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.

Москва — 2022 г.

Оглавление

Введение

В данной лабораторной работе будут рассмотрены алгоритмы умножения матриц. В программирование, как и в математике, часто приходится прибегать к использованию матриц. В настоящее время умножение матриц активно используется в компьютерной графике, криптографии.

Над матрицами существует различные операции, например: сложение, возведение в степень, умножение. В данной лабораторной работе пойдёт речь о умножении матриц и оптимизации этой операции. Матрицы A и B могут быть перемножены, если число столбцов матрицы A равно числу строк B.

Алгоритм Копперсмита-Винограда – алгоритм умножения квадратных матриц, предложенный в 1987 году Д. Копперсмитом и Ш. Виноградом.

В данной работе будут предложены реализации следующих алгоритмов:

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

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов умножения матриц.

Задачи лабораторной работы:

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

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

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

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

1.1 Стандартный алгоритм умножения матриц

Пусть даны матрицы A размерностью $n \times k$ и B размерностью $k \times m$. Тогда матрица $C = AB$ будет размерностью $n \times q$, а каждый элемент

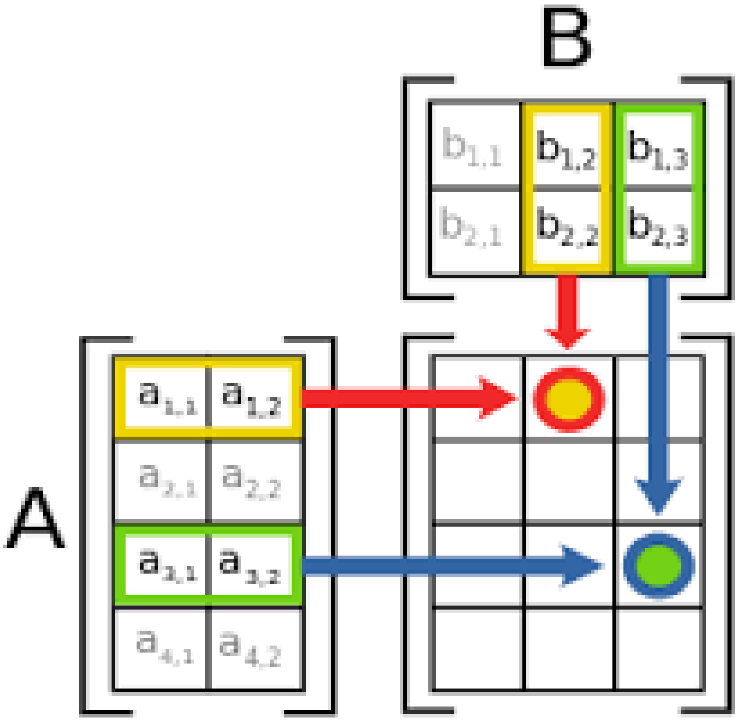

Рис. 1.1: Произведение матриц

матрицы C выражается формулой 1.2.

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj} \quad (i = \overline{1,n}; j = \overline{1,m}) \tag{1.2}$$

1.2 Умножение матриц по Винограду

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.

Каждый элемент в матрице C, которая является результатом умножения двух матриц, представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. В алгоритме умножение матриц по Винограду предложено сделать 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 вычисляется по формуле 1.3.

$$V \cdot W = v_1 w_1 + v_2 w_2 + v_3 w_3 + v_4 w_4 \tag{1.3}$$

Равенство 1.3 можно записать в виде 1.4.

$$\begin{aligned} V \cdot W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - \\ v_1 v_2 - v_3 v_4 - w_1 w_2 - w_3 w_4 \end{aligned} \tag{1.4}$$

Несмотря на то, что второе выражение 1.4 требует вычисления большего количества операций, чем стандартный алгоритм, 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, поэтому 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.

Стоит упомянуть, что при нечётном значении размера матрицы нужно дополнительно добавить произведения крайних элементов соответствующих строк и столбцов.

Вывод

В данном разделе были рассмотрены алгоритмы классического умножения матриц и алгоритм Винограда. Выявлено основное отличие, за счёт которого, алгоритм Винограда должен работать быстрее – предварительная обработка данных, а так же снижение количества операций умножения.

2 Конструкторская часть

В данном разделе представлены схемы и расчёт трудоемкости реализуемых алгоритмов.

2.1 Разработка алгоритмов

На рисунке 2.1 и 2.2 представлены схемы стандартного алгоритма и алгоритма Винограда умножения матриц соответственно. На рисунке 2.3 представлены схемы алгоритмов предварительной обработки данных, использующихся в алгоритме Винограда.

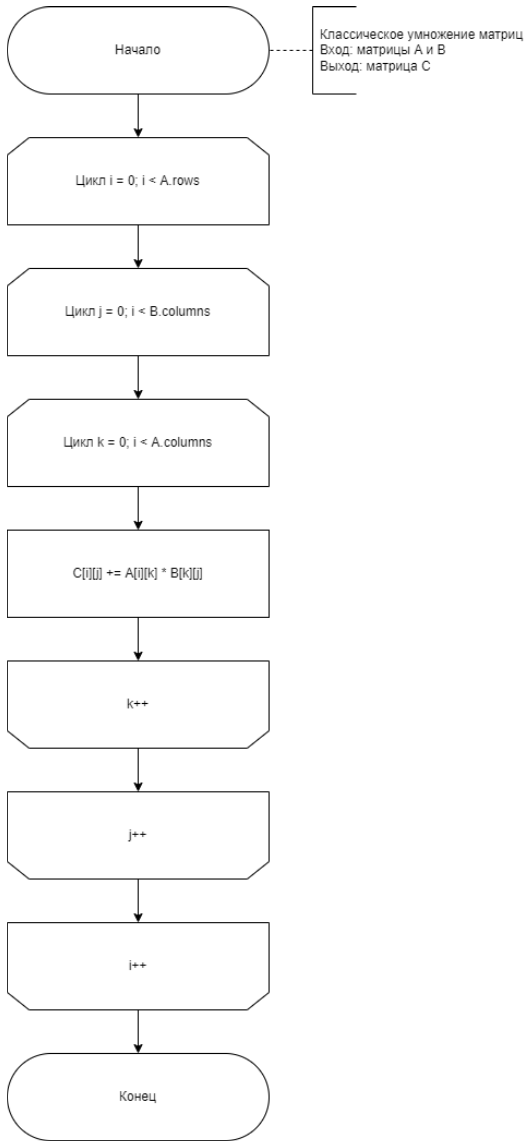

Рис. 2.1: Схема стандартного алгоритма умножения матриц

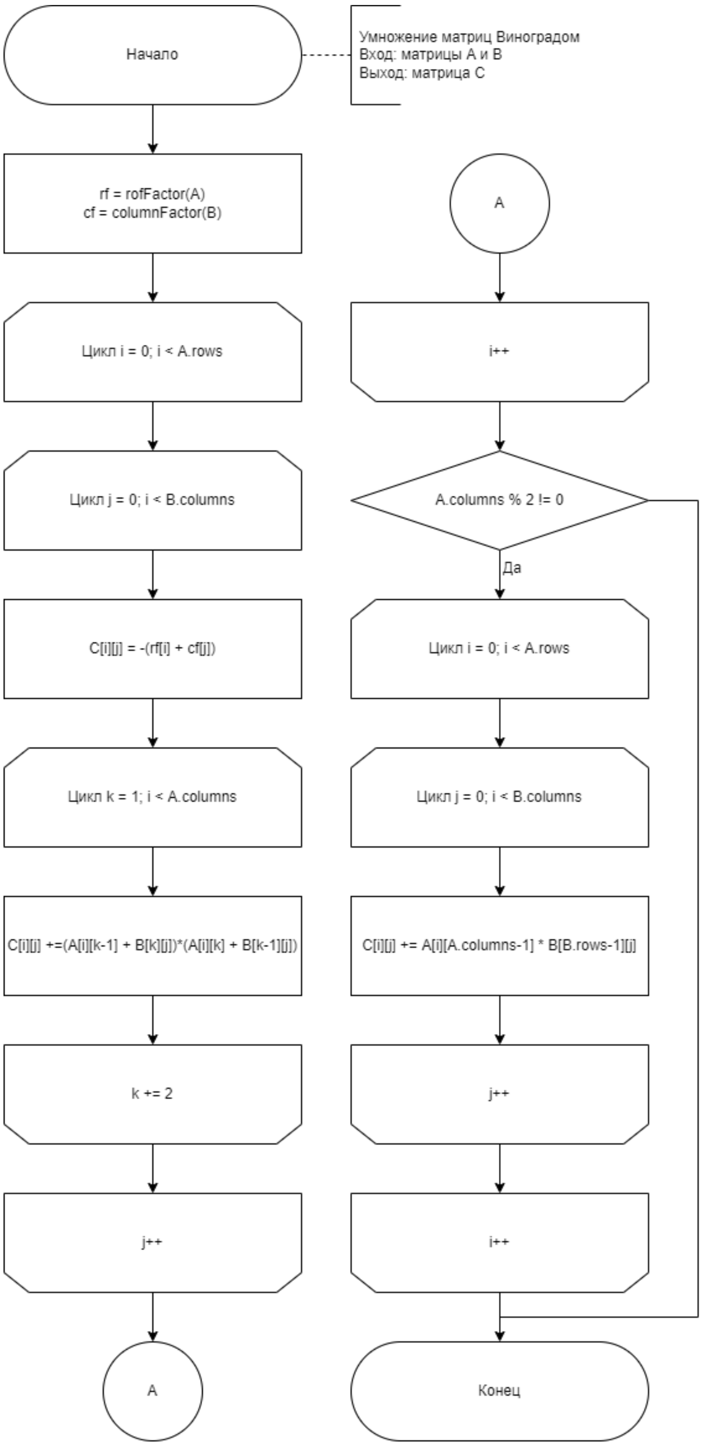

Рис. 2.2: Схема алгоритма Винограда умножения матриц

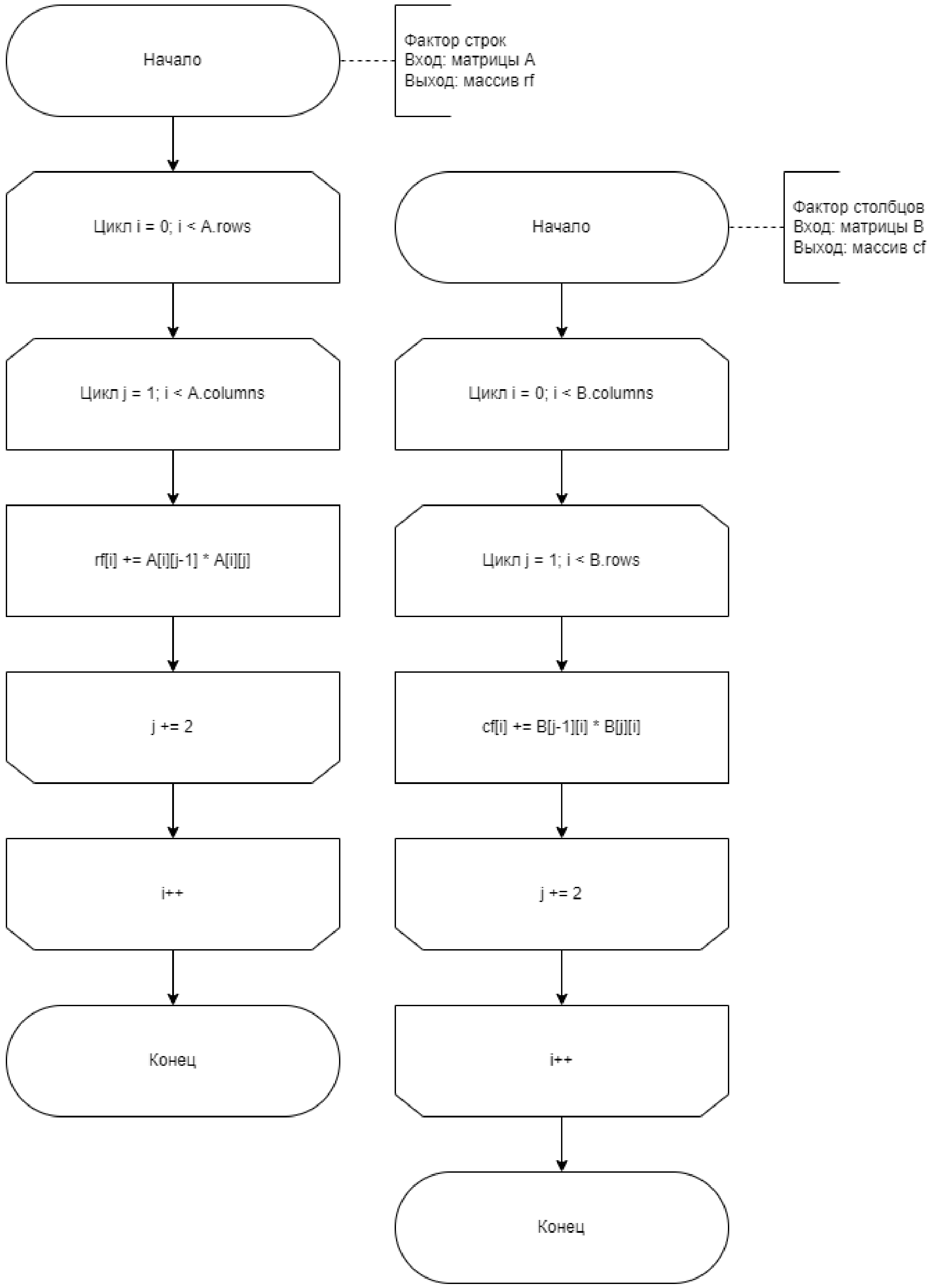

Рис. 2.3: Схемы алгоритмов предварительной обработки данных

2.2 Модель вычислений

Для последующего вычисления трудоемкости необходимо ввести модель вычислений:

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

$$+, -, =, +=, -=, ==, !=, <, >, <=, >=, [], ++, -- \tag{2.1}$$

2. операции из списка (2.2) имеют трудоемкость 2;

$$*, /, \%, *=, /=, \%= \tag{2.2}$$

3. трудоемкость оператора выбора `if условие then A else B` рассчитывается, как (2.3);

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.3}$$

4. трудоемкость цикла рассчитывается, как (2.4);

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \tag{2.4}$$

5. 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 включает в себя трудоемкости:

- внешнего цикла по $i \in [1 \dots n]$, рассчитывающегося как (2.5);

$$f_i = 2 + n(2 + f_j) \tag{2.5}$$

- цикла по $j \in [1 \dots m]$, рассчитывающегося как (2.6);

$$f_j = 2 + m(2 + f_k) \tag{2.6}$$

- скалярного умножения двух векторов – цикл по $k \in [1 \dots k]$, трудоёмкость которого равняется (2.7).

$$f_k = 2 + 14k \tag{2.7}$$

Трудоёмкость стандартного алгоритма равна трудоёмкости внешнего цикла, поэтому её можно вычислить как (2.8):

$$f_{classic} = 2 + n(4 + m(4 + 14k)) \approx 14nmk = O(N^3) \tag{2.8}$$

2.3.2 Алгоритм Винограда

Трудоёмкость алгоритма Винограда состоит из:

- инициализация массивов RF и CF, имеющая трудоёмкость (2.9);

$$f_{init} = n + m \tag{2.9}$$

- заполнение массива RF, который представляет из себя фактор строк и имеет трудоёмкость (2.10);

$$f_{RF} = 2 + n(4 + \frac{k}{2}(4 + 6 + 1 + 2 + 3 \cdot 2)) = 2 + 4n + 9.5nk \tag{2.10}$$

- заполнение массива CF, который представляет из себя фактор столбцов и имеет трудоёмкость (2.11);

$$f_{CF} = 2 + m(4 + \frac{k}{2}(4 + 6 + 1 + 2 + 3 \cdot 2)) = 2 + 4m + 9.5mk \quad (2.11)$$

- цикла заполнения ячеек матрицы C для чётных размеров, имеющий трудоёмкость (2.12);

$$f_{even} = 2+n(4+m(2+11+\frac{k}{2}(4+28))) = 2+4n+13nm+16nmk \quad (2.12)$$

- дополнительный цикл, если размер матрицы нечётный, имеющий трудоёмкость (2.13).

$$f_{odd} = 3 + \begin{cases} 0, & \text{размер матрицы чётный,} \\ 2 + 4n + 16nm, & \text{иначе.} \end{cases} \quad (2.13)$$

Трудоёмкость алгоритма Винограда равна сумме вышеперечисленных трудоёмкостей (2.14):

$$f_{win} = f_{init} + f_{RF} + f_{CF} + f_{even} + f_{onn} \quad (2.14)$$

Итого, трудоёмкость в лучшем случае (2.15):

$$f_{best} \approx 16nmk = O(N^3) \quad (2.15)$$

Трудоёмкость в худшем случае (2.16):

$$f_{worst} \approx 16nmk = O(N^3) \quad (2.16)$$

2.3.3 Оптимизированный алгоритм Винограда

Данный алгоритм можно оптимизировать:

1. заменив выражения вида $a = a + k$ на $a += k$;
2. заменив умножение на 2 на побитовый сдвиг;

3. предвычислив некоторые слагаемые для алгоритма.

Трудоёмкость инициализации массивов RF и CF никак не меняется. Заполнение массивов факторов и дополнительный цикл, для нечётных размеров матрицы, меняются незначительно для итоговой оценки трудоёмкости. Трудоёмкость цикла заполнения ячеек матрицы для чётных размеров выглядит как (2.17):

$$f_{even} = 2+n(4+m(4+7+\frac{k}{2}(4+19))) = 2+4n+11nm+11.5nmkO(N^3) \quad (2.17)$$

Итого, трудоёмкость в лучшем случае (2.18):

$$f_{best} \approx 11.5nmk = O(N^3) \quad (2.18)$$

Трудоёмкость в худшем случае (2.19):

$$f_{worst} \approx 11.5nmk = O(N^3) \quad (2.19)$$

Вывод

На основе формул и теоретических данных, полученных в аналитическом разделе, были спроектированы схемы алгоритмов. Проведена теоретическая оценка трудоёмкости и для каждого из алгоритмов 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 [1]. 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 [2].

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

```
func (m1 Matrix) MulClassic(m2 Matrix) (Matrix, error) {
	if m1.columns != m2.rows {
		return Matrix{}, fmt.Errorf("matrix1␣columns␣not␣equal␣matrix2␣rows")
	}
	res, _ := Create(m1.rows, m2.columns)
	for i := 0; i < res.rows; i++ {
		for j := 0; j < res.columns; j++ {
			for k := 0; k < m1.columns; k++ {
				res.data[i][j] = res.data[i][j] +
					m1.data[i][k]*m2.data[k][j]
			}
		}
	}
	return res, nil
}
```

Листинг 3.2: Умножение матриц алгоритмом Винограда

```
func (m1 Matrix) MulWinograd(m2 Matrix) (Matrix, error) {
	res, _ := Create(m1.rows, m2.columns)
	rf := m1.rowFactor()
	cf := m2.columnFactor()
	for i := 0; i < res.rows; i++ {
		for j := 0; j < res.columns; j++ {
			res.data[i][j] = -rf[i] - cf[j]
			for k := 0; k < m1.columns/2; k++ {
				res.data[i][j] = res.data[i][j] + (m1.data[i][k*2+1]+
					m2.data[k*2][j])*(m1.data[i][k*2]+m2.data[k*2+1][j])
			}
		}
	}
	if m1.columns%2 != 0 {
		for i := 0; i < res.rows; i++ {
			for j := 0; j < res.columns; j++ {
				res.data[i][j] = res.data[i][j] + (m1.data[i][m1.columns-1]
				    *
					m2.data[m2.rows-1][j])
			}
		}
	}
	return res, nil
}
```

Листинг 3.3: Вспомогательные функии для алгоритма Винограда

```
func (m Matrix) rowFactor() []int {
	factor := make([]int, m.rows)
	for i := 0; i < m.rows; i++ {
		for j := 0; j < m.columns/2; j++ {
			factor[i] = factor[i] + m.data[i][j*2+1]*m.data[i][j*2]
		}
	}
	return factor
}

func (m Matrix) columnFactor() []int {
	factor := make([]int, m.columns)
	for i := 0; i < m.columns; i++ {
		for j := 0; j < m.rows/2; j++ {
			factor[i] = factor[i] + m.data[j*2+1][i]*m.data[j*2][i]
		}
	}
	return factor
}
```

Листинг 3.4: Умножение матриц оптимизированным алгоритмом Винограда

```
func (m1 Matrix) MulWinogradOptimized(m2 Matrix) (Matrix, error) {
	res, _ := Create(m1.rows, m2.columns)
	rf := m1.rowFactorOptimized()
	cf := m2.columnFactorOptimized()
	for i := 0; i < m1.rows; i++ {
		for j := 0; j < m2.columns; j++ {
			res.data[i][j] -= rf[i] + cf[j]
			for k := 1; k < m1.columns; k += 2 {
				res.data[i][j] += ((m1.data[i][k-1] + m2.data[k][j]) *
					(m1.data[i][k] + m2.data[k-1][j]))
			}
		}
	}
	if m1.columns&1 != 0 {
		for i := 0; i < m1.rows; i++ {
			for j := 0; j < m2.columns; j++ {
				res.data[i][j] += (m1.data[i][m1.columns-1] *
					m2.data[m1.columns-1][j])
			}
		}
	}
	return res, nil
}
```

Листинг 3.5: Вспомогательные функии для оптимизированного алгоритма

```
func (m Matrix) rowFactorOptimized() []int {
	factor := make([]int, m.rows)
	for i := 0; i < m.rows; i++ {
		for j := 1; j < m.columns; j += 2 {
			factor[i] += m.data[i][j-1] * m.data[i][j]
		}
	}
	return factor
}

func (m Matrix) columnFactorOptimized() []int {
	factor := make([]int, m.columns)
	for i := 0; i < m.columns; i++ {
		for j := 1; j < m.rows; j += 2 {
			factor[i] += m.data[j-1][i] * m.data[j][i]
		}
	}
	return factor
}
```

Листинг 3.6: Пример реализации тестов

```
func TestMulClassic(t *testing.T) {
	for _, test := range testMulTable {
		r, err := test.in[0].MulClassic(test.in[1])
		if (err == nil) == test.err && r.Compare(test.out) {
			t.Errorf("Incorrect␣result.\ntitle:␣%v\nin1:␣%v\nin2:␣%v\nout:␣%v\nres:␣%v\n",
				test.title, test.in[0], test.in[1], test.out, r)
		} else {
			t.Logf("Test␣pass␣'%v'.\n", test.title)
		}
	}
}

func TestMulWinograd(t *testing.T) {
	for _, test := range testMulTable {
		r, err := test.in[0].MulWinograd(test.in[1])
		if (err == nil) == test.err && r.Compare(test.out) {
			t.Errorf("Incorrect␣result.\ntitle:␣%v\nin1:␣%v\nin2:␣%v\nout:␣%v\nres:␣%v\n",
				test.title, test.in[0], test.in[1], test.out, r)
		} else {
			t.Logf("Test␣pass␣'%v'.\n", test.title)
		}
	}
}

func TestMulWinogradOptimized(t *testing.T) {
	for _, test := range testMulTable {
		r, err := test.in[0].MulWinogradOptimized(test.in[1])
		if (err == nil) == test.err && r.Compare(test.out) {
			t.Errorf("Incorrect␣result.\ntitle:␣%v\nin1:␣%v\nin2:␣%v\nout:␣%v\nres:␣%v\n",
				test.title, test.in[0], test.in[1], test.out, r)
		} else {
			t.Logf("Test␣pass␣'%v'.\n", test.title)
		}
	}
}
```

Листинг 3.7: Пример реализации бенчмарка

```
func Benchmark(mats [2]matrix.Matrix, repeats int) {
	var durations [3]uint64

	for i := 0; i < repeats; i++ {
		start := C.tick()
		mats[0].MulClassic(mats[1])
		durations[0] += uint64(C.tick() - start)

		start = C.tick()
		mats[0].MulWinograd(mats[1])
		durations[1] += uint64(C.tick() - start)

		start = C.tick()
		mats[0].MulWinogradOptimized(mats[1])
		durations[2] += uint64(C.tick() - start)
	}
}
```

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 [3].
- Память: 8 GiB.
- Процессор: AMD Ryzen 5 3550H [4].

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 Время выполнения алгоритмов

Результаты тестирования приведены в таблице 4.1. На рисунке 4.1 приведена зависимость времени работы алгоритма от размера матрицы.

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

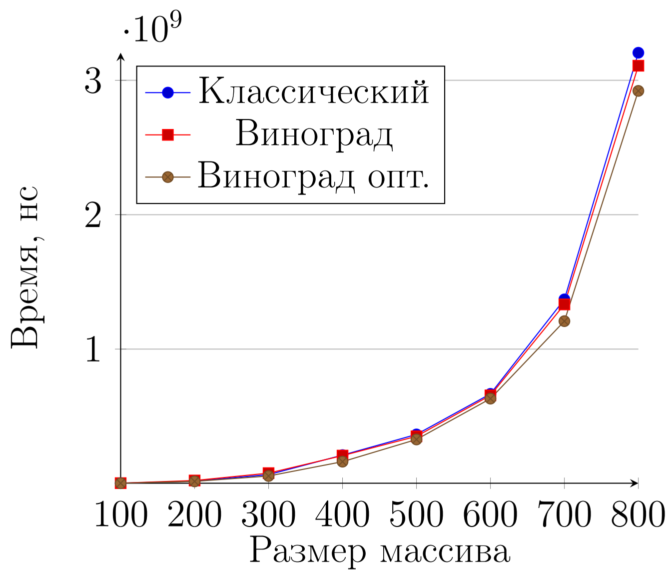

Рис. 4.1: Временные характеристики

Отдельно приведены результаты тестирования для нечётных размеров матриц в таблице 4.2. На рисунке 4.1 приведена зависимость времени работы алгоритма от размера матрицы, при условии, что этот размер нечётный.

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	1332138491	1284273841	1124143862
801	3211482918	3139582931	2917248194

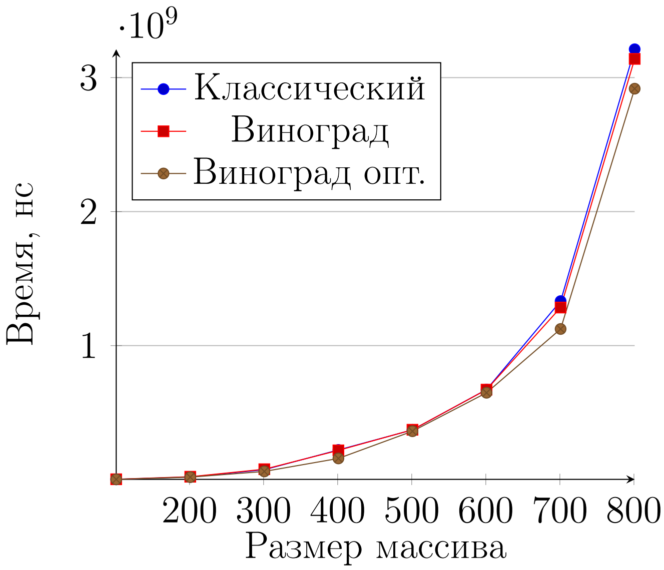

Рис. 4.2: Временные характеристики на нечетных размерах матриц

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 $n + m$ памяти.

Время работы реализации алгоритма Винограда незначительно меньше времени работы реализации простого алгоритма умножения, однако, при больших размерах, время вычислений реализации алгоритма Винограда в ~ 1.1 раза быстрее, нежели у реализации классического алгоритма.

Такой результат совпадает с теоретически полученными оценками трудоемкости алгоритмов.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- был изучен и реализован стандартный алгоритм умножения матриц;
- был изучен и реализован алгоритм Винограда умножения матриц;
- был оптимизирован алгоритм Винограда умножения матриц;
- была произведена оценка трудоемкости реализаций алгоритмов умножения матриц;
- были произведены сравнения временных характеристик вышеизложенных алгоритмов;
- подготовлен отчет о лабораторной работе.

Среди рассмотренных алгоритмов наиболее эффективным оказался алгоритм Винограда, однако незначительное (порядка 10%) улучшение характеристик по времени повлекло за собой дополнительные затраты по памяти. 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, классический алгоритм умножения матриц является более предпочтительным.

Список использованных источников

[1] Язык программирования Go [Электронный ресурс]. Режим доступа: `https://go.dev` (дата обращения: 23.10.2022).

[2] Документация по ЯП Go: бенчмарки [Электронный ресурс]. Режим доступа: `https://go.dev/doc/tutorial/add-a-test` (дата обращения: 23.10.2022).

[3] Windows 11 [Электронный ресурс]. Режим доступа: `https://www.microsoft.com/en-us/windows?wa=wsignin1.0` (дата обращения: 23.10.2022).

[4] Процессор AMD Ryzen™ 5 3550H [Электронный ресурс]. Режим доступа: `https://www.amd.com/ru/products/apu/amd-ryzen-5-3550h` (дата обращения: 23.10.2022).