

МГТУ им. БАУМАНА

ЛАБОРАТОРНАЯ РАБОТА №2

По курсу: "АНАЛИЗ АЛГОРИТМОВ"

## **Алгоритмы умножения матриц**

Работу выполнил: Чернов Даниил, ИУ7-56Б

Преподаватели: Волкова Л.Л., Строганов Ю.В.

*Москва, 2019*

# Оглавление

<b>Введение</b>	<b>3</b>
<b>1 Аналитическая часть</b>	<b>4</b>
1.1 Алгоритм Винограда . . . . .	4
1.2 Вывод . . . . .	5
<b>2 Конструкторская часть</b>	<b>6</b>
2.1 Схемы алгоритмов . . . . .	6
2.2 Трудоемкость алгоритмов . . . . .	10
2.2.1 Трудоемкость первичной проверки . . . . .	10
2.2.2 Классический алгоритм . . . . .	10
2.2.3 Алгоритм Винограда . . . . .	11
2.2.4 Оптимизированный алгоритм Винограда . . . . .	11
2.3 Вывод . . . . .	11
<b>3 Технологическая часть</b>	<b>12</b>
3.1 Выбор ЯП . . . . .	12
3.2 Сведения о модулях программы . . . . .	12
3.3 Листинг кода алгоритмов . . . . .	13
3.3.1 Оптимизация алгоритма Винограда . . . . .	16
3.4 Тестирование программы . . . . .	18
3.5 Вывод . . . . .	19
<b>4 Исследовательская часть</b>	<b>20</b>
4.1 Сравнительный анализ на основе замеров времени работы алгоритмов . . . . .	20
4.2 Вывод . . . . .	21

<b>Заключение</b>	<b>22</b>
<b>Список литературы</b>	<b>22</b>

# Введение

Цель работы: изучение алгоритмов умножения матриц. В данной лабораторной работе рассматривается стандартный алгоритм умножения матриц, алгоритм Винограда и модифицированный алгоритм Винограда. Также требуется изучить расчет сложности алгоритмов, получить навыки в улучшении алгоритмов.

В ходе лабораторной работы предстоит решить следующие задачи:

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

# 1 | Аналитическая часть

Матрицей  $A$  размера  $[m * n]$  называется прямоугольная таблица чисел, функций или алгебраических выражений, содержащая  $m$  строк и  $n$  столбцов. Числа  $m$  и  $n$  определяют размер матрицы. [1] Если число столбцов в первой матрице совпадает с числом строк во второй, то эти две матрицы можно перемножить. У произведения будет столько же строк, сколько в первой матрице, и столько же столбцов, сколько во второй.

Пусть даны две прямоугольные матрицы  $A$  и  $B$  размеров  $[m * n]$  и  $[n * k]$  соответственно. В результате произведения матриц  $A$  и  $B$  получим матрицу  $C$  размера  $[m * k]$ .

$C_{i,j} = \sum_{r=1}^n A_{i,r} \cdot B_{r,j}$  называется произведением матриц  $A$  и  $B$  [1].

## 1.1 Алгоритм Винограда

Подход Алгоритма Винограда является иллюстрацией общей методологии, начатой в 1979-х годах на основе билинейных и трилинейных форм, благодаря которым большинство усовершенствований для умножения матриц были получены [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 (1.1)

$$V \cdot W = v_1 \cdot w_1 + v_2 \cdot w_2 + v_3 \cdot w_3 + v_4 \cdot w_4 \quad (1.1)$$

Равенство (1.1) можно переписать в виде (1.2)

$$V \cdot W = (v_1 + w_2) \cdot (v_2 + w_1) + (v_3 + w_4) \cdot (v_4 + w_3) - v_1 \cdot v_2 - v_3 \cdot v_4 - w_1 \cdot w_2 - w_3 \cdot w_4 \quad (1.2)$$

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. Это означает, что над предварительно обработанными элементами нам придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

В алгоритме Винограда, результатом умножения матриц  $A$  размером  $[m * n]$  и  $B$  размером  $[n * k]$  получим матрицу  $C$  размера  $[m * k]$  которая вычисляется по следующим формулам:

Вычисление строковых множителей (1.3)

$$Rows_i = \sum_{j=1}^{n/2} A_{i,2j} \cdot A_{i,2j+1} \quad (1.3)$$

Вычисление множителей столбцов (1.4)

$$Cols_i = \sum_{j=1}^{n/2} B_{2j,i} \cdot B_{2j+1,i} \quad (1.4)$$

Вычисление результирующей матрицы  $C$  (1.5)

$$C_{i,j} = -Rows_i - Cols_j + \sum_{k=1}^{n/2} (A_{i,2k+1} + B_{2k,j}) \cdot (A_{i,2k} + B_{2k+1,j}) \quad (1.5)$$

Следует отметить, что в худшем случае (при условии того что кол-во столбцов в матрице  $A$  либо кол-во строк в матрице  $B$  - нечётное) к результирующей формуле (1.5) применяется следующее выражение (1.6)

$$C_{i,j}+ = A_{i,n} \cdot B_{n,j} \quad (1.6)$$

## 1.2 Вывод

Были рассмотрены алгоритмы классического умножения матриц и алгоритм Винограда, основное отличие которых — наличие предварительной обработки, а также количество операций умножения.

## 2 | Конструкторская часть

**Требования к вводу:** На вход подаются две матрицы

**Требования к программе:**

- корректное умножение двух матриц;
- при матрицах неправильных размеров программа не должна аварийно завершаться.

### 2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов.

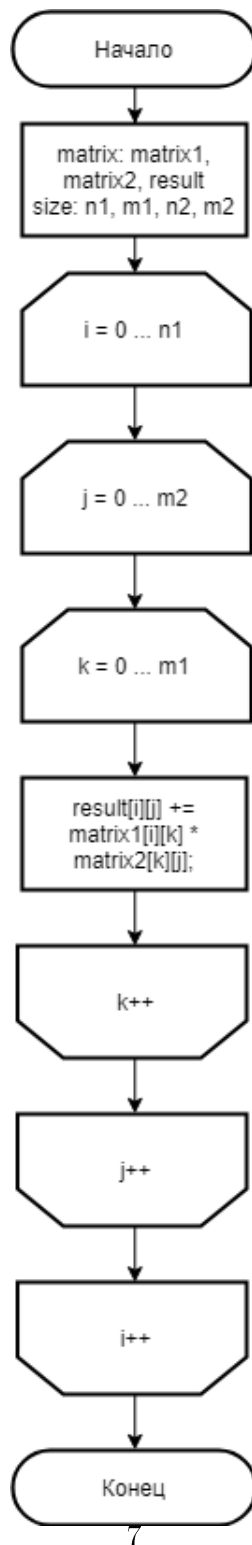


Рис. 2.1: Схема классического алгоритма умножения матриц



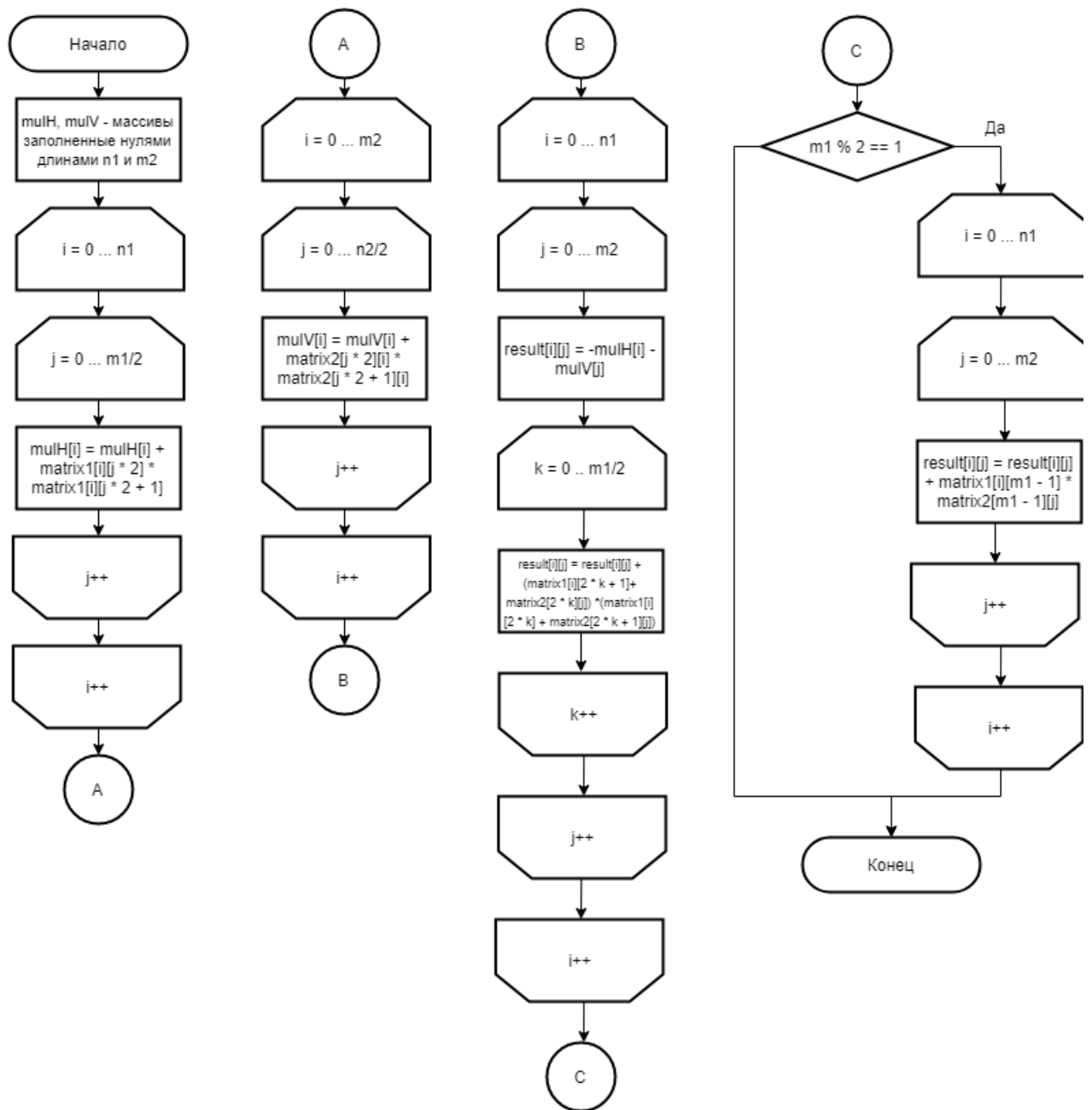


Рис. 2.2: Схема алгоритма Винограда

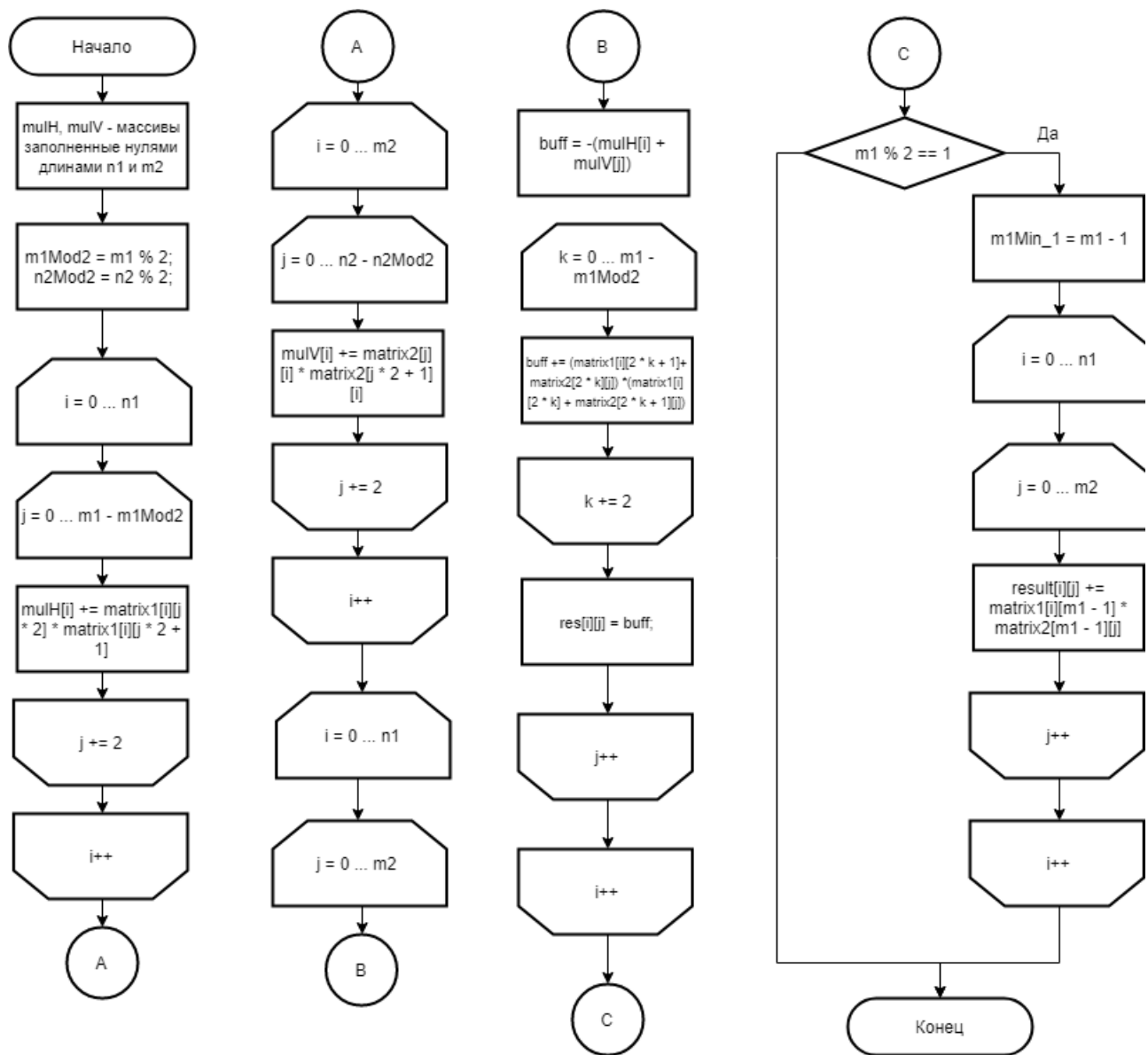


Рис. 2.3: Схема оптимизированного алгоритма Винограда

## 2.2 Трудоемкость алгоритмов

Введем модель трудоемкости для оценки алгоритмов:

- базовые операции стоимостью 1 —  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $=$ ,  $==$ ,  $<=$ ,  $>=$ ,  $!=$ ,  $+=$ ,  $||$ , получение полей класса
- оценка трудоемкости цикла:  $F_{\text{ц}} = \text{init} + N * (\text{a} + F_{\text{тела}} + \text{post}) + \text{a}$ , где  $\text{a}$  - условие цикла,  $\text{init}$  - предусловие цикла,  $\text{post}$  - постусловие цикла
- стоимость условного перехода применим за 0, стоимость вычисления условия остаётся

Оценим трудоемкость алгоритмов по коду программы.

### 2.2.1 Трудоемкость первичной проверки

Рассмотрим трудоемкость первичной проверки на возможность умножения матриц.

Табл. 2.1 Построчная оценка веса

Код	Вес
<code>int firstRowCount = firstMatrix.length;</code>	2
<code>int secondRowCount = secondMatrix.length;</code>	2
<code>if (firstRowCount == 0    secondRowCount == 0) return null;</code>	3
<code>int firstColumnCount = firstMatrix[0].length;</code>	3
<code>int secondColumnCount = secondMatrix[0].length;</code>	3
<code>if (firstColumnCount == 0    secondColumnCount == 0) return null;</code>	1
Итого	14

### 2.2.2 Классический алгоритм

Рассмотрим трудоемкость классического алгоритма:

Инициализация матрицы результата:  $1 + 1 + n_1(1 + 2 + 1) + 1 = 4n_1 + 3$

Подсчет:

$$1 + n_1(1 + (1 + m_2(1 + (1 + m_1(1 + (8) + 1) + 1) + 1) + 1) + 1) + 1 = n_1(m_2(10m_1 + 4) + 4) + 2 = 10n_1m_2m_1 + 4n_1m_2 + 4n_1 + 2$$

### 2.2.3 Алгоритм Винограда

Аналогично рассмотрим трудоемкость алгоритма Винограда.

Первый цикл:  $\frac{15}{2}n_1m_1 + 5n_1 + 2$

Второй цикл:  $\frac{15}{2}m_2n_2 + 5m_2 + 2$

Третий цикл:  $13n_1m_2m_1 + 12n_1m_2 + 4n_1 + 2$

Условный переход:  $\begin{bmatrix} 2 & , \text{ невыполнение условия} \\ 15n_1m_2 + 4n_1 + 2 & , \text{ выполнение условия} \end{bmatrix}$

Итого:  $13n_1m_2m_1 + \frac{15}{2}n_1m_1 + \frac{15}{2}m_2n_2 + 12n_1m_2 + 5n_1 + 5m_2 + 4n_1 + 6 +$   
 $\begin{bmatrix} 2 & , \text{ невыполнение условия} \\ 15n_1m_2 + 4n_1 + 2 & , \text{ выполнение условия} \end{bmatrix}$

### 2.2.4 Оптимизированный алгоритм Винограда

Аналогично Рассмотрим трудоемкость оптимизированного алгоритма Винограда:

Первый цикл:  $\frac{11}{2}n_1m_1 + 4n_1 + 2$

Второй цикл:  $\frac{11}{2}m_2n_2 + 4m_2 + 2$

Третий цикл:  $\frac{17}{2}n_1m_2m_1 + 9n_1m_2 + 4n_1 + 2$

Условный переход:  $\begin{bmatrix} 1 & , \text{ невыполнение условия} \\ 10n_1m_2 + 4n_1 + 2 & , \text{ выполнение условия} \end{bmatrix}$

Итого:  $\frac{17}{2}n_1m_2m_1 + \frac{11}{2}n_1m_1 + \frac{11}{2}m_2n_2 + 9n_1m_2 + 8n_1 + 4m_2 + 6 +$   
 $\begin{bmatrix} 1 & , \text{ невыполнение условия} \\ 10n_1m_2 + 4n_1 + 2 & , \text{ выполнение условия} \end{bmatrix}$

## 2.3 Вывод

В данном разделе были рассмотрены схемы алгоритмов умножения матриц, введена модель оценки трудоемкости алгоритма, были рассчитаны трудоемкости алгоритмов в соответствии с этой моделью.

## 3 | Технологическая часть

### 3.1 Выбор ЯП

В качестве языка программирования был выбран Java [?], а средой разработки IntelliJ IDEA. Время работы алгоритмов было замерено с помощью класса `System.Time`.

### 3.2 Сведения о модулях программы

Программа состоит из:

- `Main.java` - главный файл программы, в котором располагается точка входа в программу и функция замера времени.
- `MatrixMultiplication.java` - класс, реализующий стандартный алгоритм умножения матриц.
- `WinogradMultiplication.java` - класс, реализующий алгоритм умножения матриц по Винограду.
- `WinogradOptimizedMultiplication.java` - класс, реализующий улучшенный алгоритм Винограда для умножения матриц.
- `RandomMatrix.java` - класс, реализующий генерацию случайных матрицы заданного размера.
- `TestCases.java` - класс, реализовывающий юнит-тестирование.

### 3.3 Листинг кода алгоритмов

В листингах 3.1 - 3.3 будет рассмотрена реализация описанных алгоритмов.

Листинг 3.1: Стандартный алгоритм умножения матриц

```
1 public class MatrixMultiplication {
2     static public int [][] Multiply(@NotNull int [][] matrixA
3     , @NotNull int [][] matrixB) {
4         if (matrixA.length == 0 || matrixB.length == 0)
5             return null;
6         if (matrixA[0].length != matrixB.length)
7             return null;
8
9         int [][] matrixResult = new int[matrixA.length][
10            matrixB[0].length];
11
12         for (int i = 0; i < matrixA.length; i++)
13             for (int j = 0; j < matrixB[0].length; j++)
14                 for (int k = 0; k < matrixB.length; k++)
15                     matrixResult[i][j] = matrixResult[i][j]
16                     + matrixA[i][k] * matrixB[k][j];
17         return matrixResult;
18     }
19 }
```

Листинг 3.2: Алгоритм Винограда

```
1 public class WinogradMultiplication extends
2     MatrixMultiplication {
3     static public int [][] Multiply(@NotNull int [][] matrixA
4     , @NotNull int [][] matrixB) {
5         if (matrixA.length == 0 || matrixB.length == 0)
6             return null;
7         if (matrixA[0].length != matrixB.length)
8             return null;
9
10        int [][] matrixResult = new int[matrixA.length][
11            matrixB[0].length];
12        int firstRowCount = matrixA.length;
```

```

10     int firstColumnCount = matrixA[0].length;
11     int secondRowCount = matrixB.length;
12     int secondColumnCount = matrixB[0].length;
13     int[] rowFactor = new int[firstRowCount];
14     int[] columnFactor = new int[secondColumnCount];
15
16     for(int i = 0; i < firstRowCount; i++)
17         for (int j = 0; j < firstColumnCount / 2; j++)
18             rowFactor[i] += matrixA[i][2 * j] * matrixA
19                 [i][2 * j + 1];
20
21     for(int i = 0; i < secondColumnCount; i++)
22         for (int j = 0; j < secondRowCount / 2; j++)
23             columnFactor[i] += matrixB[2 * j][i] *
24                 matrixB[2 * j + 1][i];
25
26     for(int i = 0; i < firstRowCount; i++)
27         for(int j = 0; j < secondColumnCount; j++) {
28             matrixResult[i][j] = -rowFactor[i] -
29                 columnFactor[j];
30             for(int k = 0; k < firstColumnCount / 2; k
31                 ++){
32                 matrixResult[i][j] += + (matrixA[i][2 *
33                     k + 1] + matrixB[2 * k][j]) *
34                     (matrixA[i][2 * k] + matrixB[2
35                         * k + 1][j]);
36             }
37
38     if (firstColumnCount % 2 == 1)
39         for (int i = 0; i < firstRowCount; i++)
40             for (int j = 0; j < secondColumnCount; j++)
41                 matrixResult[i][j] += matrixA[i][
42                     firstColumnCount - 1] * matrixB[
43                         firstColumnCount - 1][j];
44
45     return matrixResult;
46 }
47 }

```

Листинг 3.3: Оптимизированный алгоритм Винограда

```

1 public class WinogradOptimizedMultiplication extends
   WinogradMultiplication {
2     static public int [][] Multiply(@NotNull int [][] matrixA
   , @NotNull int [][] matrixB) {
3         if (matrixA.length == 0 || matrixB.length == 0)
4             return null;
5         if (matrixA[0].length != matrixB.length)
6             return null;
7
8         int [][] matrixResult = new int[matrixA.length][
   matrixB[0].length];
9         int firstRowCount = matrixA.length;
10        int secondRowCount = matrixB.length;
11        int firstColumnCount = matrixA[0].length;
12        int secondColumnCount = matrixB[0].length;
13        int [] rowFactors = new int[firstRowCount];
14        int [] columnFactors = new int[secondColumnCount];
15        int fColumnMod2 = firstColumnCount % 2;
16        int sRowMod2 = secondRowCount % 2;
17
18        for (int i = 0; i < firstRowCount; i++)
19            for (int j = 0; j < (firstColumnCount -
   fColumnMod2); j += 2)
20                rowFactors[i] += matrixA[i][j] * matrixA[i
   ][j + 1];
21
22        for (int i = 0; i < secondColumnCount; i++)
23            for (int j = 0; j < (secondRowCount - sRowMod2)
   ; j += 2)
24                columnFactors[i] += matrixB[j][i] * matrixB
   [j + 1][i];
25
26        for (int i = 0; i < firstRowCount; i++)
27            for (int j = 0; j < secondColumnCount; j++) {
28                int buff = -(rowFactors[i] + columnFactors[
   j]);
29                for (int k = 0; k < (firstColumnCount -
   fColumnMod2); k += 2)
30                    buff += (matrixA[i][k + 1] + matrixB[k

```



```

31         ][j]) * (matrixA[i][k] + matrixB[k +
32         1][j]);
33     matrixResult[i][j] = buff;
34 }
35 if (fColumnMod2 == 1)
36     for (int i = 0; i < firstRowCount; i++)
37         for (int j = 0; j < secondColumnCount; j++)
38             matrixResult[i][j] += matrixA[i][
39                 firstColumnCount - 1] * matrixB[
40                 firstColumnCount - 1][j];
41 }

```

### 3.3.1 Оптимизация алгоритма Винограда

В рамках данной лабораторной работы было предложено 3 оптимизации:

1. Избавление от деления в условии цикла;
2. Замена  $rowFactors[i] += \dots$  на  $rowFactors[i] += \dots$  (аналогично для  $columnFactors[i]$ );

Листинг 3.4: Оптимизации алгоритма Винограда №1 и №2

```

1      int fColumnMod2 = firstColumnCount \% 2;
2      int sRowMod2 = secondRowCount \% 2;
3
4      for (int i = 0; i < firstRowCount; i++) {
5          for (int j = 0; j < (firstColumnCount -
6              fColumnMod2); j += 2) {
7              rowFactors[i] += firstMatrix[i][j] *
8                  firstMatrix[i][j + 1];
9          }
10     }
11
12     for (int i = 0; i < secondColumnCount; i++) {
13         for (int j = 0; j < (secondRowCount -
14             sRowMod2); j += 2) {
15             columnFactors[i] += secondMatrix[j][i]
16                 * secondMatrix[j + 1][i];
17         }
18     }

```

3. Накопление результата в буфер, чтобы не обращаться каждый раз к одной и той же ячейке памяти. Сброс буфера в ячейку матрицы после цикла.

Листинг 3.5: Оптимизации алгоритма Винограда №3

```

1      for (int i = 0; i < firstRowCount; i++) {
2          for (int j = 0; j < secondColumnCount; j++)
3              {
4                  int buff = -(rowFactors[i] +
5                      columnFactors[j]);
6                  for (int k = 0; k < (firstColumnCount -
7                      fColumnMod2); k += 2) {
8                      buff += (firstMatrix[i][k + 1] +
9                          secondMatrix[k][j]) * (
10                             firstMatrix[i][k] + secondMatrix
11                                 [k + 1][j]);
12                  }
13                  result[i][j] = buff;
14              }
15     }

```

### 3.4 Тестирование программы

Было произведено тестирование реализованных алгоритмов с помощью библиотеки JUnit.

Всего было реализованно 7 тестовых случаев:

- Некорректный размер матриц. Алгоритм должен возвращать Null
- Размер матриц равен 0
- Размер матриц равен 1
- Сравнение работы стандартной реализации с Виноградом на случайных значениях

Четный размер

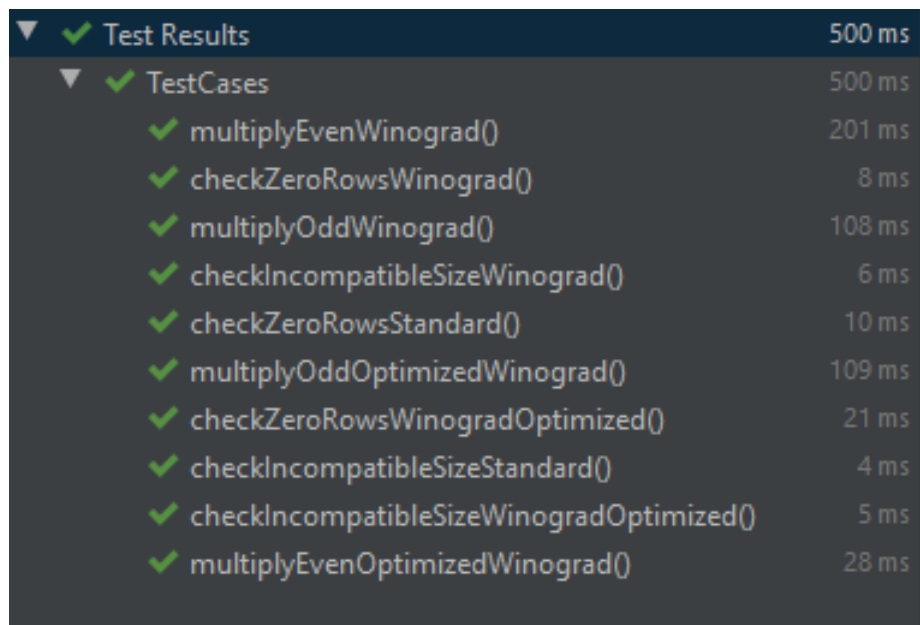
Нечетный размер

- Сравнение работы стандартной реализации с оптимизированным Виноградом на случайных значениях

Четный размер

Нечетный размер

На 3.1 будут предоставлены результаты тестирования программы. Откуда видно, что тестирование пройдено, алгоритмы реализованы правильно.



▼ ✓ Test Results	500 ms
▼ ✓ TestCases	500 ms
✓ multiplyEvenWinograd()	201 ms
✓ checkZeroRowsWinograd()	8 ms
✓ multiplyOddWinograd()	108 ms
✓ checkIncompatibleSizeWinograd()	6 ms
✓ checkZeroRowsStandard()	10 ms
✓ multiplyOddOptimizedWinograd()	109 ms
✓ checkZeroRowsWinogradOptimized()	21 ms
✓ checkIncompatibleSizeStandard()	4 ms
✓ checkIncompatibleSizeWinogradOptimized()	5 ms
✓ multiplyEvenOptimizedWinograd()	28 ms

Рис. 3.1: Результаты работы тестов

## 3.5 Вывод

В данном разделе была рассмотрена реализация и тестирование ПО, а так же листинги кода программы.

## 4 | Исследовательская часть

### 4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов.

Первый эксперимент производится для лучшего случая на квадратных матрицах размером от 100 x 100 до 1000 x 1000 с шагом 100. Сравним результаты для разных алгоритмов:

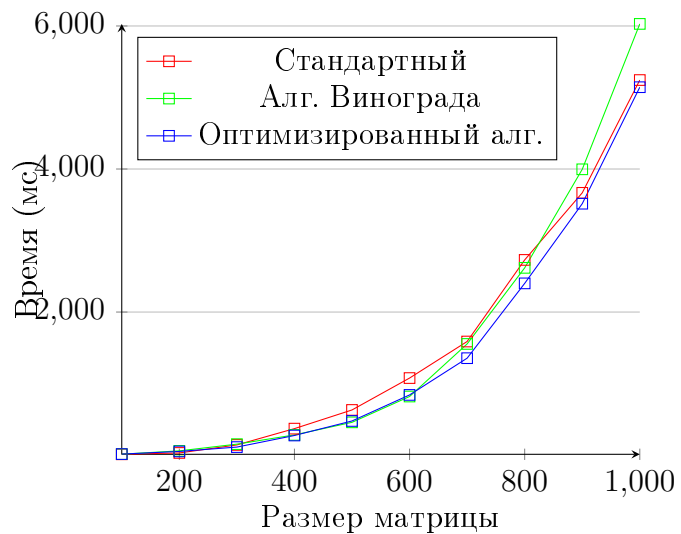


Рис. 4.1: Сравнение времени работы алгоритмов при четном размере матрицы

Второй эксперимент производится для худшего случая, когда поданы квадратные матрицы с нечетными размерами от 101 x 101 до 1001 x 1001 с шагом 100. Сравним результаты для разных алгоритмов:

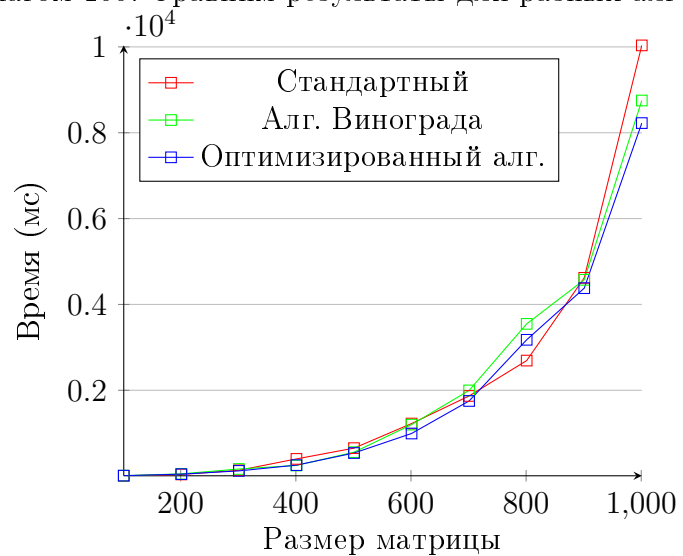


Рис. 4.2: Сравнение времени работы алгоритмов при нечетном размере матрицы

## 4.2 Вывод

По результатам тестирования все рассматриваемые алгоритмы реализованы правильно. Самым медленным алгоритмом оказался алгоритм классического умножения матриц, а самым быстрым — оптимизированный алгоритм Винограда.

## Заключение

В ходе лабораторной работы я изучил алгоритмы умножения матриц: стандартный и Винограда, оптимизировал алгоритм Винограда, дал теоретическую оценку алгоритмов стандартного умножения матриц, Винограда и улучшенного Винограда, реализовал три алгоритма умножения матриц на языке программирования Java и сравнил эти алгоритмы.

# Литература

- [1] И. В. Белоусов(2006), Матрицы и определители, учебное пособие по линейной алгебре, с. 1 - 16
- [2] Le Gall, F. (2012), "Faster algorithms for rectangular matrix multiplication Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2012), pp. 514–523
- [3] Документация по языку Java[Электронный ресурс], - режим доступа: <https://docs.oracle.com/en/java/javase/13/>