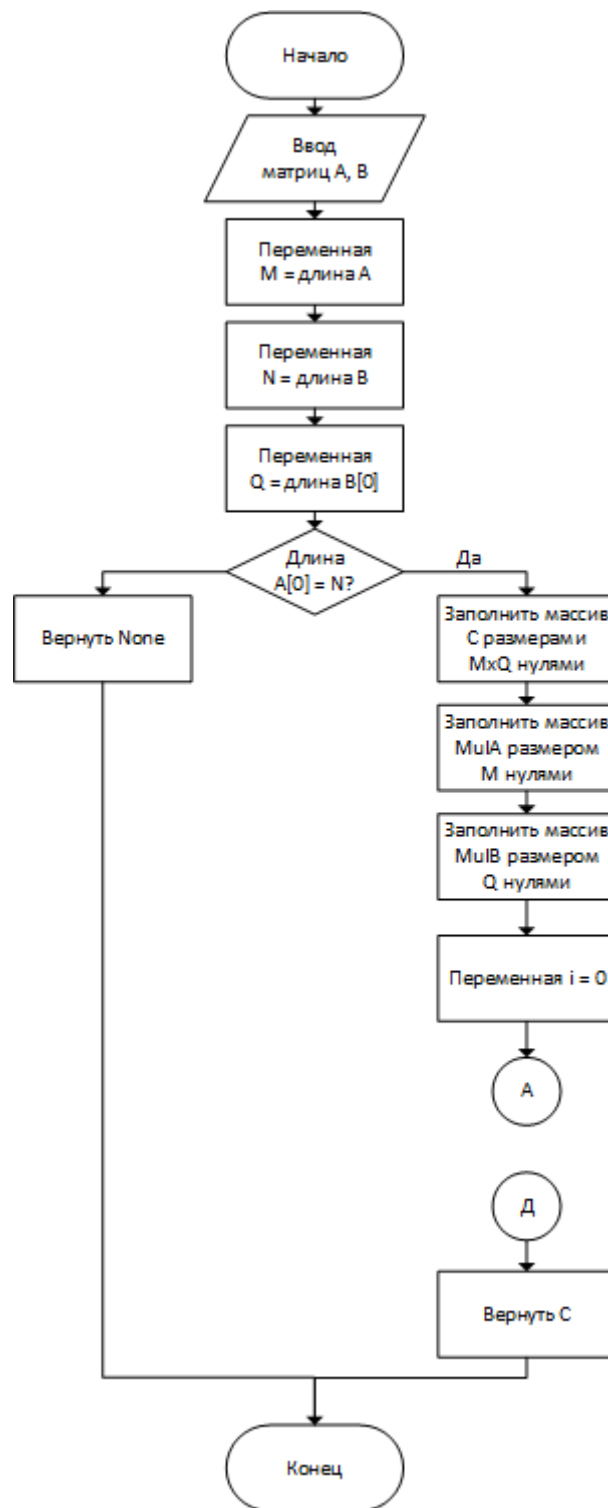


Рис. 2.7: Оптимизированный алгоритм Винограда умножения матриц

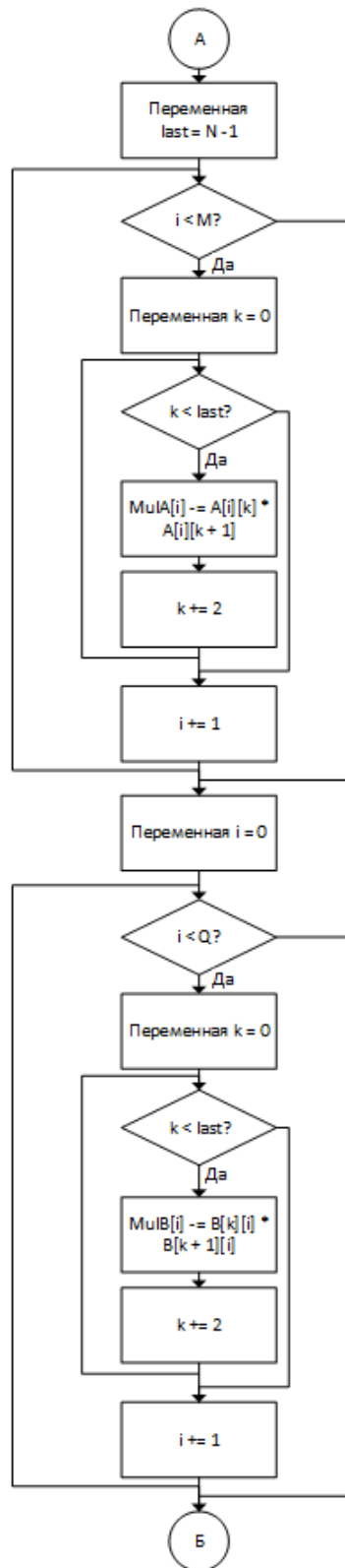


Рис. 2.8: Оптимизированный алгоритм Винограда умножения матриц(продолжение 1)

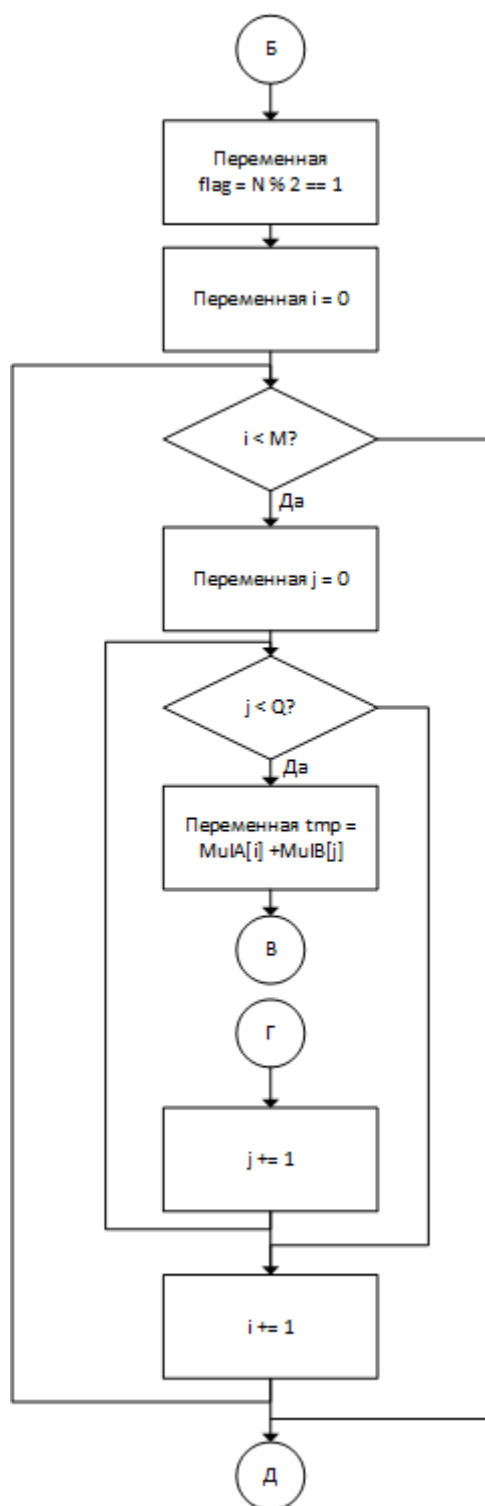


Рис. 2.9: Оптимизированный алгоритм Винограда умножения матриц(продолжение 2)

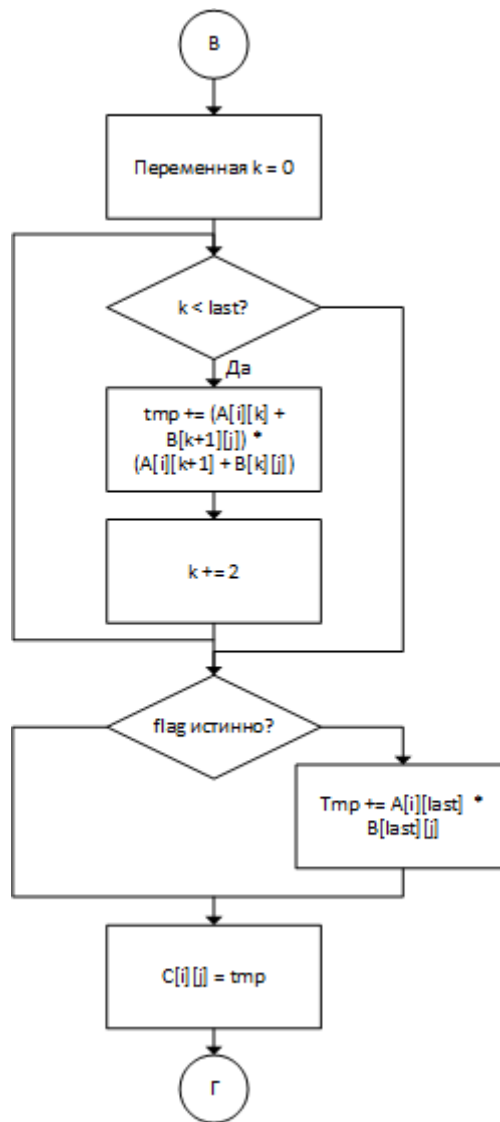


Рис. 2.10: Оптимизированный алгоритм Винограда умножения матриц(продолжение 3)

2.2 Расчет трудоемкости

Пусть заданы матрицы размерами $m \times n$, $n \times q$. Используя модель вычислений, заданную ранее, произведем подсчет трудоемкости алгоритмов умножения матриц:

- Стандартный алгоритм

$$f_{\text{std}} = 2 + M(2 + 2 + Q(2 + 2 + N(2 + 8 + 1 + 1 + 1))) = 13MNQ + 4MQ + 4M + 2 \approx 13MNQ$$

- Алгоритм Винограда

$$f_I = 2 + M(2 + 3 + \frac{N}{2}(3 + 1 + 6 + 2 + 3)) = \frac{15}{2}MN + 5M + 2$$

$$f_{II} = 2 + Q(2 + 3 + \frac{N}{2}(3 + 1 + 6 + 2 + 3)) = \frac{15}{2}QN + 5Q + 2$$

$$f_{III} = 2 + M(2 + 2 + Q(2 + 7 + 3 + \frac{N}{2}(3 + 6 + 17))) = 13MNQ + 12MQ + 4M + 2$$

$$f_{IV} = 2 + \begin{cases} 0, & N \text{ четное} \\ 12MQ + 4M + 2, & N \text{ нечетное} \end{cases}$$

$$\begin{aligned} f_{\text{вин}} &= f_I + f_{II} + f_{III} + f_{IV} = \frac{15}{2}MN + \frac{15}{2}QN + 9M + 8 + 5Q \\ &+ 13MNQ + 12MQ + \begin{cases} 0, & N \text{ четное} \\ 12MQ + 4M + 2, & N \text{ нечетное} \end{cases} \approx 13MNQ \end{aligned}$$

- Оптимизированный алгоритм Винограда

$$f_I^* = 2 + M(2 + 2 + \frac{N}{2}(2 + 1 + 5 + 1 + 1)) = 5MN + 4M + 2$$

$$f_{II}^* = 2 + Q(2 + 2 + \frac{N}{2}(2 + 1 + 5 + 1 + 1)) = 5QN + 4Q + 2$$

$$f_{III}^* = 5 + 2 + M(2 + 2 + Q(2 + 4 + 2 + \frac{N}{2}(2 + 14) + 1 + 3 + \begin{cases} 0, & N \text{ четное} \\ 6, & N \text{ нечетное} \end{cases})) =$$

$$= 7 + 4M + 12MQ + 8MNQ + \begin{cases} 0, & N \text{ четное} \\ 6MQ, & N \text{ нечетное} \end{cases}$$

$$\begin{aligned} f_{\text{вин}}^* &= f_I^* + f_{II}^* + f_{III}^* = 5MN + 8M + 11 + \\ &5QN + 4Q + 12MQ + 8MNQ + \begin{cases} 0, & N \text{ четное} \\ 6MQ, & N \text{ нечетное} \end{cases} \approx 8MNQ \end{aligned}$$

Как видно из вычислений, наиболее эффективным по трудоемкости является оптимизированный алгоритм Винограда.

Трудоемкость удалось снизить за счет следующих оптимизаций:

1. Замена $C[i][j] = C[i][j] + \dots$ на $C[i][j] += \dots$
2. Замена цикла по k от 0 до $N/2$ с шагом 1 на цикл от 0 до N с шагом 2, что убрало лишние умножения на 2
3. Элементы $MulA$ и $MulB$ сразу высчитываются отрицательными, что убирает 1 операцию отрицания в цикле
4. Замена $C[i][j] = \dots$ в цикле по k на буферную переменную, что убирает 2 операции индексирования во внутреннем цикле, но добавляет $C[i][j] = tmp$ во внешний цикл
5. Перенос проверки четности внутрь основного цикла, что ухудшило лучший случай, но улучшило худший
6. Вычисление условия четности и значения $N - 1$, тем самым улучшая и лучший, и худший случаи

3 | Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода

3.1 Требования к ПО

На вход поступают две целочисленные матрицы, на выходе должен возвращаться результат их умножения, либо сообщение о невозможности их умножения.

3.2 Средства реализации

Для реализации представленных алгоритмов был выбран язык Python. Время работы алгоритмов было замерено с помощью функции `process_time()` из библиотеки `time`.

3.3 Листинги кода

В Листинге 3.1 показана реализация стандартного алгоритма умножения матриц.

Листинг 3.1: Функция стандартного умножения матриц

```
1 def standard_multiply(a, b):
2     m = len(a)
3     n = len(b)
4     if m == 0 or n == 0 or len(a[0]) != n:
5         return None
6     q = len(b[0])
7     c = [[0 for i in range(q)] for j in range(m)]
8     for i in range(m):
9         for j in range(q):
10             for k in range(n):
11                 c[i][j] = c[i][j] + a[i][k] * b[k][j]
12     return c
```

В Листинге 3.2 показана реализация алгоритма Винограда умножения матриц.

Листинг 3.2: Функция умножения матриц алгоритмом Винограда

```
1 def vinograd_multiply(a, b):
2     m = len(a)
3     n = len(b)
4     if m == 0 or n == 0 or len(a[0]) != n:
5         return None
6     q = len(b[0])
7     # Part I
8     r = n // 2
9     mul_a = [0] * m
10    for i in range(m):
11        for j in range(r):
12            mul_a[i] = mul_a[i] + a[i][j * 2] * a[i][j * 2 + 1]
13
14    # Part II
15    mul_b = [0] * q
16    for i in range(q):
17        for j in range(r):
18            mul_b[i] = mul_b[i] + b[j * 2][i] * b[j * 2 + 1][i]
19
20    # Part III
21    c = [[0 for i in range(q)] for j in range(m)]
22    for i in range(m):
23        for j in range(q):
24            c[i][j] = -mul_a[i] - mul_b[j]
25            for k in range(r):
26                c[i][j] = c[i][j] + (a[i][2 * k] + b[2 * k + 1][j]) \
27                    * (a[i][2 * k + 1] + b[2 * k][j])
28
29    # Part IV
30    if n % 2 == 1:
31        for i in range(m):
32            for j in range(q):
33                c[i][j] = c[i][j] + a[i][n - 1] * b[n - 1][j]
34    return c
```

В Листинге 3.3 показана реализация оптимизированного алгоритма Винограда умножения матриц.

Листинг 3.3: Функция умножения матриц оптимизированным алгоритмом Винограда

```
1 def optimized_vinograd_multiply(a, b):
2     m = len(a)
3     n = len(b)
4     if m == 0 or n == 0 or len(a[0]) != n:
5         return None
6     q = len(b[0])
7     last = n - 1 # Optimization for odd numbers
8     # Part I
9     mul_a = [0] * m
10    for i in range(m):
11        for j in range(0, last, 2): # Optimization for n // 2
12            mul_a[i] -= a[i][j] * a[i][j + 1] # Optimization for negative and +=
13
14    # Part II
15    mul_b = [0] * q
16    for i in range(q):
17        for j in range(0, last, 2): # Optimization for n // 2
18            mul_b[i] -= b[j][i] * b[j + 1][i] # Optimization for negative and +=
19
20    flag = n % 2 == 1
21    # Part III
22    c = [[0 for i in range(q)] for j in range(m)]
23    for i in range(m):
24        for j in range(q):
25            tmp = mul_a[i] + mul_b[j] # Optimization for buffer
26            for k in range(0, last, 2): # Optimization for n // 2
27                tmp += (a[i][k] + b[k + 1][j]) \
28                    * (a[i][k + 1] + b[k][j]) # Optimization for +=
29            if flag: # Optimization for odd numbers
30                tmp += a[i][last] * b[last][j]
31            c[i][j] = tmp
32
33    return c
```

4 | Экспериментальная часть

В данном разделе приведены примеры работы программы, постановка эксперимента и сравнительный анализ алгоритмов на основе экспериментальных данных.

4.1 Примеры работы

Пример 1

Матрица A:

1 2 3

4 5 6

Матрица B:

1

2

3

Результирующая матрица:

14

32

Пример 2

Матрица A:

5 2

1 4

Матрица B:

0 3

-6 1

Результирующая матрица:

-12 17

-24 7

Пример 3

Матрица A:

2 7

1 3

Матрица B:

-3 7

1 -2

Результирующая матрица:

1 0

0 1

4.2 Функциональное тестирование

Было проведено функциональное тестирование программы, результаты которого занесены в Таблицу 4.1, 1 столбец которой - номер тестового случая, 2 и 3 столбцы - виды матриц, поступающих на вход, 4 столбец - ожидаемый результат, где *None* означает, что матрицы несовместимы по размеру, 5 столбец - полученный результат.

№	A	B	Ожидаемый результат	Полученный результат
1	Случайная	Пустая	None	None
2	Пустая	Случайная	None	None
3	Пустая	Пустая	None	None
4	Случайная	Нулевая	Нулевая	Нулевая
5	Нулевая	Случайная	Нулевая	Нулевая
6	Единичная	Квадратная	B	B
7	Квадратная	Единичная	A	A
8	Размера $M \times N$	Размера $M \times N$	None	None

Таблица 4.1: Тестовые случаи

Программа успешно прошла все тестовые случаи, все полученные результаты совпали с ожидаемыми.

4.3 Сравнение алгоритмов по времени

Для экспериментов использовались матрицы, размер которых варьируется от 100×100 до 1000×1000 с шагом 100 для матриц четных размеров и от 101×101 до 1001×1001 с шагом 100 для матриц нечетных размеров. Количество повторов каждого эксперимента = 100. Результат одного эксперимента рассчитывается как средний из результатов проведенных испытаний с одинаковыми входными данными.

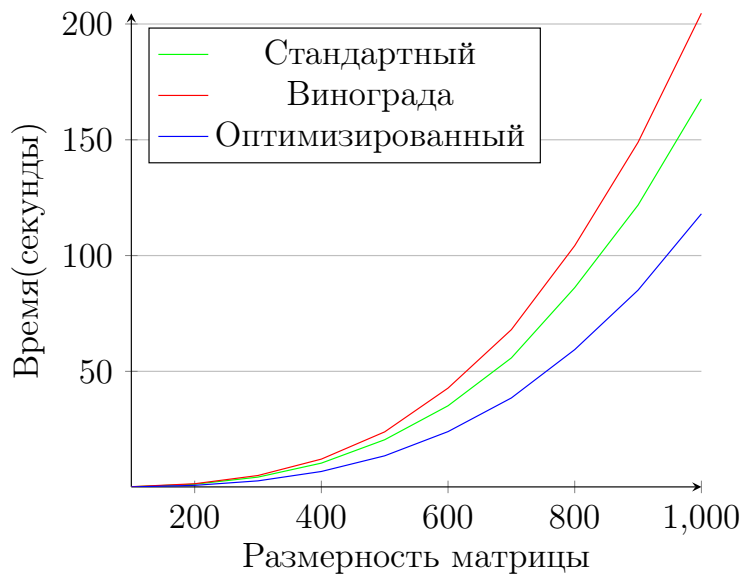


Рис. 4.1: Алгоритмы умножения матриц(четных размеров)

На Рис. 4.1 видно, что оптимизированный алгоритм Винограда превосходит стандартный алгоритм умножения на $\approx 30\%$, а алгоритм Винограда на $\approx 45\%$.

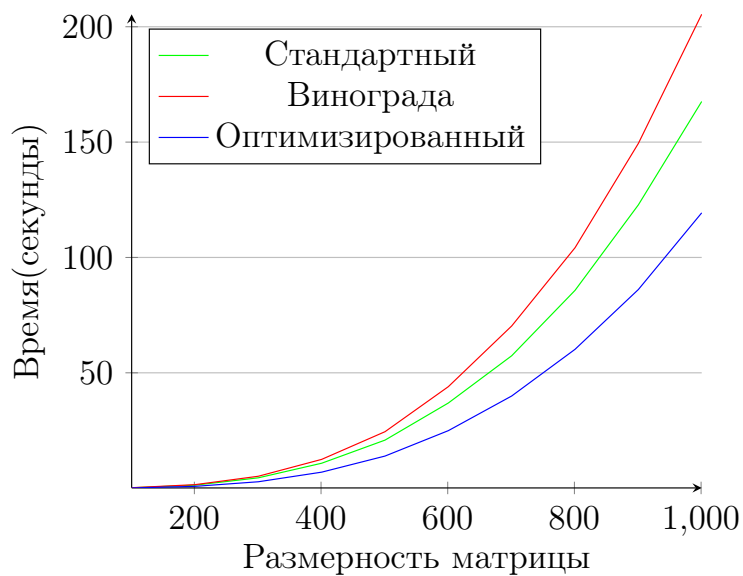


Рис. 4.2: Алгоритмы умножения матриц(нечетных размеров)

На Рис. 4.2 видно, что оптимизированный алгоритм Винограда сохраняет свое превосходство и при нечетных размерах матриц, стандартный алгоритм не изменил своего времени работы, алгоритм Винограда стал работать на пре-небрежимо малое количество времени дольше.

Заключение

В ходе работы были изучены и реализованы алгоритмы стандартного умножения матриц, алгоритма Винограда и оптимизированного алгоритма Винограда. Был проведен сравнительный анализ перечисленных алгоритмов по трудоемкости и экспериментально подтверждено различие во временной эффективности. Классический алгоритм в неоптимизированном виде является более эффективным, чем алгоритм Винограда, однако при ряде оптимизаций алгоритм Винограда становится значительно быстрее классического. При этом разница лучшего и худшего случая алгоритма Винограда пренебрежимо мала.