



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 Лемешев А. П.

Группа ИУ7-52Б

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

Оглавление

Введение	2
1 Аналитическая часть	3
1.1 Стандартный алгоритм умножения матриц	4
1.2 Умножение матриц по Винограду	4
2 Конструкторская часть	6
2.1 Разработка алгоритмов	6
2.2 Модель вычислений	9
2.3 Трудоёмкость алгоритмов	9
2.3.1 Стандартный алгоритм умножения матриц	9
2.3.2 Алгоритм Винограда	10
2.3.3 Оптимизированный алгоритм Винограда	11
3 Технологическая часть	13
3.1 Требования к ПО	13
3.2 Средства реализации	13
3.3 Листинг кода	13
3.4 Тестирование функций	19
4 Исследовательская часть	20
4.1 Технические характеристики	20
4.2 Пример работы программы	20
4.3 Время выполнения алгоритмов	20
Заключение	24

Введение

В данной лабораторной работе будут рассмотрены алгоритмы умножения матриц. Матрицы A и B могут быть перемножены, если число столбцов матрицы A равно числу строк B .

Алгоритм Винограда – алгоритм умножения квадратных 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 Винограда, с учетом серии улучшений и доработок в последующие годы, обладает лучшей асимптотикой среди известных алгоритмов умножения матриц. На практике алгоритм Винограда не используется [?], так как он имеет очень большую константу пропорциональности и начинает выигрывать в быстродействии у других известных алгоритмов только для матриц, размер которых превышает память современных компьютеров.

В настоящее время умножение матриц активно используется в компьютерной графике, криптографии.

В данной работе будут предложены реализации следующих алгоритмов:

- стандартный алгоритм умножения матриц;
- алгоритм Винограда;
- оптимизированный алгоритм Винограда.

Задачи лабораторной работы:

- изучить и реализовать стандартный алгоритм умножения матриц;
- изучить и реализовать алгоритм Винограда умножения матриц;
- оптимизировать алгоритм Винограда умножения матриц;
- оценить трудоемкость реализаций алгоритмов умножения матриц теоретически;
- сравнить временные характеристики вышеизложенных алгоритмов экспериментально.

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов умножения матриц.

Матрица – объект, записываемый в виде прямоугольной таблицы элементов, которая представляет собой совокупность строк и столбцов, на пересечении которых находятся её элементы 1.1.

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \quad (1.1)$$

Произведение матриц AB состоит из всех возможных комбинаций скалярных произведений вектор-строк матрицы A и вектор-столбцов матрицы B 1.1.

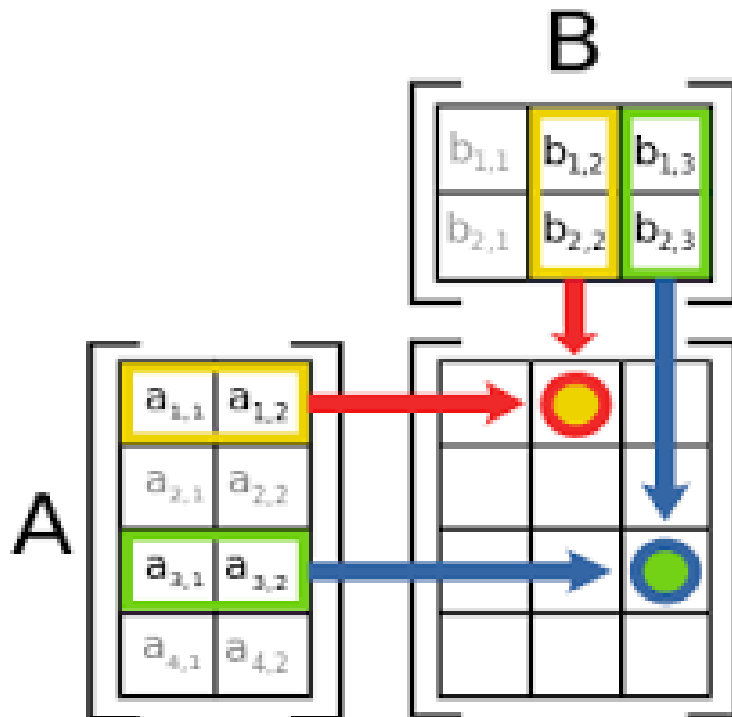


Рис. 1.1: Произведение матриц

1.1 Стандартный алгоритм умножения матриц

Пусть даны матрицы A размерностью $n \times m$ и B $m \times q$. Матрица $C = AB$ будет размерностью $n \times q$. Тогда каждый элемент матрицы C выражается формулой 1.2.

$$c_{ij} = \sum_{k=1}^m a_{ik} b_{kj} \quad (i = 1, 2, \dots, n; j = 1, 2, \dots, q) \quad (1.2)$$

1.2 Умножение матриц по Винограду

Каждый элемент в матрице C , которая является результатом умножения двух матриц, представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. В алгоритме умножения матриц по Винограду предложено сделать предварительную обработку, позволяющую часть работы выполнить заранее.

Рассмотрим два вектора V (формула 1.3) и W (формула 1.4).

$$V = (v_1, v_2, v_3, v_4) \quad (1.3)$$

$$W = (w_1, w_2, w_3, w_4) \quad (1.4)$$

Их скалярное произведение вычисляется по формуле 1.5.

$$V * W = v_1 w_1 + v_2 w_2 + v_3 w_3 + v_4 w_4 \quad (1.5)$$

Равенство 1.5 можно записать в виде 1.6.

$$V * W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1 v_2 - v_3 v_4 - w_1 w_2 - w_3 w_4 \quad (1.6)$$

Несмотря на то, что второе выражение 1.6 требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырех умножений - шесть, а вместо трех сложений - десять, выражение в пра-

вой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй, то для каждого элемента будет необходимо выполнить лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения. Из-за того, что операция сложения быстрее операции умножения, алгоритм должен работать быстрее стандартного.

Вывод

Были рассмотрены основополагающие материалы, которые в дальнейшем потребуются при реализации алгоритмов умножения матриц.

2 Конструкторская часть

В данном разделе представлены схемы и трудоемкости реализуемых алгоритмов.

2.1 Разработка алгоритмов

На рисунке 2.1 и 2.2 представлены схемы стандартного алгоритма и алгоритма Винограда умножения матриц. На рисунке 2.3 представлены схемы дополнительных алгоритмов, использующихся в алгоритме Винограда.

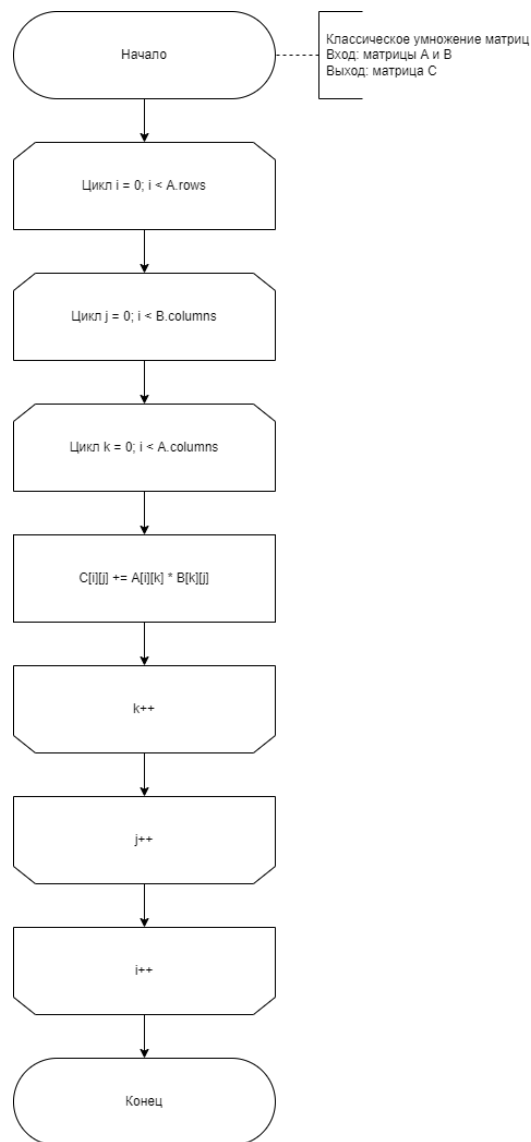


Рис. 2.1: Схема стандартного алгоритма умножения матриц

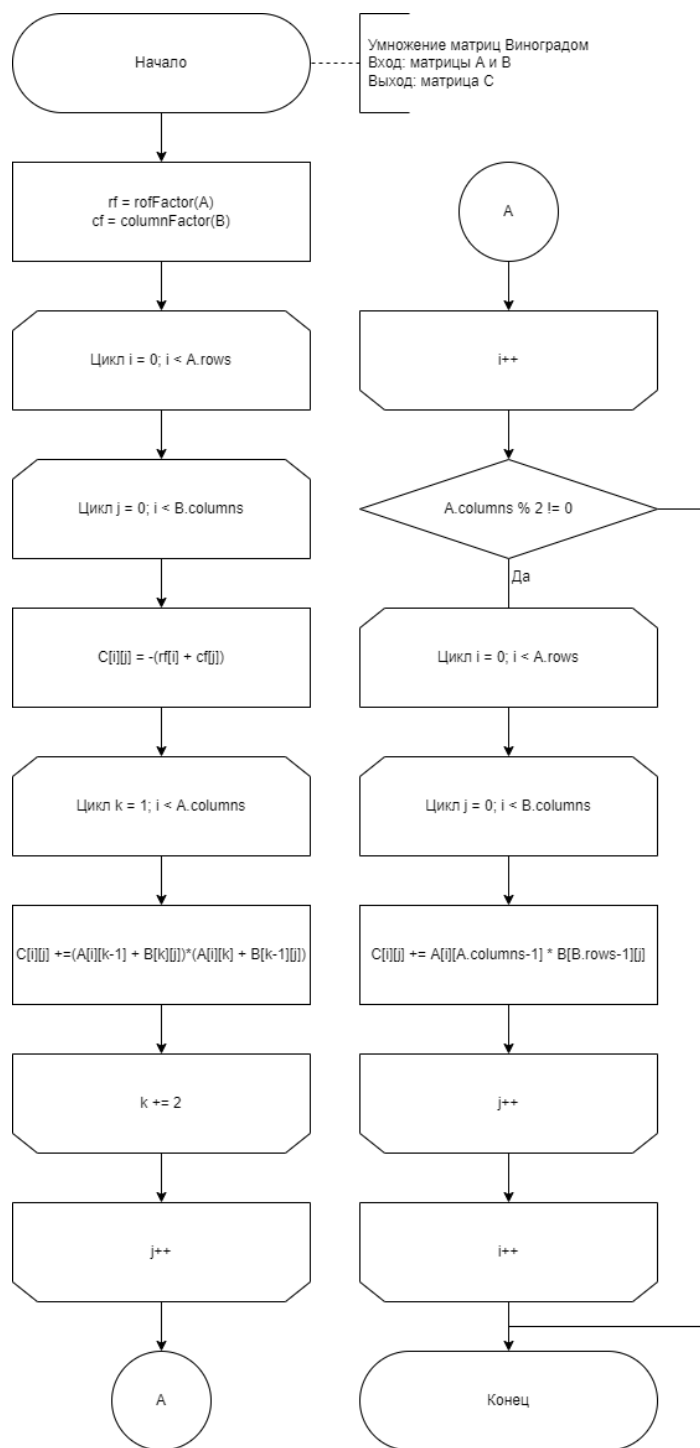


Рис. 2.2: Схема алгоритма Винограда умножения матриц

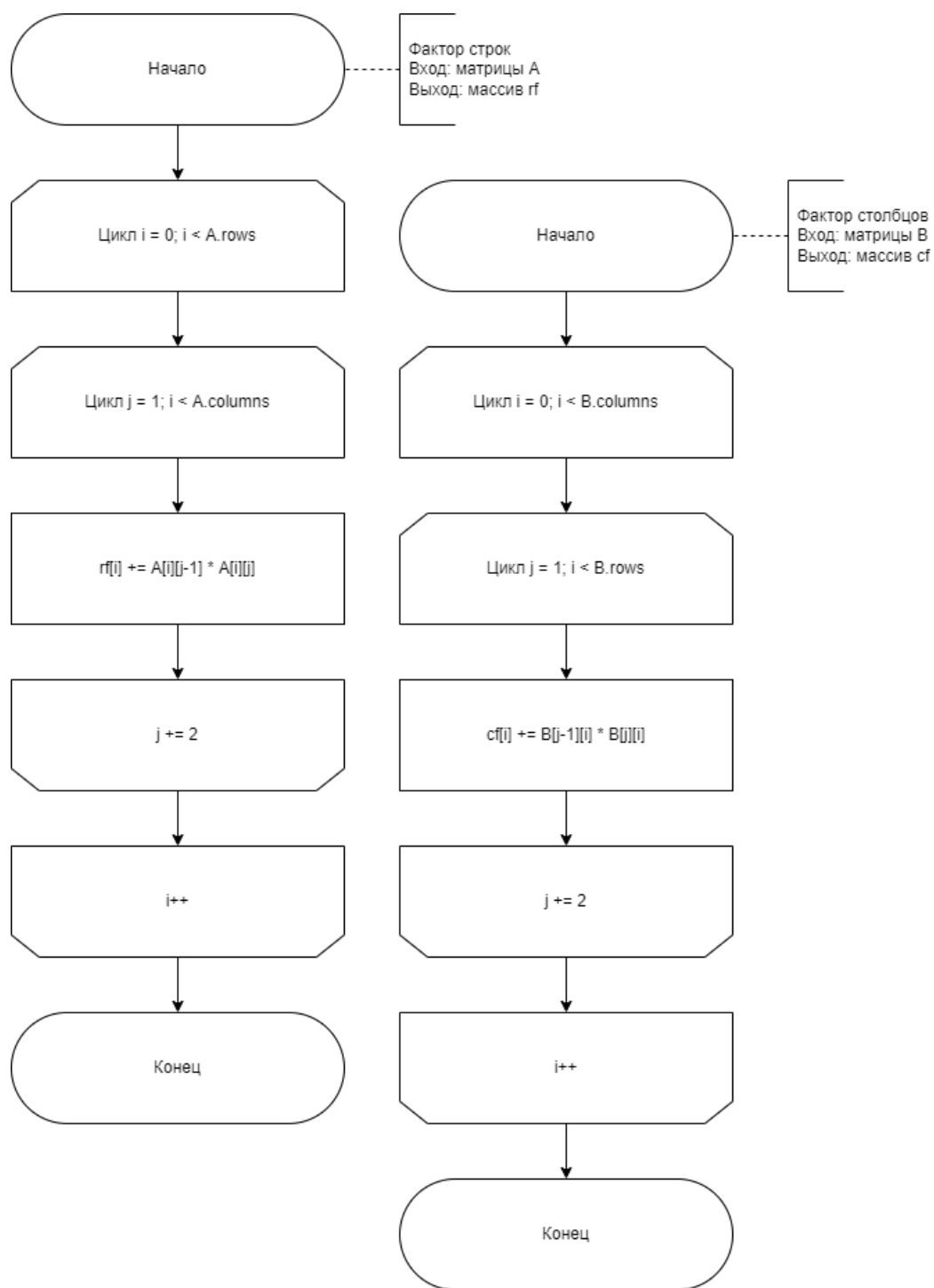


Рис. 2.3: Дополнительные алгоритмы

2.2 Модель вычислений

Для последующего вычисления трудоемкости необходимо ввести модель вычислений:

1. операции из списка (2.1) имеют трудоемкость 1;

$$+, -, /, *, =, + =, - =, ==, !=, <, >, <=, >=, [], ++, -- \quad (2.1)$$

2. трудоемкость оператора выбора `if условие then A else B` рассчитывается, как (2.2);

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \quad (2.2)$$

3. трудоемкость цикла рассчитывается, как (2.3);

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \quad (2.3)$$

4. трудоемкость вызова функции равна 0.

2.3 Трудоёмкость алгоритмов

Обозначим во всех последующих вычислениях размерность матрицы A как $n \times k$, а матрицы B как $k \times m$.

2.3.1 Стандартный алгоритм умножения матриц

Трудоёмкость стандартного алгоритма умножения матриц состоит из:

- внешнего цикла по $i \in [1 \dots n]$;
- цикла по $j \in [1 \dots m]$;
- скалярного умножения двух векторов – цикл по $k \in [1 \dots k]$.

Трудоёмкость стандартного алгоритма равна трудоёмкости внешнего цикла, можно вычислить ее, подставив циклы тела (2.4):

$$f_{base} = 2 + n \cdot (4 + m \cdot (4 + 11k)) = 2 + 4 \cdot n + 4 \cdot nm + 11 \cdot nmk \approx 11 \cdot nmk \quad (2.4)$$

2.3.2 Алгоритм Винограда

Трудоёмкость алгоритма Винограда состоит из:

- формирования массива сумм произведений пар соседних элементов строк матрицы А;
- формирования массива сумм произведений пар соседних элементов строк матрицы В;
- цикла заполнения ячеек матрицы С для чётных размеров;
- дополнительный цикл, если размер матрицы нечётный.

Формирование массива сумм произведений пар соседних элементов строк матрицы А можно посчитать как 2.5:

$$f_1 = 2 + n \cdot (2 + 4 + q/2 \cdot (4 + 1 + 6 + 3 \cdot 2 + 2)) = 19/2 \cdot nm + 6 \cdot n + 2; \quad (2.5)$$

Формирования массива сумм произведений пар соседних элементов строк матрицы В можно посчитать как 2.6:

$$f_2 = 2 + n \cdot (2 + 4 + q/2 \cdot (4 + 1 + 6 + 3 \cdot 2 + 2)) = 19/2 \cdot nm + 6 \cdot n + 2; \quad (2.6)$$

Цикла заполнения ячеек матрицы С для чётных размеров можно посчитать как 2.7:

$$f_3 = 2 + n \cdot (2 + 2 + m \cdot (2 + 7 + 4 + q/2 \cdot (4 + 1 + 12 + 5 + 5 \cdot 2))) = 32/2 \cdot nqm + 13 \cdot nm + 4 \cdot n + 2; \quad (2.7)$$

Дополнительный цикл 2.8:

$$f_4 = 3 + \begin{cases} 0, & \text{чётная,} \\ 2 + 4 \cdot n + 13 \cdot nm, & \text{иначе.} \end{cases} \quad (2.8)$$

Итого, трудоёмкость в лучшем случае (2.9):

$$f_{best} = 10nmk \quad (2.9)$$

Трудоёмкость в худшем случае (2.10):

$$f_{worst} = 8nmk \quad (2.10)$$

2.3.3 Оптимизированный алгоритм Винограда

Данный алгоритм можно оптимизировать:

1. заменив выражения вида $\mathbf{a} = \mathbf{a} + \dots$ на $\mathbf{a} += \dots$;
2. введя промежуточный буфер для подсчета промежуточных значений;
3. сделав в циклах по k шаг 2, избавившись тем самым от двух операций умножения на каждую итерацию.

Тогда трудоёмкости выглядят так: ...

Итого, трудоёмкость в лучшем случае (2.11):

$$f_{best} = 9nmk \quad (2.11)$$

Трудоёмкость в худшем случае (2.12):

$$f_{worst} = 7nmk \quad (2.12)$$

Вывод

На основе формул и теоретических данных, полученных в аналитическом разделе, были спроектированы схемы алгоритмов. Для каждого из

них были рассчитаны и оценены лучшие и худшие случаи.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации и листинга кода.

3.1 Требования к ПО

Программа должна отвечать следующим требованиям:

- программа на вход принимает две матрицы A и B ;
- количество столбцов матрицы A должно быть равно количеству строк матрицы B ;
- программа выдает результат умножения введенных пользователем матриц.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран современный компилируемый ЯП Golang [?]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного языка, а также тем, что данный язык предоставляет широкие возможности для написания тестов [?].

3.3 Листинг кода

В листингах 3.1, 3.2 и 3.4 приведены листинги описанных алгоритмов умножения матриц. В листингах 3.3 и 3.5 приведены вспомогательные функции, использующиеся в алгоритмах Винограда. В листингах 3.6 и 3.7 приведены примеры реализации тестов и бенчмарков.

Листинг 3.1: Классический алгоритм умножения матриц

```
1 func (m1 Matrix) MulClassic(m2 Matrix) (Matrix, error) {
2     if m1.columns != m2.rows {
3         return Matrix{}, fmt.Errorf("matrix1_columns_not_equal_matrix2_rows")
4     }
5     res, _ := Create(m1.rows, m2.columns)
6     for i := 0; i < res.rows; i++ {
7         for j := 0; j < res.columns; j++ {
8             for k := 0; k < m1.columns; k++ {
9                 res.data[i][j] = res.data[i][j] +
10                     m1.data[i][k]*m2.data[k][j]
11             }
12         }
13     }
14     return res, nil
15 }
```

Листинг 3.2: Умножение матриц алгоритмом Винограда

```

1 func (m1 Matrix) MulWinograd(m2 Matrix) (Matrix, error) {
2     res, _ := Create(m1.rows, m2.columns)
3     rf := m1.rowFactor()
4     cf := m2.columnFactor()
5     for i := 0; i < res.rows; i++ {
6         for j := 0; j < res.columns; j++ {
7             res.data[i][j] = -rf[i] - cf[j]
8             for k := 0; k < m1.columns/2; k++ {
9                 res.data[i][j] = res.data[i][j] + (m1.data[i][k*2+1]+
10                     m2.data[k*2][j])*(m1.data[i][k*2]+m2.data[k*2+1][j])
11             }
12         }
13     }
14     if m1.columns%2 != 0 {
15         for i := 0; i < res.rows; i++ {
16             for j := 0; j < res.columns; j++ {
17                 res.data[i][j] = res.data[i][j] + (m1.data[i][m1.columns-1]
18                     *
19                     m2.data[m2.rows-1][j])
20             }
21         }
22     }
23     return res, nil
24 }

```

Листинг 3.3: Вспомогательные функции для алгоритма Винограда

```

1 func (m Matrix) rowFactor() []int {
2     factor := make([]int, m.rows)
3     for i := 0; i < m.rows; i++ {
4         for j := 0; j < m.columns/2; j++ {
5             factor[i] = factor[i] + m.data[i][j*2+1]*m.data[i][j*2]
6         }
7     }
8     return factor
9 }
10
11 func (m Matrix) columnFactor() []int {
12     factor := make([]int, m.columns)
13     for i := 0; i < m.columns; i++ {
14         for j := 0; j < m.rows/2; j++ {
15             factor[i] = factor[i] + m.data[j*2+1][i]*m.data[j*2][i]
16         }
17     }
18     return factor
19 }

```


Листинг 3.4: Умножение матриц оптимизированным алгоритмом Винограда

```

1 func (m1 Matrix) MulWinogradOptimized(m2 Matrix) (Matrix, error) {
2     res, _ := Create(m1.rows, m2.columns)
3     rf := m1.rowFactorOptimized()
4     cf := m2.columnFactorOptimized()
5     for i := 0; i < m1.rows; i++ {
6         for j := 0; j < m2.columns; j++ {
7             res.data[i][j] -= rf[i] + cf[j]
8             for k := 1; k < m1.columns; k += 2 {
9                 res.data[i][j] += ((m1.data[i][k-1] + m2.data[k][j]) *
10                     (m1.data[i][k] + m2.data[k-1][j]))
11             }
12         }
13     }
14     if m1.columns&1 != 0 {
15         for i := 0; i < m1.rows; i++ {
16             for j := 0; j < m2.columns; j++ {
17                 res.data[i][j] += (m1.data[i][m1.columns-1] *
18                     m2.data[m1.columns-1][j])
19             }
20         }
21     }
22     return res, nil
23 }

```

Листинг 3.5: Вспомогательные функции для оптимизированного алгоритма

```

1 func (m Matrix) rowFactorOptimized() []int {
2     factor := make([]int, m.rows)
3     for i := 0; i < m.rows; i++ {
4         for j := 1; j < m.columns; j += 2 {
5             factor[i] += m.data[i][j-1] * m.data[i][j]
6         }
7     }
8     return factor
9 }
10
11 func (m Matrix) columnFactorOptimized() []int {
12     factor := make([]int, m.columns)
13     for i := 0; i < m.columns; i++ {
14         for j := 1; j < m.rows; j += 2 {
15             factor[i] += m.data[j-1][i] * m.data[j][i]
16         }
17     }
18     return factor
19 }

```

Листинг 3.6: Пример реализации тестов

```

1 func TestMulClassic(t *testing.T) {
2     for _, test := range testMulTable {
3         r, err := test.in[0].MulClassic(test.in[1])
4         if (err == nil) == test.err && r.Compare(test.out) {
5             t.Errorf("Incorrect result.\ntitle:_%v\nin1:_%v\nin2:_%v\nout:_%v\nres:_%v\n",
6                 test.title, test.in[0], test.in[1], test.out, r)
7         } else {
8             t.Logf("Test pass_%v'\n", test.title)
9         }
10    }
11 }
12
13 func TestMulWinograd(t *testing.T) {
14     for _, test := range testMulTable {
15         r, err := test.in[0].MulWinograd(test.in[1])
16         if (err == nil) == test.err && r.Compare(test.out) {
17             t.Errorf("Incorrect result.\ntitle:_%v\nin1:_%v\nin2:_%v\nout:_%v\nres:_%v\n",
18                 test.title, test.in[0], test.in[1], test.out, r)
19         } else {
20             t.Logf("Test pass_%v'\n", test.title)
21         }
22    }
23 }
24
25 func TestMulWinogradOptimized(t *testing.T) {
26     for _, test := range testMulTable {
27         r, err := test.in[0].MulWinogradOptimized(test.in[1])
28         if (err == nil) == test.err && r.Compare(test.out) {
29             t.Errorf("Incorrect result.\ntitle:_%v\nin1:_%v\nin2:_%v\nout:_%v\nres:_%v\n",
30                 test.title, test.in[0], test.in[1], test.out, r)
31         } else {
32             t.Logf("Test pass_%v'\n", test.title)
33         }
34    }
35 }

```

Листинг 3.7: Пример реализации бенчмарка

```
1 func Benchmark(mats [2]matrix.Matrix, repeats int) {  
2     var durations [3]uint64  
3  
4     for i := 0; i < repeats; i++ {  
5         start := C.tick()  
6         mats[0].MulClassic(mats[1])  
7         durations[0] += uint64(C.tick() - start)  
8  
9         start = C.tick()  
10        mats[0].MulWinograd(mats[1])  
11        durations[1] += uint64(C.tick() - start)  
12  
13        start = C.tick()  
14        mats[0].MulWinogradOptimized(mats[1])  
15        durations[2] += uint64(C.tick() - start)  
16    }  
17 }
```

3.4 Тестирование функций

В таблице 3.1 приведены тесты для функций, реализующих умножение матриц. Все тесты пройдены успешно.

Таблица 3.1: Тестовые данные

Первая матрица	Вторая матрица	Ожидаемый результат
$\begin{pmatrix} 5 \end{pmatrix}$	$\begin{pmatrix} 5 \end{pmatrix}$	$\begin{pmatrix} 10 \end{pmatrix}$
$\begin{pmatrix} 5 \\ 5 \end{pmatrix}$	$\begin{pmatrix} 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 10 & 15 \\ 10 & 15 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	$\begin{pmatrix} 2 & 4 \\ 6 & 8 \\ 10 & 12 \end{pmatrix}$	$\begin{pmatrix} 44 & 56 \\ 98 & 128 \end{pmatrix}$
$\begin{pmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \\ -7 & -8 & -9 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} -30 & -36 & -42 \\ -66 & -81 & -96 \\ -102 & -126 & -150 \end{pmatrix}$

Вывод

На основе схем из конструкторского раздела были разработаны и протестированы спроектированные алгоритмы.

4 Исследовательская часть

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

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система: Windows 11 x64 [?].
- Память: 8 GiB.
- Процессор: AMD Ryzen 5 3550H [?].

Замеры проводились на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.2 Пример работы программы

На рисунке 4.1 представлен результат работы программы.

Рис. 4.1: Пример работы программы

4.3 Время выполнения алгоритмов

Результаты тестирования приведены в таблице 4.1. На рисунках ?? и ?? приведены зависимости времени работы алгоритма от длины строк.

Таблица 4.1: Время работы алгоритмов

Размерность $n \times n$	Время работы, нс		
	Классический	Виноград	Виноград опт.
100	2169144	2424132	1811016
200	17680401	22032658	15485909
300	65838464	76854395	55530661
400	210892129	207347115	162146803
500	364922866	350837776	327348293
600	666673788	655599427	631239232
700	1369979155	1332273841	1207834761
800	3205123980	3108582931	2921219674

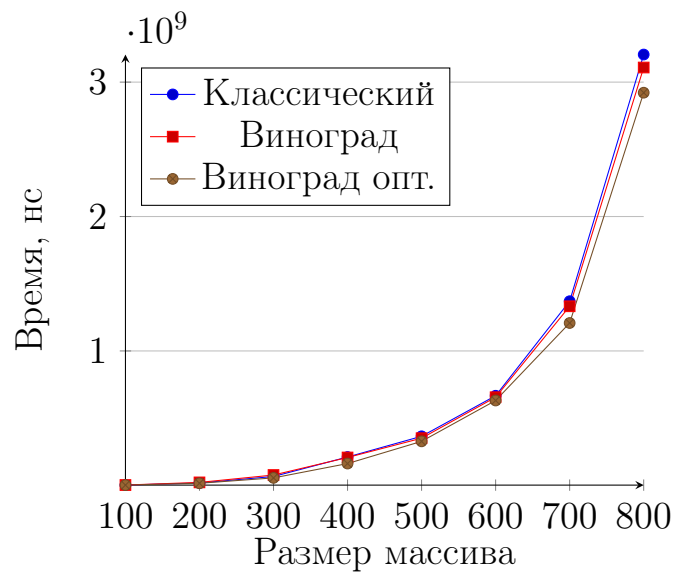


Рис. 4.2: Временные характеристики

Результаты тестирования приведены в таблице 4.1. На рисунках ?? и ?? приведены зависимости времени работы алгоритма от длины строк.

Таблица 4.2: Время работы алгоритмов при нечетной размерности

Размерность $n \times n$	Время работы, нс		
	Классический	Виноград	Виноград опт.
101	2259672	2796926	2018691
201	20071583	21688038	18481951
301	74128394	77705821	61183913
401	221489215	218382194	158158291
501	371283268	372238416	361183214
601	672286913	671599427	648239232
701	1382138491	1384273841	1324143862
801	3291482918	3389582931	3217248194

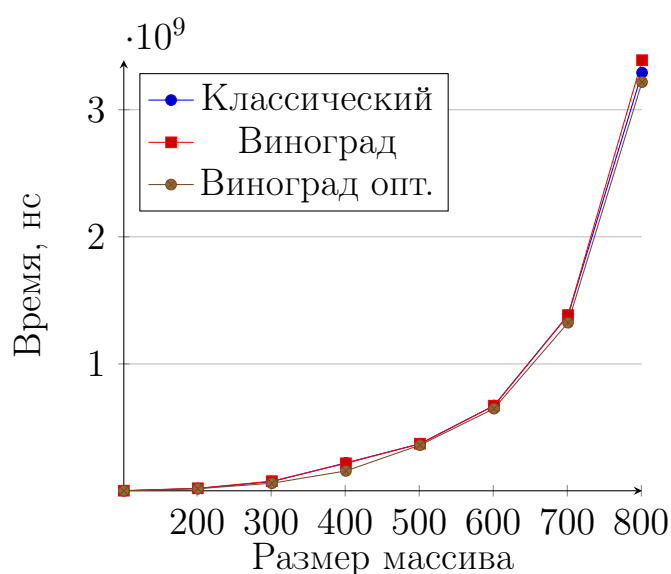


Рис. 4.3: Временные характеристики на нечетных размерах матриц

Вывод

В данном разделе было произведено сравнение количества затраченного времени вышеизложенных алгоритмов. Наименее затратным по времени оказался оптимизированный алгоритм Винограда. Но при этом ему дополнительно требуется $n + t$ памяти под результат (т.е. $n + t$ прибавляется к

тому количеству памяти, которое выделяется под результат в реализации стандартного алгоритма умножения матриц - $n \cdot m$).

Время работы реализации алгоритма Винограда незначительно меньше времени работы реализации простого алгоритма умножения, однако при размере 800×800 время вычислений реализации алгоритма Винограда на 0.3 секунды меньше, нежели у реализации простого алгоритма (что составляет в данном случае порядка 10%).

Такой результат совпадает с теоретически полученными оценками трудоемкости алгоритмов

Заключение

В рамках лабораторной работы были:

- изучен и реализован стандартный алгоритм умножения матриц;
- изучен и реализован алгоритм Винограда умножения матриц;
- оптимизирован алгоритм Винограда умножения матриц;
- произведена оценка трудоемкости реализаций алгоритмов умножения матриц;
- произведены сравнения временных характеристик вышеизложенных алгоритмов.

Среди рассмотренных алгоритмов наиболее эффективным оказался алгоритм Винограда, однако незначительное (порядка 10%) улучшение характеристик по времени повлекло за собой дополнительные затраты по памяти ($n + m$, где n и m - соответствующие размерности матриц). Алгоритм Винограда становится тем эффективнее по времени, чем большие размерности матриц подаются на вход алгоритма. В связи с вышеуказанным, оптимизированный алгоритм Винограда является предпочтительным при обработке больших матриц, однако, при строгих ограничениях на затраты по памяти, простой (или наивный) алгоритм умножения матриц является более предпочтительным, т.к. расходует лишь $n \cdot m$ памяти для хранения результирующей матрицы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Group-theoretic Algorithms for Matrix Multiplication / Henry Cohn, Robert Kleinberg, Balazs Szegedy [и др.] // Proceedings of the 46th Annual Symposium on Foundations of Computer Science. 2005. С. 379–388.
- [2] Язык программирования Go [Электронный ресурс]. Режим доступа: <https://go.dev> (дата обращения: 21.09.2022).
- [3] Документация по ЯП Go: бенчмарки [Электронный ресурс]. Режим доступа: <https://go.dev/doc/tutorial/add-a-test> (дата обращения: 21.09.2022).
- [4] Windows 11 [Электронный ресурс]. Режим доступа: <https://www.microsoft.com/en-us/windows?wa=wsignin1.0> (дата обращения: 21.09.2022).
- [5] Процессор AMD Ryzen™ 5 3550H [Электронный ресурс]. Режим доступа: <https://www.amd.com/ru/products/apu/amd-ryzen-5-3550h> (дата обращения: 21.09.2022).