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»

Отчёт по лабораторной работе №2
по дисциплине «Анализ алгоритмов»

Тема: Алгоритмы умножения матриц
Студент: Карпова Е. О.
Группа: ИУ7-52Б
Оценка (баллы): _____
Преподаватели: Волкова Л. Л., Строганов Ю. В.

Москва — 2022 г.

Оглавление

Введение

Матрица — это математический объект, представляемый в виде прямоугольной таблицы элементов (например, целых чисел), имеющей определённое количество строк и столбцов, на пересечении которых находятся его элементы. Количество строк и столбцов задает размерность матрицы.

Для матрицы определены нижеперечисленные алгебраические операции:

- умножение матриц;
- сложение матриц;
- умножение матрицы на скаляр.

Матричное умножение считается одной из самых фундаментальных операций в современных вычислениях [1]. Например, оно широко применяется в:

- задачах линейной алгебры;
- задачах полилинейной алгебры;
- задачах полиномиальной алгебры;
- решении обыкновенных дифференциальных уравнений;
- решении уравнений в частных производных;
- решении интегральных уравнений;
- комбинаторике;
- статистике;
- биоинформатике.

Цель работы: получение навыков программирования, тестирования полученного программного продукта и проведения замеров времени выполнения и объёма потребляемой памяти по результатам работы программы на примере реализации алгоритмов умножения матриц.

Задачи работы:

1) изучение алгоритмов стандартного умножения матриц, Винограда и оптимизированного алгоритма Винограда;

2) разработка схем данных алгоритмов и анализ их трудоёмкости;

3) реализация данных алгоритмов;

4) проведение замеров потребления памяти (в байтах) для данных алгоритмов;

5) проведение замеров времени работы (в нс) данных алгоритмов;

6) получение графической зависимости замеряемых величин от размерности квадратной матрицы предоставляемой на вход алгоритмам;

7) проведение сравнительного анализа данных алгоритмов на основе полученных зависимостей.

1. Аналитическая часть

В данном разделе будут рассмотрены теоретические основы алгоритмов умножения матриц: стандартного, Винограда и Винограда с оптимизациями. Все алгоритмы рассматриваются для матриц, включающих в себя только целочисленные элементы.

Для программной реализации матрицу можно рассматривать как двумерный массив, или массив, содержащий элементы типа массив элементов заданного типа.

Соответственно, две матрицы можно перемножить тогда и только тогда, когда количество столбцов (количество элементов в массивах, хранящихся в массиве массивов) первой матрицы-множителя совпадает с количеством строк (количеством массивов в массиве массивов) второй матрицы-множителя. У результирующей матрицы количество строк будет совпадать с количеством строк в первой матрице, а количество столбцов с количеством столбцов во второй матрице.

1.1. Стандартный алгоритм умножения матриц

Стандартный алгоритм умножения матриц можно описать следующими соображениями.

Если существуют две матрицы, A и B (1.1), для которых количество строк и столбцов обозначить как m_A,n_A и m_B,n_B соответственно, и при этом $n_A = m_B$, то результатом их умножения (1.2) будет матрица C (1.3). Для матрицы C количество строк $m_C = m_A$, а количество столбцов $n_C = n_B$.

Данные соображения представимы формулами

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n_A} \\ a_{21} & a_{22} & \dots & a_{2n_A} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m_A1} & a_{m_A2} & \dots & a_{m_An_A} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n_B} \\ b_{21} & b_{22} & \dots & b_{2n_B} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m_B1} & b_{m_B2} & \dots & b_{m_Bn_B} \end{pmatrix}, \tag{1.1}$$

$$A \cdot B = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n_A} \\ a_{21} & a_{22} & \dots & a_{2n_A} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m_A1} & a_{m_A2} & \dots & a_{m_An_A} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n_B} \\ b_{21} & b_{22} & \dots & b_{2n_B} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m_B1} & b_{m_B2} & \dots & b_{m_Bn_B} \end{pmatrix} = C, \tag{1.2}$$

где

$$C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n_C} \\ c_{21} & c_{22} & \dots & c_{2n_C} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m_C1} & c_{m_C2} & \dots & c_{m_Cn_C} \end{pmatrix}. \tag{1.3}$$

Для каждого элемента c_{ij} матрицы C, где $i = \overline{1, m_C}$,$j = \overline{1, n_C}$:

$$c_{ij} = \sum_{k=1}^{m_B} a_{ik} b_{kj}. \tag{1.4}$$

1.2. Алгоритм Винограда для умножения матриц

Одним из самых эффективных по времени алгоритмов умножения матриц является алгоритм Винограда, имеющий асимптотическую сложность $O(n^{3.755})$ [6]. Описать основные этапы алгоритма можно при помощи следующих утверждений.

Пусть дан вектор $A = (a_1, a_2, a_3, a_4)$ и дан вектор $B = (b_1, b_2, b_3, b_4)$, где a_i, b_i для $i = \overline{1, 4}$ — некоторые целые числа, например. Произведение этих векторов можно записать в виде:

$$A \cdot B = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3 + a_4 \cdot b_4. \tag{1.5}$$

Формулу (1.5) можно также представить в следующем виде:

$$A \cdot B = (a_1 + b_2)(a_2 + b_1) + (a_3 + b_4)(a_4 + b_3) - a_1 \cdot a_2 - a_3 \cdot a_4 - b_1 \cdot b_2 - b_3 \cdot b_4. \tag{1.6}$$

В (1.6) количество умножений (6 операций) и сложений (9 операций) превышает число аналогичных операций в (1.5), где 4 умножения и 3 сложения, но вычисление результата по (1.6) допускает предварительную обработку.

Наибольшей эффективности по времени можно достичь, исключая наиболее трудоёмкие для ЭВМ операции, в данном случае — умножение, поэтому коэффициенты $a_1 \cdot a_2$, $a_3 \cdot a_4$, $b_1 \cdot b_2$, $b_3 \cdot b_4$ из (1.6) можно вычислять заранее, перед основным циклом, и использовать повторно для вычисления соответствующих значений.

Таким образом, количество операций в формуле на каждой итерации основного цикла сокращается до 2 для умножения и до 5 для сложения, без учёта двух дополнительных сложений с произведениями соседних элементов соответствующих строк и столбцов.

Для случая, когда совпадающая размерность для изначальных матриц — нечётная, требуется на каждой итерации производить дополнительное сложение с произведением последних элементов соответствующих строк и столбцов.

1.3. Оптимизированный алгоритм Винограда для умножения матриц

Для алгоритма Винограда существует множество модификаций, позволивших увеличить его эффективность, например:

- в 2010 Эндрю Стотерс усовершенствовал алгоритм до $O(n^{2.374})$ [7];
- в 2011 году Вирджиния Вильямс (англ.) усовершенствовала алгоритм до $O(n^{2.3728642})$ [8];
- 5 октября 2022 компанией $DeepMind$ при помощи нейросетевого алгоритма $AlphaZero$ были найдены несколько новых алгоритмов перемножения матриц различных размерностей: при размере 9×9 число операций уменьшилось с 511 до 498, а при 11×11 — с 919 до 896, а в ряде других случаев $AlphaTensor$ повторил лучшие из известных алгоритмов [9].

В данной лабораторной работе были реализованы следующие улучшения:

- операция $x = x + k$, где x, k — некоторые числа, была заменена на $x += k$;
- умножение на 2 было заменено на побитовый сдвиг операнда влево на 1 бит;
- вычисления горизонтального индекса предпоследнего элемента строки матрицы и элемента в середине строки, булевого значения кратности длины строки матрицы двум были вынесены из основного цикла;
- цикл, вход в который осуществляется программой в случае нечётности размерности исходной матрицы, был внесён внутрь основного, для чего введена булевая переменная-флаг.

2. Конструкторская часть

В данном разделе будут представлены схемы реализаций алгоритмов умножения матриц, среди которых стандартный, Винограда и Винограда с улучшениями.

2.1. Разработка стандартного алгоритма умножения матриц

На рисунке 2.1 представлена схема реализации стандартного алгоритма умножения матриц.

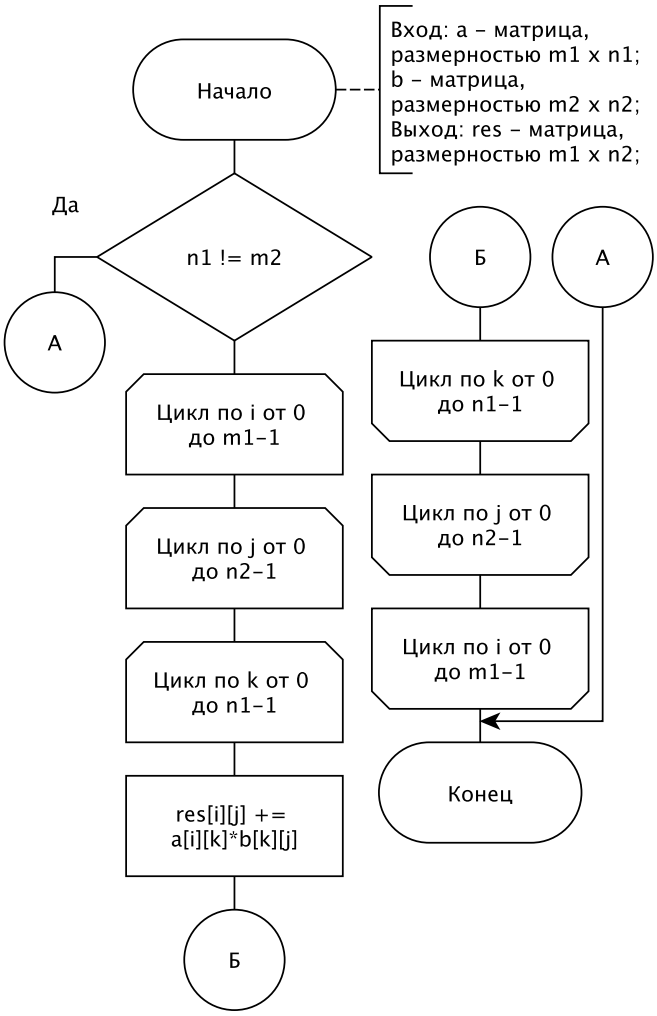

Рисунок 2.1 — Схема реализации стандартного алгоритма умножения матриц

2.2. Разработка алгоритма Винограда умножения матриц

На рисунке 2.2 представлена схема реализации алгоритма Винограда умножения матриц. На рисунках 2.3 и 2.4 представлены схемы реализаций алгоритмов подпрограмм поиска произведений соседних элементов строк матрицы (рисунок 2.3) и поиска произведений соседних элементов столбцов матрицы (рисунок 2.4), необходимых для реализации алгоритма Винограда умножения матриц.

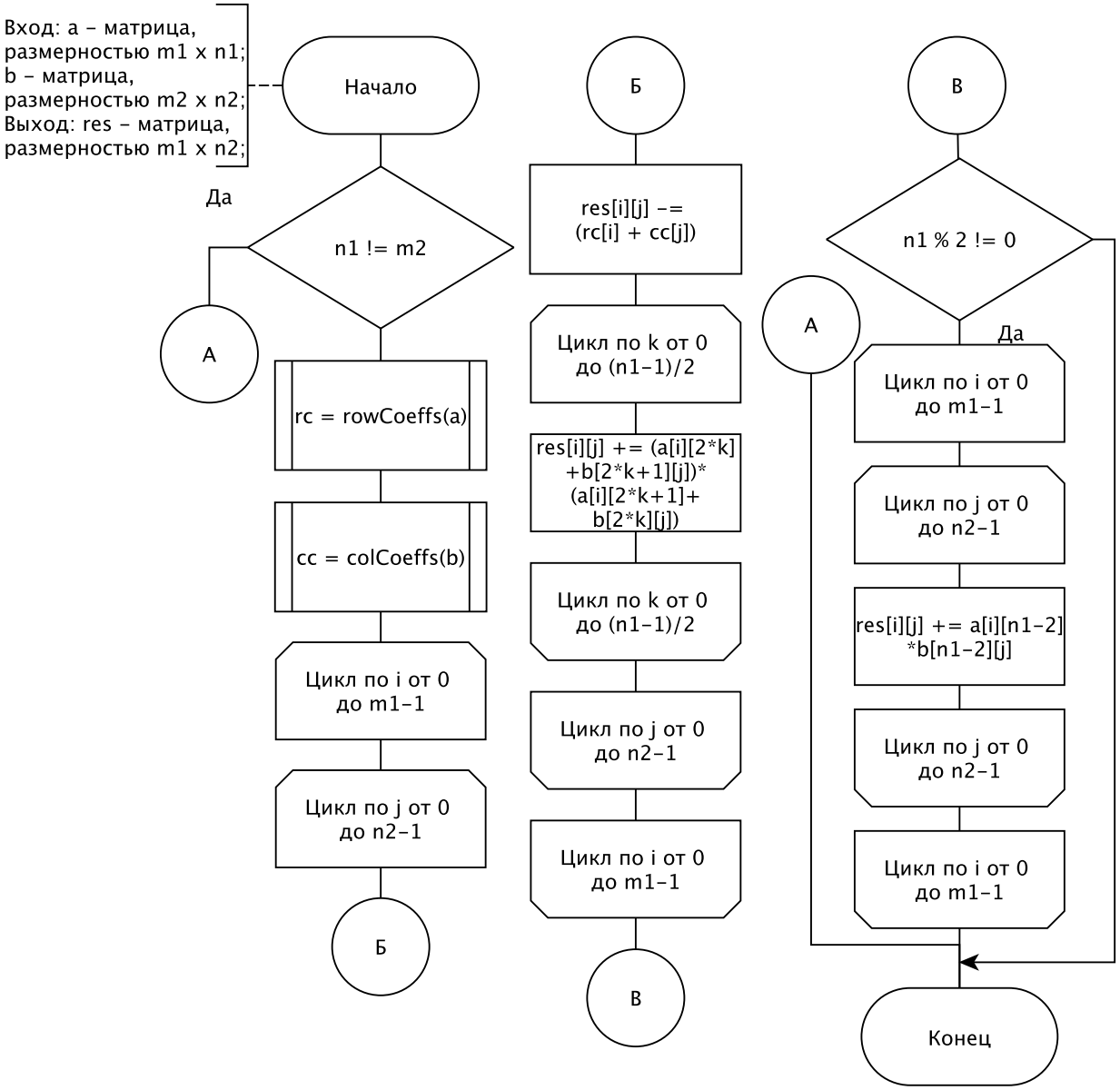

Рисунок 2.2 — Схема реализации алгоритма Винограда умножения матриц

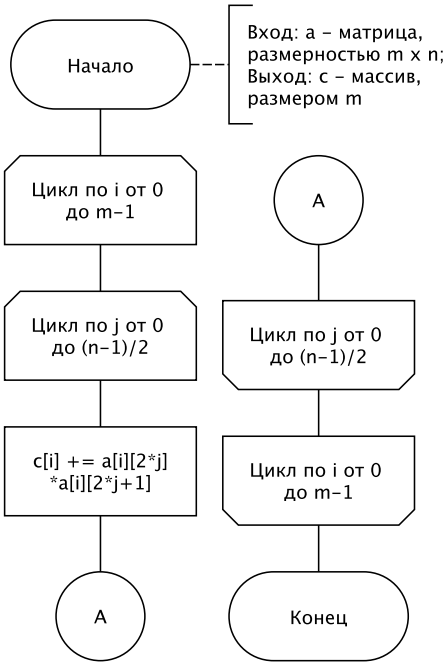

Рисунок 2.3 — Схема реализации алгоритма поиска произведений соседних элементов строк матрицы

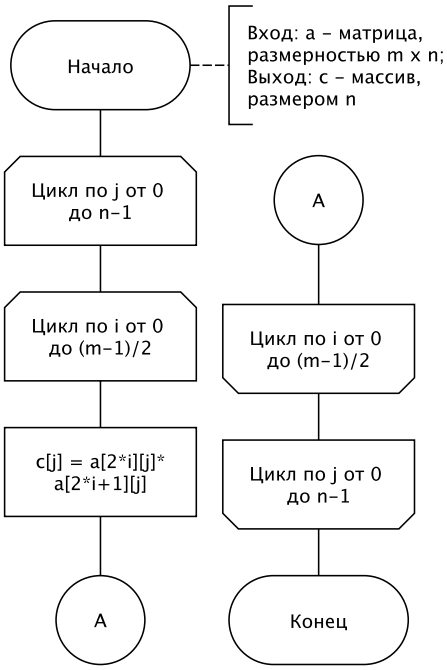

Рисунок 2.4 — Схема реализации алгоритма поиска произведений соседних элементов столбцов матрицы

2.3. Разработка оптимизированного алгоритма Винограда умножения матриц

На рисунке 2.5 представлена схема реализации оптимизированного алгоритма Винограда умножения матриц. На рисунке 2.6 представлены схемы реализаций алгоритмов подпрограмм улучшенного поиска произведений соседних элементов строк матрицы и улучшенного поиска произведений соседних элементов столбцов матрицы, необходимых для реализации оптимизированного алгоритма Винограда умножения матриц.

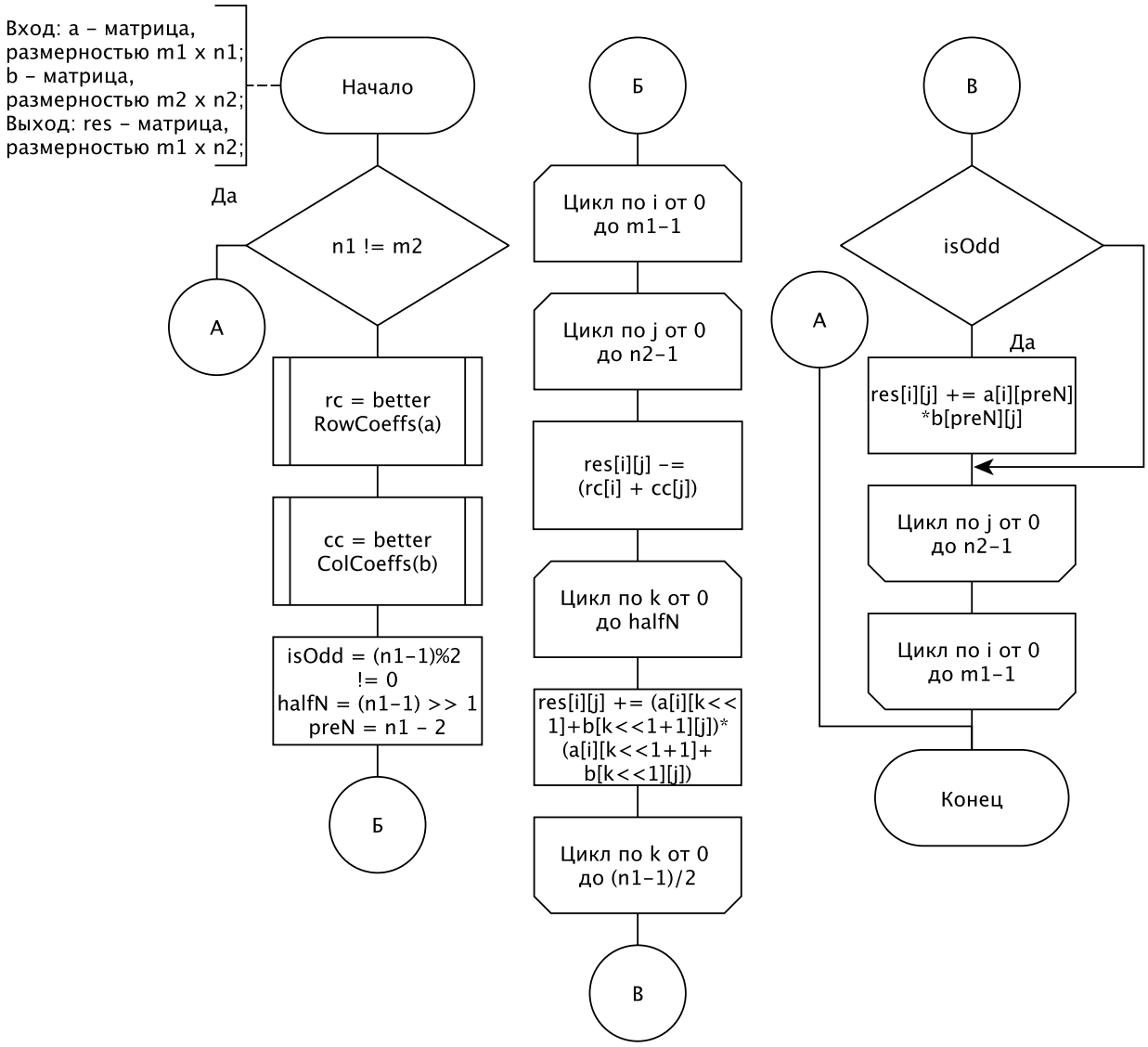

Рисунок 2.5 — Схема реализации оптимизированного алгоритма Винограда умножения матриц

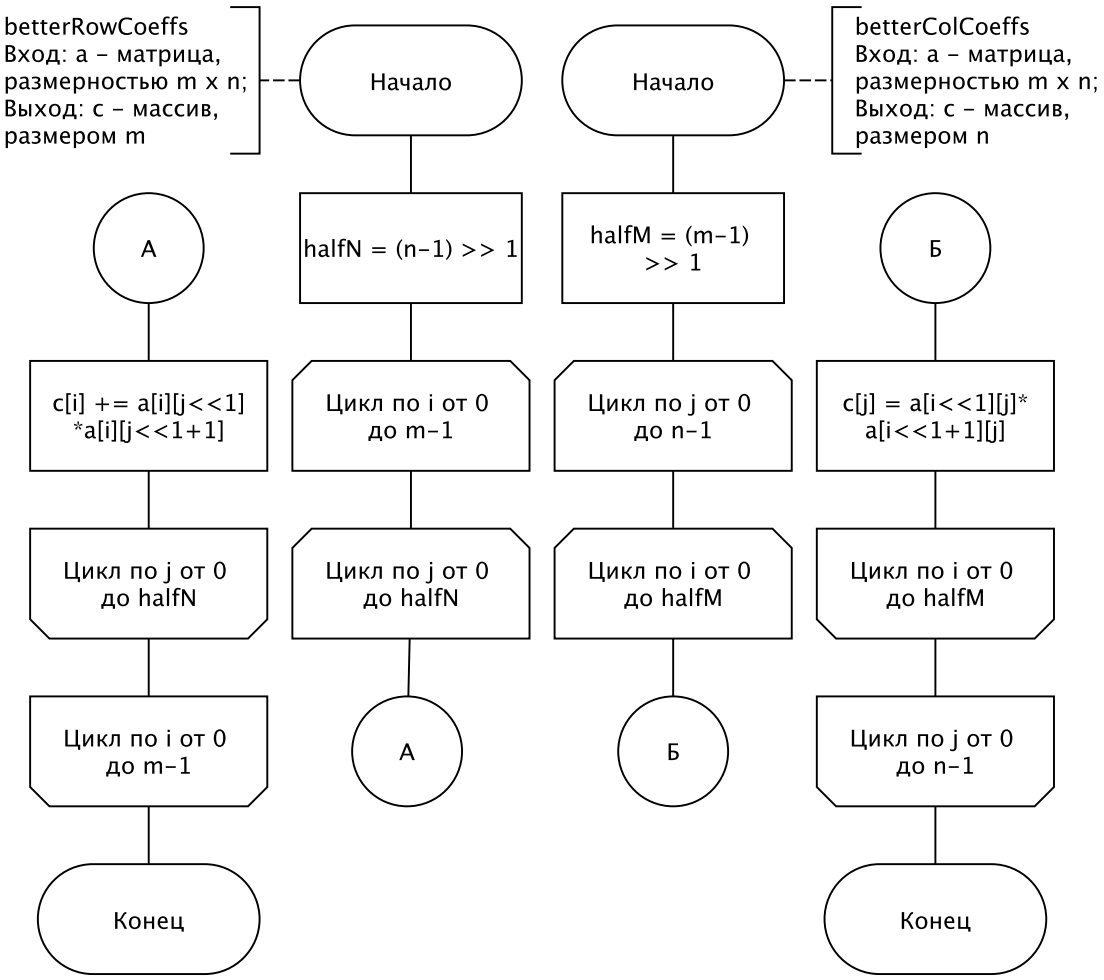

Рисунок 2.6 — Схема реализации улучшенного поиска произведений соседних элементов строк и столбцов матрицы

2.4. Модель вычислений

Для вычисления трудоёмкости данных алгоритмов необходимо ввести модель вычислений.

Обозначим трудоёмкость как f_a, где a — индекс, указывающий операцию, блок кода или оператор, для которого вычисляется трудоёмкость.

Определим трудоёмкость базовых операций как:

$$\begin{array}{rrrr} f_{+} = 1 & f_{-} = 1 & f_{+=} = 1 & f_{-=} = 1 \\ f_{:=} = 1 & f_{<<} = 1 & f_{>>} = 1 & f_{[]} = 1 \\ f_{++} = 1 & f_{--} = 1 & f_{>} = 1 & f_{<} = 1 \\ f_{>=} = 1 & f_{<=} = 1 & f_{!=} = 1 & f_{==} = 1 \\ f_{.} = 2 & f_{/} = 2 & f_{\%} = 2 & \end{array} \tag{2.1}$$

Определим трудоёмкость вызова функции как 0.

Определим трудоёмкость условия как

$$f_{if} = f_{cc} + \begin{cases} \min(f_1, f_2), & \text{в лучшем случае,} \\ \max(f_1, f_2), & \text{в худшем случае,} \end{cases} \tag{2.2}$$

где приняты следующие обозначения:

- f_{cc} — трудоёмкость вычисления условия;
- f_1 — трудоёмкость блока после if;
- f_2 — трудоёмкость блока после $else$.

Определим трудоёмкость цикла как

$$f_{loop} = f_{init} + f_{first-cmp} + n \cdot (f_{body} + f_{inc} + f_{cmp}), \tag{2.3}$$

где приняты следующие обозначения:

- f_{init} — трудоёмкость инициализации;
- $f_{first-cmp}$ — трудоёмкость первого сравнения;
- f_{body} — трудоёмкость тела цикла;
- n — количество итераций цикла;
- f_{inc} — трудоёмкость изменения индекса;
- f_{cmp} — трудоёмкость сравнения.

2.5. Трудоёмкость алгоритмов умножения матриц

Введём некоторые обозначения:

- M — размерность первой матрицы по количеству строк;
- N — размерность второй матрицы по количеству столбцов;
- K — размерность первой матрицы по количеству столбцов (второй матрицы по количеству строк);
- f_{best} — трудоёмкость алгоритма в лучшем случае;
- f_{worst} — трудоёмкость алгоритма в худшем случае.

2.5.1. Трудоёмкость стандартного алгоритма умножения матриц

Далее приведён расчёт трудоёмкость алгоритма алгоритма умножения матриц. Для данного алгоритма не выделяются худший и лучший случаи в связи с тем, что его быстродействие зависит лишь от размерности исходных матриц.

Трудоёмкость стандартного алгоритма умножения матриц включает в себя:

- трудоёмкость внешнего цикла по $i = \overline{0, M-1}$:

$$f_{i-loop} = 2 + M \cdot (2 + f_{body}); \tag{2.4}$$

- трудоёмкость внутреннего цикла по $j = \overline{0..N-1}$:

$$f_{j-loop} = 2 + N \cdot (2 + f_{body}); \tag{2.5}$$

- трудоёмкость внутреннего цикла по $k = \overline{0..K-1}$:

$$f_{k-loop} = 2 + K \cdot (2 + 12). \tag{2.6}$$

Так как трудоёмкость стандартного алгоритма умножения матриц равна трудоёмкости внешнего цикла, можно вычислить ее, подставив в (2.4) трудоёмкости внутренних циклов:

$$f_{standard} = 2 + M \cdot (4 + N \cdot (4 + 14 \cdot K)). \tag{2.7}$$

Таким образом, для стандартного алгоритма умножения матриц трудоёмкость составит:

$$f_{standard} = 2 + 4 \cdot M + 4 \cdot M \cdot N + 14 \cdot M \cdot N \cdot K \approx 14 \cdot M \cdot N \cdot K. \tag{2.8}$$

Такую трудоёмкость можно оценить как $O(M \cdot N \cdot K)$.

2.5.2. Трудоёмкость алгоритма Винограда для умножения матриц

Далее приведён расчёт трудоёмкости алгоритма Винограда для умножения матриц. Лучшим случаем считается ситуация, когда размерность исходной матрицы чётная, и худшим, когда нечётная.

Трудоёмкость алгоритма Винограда включает в себя:

- трудоёмкость создания и заполнения массивов $rowCoeffs$ и $colCoeffs$:

$$f_{rowCoeffs} = M + (2 + M \cdot (5 + \frac{K}{2} \cdot 18)), \tag{2.9}$$

$$f_{colCoeffs} = N + (2 + N \cdot (5 + \frac{K}{2} \cdot 18)); \tag{2.10}$$

- трудоёмкость цикла заполнения матрицы-результата для чётных размеров исходной матрицы:

$$f_{loop} = 2 + M \cdot (4 + N \cdot (14 + \frac{K}{2} \cdot 31)); \tag{2.11}$$

- трудоёмкость цикла для дополнения умножения суммой последних нечётных строки и столбца при нечётном размере исходной матрицы:

$$f_{loop-odd} = 3 + \begin{cases} 0, & \text{если размер чётный,} \\ 2 + M \cdot (4 + N \cdot 16), & \text{иначе.} \end{cases} \tag{2.12}$$

Таким образом, для худшего случая (нечётный общий размер матриц) трудоёмкость составит:

$$f_{Win-worst} \approx 15.5 \cdot M \cdot N \cdot K = O(M \cdot N \cdot K), \tag{2.13}$$

а лучшего случая (чётный общий размер матриц) трудоёмкость составит:

$$f_{Win-best} \approx 15.5 \cdot M \cdot N \cdot K = O(M \cdot N \cdot K). \tag{2.14}$$

2.5.3. Трудоёмкость оптимизированного алгоритма Винограда для умножения матриц

Рассчитаем трудоёмкость оптимизированного алгоритма Винограда для умножения матриц. Как и для обычного алгоритма Винограда (см. п. 2.5.2.), лучшим случаем считается ситуация, когда размерность исходной матрицы чётная, и худшим, когда нечётная. Применённые улучшения приведены в пункте 1.3.

Трудоёмкость оптимизированного алгоритма Винограда включает в себя:

- трудоёмкость создания и заполнения массивов $rowCoeffs$ и $colCoeffs$:

$$f_{rowOCoeffs} = M + 2 + (2 + M \cdot (4 + \frac{K}{2} \cdot 15)), \tag{2.15}$$

$$f_{colOCoeffs} = N + (2 + N \cdot (4 + \frac{K}{2} \cdot 15)); \tag{2.16}$$

- трудоёмкость предвычисления переменных: $f_{val} = 5$;

- трудоёмкость цикла заполнения матрицы-результата для чётных размеров исходной матрицы с учётом ситуации с нечётным размером:

$$f_{loopO} = 2 + M \cdot (4 + N \cdot (17 + \frac{K}{2} \cdot 23)). \tag{2.17}$$

Таким образом, для худшего случая (нечётный общий размер матриц) трудоёмкость составит:

$$f_{WinO-worst} \approx 11.5 \cdot M \cdot N \cdot K = O(M \cdot N \cdot K), \tag{2.18}$$

а лучшего случая (чётный общий размер матриц) трудоёмкость составит:

$$f_{WinO-best} \approx 11.5 \cdot M \cdot N \cdot K = O(M \cdot N \cdot K). \tag{2.19}$$

3. Технологическая часть

В данном разделе будет представлена реализация алгоритмов решения задачи коммивояджёра. Также будут указаны обязательные требования к ПО, средства реализации алгоритмов и результаты проведённого тестирования программы.

3.1. Требования к ПО

Для программы выделен перечень требований:

- предоставляется интерфейс в формате меню с возможностью ввода и изменения обрабатываемых матриц, завершения работы с программой и выбора используемого для умножения введённых матриц алгоритма;
- предлагается повторно выполнить ввод при невалидном выборе пункта меню;
- производится аварийное завершение с текстом об ошибке при иных ошибках;
- проводится модульное тестирование функций умножения матриц;
- производятся замеры времени выполнения и потребления памяти функциями умножения матриц;
- производятся расчёты только для матриц, содержащих целые числа.

3.2. Средства реализации

Для реализации данной работы был выбран язык программирования Go [2]. Выбор обусловлен наличием в *Go* библиотек для тестирования ПО и проведения замеров времени выполнения и потребления памяти функциями, а также необходимых для реализации поставленных цели и задач средств. В качестве среды разработки была выбрана *GoLand* [4].

3.3. Реализация алгоритмов

В листингах 3.1 – 3.5 представлены реализации алгоритмов умножения матриц: стандартного, Винограда и оптимизированного алгоритма Винограда, и некоторых нужных подпрограмм в листингах 3.6 – 3.9.

Листинг 3.1 — Листинг функции реализации алгоритма полного перебора для решения задачи коммивояджёра

```
func TravellingSalesmanBF(g graph.Graph) ([]int, int) {
        var minRoute []int
        verts := make([]int, g.Size)
        for i := range verts {
                verts[i] = i
        }

        permuts := permutations(verts)
        minTax := math.MaxInt
        for i := range permuts {
                if g.IsOKRoute(permuts[i]) {
                        cost := g.RouteTotalTax(permuts[i])

                        if cost < minTax {
                                minTax = cost
                                minRoute = permuts[i]
                        }
                }
        }

        return minRoute, minTax
}
```

Листинг 3.2 — Листинг функции реализации алгоритма Винограда умножения матриц

```
func WinogradMulMatrix(m1 matrix.Matrix, m2 matrix.Matrix)
(matrix.Matrix, error) {
        if m1.N != m2.M {
                return matrix.Matrix{}, errors.New("mul sizes error")
        }

        res := matrix.CreateEmpty(m1.M, m2.N)

        rowCoefs := rowCoefs(m1)
        columnCoefs := columnCoefs(m2)
```

Листинг 3.3 — Листинг функции реализации алгоритма Винограда умножения матриц (продолжение листинга 3.2)

```
        for i := 0; i < res.M; i++ {
                for j := 0; j < res.N; j++ {
                        res.Data[i][j] = res.Data[i][j] - rowCoefs[i] -
                         columnCoefs[j]

                        for k := 0; k < m1.N/2; k++ {
                                res.Data[i][j] = res.Data[i][j] +
                                (m1.Data[i][2*k]+m2.Data[2*k+1][j])*
                                        (m1.Data[i][2*k+1]+m2.Data[2*k][j])
                        }
                }
        }

        if m1.N%2 != 0 {
                for i := 0; i < res.M; i++ {
                        for j := 0; j < res.N; j++ {
                                res.Data[i][j] = res.Data[i][j] +
                                m1.Data[i][m1.N-1]*m2.Data[m1.N-1][j]
                        }
                }
        }

        return res, nil
}
```

Листинг 3.4 — Листинг функции реализации оптимизированного алгоритма Винограда умножения матриц

```
func WinogradBetterMulMatrix(m1 matrix.Matrix, m2 matrix.Matrix)
(matrix.Matrix, error) {
        if m1.N != m2.M {
                return matrix.Matrix{}, errors.New("mul sizes error")
        }
        res := matrix.CreateEmpty(m1.M, m2.N)
        rowCoefs := betterRowCoefs(m1)
        columnCoefs := betterColumnCoefs(m2)
```

Листинг 3.5 — Листинг функции реализации оптимизированного алгоритма Винограда умножения матриц (продолжение листинга 3.4)

```
        isOdd := m1.N%2 != 0
        halfN := m1.N >> 1
        preN := m1.N - 1

        for i := 0; i < res.M; i++ {
                for j := 0; j < res.N; j++ {
                        res.Data[i][j] -= rowCoefs[i] + columnCoefs[j]

                        for k := 0; k < halfN; k++ {
                                res.Data[i][j] += (m1.Data[i][k<<1] +
                                m2.Data[(k<<1)+1][j])*
                                (m1.Data[i][(k<<1)+1] + m2.Data[k<<1][j])
                        }

                        if isOdd {
                                res.Data[i][j] += m1.Data[i][preN] *
                                m2.Data[preN][j]
                        }
                }
        }

        return res, nil
}
```

Листинг 3.6 — Листинг функции реализации алгоритма поиска произведений соседних элементов строк матрицы

```
func rowCoefs(m matrix.Matrix) []int {
        coefs := make([]int, m.M)
        for i := 0; i < m.M; i++ {
                for j := 0; j < m.N/2; j++ {
                        coefs[i] = coefs[i] + m.Data[i][2*j]*m.Data[i][2*j+1]
                }
        }
        return coefs
}
```

Листинг 3.7 — Листинг функции реализации алгоритма поиска произведений соседних элементов столбцов матрицы

```
func columnCoefs(m matrix.Matrix) []int {
        coefs := make([]int, m.N)
        for j := 0; j < m.N; j++ {
                for i := 0; i < m.M/2; i++ {
                        coefs[j] = coefs[j] + m.Data[2*i][j]*m.Data[2*i+1][j]
                }
        }
        return coefs
}
```

Листинг 3.8 — Листинг функции реализации оптимизированного алгоритма поиска произведений соседних элементов строк матрицы

```
func betterRowCoefs(m matrix.Matrix) []int {
        coefs := make([]int, m.M)
        halfN := m.N >> 1
        for i := 0; i < m.M; i++ {
                for j := 0; j < halfN; j++ {
                        coefs[i] += m.Data[i][j<<1] * m.Data[i][(j<<1)+1]
                }
        }
        return coefs
}
```

Листинг 3.9 — Листинг функции реализации оптимизированного алгоритма поиска произведений соседних элементов столбцов матрицы

```
func betterColumnCoefs(m matrix.Matrix) []int {
        coefs := make([]int, m.N)
        halfM := m.M >> 1
        for j := 0; j < m.N; j++ {
                for i := 0; i < halfM; i++ {
                        coefs[j] += m.Data[i<<1][j] * m.Data[(i<<1)+1][j]
                }
        }
        return coefs
}
```

3.4. Тестирование

В таблице представлены тесты для алгоритмов умножения матриц: стандартного, Винограда и оптимизированного алгоритма Винограда. Тестирование проводилось по методологии чёрного ящика. Все тесты пройдены успешно.

Таблица 3.1 — Тесты для алгоритмов умножения матриц: стандартного, Винограда и оптимизированного алгоритма Винограда

№	Матрица A	Матрица B	Результат
1	$\begin{pmatrix}1\end{pmatrix}$	$\begin{pmatrix}1\end{pmatrix}$	$\begin{pmatrix}1\end{pmatrix}$
2	$\begin{pmatrix}1\end{pmatrix}$	$\begin{pmatrix}0\end{pmatrix}$	$\begin{pmatrix}0\end{pmatrix}$
3	$\begin{pmatrix}0 & 0 & 0\\0 & 0 & 0\end{pmatrix}$	$\begin{pmatrix}0 & 0\\0 & 0\\0 & 0\end{pmatrix}$	$\begin{pmatrix}0 & 0\\0 & 0\end{pmatrix}$
4	$\begin{pmatrix}0 & 3 & 0\\0 & 0 & 4\end{pmatrix}$	$\begin{pmatrix}0 & 5\\0 & 0\end{pmatrix}$	mul sizes error
5	$\begin{pmatrix}1 & 2\\3 & 4\\5 & 6\\7 & 8\end{pmatrix}$	$\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}$	$\begin{pmatrix}7 & 10\\15 & 22\\23 & 34\\31 & 46\end{pmatrix}$
6	$\begin{pmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{pmatrix}$	$\begin{pmatrix}1\\2\\3\end{pmatrix}$	$\begin{pmatrix}14\\32\\50\end{pmatrix}$
7	$\begin{pmatrix}5 & 0 & 0 & 1\\9 & 7 & 0 & 8\\3 & 6 & 4 & 7\\0 & 0 & 0 & 0\\2 & 2 & 8 & 2\end{pmatrix}$	$\begin{pmatrix}0 & 7 & 5\\0 & 0 & 0\\0 & 9 & 5\\0 & 1 & 0\end{pmatrix}$	$\begin{pmatrix}0 & 36 & 25\\0 & 71 & 45\\0 & 64 & 35\\0 & 0 & 0\\0 & 88 & 50\end{pmatrix}$

4. Экспериментальная часть

В данном разделе описаны проведённые замеры и представлены их результаты. Также будут уточнены характеристики устройства, на котором проводились замеры.

4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование [5]:

- операционная система macOS Monterey 12.4;
- 8 ГБ оперативной памяти;
- процессор Apple M2 (базовая частота — 2400 МГц, но поддержка технологии Turbo Boost позволяет достигать частоты в 3500 МГц [10]).

4.2. Измерение времени выполнения реализаций алгоритмов

Замеры времени реализаций алгоритмов производилось при помощи импортируемой в *Go* функции *clock_gettime* языка *C* [12]. Эта функция при использовании макроса $CLOCK\_PROCESS\_CPUTIME\_ID$ возвращает затраченное на работу процесса процессорное время в формате структуры *struct timespec*, в которой хранятся результаты замеров из двух частей — в секундах и наносекундах (см. листинг 4.10).

Листинг 4.10 — Листинг структуры *struct timespec*

```
struct timespec {
        time_t   tv_sec;        /* seconds */
        long     tv_nsec;       /* nanoseconds */
};
```

Количество повторений замера для одинаковых входных данных — 100.

Функция, возвращающая текущее процессорное время, приведена в листинге 4.11.

Листинг 4.11 — Листинг функции, возвращающей текущее процессорное время

```
#include <pthread.h>
#include <time.h>
#include <stdio.h>

static long long getCPUNs(){
        struct timespec time;
        if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time)) {
                perror("can't measure time");
                return 0;
        }
        return time.tv_sec * 1000000000LL + time.tv_nsec;
}
```

Функция единичного замера времени выполнения в наносекундах приведена в листинге 4.12, где $alg.function$ — объект типа функция (в данной реализации — функция, описывающая один из алгоритмов умножения матриц).

Листинг 4.12 — Листинг функции единичного замера времени выполнения в наносекундах

```
func RunBenchmark(m1, m2 matrix.Matrix, alg algorithm) int {
        t := 0

        start := C.getCPUNs()
        for i := 0; i < NumTests; i++ {
                alg.function(m1, m2)
        }
        end := C.getCPUNs()
        t = int(end-start) / NumTests

        return t
}
```

Входные матрицы для проведения замеров заполняются случайными числами с помо-

щью функции, реализация которой представлена в листинге 4.13.

Листинг 4.13 — Листинг примера функции для заполнения матрицы случайными числами в диапазоне от -500 до 500

```
const (
        maxNum   = 1000
        interval = 500
)

func FillRandom(m Matrix) {
        for i := 0; i < m.M; i++ {
                for j := 0; j < m.N; j++ {
                        m.Data[i][j] = rand.Intn(maxNum+1) - interval
                }
        }
}
```

Результаты замеров времени выполнения (в нс) приведены в таблицах 4.1 — 4.2. Оптимизации компилятора были отключены с помощью флага $-gcflags' - N - l'$. Для стандартного алгоритма умножения матриц невозможно выделить худший и лучший случай в зависимости от входных данных, поэтому понятия «худший» и «лучший случай», применительно к нижеописанным результатам замеров, относятся только к реализациям алгоритма Винограда. На рисунках 4.1 — 4.4 приведены графики, отображающие зависимость времени работы алгоритмов от размера матриц для лучшего и худшего случаев. Результаты замеров для реализаций алгоритмов Винограда вынесены на отдельный рисунок для каждого случая, так как разница в быстродействии невелика, что делает невозможность наглядно продемонстрировать её на большом диапазоне значений размерностей входных матриц. Матрицы заполнялись случайными числами.

Таблица 4.1 — Результаты замеров времени для лучшего случая (чётная размерность) (нс)

Размерность матриц	Стандартный	Виноград	Опт. Виноград
2	180	330	280
10	9220	3350	2790
50	503 320	277 690	260 000
100	4 261 770	2 178 990	2 159 120
200	36 068 900	19 995 980	19 371 750
300	130 022 190	74 695 990	71 111 840
500	599 444 020	334 126 470	327 040 510

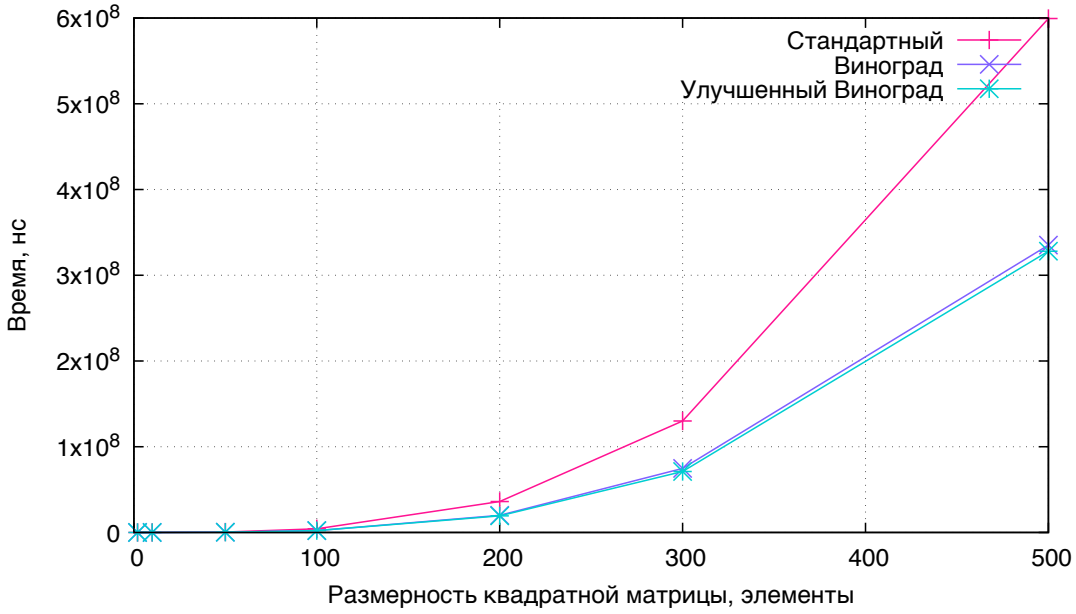

Рисунок 4.1 — Зависимость времени работы алгоритмов умножения матриц от размерности матриц (лучший случай)

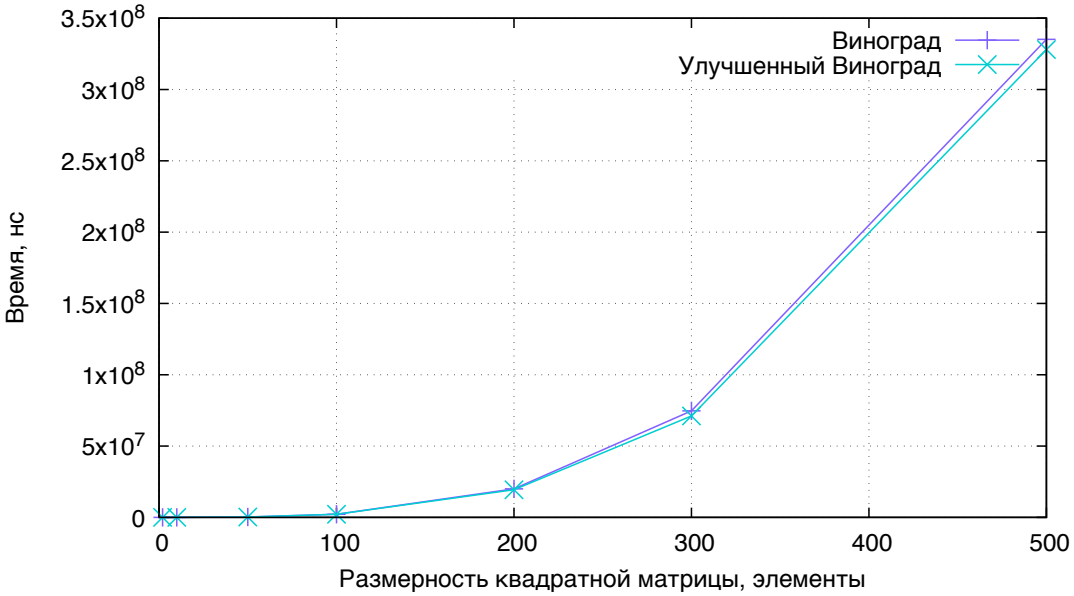

Рисунок 4.2 — Зависимость времени работы алгоритмов умножения матриц от размерности матриц (лучший случай, алгоритм Винограда)

Таблица 4.2 — Результаты замеров времени для худшего случая (нечётная размерность) (нс)

Размерность матриц	Стандартный	Виноград	Опт. Виноград
2	330	430	333
10	4940	3890	3590
50	538 720	290 780	288 490
100	4 401 290	2 264 800	2 223 520
200	36 774 650	20 298 640	19 554 690
300	131 696 540	74 210 140	71 339 110
500	604 239 010	335 168 220	328 529 920

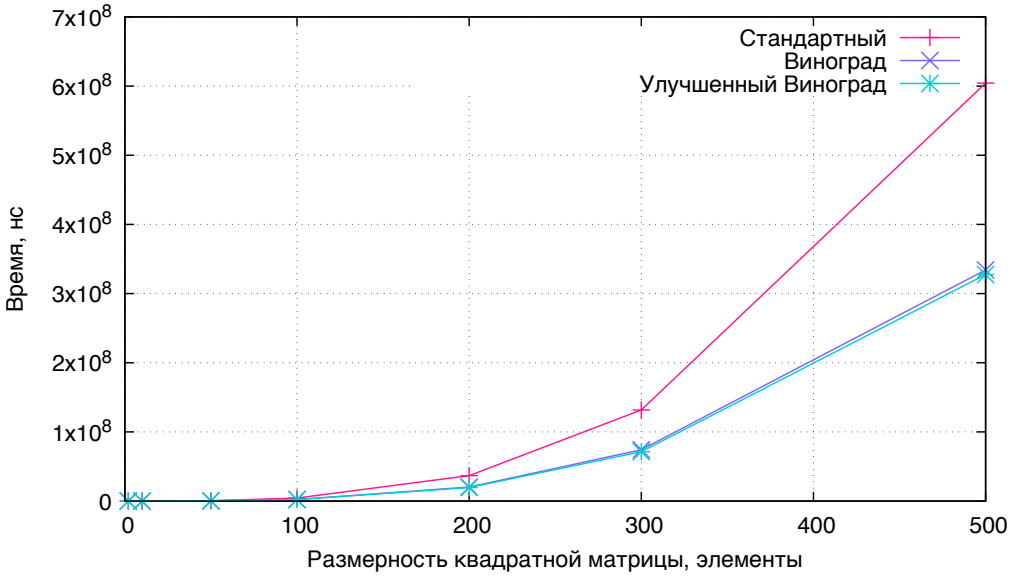

Рисунок 4.3 — Зависимость времени работы алгоритмов умножения матриц от размерности матриц (худший случай)

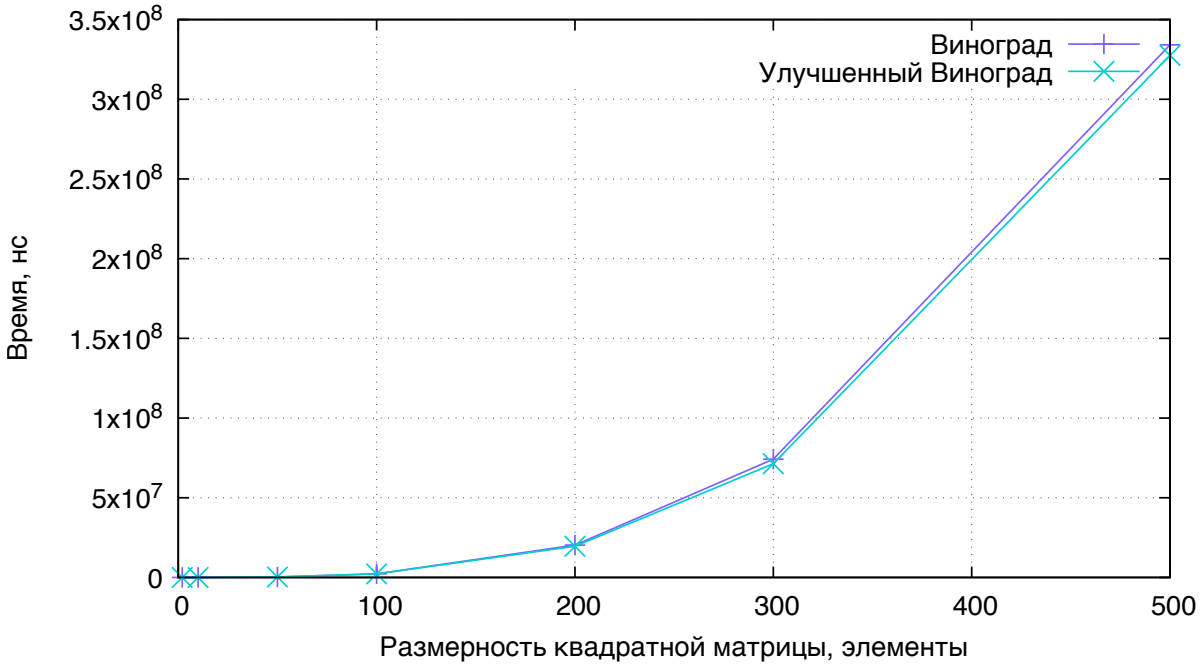

Рисунок 4.4 — Зависимость времени работы алгоритмов умножения матриц от размерности матриц (худший случай, алгоритм Винограда)

4.3. Измерение объёма потребляемой памяти реализаций

Измерение объёма потребляемой памяти производилось посредством создания собственного программного модуля memory, использующего функции пакета unsafe языка *Go* [3]. Реализация основного функционала модуля, а также пример функции расчёта памяти, затрачиваемой на стандартный алгоритм умножения матриц, и пример использования указанных функций приведены в Приложении А.

Результаты замеров потребляемой памяти (в байтах) приведены в таблице 4.3. На рисунке 4.5 приведён график, отображающий зависимость потребляемой памяти от размерности матриц. Матрицы заполнялись случайными числами. Результаты замеров на графике представлены на диапазоне размерностей матриц до 100 из-за незначительности разницы в потреблении памяти, что не позволяет наглядно продемонстрировать её на большом диапазоне значений размерностей входных матриц.

Таблица 4.3 — Результаты замеров потребляемой памяти (в байтах)

Размерность матриц	Стандартный	Виноград	Опт. Виноград
2	272	544	552
10	1232	1632	1640
50	21 392	22 432	22 440
100	82 592	84 432	84 440
200	324 992	328 432	328 440
300	727 392	732 432	732 440
500	2 012 192	2 020 432	2 020 440

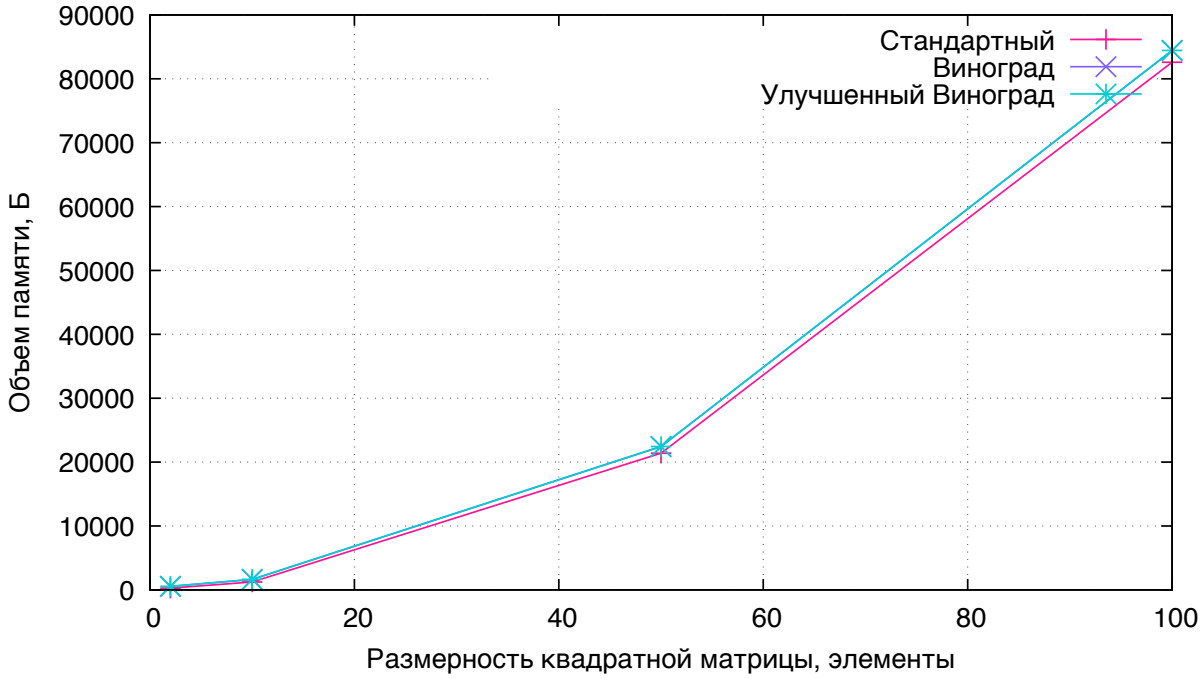

Рисунок 4.5 — Зависимость потребляемой памяти от размерности матриц при умножении

Заключение

Для каждого алгоритма, кроме стандартного, время работы и объём потребляемой памяти для которого зависит только от размерности обрабатываемой матрицы, были выявлены лучший и худший случаи (данные, при которых алгоритмы работают быстрее всего и медленнее всего, соответственно).

Для алгоритма Винограда лучшим случаем считается тот, при котором на вход подаются матрицы с нечётной размерностью, так как это влечёт за собой выполнение дополнительного цикла для коррекции результата, полученного после выполнения основного цикла.

В результате проведения замеров времени выполнения и потребляемой памяти с учётом разных случаев были сформулированы нижеперечисленные выводы.

Для любых случаев время работы стандартного алгоритма превосходит время работы алгоритмов Винограда. Например, для лучших случаев алгоритмов Винограда, стандартный на размерности матриц в 10 элементов работает медленнее в 1.2 раза, а на размерности 500 в 1,7 раза. Для худших случаев при тех же условиях разница составляет 1,3 и 1,7 раза.

В лучших случаях скорость работы алгоритма Винограда превышает скорость работы этого же алгоритма для худших случаев: для размерности 10 в 1.95 раза, для размерности 500 в 1.003 раза. Для оптимизированного алгоритма Винограда такая разница составляет 1.009 раза для размерности 10 и 1.004 раза для размерности 500.

В лучших случаях скорость работы оптимизированного алгоритма Винограда превышает скорость работы неоптимизированного алгоритма в 1.17 раза для размерности 10 и в 1.02 раза для размерности 500. Для худших случаев такая разница составляет 1.08 раза для размерности 10 и 1.03 раза для размерности 500.

Если говорить об объёме занимаемой памяти, то стандартный алгоритм выигрывает по сравнению с реализациями алгоритма Винограда только на размерностях матриц до 10 элементов. Например, при размерности 2 стандартный алгоритм потребляет примерно в 1.2 раза меньше памяти, чем реализации алгоритма Винограда. На размерности в 50 элементов стандартный алгоритм потребляет в 1.9 раза больше памяти, чем реализации алгоритма Винограда. Разница между двумя реализациями незначительна и составляет в среднем 1.01 раза в пользу неоптимизированного алгоритма.

В ходе выполнения лабораторной работы была достигнута поставленная цель: были получены навыки программирования, тестирования полученного программного продукта и проведения замеров по результатам работы программы на примере реализации алгоритмов

умножения матриц.

В процессе выполнения лабораторной работы были также реализованы все поставленные задачи, а именно:

- были изучены алгоритмы стандартного умножения матриц, Винограда и оптимизированный алгоритм Винограда;
- были разработаны схемы данных алгоритмов и проведён анализ их трудоёмкости;
- была выполнена программная реализация данных алгоритмов;
- были проведены замеры потребления памяти (в байтах) для данных алгоритмов;
- были проведены замеры времени работы (в нс) данных алгоритмов;
- была получены графическая зависимость замеряемых величин от размерности квадратной матрицы предоставляемой на вход алгоритмам;
- был проведён сравнительный анализ данных алгоритмов на основе полученных зависимостей.

Список использованных источников

[1] Быстрое умножение матриц и смежные вопросы алгебры [Электронный ресурс]. Режим доступа: *http://www.mathnet.ru/links/8d77bce10947b9bc83ba153fee3e56f9/sm8833.pdf* (дата обращения: 10.10.2022).

[2] Документация по языку программирования *Go* [Электронный ресурс]. Режим доступа: *https://go.dev/doc/* (дата обращения: 10.10.2022).

[3] Документация по пакетам языка программирования *Go* [Электронный ресурс]. Режим доступа: *https://pkg.go.dev* (дата обращения: 10.10.2022).

[4] GoLand: IDE для профессиональной разработки на *Go* [Электронный ресурс]. Режим доступа: *https://www.jetbrains.com/ru-ru/go/* (дата обращения: 10.10.2022).

[5] Техническая спецификация ноутбука *MacBookAir* [Электронный ресурс]. Режим доступа: *https://support.apple.com/kb/SP869* (дата обращения: 10.10.2022).

[6] Реализация алгоритма умножения матриц по винограду на языке Haskell [Электронный ресурс]. Режим доступа: *https://cyberleninka.ru/article/n/realizatsiya-algoritma-umnozheniya-matrits-po-vinogradu-na-yazyke-haskell* (дата обращения: 10.10.2022).

[7] Multiplying matrices in $O(n^{2.373})$ time [Электронный ресурс]. Режим доступа: *http://theory.stanford.edu/ virgi/matrixmult-f.pdf* (дата обращения: 10.10.2022).

[8] On the Complexity of Matrix Multiplication [Электронный ресурс]. Режим доступа: *https://era.ed.ac.uk/bitstream/handle/1842/4734/Stothers2010.pdf* (дата обращения: 10.10.2022).

[9] Discovering novel algorithms with AlphaTensor [Электронный ресурс]. Режим доступа: *https://www.deepmind.com/blog/discovering-novel-algorithms-with-alphatensor* (дата обращения: 10.10.2022).

[10] *AppleM2* [Электронный ресурс]. Режим доступа: *https://www.notebookcheck.net/Apple-M2-Processor-Benchmarks-and-Specs.632312.0.html* (дата обращения: 10.10.2022).

[11] Исходный код *src/testing/benchmark.go* [Электронный ресурс]. Режим доступа: *https://go.dev/src/testing/benchmark.go* (дата обращения: 10.10.2022).

[12] clock_gettime(3) — Linux manual page [Электронный ресурс]. Режим доступа: *https://man7.org/linux/man-pages/man3/clock_gettime.3.html* (дата обращения: 10.10.2022).

Приложение А

В ходе выполнения лабораторной работы в соответствии с поставленными задачами было необходимо произвести замеры потребляемой при выполнении функций, реализующих заданные алгоритмы, памяти. В связи с этим был разработан программный модуль *memory* для измерения потребляемой функцией памяти в байтах (замеры реализованы только для функций, представляющих алгоритмы по заданию лабораторной работы).

При вызове функции в языке *Go* для неё выделяется область собственного стека, что особенно критично для рекурсивных алгоритмах. Соответственно, необходимо иметь возможность измерять потребляемую память и на стеке, так как в ином случае не будут получены реалистичные результаты замеров потребления памяти и построить зависимость, отражающую действительное потребление памяти в зависимости от размерности квадратной матрицы, будет невозможно.

В листингах 5.1 — 5.3 приведена реализация основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах.

Листинг 5.1 — Листинг основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах (начало)

```
package algorithms

import (
        "unsafe"
)

var MemoryInfo Metrics

type Metrics struct {
        current int
        max     int
}
```

Листинг 5.2 — Листинг основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах (продолжение листинга 5.1)

```
// сброс рассчитанных значений
func (m *Metrics) Reset() {
        m.current = 0
        m.max = 0
}

// добавление значения к общей сумме потребляемой памяти,
// обновление максимума
func (m *Metrics) Add(v int) {
        m.current += v
        if m.current > m.max {
                m.max = m.current
        }
}

// вычитание значения из общей суммы потребляемой памяти
func (m *Metrics) Done(v int) {
        m.current -= v
}

// получение значения макс. потребления памяти за выполнение функции
func (m *Metrics) Max() int64 {
        return int64(m.max)
}

// получение размера типа данных
// пример вызова: sizeOf[int]()
func sizeOf[T any]() int {
        var v T
        return int(unsafe.Sizeof(v))
}
```

Листинг 5.3 — Листинг основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах (окончание листинга 5.2)

```
// получение полного размера среза (заголовок + элементы)
// пример вызова: sizeOfArray[int](10)
func sizeOfArray[T any](n int) int {
        return sizeOf[[]T]() + n*sizeOf[T]()
}

// получение полного размера матрицы (заголовок +
//заголовки массивов + элементы)
// sizeOfMatrix[int](10, 10)
func sizeOfMatrix[T any](m, n int) int {
        return sizeOf[[][]T]() + m*sizeOf[[]T]() + m*n*sizeOf[T]()
}
```

В листинге 5.4 приведена реализация одной из функций (в данном случае функции, реализующей алгоритм блинной сортировки), вычисляющих потребление памяти конкретной функцией, модуля *memory*.

Листинг 5.4 — Листинг функции, вычисляющей потребление памяти функцией, реализующей стандартный алгоритм умножения матриц

```
func MemoryUsual(m, n int) int {
        args := 3 * sizeOf[[][]int]()
        res := sizeOf[[][]int]()
        loop := 3 * sizeOf[int]()
        create := 3*sizeOf[int]() + sizeOfMatrix[int](m, n)
        + sizeOf[[][]int]()
        return args + res + loop + create
}
```

В листинге 5.5 приведен пример использования функций модуля *memory* в реализации стандартного алгоритма умножения матриц.

Листинг 5.5 — Листинг использования функций модуля *memory* в реализации стандартного алгоритма умножения матриц

```
func Mul(m1, m2 Matrix) (Matrix, error) {
        memory.MemoryInfo.Reset()
        memory.MemoryInfo.Add(memory.MemoryUsual(m1.M, m2.N))
        defer memory.MemoryInfo.Done(memory.MemoryUsual(m1.M, m2.N))

        ...
}
```

Таким образом, данный модуль позволяет измерить полное потребление памяти функциями, реализующими алгоритмы умножения матриц, и получить реалистичные данные по результатам измерений.