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

Целью данной лабораторной работы является исследование существующих алгоритмов умножения матриц и их трудоемкости.
Примем следующую модель вычислений:

1. Трудоемкость базовых операций
   Операции $+$, $-$, $*$, $/$, $\%$, $=$, $>$, $<$, $\le$, $\ge$, $==$, $\ne$, $[]$, $+=$, $-=$ - имеют стоимость 1.

2. Трудоемкость условного перехода
   Условный переход имеет стоимость 0, при этом оцениваем расчет условия:

$$
\begin{array}{l}
if\ (n \,\%\, 2\ ==\ 1) \\
\{ \\
\quad //\ \text{Тело 1} \\
\} \\
else \\
\{ \\
\quad //\ \text{Тело 2} \\
\}
\end{array}
$$

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

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 выполнима 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.

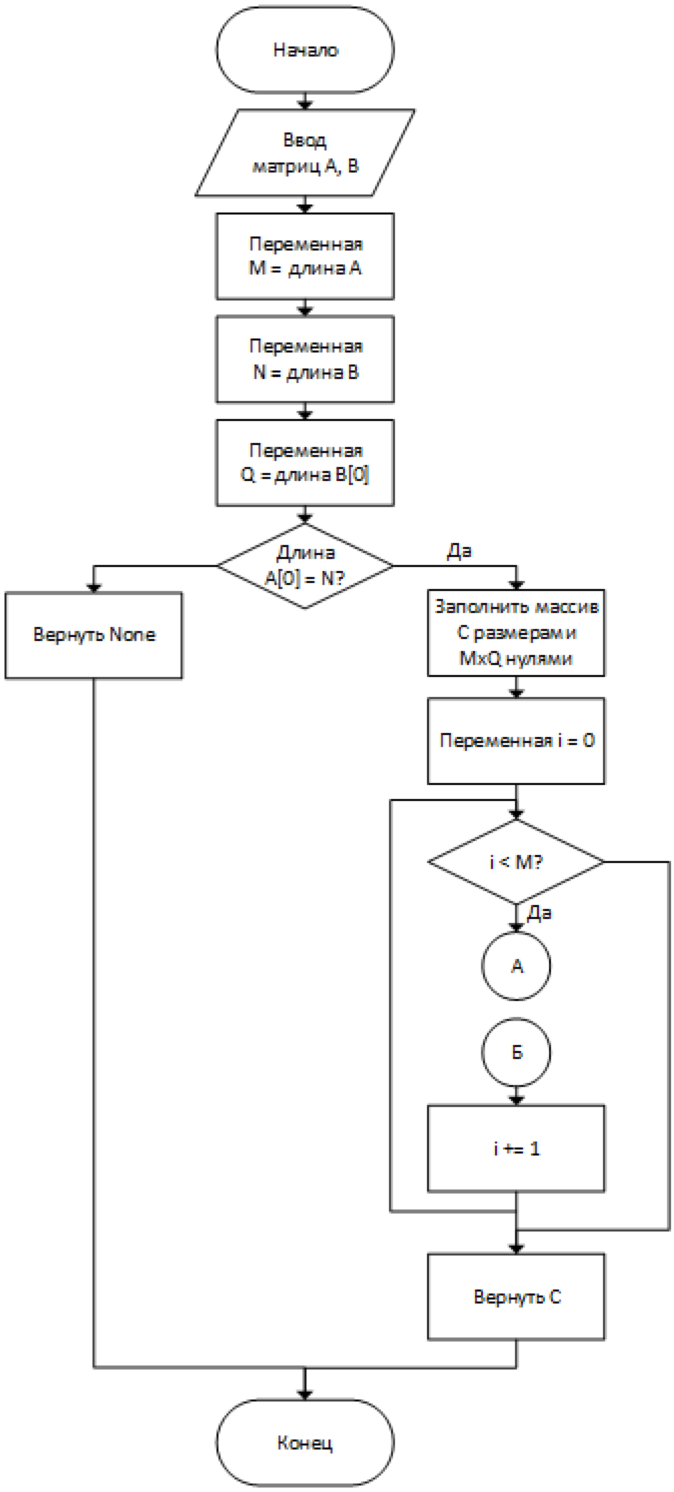

Рис. 2.1: Стандартный алгоритм умножения матриц

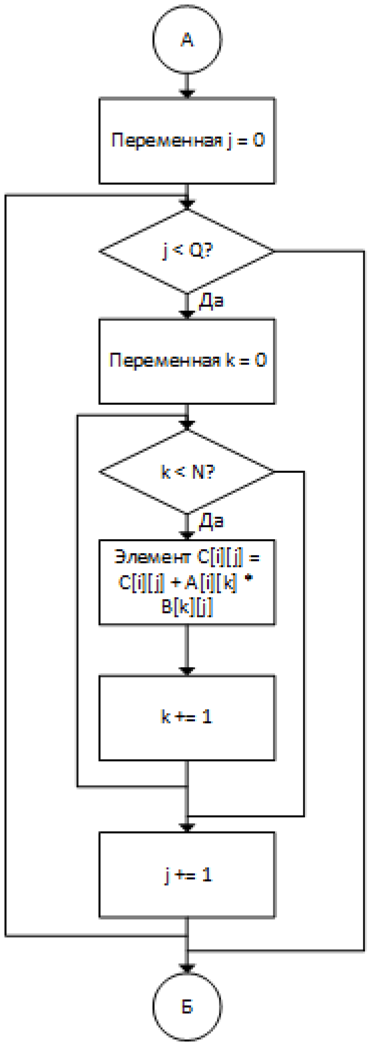

Рис. 2.2: Стандартный алгоритм умножения матриц(продолжение)

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

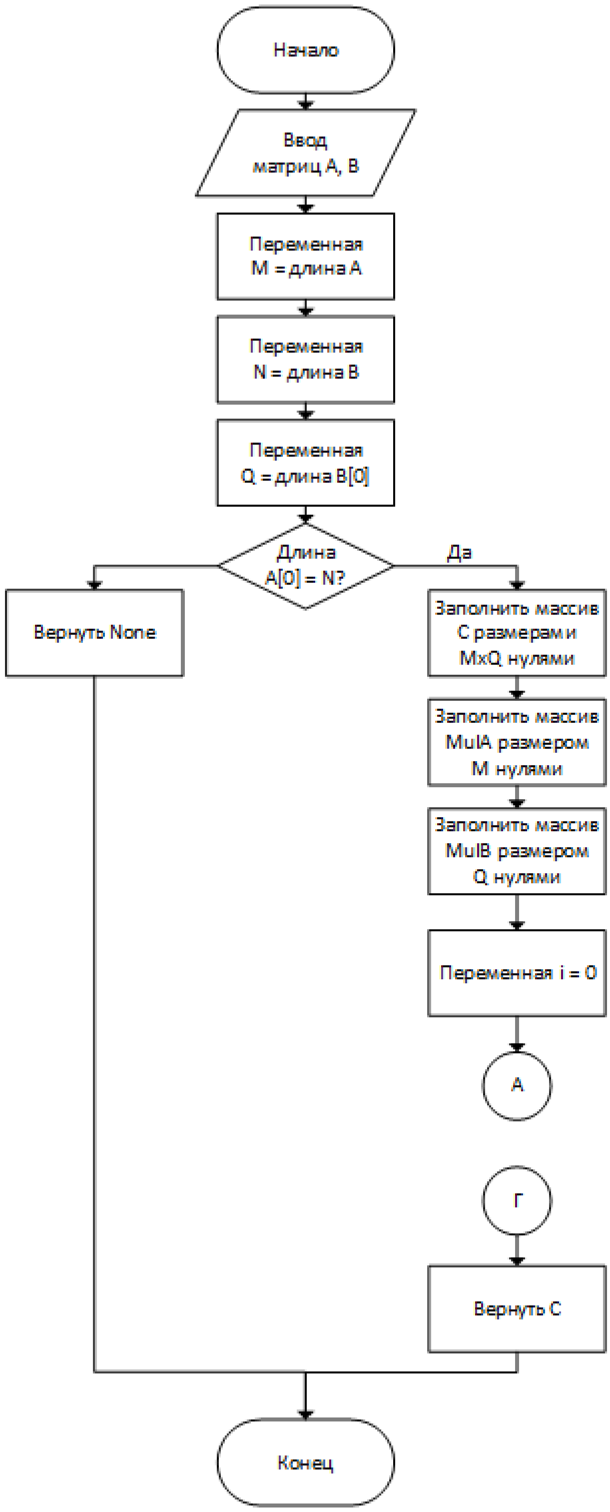

Рис. 2.3: Алгоритм Винограда умножения матриц

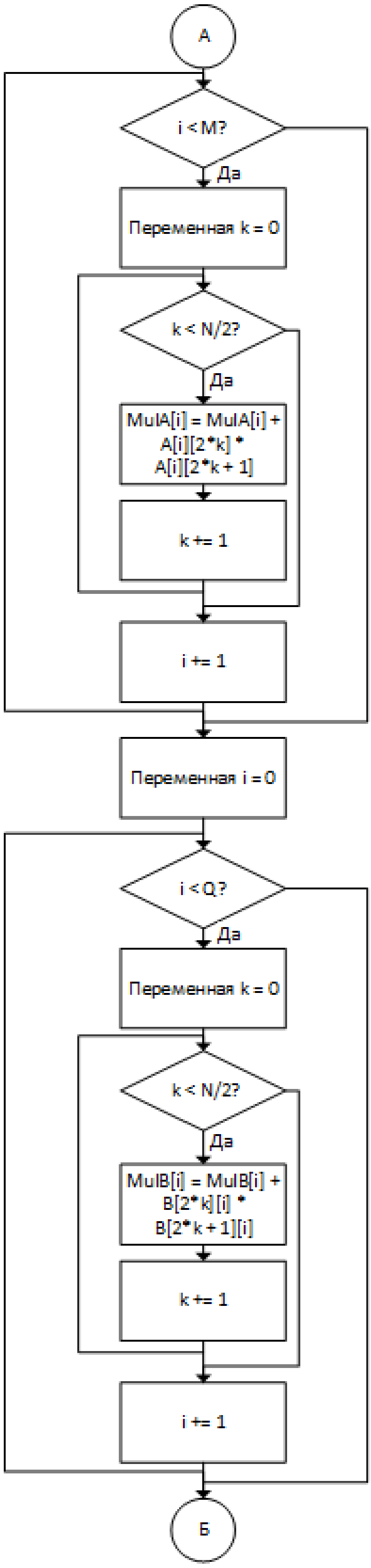

Рис. 2.4: Алгоритм Винограда умножения матриц(продолжение 1)

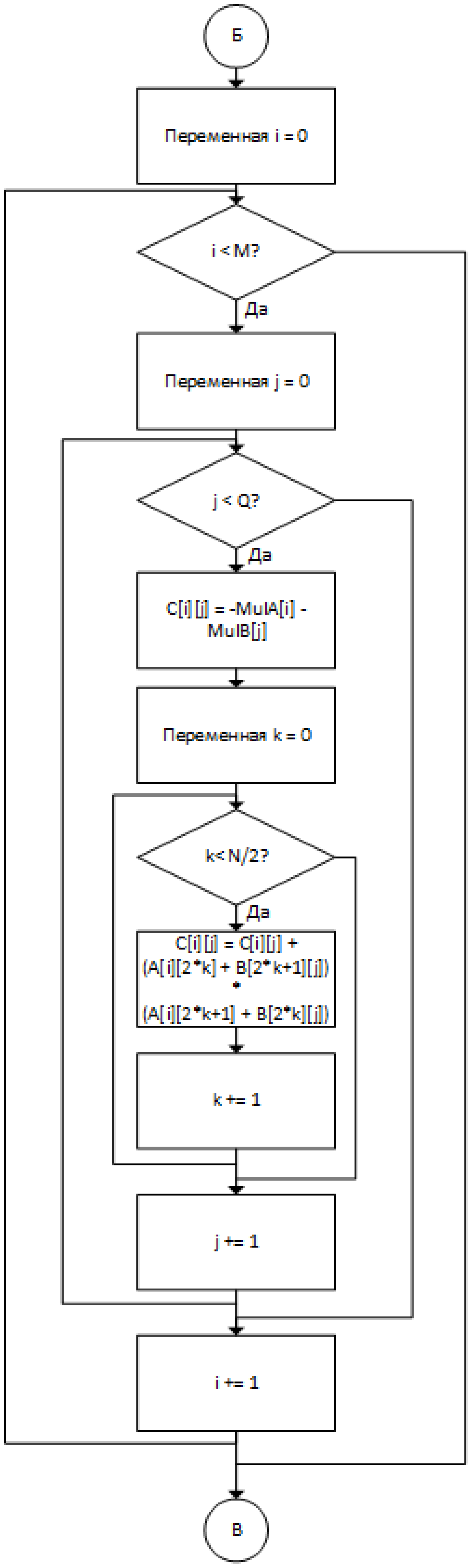

Рис. 2.5: Алгоритм Винограда умножения матриц(продолжение 2)

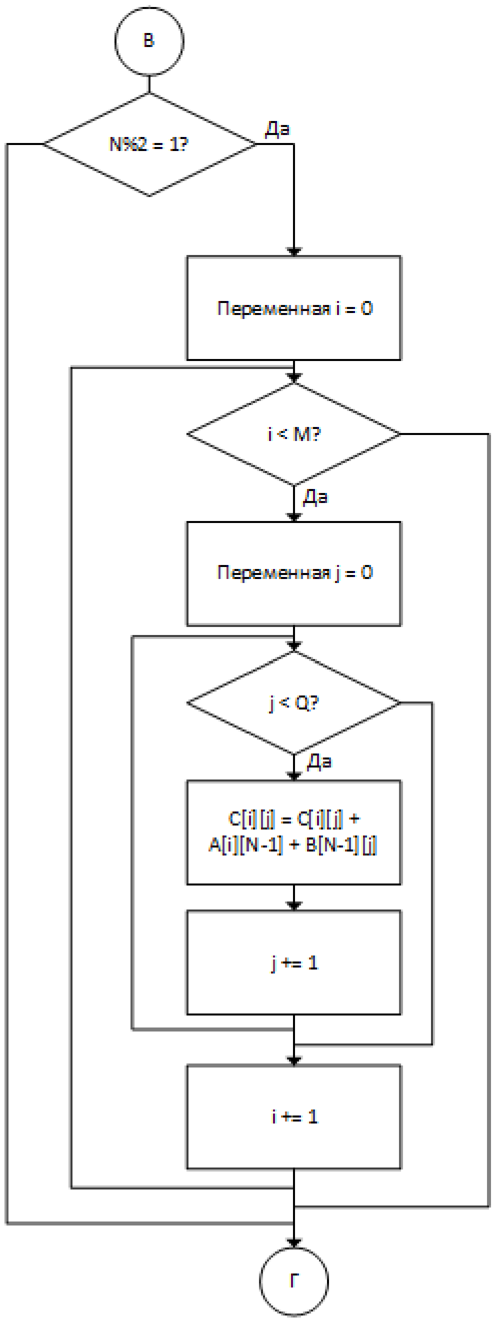

Рис. 2.6: Алгоритм Винограда умножения матриц(продолжение 3)

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

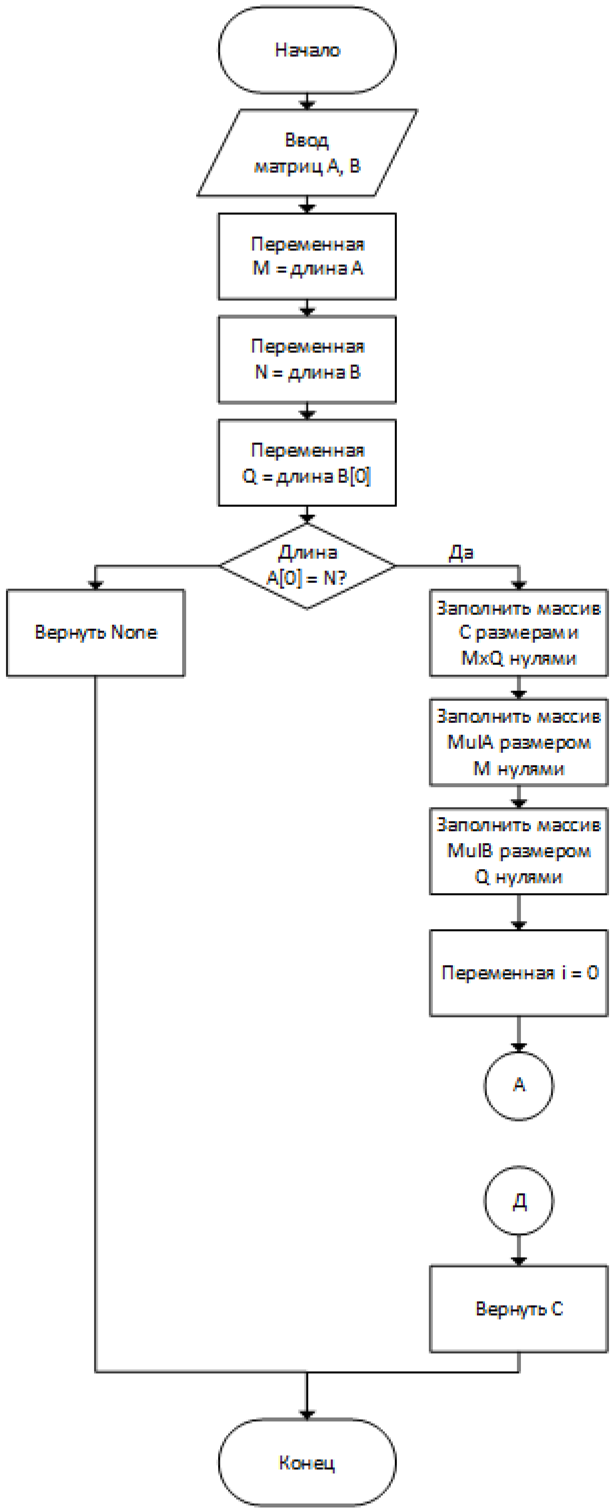

Рис. 2.7: Оптимизированный алгоритм Винограда умножения матриц

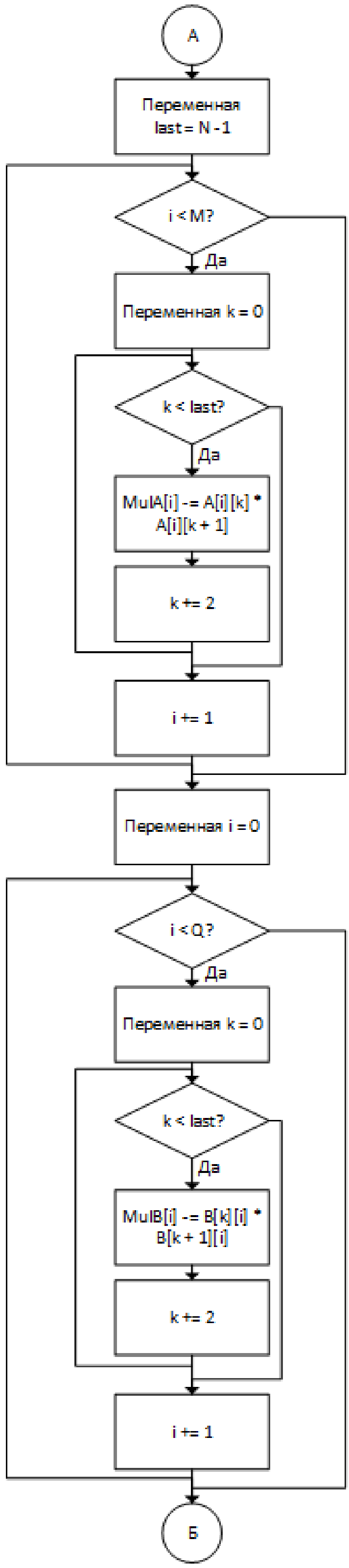

Рис. 2.8: Оптимизированный алгоритм Винограда умножения матриц(продолжение 1)

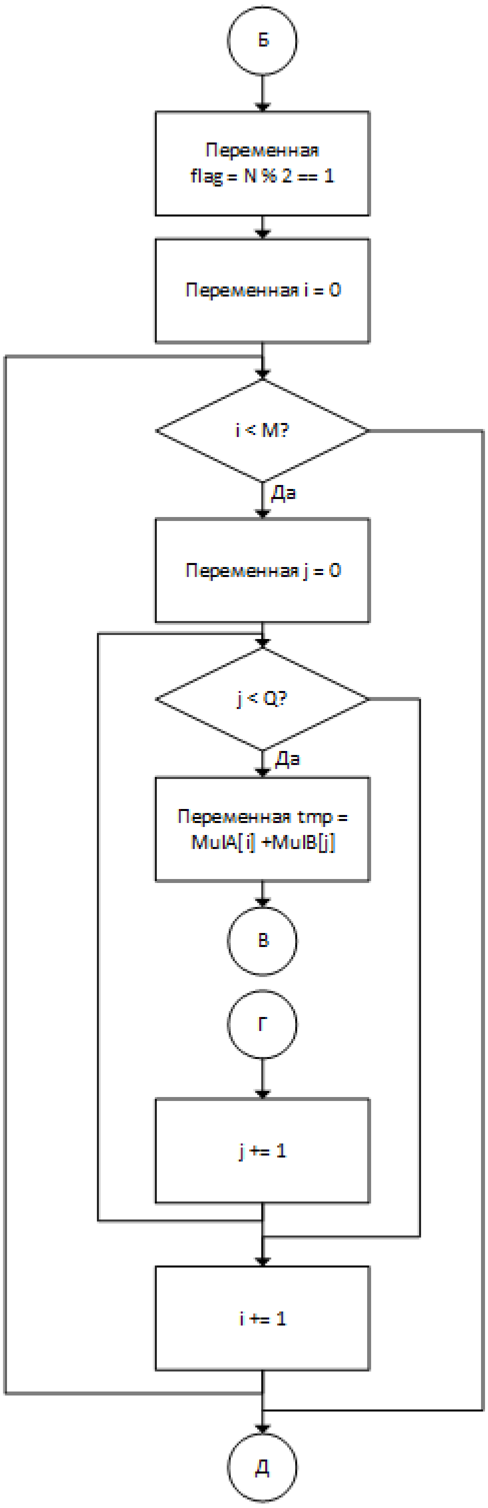

Рис. 2.9: Оптимизированный алгоритм Винограда умножения матриц(продолжение 2)

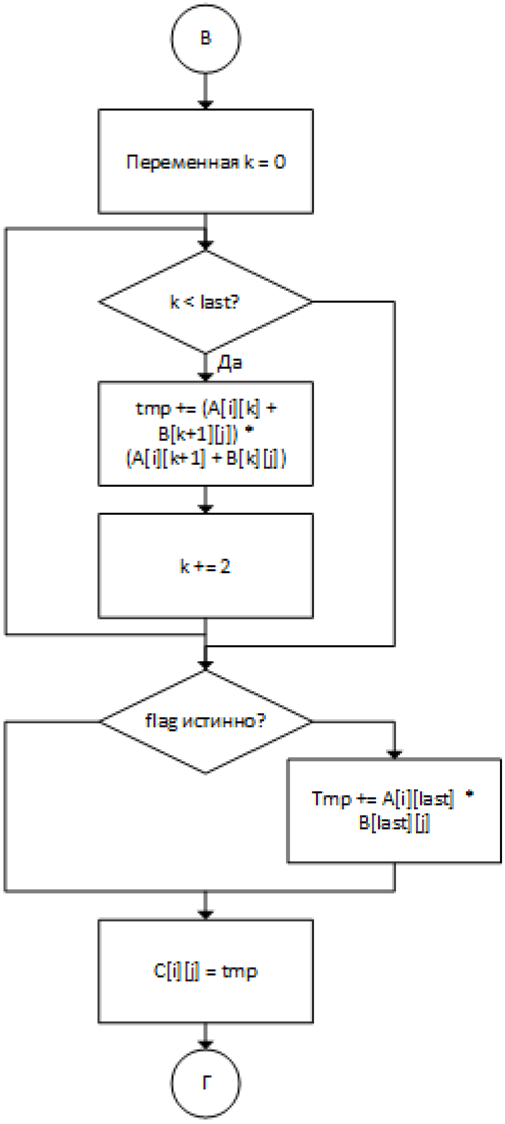

Рис. 2.10: Оптимизированный алгоритм Винограда умножения матриц(продолжение 3)

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

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

- Алгоритм Винограда

$$f_I = 2 + M(2 + 3 + \frac{N}{2}(3 + 1 + 6 + 2 + 3)) = \frac{15}{2}MN + 5M + 2$$
$$f_{II} = 2 + Q(2 + 3 + \frac{N}{2}(3 + 1 + 6 + 2 + 3)) = \frac{15}{2}QN + 5Q + 2$$
$$f_{III} = 2 + M(2 + 2 + Q(2 + 7 + 3 + \frac{N}{2}(3 + 6 + 17))) = 13MNQ + 12MQ + 4M + 2$$
$$f_{IV} = 2 + \left[\begin{array}{cc} 0, & N \text{ четное} \\ 12MQ + 4M + 2, & N \text{ нечетное} \end{array} \right.$$

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

- Оптимизированный алгоритм Винограда

$$f_I^* = 2 + M(2 + 2 + \frac{N}{2}(2 + 1 + 5 + 1 + 1)) = 5MN + 4M + 2$$
$$f_{II}^* = 2 + Q(2 + 2 + \frac{N}{2}(2 + 1 + 5 + 1 + 1)) = 5QN + 4Q + 2$$
$$f_{III}^* = 5 + 2 + M(2 + 2 + Q(2 + 4 + 2 + \frac{N}{2}(2 + 14) + 1 + 3 + \left[\begin{array}{cc} 0, & N \text{ четное} \\ 6, & N \text{ нечетное} \end{array} \right.)) =$$
$$= 7 + 4M + 12MQ + 8MNQ + \left[\begin{array}{cc} 0, & N \text{ четное} \\ 6MQ, & N \text{ нечетное} \end{array} \right.$$

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

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

Трудоемкость удалось снизить за счет следующих оптимизаций:

1. Замена $C[i][j] = C[i][j] + \ldots$ на $C[i][j]{+}{=}\ldots$
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 $C[i][j] = \ldots$ в цикле по k на буферную переменную, что убирает 2 операции индлексирования 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 программнму 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

```
def standard_multiply(a, b):
    m = len(a)
    n = len(b)
    q = len(b[0])
    if len(a[0]) != n:
        return None
    c = [[0 for i in range(q)] for j in range(m)]
    for i in range(m):
        for j in range(q):
            for k in range(n):
                c[i][j] = c[i][j] + a[i][k] * b[k][j]
    return c
```

В Листинге 3.2 показана реализация алгоритма Винограда умножения матриц.

Листинг 3.2: Функция умножения матриц алгоритмом Винограда

```
def vinograd_multiply(a, b):
    m = len(a)
    n = len(b)
    q = len(b[0])
    if len(a[0]) != n:
        return None
    # Part I
    r = n // 2
    mul_a = [0] * m
    for i in range(m):
        for j in range(r):
            mul_a[i] = mul_a[i] + a[i][j * 2] * a[i][j * 2 + 1]

    # Part II
    mul_b = [0] * q
    for i in range(q):
        for j in range(r):
            mul_b[i] = mul_b[i] + b[j * 2][i] * b[j * 2 + 1][i]

    # Part III
    c = [[0 for i in range(q)] for j in range(m)]
    for i in range(m):
        for j in range(q):
            c[i][j] = -mul_a[i] - mul_b[j]
            for k in range(r):
                c[i][j] = c[i][j] + (a[i][2 * k] + b[2 * k + 1][j]) \
                          * (a[i][2 * k + 1] + b[2 * k][j])

    # Part IV
    if n % 2 == 1:
        for i in range(m):
            for j in range(q):
                c[i][j] = c[i][j] + a[i][n - 1] * b[n - 1][j]
    return c
```

В Листинге 3.3 показана реализация оптимизированного алгоритма Винограда умножения матриц.

Листинг 3.3: Функция умножения матриц оптимизированным алгоритмом Винограда

```python
def optimized_vinograd_multiply(a, b):
    m = len(a)
    n = len(b)
    q = len(b[0])
    if len(a[0]) != n:
        return None
    last = n - 1 # Optimization for odd numbers
    # Part I
    mul_a = [0] * m
    for i in range(m):
        for j in range(0, last, 2): # Optimization for n // 2
            mul_a[i] -= a[i][j] * a[i][j + 1] # Optimization for negative and +=

    # Part II
    mul_b = [0] * q
    for i in range(q):
        for j in range(0, last, 2): # Optimization for n // 2
            mul_b[i] -= b[j][i] * b[j + 1][i] # Optimization for negative and +=

    flag = n % 2 == 1
    # Part III
    c = [[0 for i in range(q)] for j in range(m)]
    for i in range(m):
        for j in range(q):
            tmp = mul_a[i] + mul_b[j] # Optimization for buffer
            for k in range(0, last, 2): # Optimization for n // 2
                tmp += (a[i][k] + b[k + 1][j]) \
                      * (a[i][k + 1] + b[k][j]) # Optimization for +=
            if flag: # Optimization for odd numbers
                tmp += a[i][last] * b[last][j]
            c[i][j] = tmp

    return c
```

4 Эксперементальная часть

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

4.1 Примеры работы

Пример 1
Строка s1 = кот
Строка s2 = скат
Расстояние Левенштейна(матричный алгоритм): 2
0 1 2 3
1 1 2 3
2 1 2 3
3 2 2 3
4 3 3 2
Расстояние Дамерау-Левенштейна(матричный алгоритм):2
0 1 2 3
1 1 2 3
2 1 2 3
3 2 2 3
4 3 3 2
Расстояние Дамерау-Левенштейна(рекурсивный алгоритм):2

Пример 2
Строка s1 = отчет
Строка s2 = спать
Расстояние Левенштейна(матричный алгоритм): 5
0 1 2 3 4 5
1 1 2 3 4 5
2 2 2 3 4 5
3 3 3 3 4 5
4 4 3 4 4 4
5 5 4 4 5 5
Расстояние Дамерау-Левенштейна(матричный алгоритм):5
0 1 2 3 4 5
1 1 2 3 4 5
2 2 2 3 4 5

3 3 3 3 4 5
4 4 3 4 4 4
5 5 4 4 5 5
Расстояние Дамерау-Левенштейна(рекурсивный алгоритм):5

Пример 3
Строка s1 = кофе
Строка s2 = коеф
Расстояние Левенштейна(матричный алгоритм): 2
0 1 2 3 4
1 0 1 2 3
2 1 0 1 2
3 2 1 1 1
4 3 2 1 2
Расстояние Дамерау-Левенштейна(матричный алгоритм):1
0 1 2 3 4
1 0 1 2 3
2 1 0 1 2
3 2 1 1 1
4 3 2 1 1
Расстояние Дамерау-Левенштейна(рекурсивный алгоритм):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 - строки, поступающие на вход; 4 столбец - ожидаемый результат, где

- 1-ая цифра - результат работы матричного алгоритма Левенштейна
- 2-ая цифра - результат работы матричного алгоритма Дамерау-Левенштейна
- 3-я цифра - результат работы рекурсивного алгоритма Дамерау-Левенштейна

5 столбец - полученный результат.

№	S1	S2	Ожидаемый результат	Полученный результат
1	пустая строка	пустая строка	0, 0, 0	0, 0, 0
2	кот	пустая строка	3, 3, 3	3, 3, 3
3	пустая строка	кот	3, 3, 3	3, 3, 3
4	кот	кот	0, 0, 0	0, 0, 0
5	кот	кит	1, 1, 1	1, 1, 1
6	от	кот	1, 1, 1	1, 1, 1
7	кот	кота	1, 1, 1	1, 1, 1
7	кот	кто	2, 1, 1	2, 1, 1

Таблица 4.1: Тестовые случаи

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

4.3 Сравение 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.

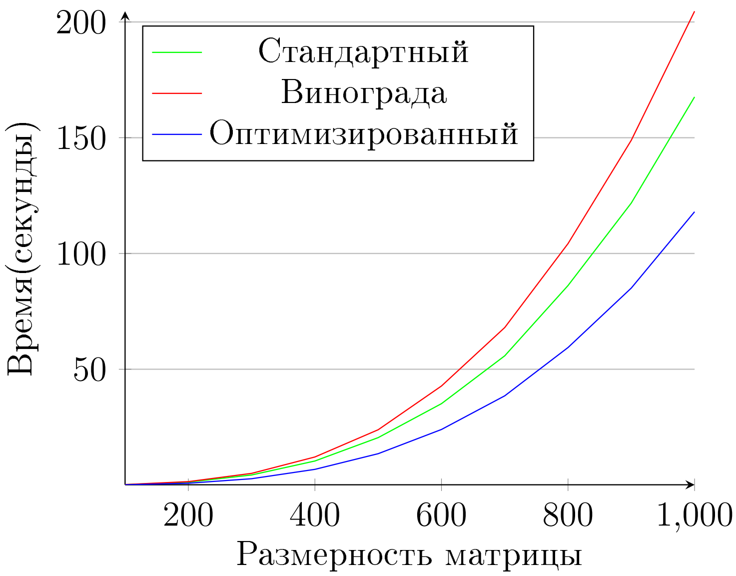

Рис. 4.1: Алгоритмы умножения матриц(четных размеров)

На Рис. 4.1 видно, что матричный алгоритм Левенштейна превосходит матричный алгоритм Дамерау-Левенштейна примерно на 20%. Эта разница объясняется необходимостью дополнительных проверок на возможность транспозиции.

Заключение

В ходе работы были достигнуты все поставленные задачи. Были изучены и реализованы в матричной и рекурсивной форме алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками. Также был проведен сравнительный анализ матричной и рекурсивной реализаций алгоритма Левенштейна и матричных реализаций алгоритмов Левенштейна и Дамерау-Левенштейна по затрачиваемым ресурсам. Экспериментально подтверждено различие во временной эффективности рекурсивной и матричной реализаций путем замеров времени работы алгоритмов.