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» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

О Т Ч Е Т

по лабораторной работе № 2

Название: Анализ алгоритмов умножения матриц

Дисциплина: Анализ алгоритмов

Студент	ИУ7-52Б		Сучков А.Д.
	(Группа)	(Подпись, дата)	(И.О. Фамилия)
Преподаватель			Волкова Л.Л.
		(Подпись, дата)	(И.О. Фамилия)

Москва, 2020

Оглавление

Введение

Умножение матриц - это один из базовых алгоритмов, который широко применяется в различных численных методах, и в частности в алгоритмах машинного обучения. Многие реализации прямого и обратного распространения сигнала в сверточных слоях неронной сети базируются на этой операции. Для перемножения двух матриц необходимо, чтобы количество столбцов в первой матрице совпадало с количеством строк во второй. У результирующей матрицы будет столько же строк, сколько в первой матрице, и столько же столбцов, сколько во второй.

Сложность вычисления произведения матриц по определению составляет $O(n^3)$, однако существуют более эффективные алгоритмы, которые применяются для больших матриц. Вопрос о предельной скорости умножения больших матриц, также как и вопрос о построении наиболее быстрых и устойчивых практических алгоритмов умножения больших матриц остаётся одной из нерешённых проблем линейной алгебры.

1. Аналитическая часть

Цель данной лабораторной работы заключается в изучении алгоритмов умножения матриц. Рассматриваются стандартный алгоритм умножения матриц, а также алгоритм Винограда и модифицированный алгоритм Винограда. Требуется рассчитать и изучить затрачиваемое каждым алгоритмом время.

В данной лабораторной работе выделено несколько задач:

- изучить алгоритмы умножения матриц: стандартный и алгоритм Винограда;
- модифицировать алгоритм Винограда;
- дать теоретическую оценку базового алгоритма умножения матриц, алгоритму Винограда и модифированному алгоритму Винограда;
- реализовать три алгоритма умножения матриц на одном из языков программирования;
- сравнить алгоритмы умножения матриц.

1.1. Стандартный алгоритм умножения матриц

Пусть даны две матрицы A и B с размерностями $m \times n$ и $n \times l$ соответственно (1.1) и (1.2):

$$\begin{bmatrix} a_{1,1} & ... & a_{1,n} \\ ... & ... & ... \\ a_{m,1} & ... & a_{m,n} \end{bmatrix} \tag{1.1}$$

$$\begin{bmatrix} b_{1,1} & ... & b_{1,l} \\ ... & ... & ... \\ b_{n,1} & ... & b_{n,l} \end{bmatrix} \tag{1.2}$$

В результате умножения, получим матрицу C размерностью $m \times l$ (1.3):

$$\begin{bmatrix} c_{1,1} & ... & c_{1,l} \\ ... & ... & ... \\ c_{m,1} & ... & c_{m,l} \end{bmatrix} \tag{1.3}$$

$c_{i,j} = \sum\limits_{r=1}^{n} a_{i,r} \cdot b_{r,j}$ называется произведением матриц A и B.

1.2. Алгоритм Винограда

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 $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение (1.4).

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

Это равенство можно переписать в виде ()1.5)

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

Кажется, что формула 1.5 задает больше работы, чем первое: вместо четырех умножений мы насчитываем их шесть, а вместо трех сложений - десять. Менее очевидно, что 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.

1.3. Вывод

Алгоритм Винограда предлагает подсчитывать значения заранее перед основными вычислениями, что может повысить производительность при перемножении достаточно больших матриц. Однако с малыми размерами, он может справляться хуже чем другие, но алгоритм можно оптимизировать и добиться более высокой скорости подсчёта.

2. Конструкторская часть

2.1. Требования к программе

Для дальнейшего тестирования программы необходимо обеспечить консольный ввод размерностей двух матриц и их содержимого, а также обеспечить выбор алгоритма поиска. На выходе должны получить результирующую матрицу. Также необходимо реализовать функцию подсчёта процессорного времени, которое могут затрачивать функции.

2.2. Схемы алгоритмов

На рисунках 2.1 - 2.5 приведены схемы алгоритмов умножения матриц.

2.3. Подсчёт трудоёмкости алгоритмов

Для начала оценки алгоритмов, можно ввести специальную модель трудоёмкости:

- стоимость базовых операций 1 — +, -, *, /, =, == ...;
- оценка цикла — $f_{for} = f_{init} + N \cdot (f + f_{body} + f_{post}) + f$, где f - условие цикла, f_{init} - предусловие цикла, f_{post} - постусловие цикла;
- стоимость условного перехода примем за 0, стоимость вычисления условия остаётся.

Для стандартного алгоритма умножения с матрицами A и B и размерами $n \times m$ и $m \times l$ соответственно:

$$f = 2 + n \cdot (2 + 2 + l \cdot (2 + 2 + m \cdot (2 + 6 + 2))) = 10nlm + 4ln + 4n + 2$$

Для алгоритма Винограда при тех же матрицах и их размерах.

Для более понятного подсчёта, можно составить таблицу (таблица 2.1), а затем подсчитать общую трудоёмкость:

$$f = 13mnl + 7.5mn + 7.5lm + 11ln + 8n + 4l + 14 + \begin{cases} 0, \text{если m чётное} \\ 15 \cdot l \cdot n + 4 \cdot n + 2, \text{иначе} \end{cases}$$

Таблица 2.1: трудоёмкость алгоритма Винограда

Часть алгоритма	Трудоёмкость
Инициализация mulH и mulV	$2 \cdot 3$
Заполнение mulH	$2 + n \cdot (2 + 2 + m/2 \cdot (3 + 6 + 6))$
Заполнение mulV	$2 + l \cdot (2 + 2 + m/2 \cdot (3 + 6 + 6))$
Подсчёт результата	$2 + n \cdot (2 + 2 + l \cdot (2 + 7 + 2 + m/2 \cdot (3 + 23)))$
Условный оператор нечёт. m	2
Для матриц с нечёт m	$2 + n \cdot (2 + 2 + l \cdot (2 + 8 + 5))$

Для оптимизированного алгоритма Винограда при тех же матрицах и размерах. Для более понятного подсчёта, можно тоже составить таблицу (таблица 2.2).

$$f = 8mnl + 5mn + 5lm + 12ln + 8n + 4l + 18 + \begin{cases} 0, \text{если m чётное} \\ 10 \cdot l \cdot n + 4 \cdot n + 4, \text{иначе} \end{cases}$$

Таблица 2.2: трудоёмкость оптимизированного алгоритма Винограда

Часть алгоритма	Трудоёмкость
Инициализация mulH и mulV	$2 \cdot 3$
Инициализация m1Mod2 и n2Mod2	$2 \cdot 2$
Заполнение mulH	$2 + n \cdot (2 + 2 + m/2 \cdot (2 + 5 + 3))$
Заполнение mulV	$2 + l \cdot (2 + 2 + m/2 \cdot (2 + 5 + 3))$
Подсчёт результата	$2 + n \cdot (2 + 2 + l \cdot (2 + 5 + 3 + 2+$ $+m/2 \cdot (2 + 14)))$
Условный оператор нечёт. m	2
Для матриц с нечёт m	$2 + 2 + n \cdot (2 + 2 + l \cdot (2 + 6 + 2))$

2.4. Вывод

Были составлены схемы и подсчитана трудоёмкость для каждого алгоритма. Из последнего можно увидеть, что оптимизированный алгоритм Винограда менее трудоёмкий, чем неоптимизированный.

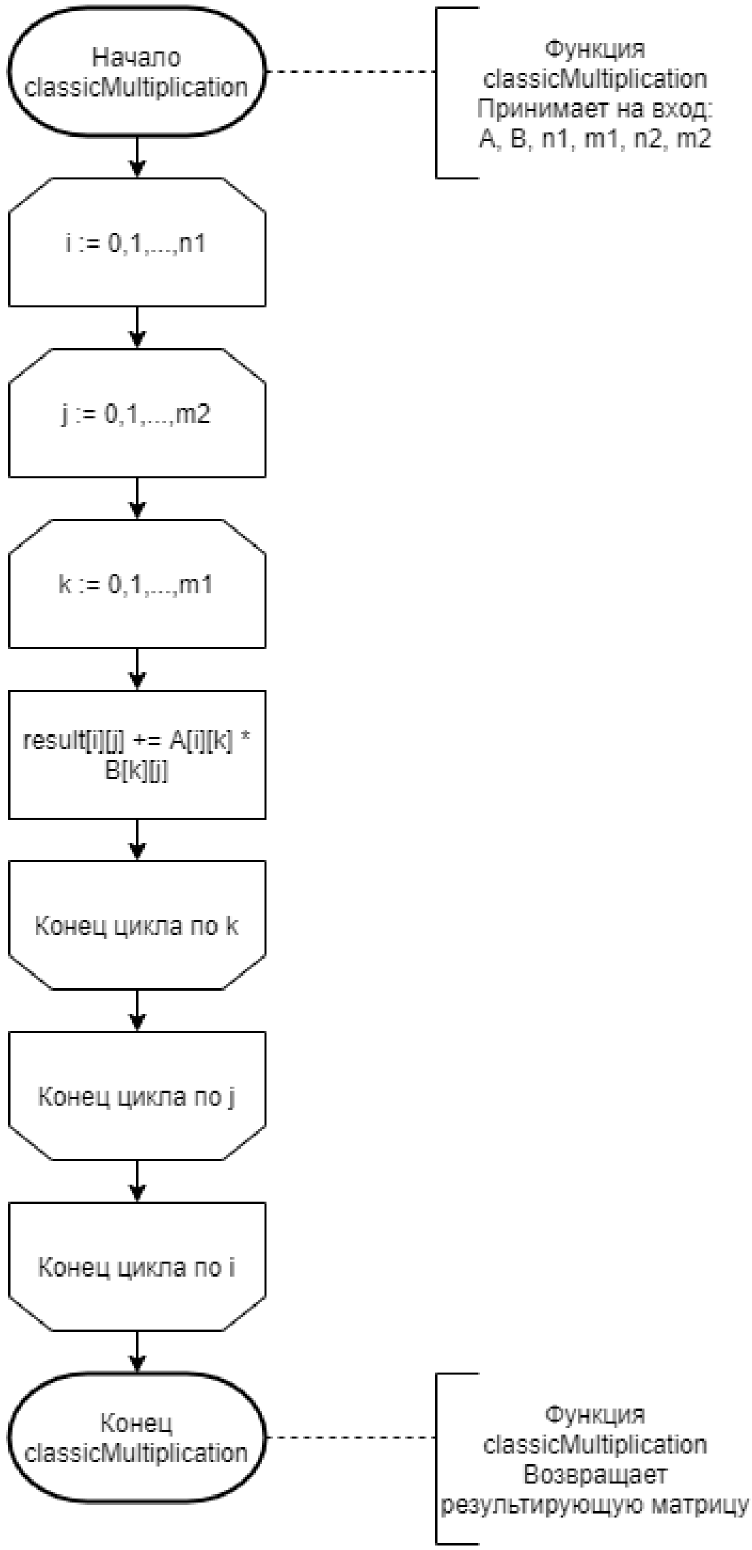

Рис. 2.1: Схема стандартного алгоритма умножения матриц

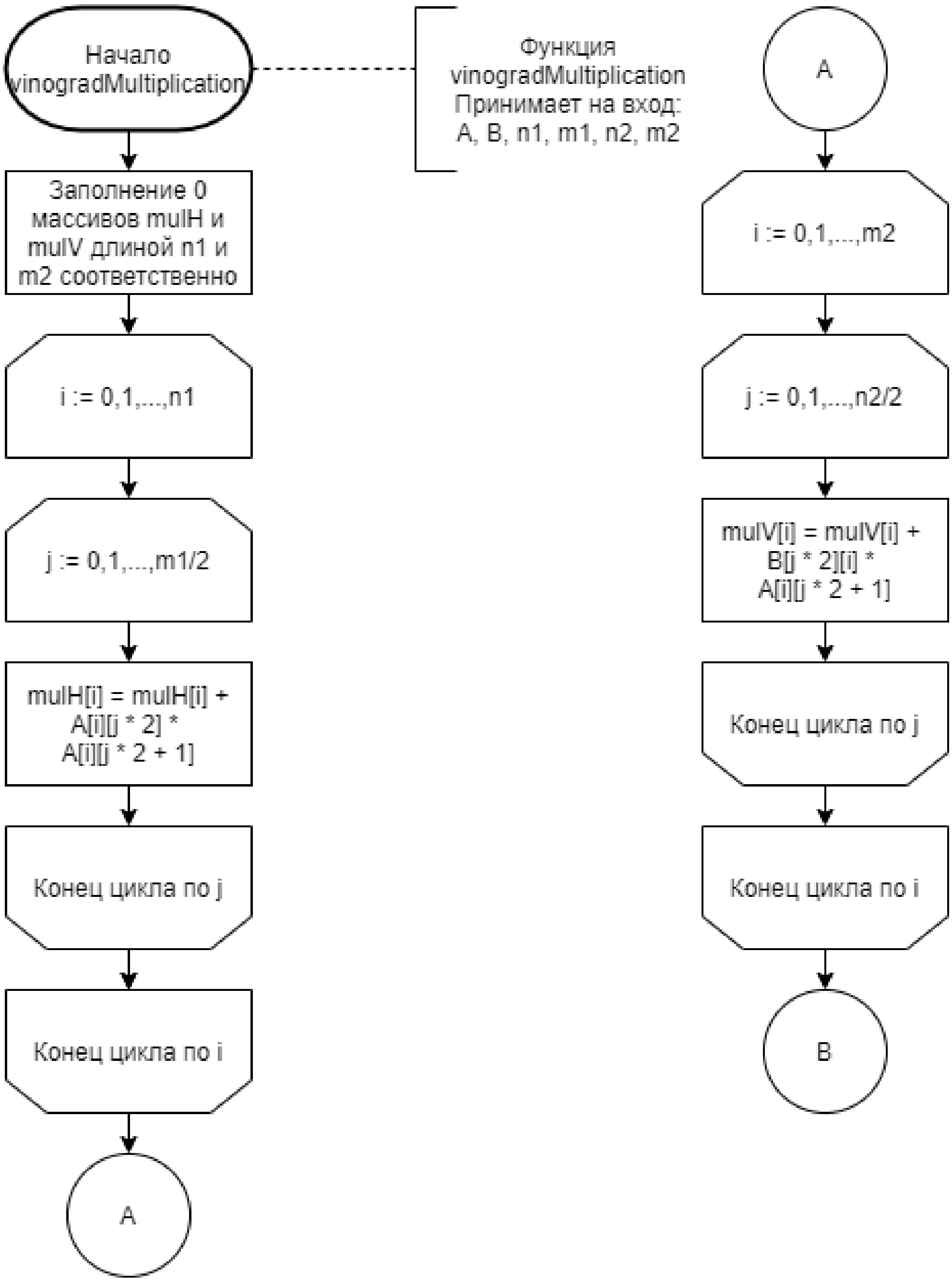

Рис. 2.2: Схема алгоритма Винограда умножения матриц, часть 1

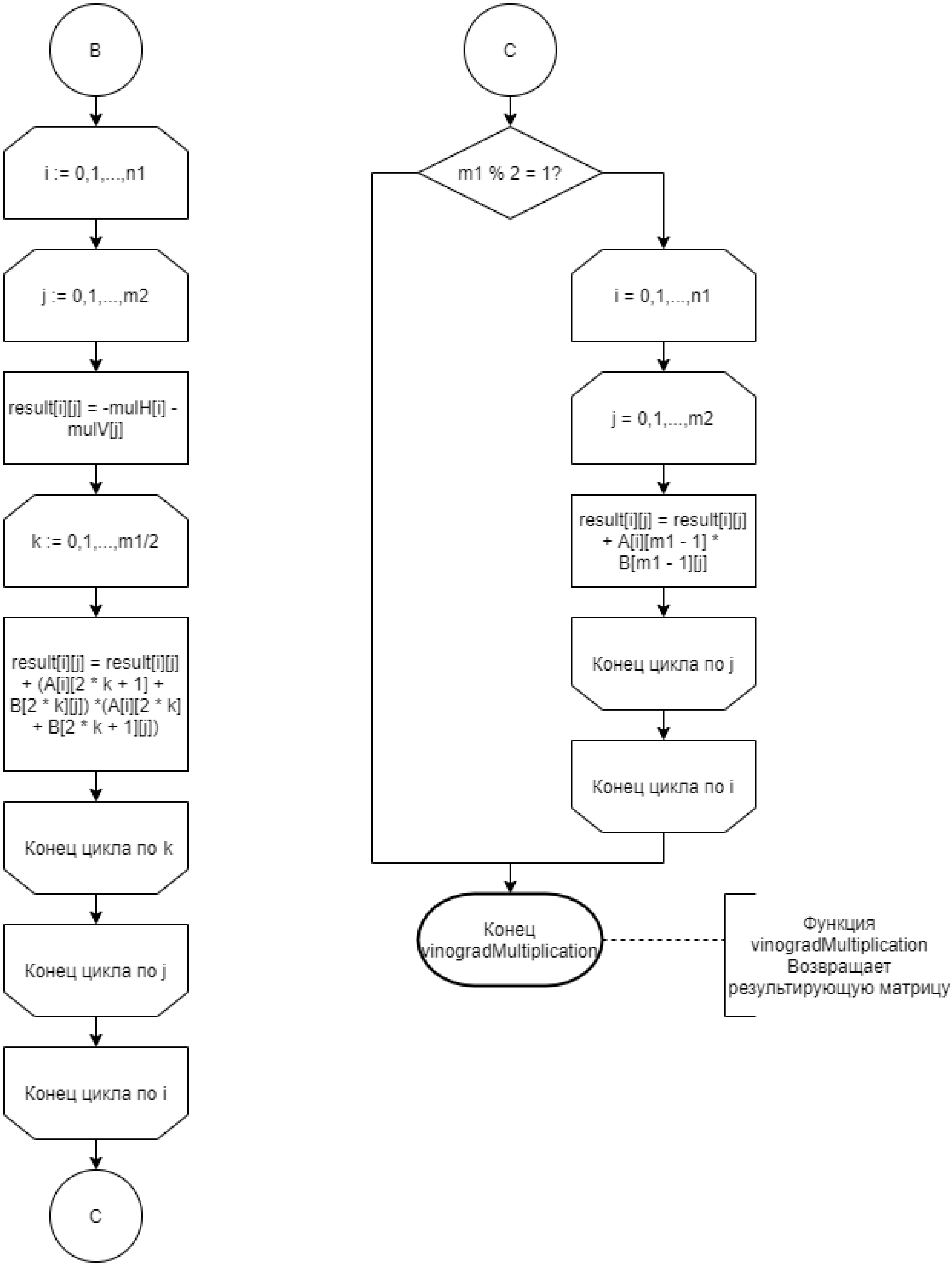

Рис. 2.3: Схема алгоритма Винограда умножения матриц, часть 2

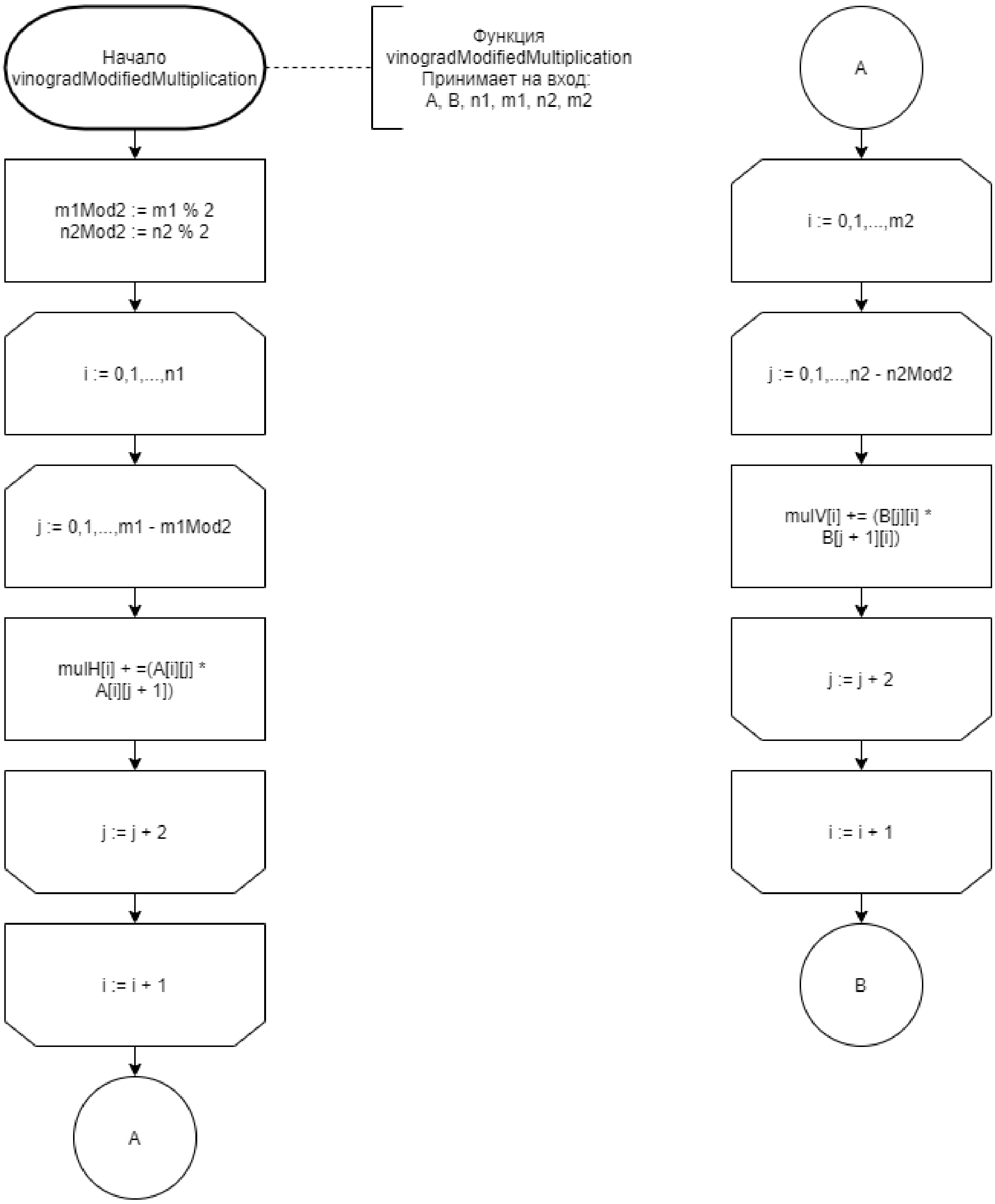

Рис. 2.4: Схема модифицированного алгоритма Винограда умножения матриц, часть 1

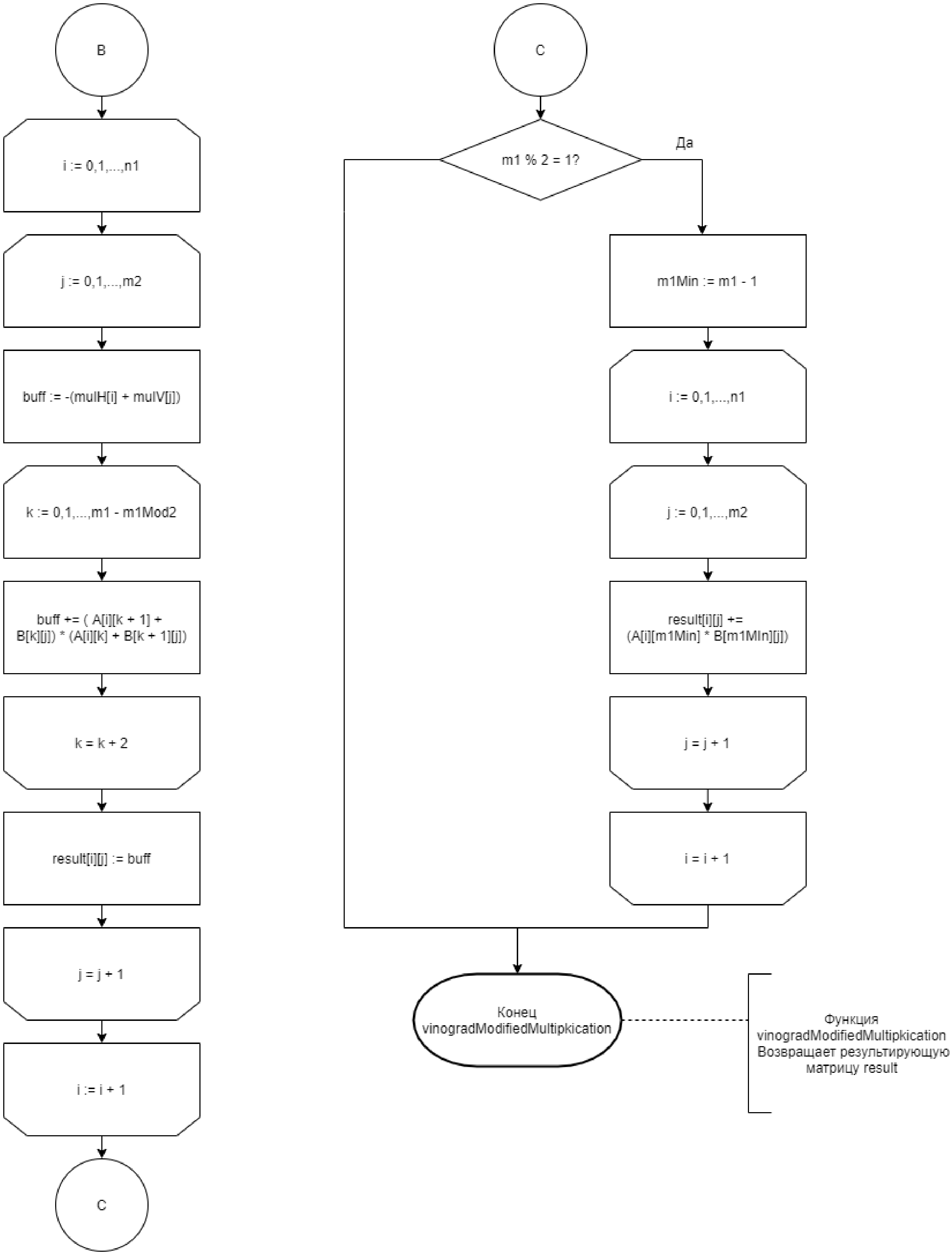

Рис. 2.5: Схема модифицированного алгоритма Винограда умножения матриц, часть 2

3. Технологическая часть

3.1. Выбор языка программирования

В качестве языка программирования было решено выбрать Python 3, так как уже имеется опыт работы с библиотеками и инструмантами языка, которые позволяют реализовать и провести исследования над умножением матриц.

3.2. Реализации алгоритмов

В листингах 3.1 - 3.3 приведены реализации алгоритмов умножения на Python.

Листинг 3.1: классический алгоритм умножения матриц

```
def classicMultiplication(A, B, isPrint):
    result = [[0 for j in range(len(A))]
                for i in range(len(B[0]))]

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

    if (isPrint):
        print("\n>>> Result with classic method:")
        printMatrix(result)
```

Листинг 3.2: алгоритм умножения матриц Винограда

```
def vinogradMultiplication(A, B, isPrint):
    n1, m1 = len(A), len(A[0])
    n2, m2 = len(B), len(B[0])

    mulH = [0 for _ in range(n1)]
    mulV = [0 for _ in range(m2)]

    result = [[0 for j in range(n1)] for i in range(m2)]
```

```

    for i in range(n1):
        for j in range(int(m1 / 2)):
            mulH[i] = mulH[i] + A[i][j * 2] *
                                A[i][j * 2 + 1]

    for i in range(m2):
        for j in range(int(n2 / 2)):
            mulV[i] = mulV[i] + B[j * 2][i] *
                                B[j * 2 + 1][i]

    for i in range(n1):
        for j in range(m2):
            result[i][j] = -mulH[i] - mulV[j]

            for k in range(int(m1 / 2)):
                result[i][j] = result[i][j] +
                    ((A[i][2 * k + 1] + B[2 * k][j]) *
                     (A[i][2 * k] + B[2 * k + 1][j]))

    if (m1 % 2):
        for i in range(n1):
            for j in range(m2):
                result[i][j] = result[i][j] +
                (A[i][m1 - 1] * B[m1 - 1][j])

    if (isPrint):
        print("\n>>> Result with vinograd method:")
        printMatrix(result)
```

Листинг 3.3: оптимизированный алгоритм Винограда

```
def vinogradMultiplicationModified(A, B, isPrint):
    n1, m1 = len(A), len(A[0])
    n2, m2 = len(B), len(B[0])

```

```
    mulH = [0 for _ in range(n1)]
    mulV = [0 for _ in range(m2)]

    result = [[0 for j in range(n1)]
                 for i in range(m2)]

    m1Mod2 = m1 % 2
    n2Mod2 = n2 % 2

    for i in range(n1):
        for j in range(0, m1 - m1Mod2, 2):
            mulH[i] += A[i][j] * A[i][j + 1]

    for i in range(m2):
        for j in range(0, n2 - n2Mod2, 2):
            mulV[i] += B[j][i] * B[j + 1][i]

    for i in range(n1):
        for j in range(m2):
            buff = -(mulH[i] + mulV[j])

            for k in range(0, m1 - m1Mod2, 2):
                buff += ((A[i][k + 1] + B[k][j]) *
                         (A[i][k] + B[k + 1][j]))
            result[i][j] = buff

    if m1Mod2:
        m1Min = m1 - 1
        for i in range(n1):
            for j in range(m2):
                result[i][j] += A[i][m1Min] * B[m1Min][j]
    if isPrint:
        print("\nResult with modified vinograd method:")
        printMatrix(result)
```

3.3. Оптимизация алгоритма Винограда

Для оптимизации алгоритма были внесены некоторые изменения.

Заранее высчитываются значения, для избавления от деления в цикле. См. листинг 3.4.

Листинг 3.4: избавление от деления в цикле

```
def vinogradMultiplicationModified(A, B, isPrint):
    ...

    m1Mod2 = m1 % 2
    n2Mod2 = n2 % 2

    for i in range(n1):
        for j in range(0, m1 - m1Mod2, 2):
            mulH[i] += A[i][j] * A[i][j + 1]

    for i in range(m2):
        for j in range(0, n2 - n2Mod2, 2):
            mulV[i] += B[j][i] * B[j + 1][i]

    ...
```

Заменяется обычное сложение (mulV[i] = mulV[i] + ...) на более оптимизированный вариант (mulV[i] += ..., аналогично и в других подобных местах). См. листинг 3.5.

Листинг 3.5: оптимизация сложения

```
def vinogradMultiplicationModified(A, B, isPrint):
    ...

    for i in range(n1):
        for j in range(0, m1 - m1Mod2, 2):
            mulH[i] += A[i][j] * A[i][j + 1]


```

```
    for i in range(m2):
        for j in range(0, n2 - n2Mod2, 2):
            mulV[i] += B[j][i] * B[j + 1][i]

    for i in range(n1):
        for j in range(m2):
            buff = -(mulH[i] + mulV[j])

            for k in range(0, m1 - m1Mod2, 2):
                buff += ((A[i][k + 1] + B[k][j]) *
                         (A[i][k] + B[k + 1][j]))

            result[i][j] = buff

    if m1Mod2:
        m1Min = m1 - 1

        for i in range(n1):
            for j in range(m2):
                result[i][j] += A[i][m1Min] * B[m1Min][j]

    ...
```

Инициализируется специальный буфер для накопления результата умножения, который позже сбрасывается в ячейку матрицы. См. листинг 3.6.

Листинг 3.6: оптимизация, при помощи буфера

```
def vinogradMultiplicationModified(A, B, isPrint):
    ...

    for i in range(n1):
        for j in range(m2):
            buff = -(mulH[i] + mulV[j])

            for k in range(0, m1 - m1Mod2, 2):
                buff += ((A[i][k + 1] + B[k][j]) *
```

```
                          (A[i][k] + B[k + 1][j]))

            result[i][j] = buff

    ...
```

3.4. Оценка затрачиваемого времени

Для замера процессорного времени выполнения алгоритмов используется библиотека time [2]. В листинге 3.7 приведена функция наполнения случайными числами матрицы с заданным размером. В листинге 3.8 приведены функции с помощью которых производятся замеры времени.

Листинг 3.7: наполнение матрицы случайными числами

```
def generateMatrix(size):
    return [[random.randint(0, 9) for _ in range(size)]
                                  for _ in range(size)]
```

Листинг 3.8: функции для замера времени

```
def doTimeTest(method, A, B):
    t1 = process_time()
    method(A, B, False)
    t2 = process_time()

    return t2 - t1

def multiplicationTimeTest():
    sizesMod2 = [100, 150, 200]
    sizesNotMod2 = [101, 151, 201]

    for size in sizesMod2:
        A = generateMatrix(size)
        B = generateMatrix(size)

        print(">>> For classic with      len = ", size,
```

```
        "Time:", doTimeTest(classicMultiplication, A, B))

        print(">>> For Vinograd with      len = ", size,
        "Time:", doTimeTest(vinogradMultiplication, A, B))

        print(">>> For Vinograd mod. with len = ", size,
        "Time:", doTimeTest(vinogradMultiplicationModified,
                            A, B))

    for size in sizesNotMod2:
        A = generateMatrix(size)
        B = generateMatrix(size)

        print(">>> For classic with       len = ", size,
        "Time:", doTimeTest(classicMultiplication, A, B))

        print(">>> For Vinograd with      len = ", size,
        "Time:", doTimeTest(vinogradMultiplication, A, B))

        print(">>> For Vinograd mod. with len = ", size,
        "Time:", doTimeTest(vinogradMultiplicationModified,
                            A, B))
```

3.5. Вывод

Были реализованы функции алгоритмов перемножения матриц на языке Python 3, а также функции тестирования и подсчёта процессорного времени.

4. Исследовательская часть

Измерения процессорного времени проводятся при одинаковых размерах матриц и при этом тестируются как чётные размеры, так и нечётные: 100, 101, 150, 151, 200, 201.

4.1. Результаты экспериментов

Проведя измерения процессорного времени выполнения реализованных алгоритмов, можно составить для чётных размеров таблицу 4.1 и для нечётных таблицу 4.2.

Таблица 4.1: Результаты замеров процессорного времени в секундах, для чётных размеров

Название метода \ Размер	100	150	200
Стандартный	0.421	1.578	3.406
алгоритм Винограда	0.453	1.563	3.625
оптимизированный алг. Винограда	0.328	0.969	2.422

Таблица 4.2: Результаты замеров процессорного времени в секундах, для нечётных размеров

Название метода \ Размер	101	151	201
Стандартный	0.391	1.469	3.422
алгоритм Винограда	0.516	1.813	3.813
оптимизированный алг. Винограда	0.313	1.125	2.766

4.2. Вывод

Анализируя результаты замеров затрачиваемого времени и тестируя алгоиртмы на разных размерах, можно сказать, что стандартный алгоритм умножения матриц выигрывает по времени у остальных, но на малых размерах, однако его главным преимущестом является стабильность, то есть алгоритм не зависит от чётности размера матрицы.

Также стоит заметить, что модифицированный алгоритм Винограда показывает наилучшее время на средних размерах, в то время, как неоптимизированный алгоритм Винограда показывает наихудшее. В неоптимизированном алгоритме Винограда приходится неоднократно вычислять одни и те же значения, что может замедлять его.

Заключение

В ходе работы были изучены алгоритмы умножения матриц. Реализованы 3 алгоритма, приведен программный код реализации алгоритмов по умножению матриц. Была подсчитана трудоемкость каждого из алгоритмов. А также было проведено сравнение алгоритмов по времени и трудоемкости. Показано, что наименее трудоемкий и наименее затратный по времени при больших размерах матриц алгоритм Винограда оптимизированный.

Цель работы достигнута, решены поставленные задачи. Получены практические навыки реализации алгоритмов Винограда и стандартного алгоритма, а также проведена исследовательская работа по оптимизации и вычислении трудоемкости алгоритмов.

Список литературы

1. Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход. – М.: Техносфера, 2017. – 267с.

2. Документация на официальном сайте Python про библиотеку time [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html (дата обращения 23.09.2020)