

МГТУ им. БАУМАНА

ЛАБОРАТОРНАЯ РАБОТА №2

По курсу: "АНАЛИЗ АЛГОРИТМОВ"

**Оценка трудоемкости
алгоритмов на примере
алгоритмов умножения матриц**

Работу выполнил: Левушкин Илья, ИУ7-52Б

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

Москва, 2019

Оглавление

Введение	3
1 Цель	6
2 Задачи	7
3 Аналитическая часть	8
3.1 Определение	8
3.2 Алгоритм Виноградова	9
3.2.1 Оптимизированный алгоритм Винограда	10
3.2.2 Модель вычислений	10
3.3 Выводы по аналитической части	11
4 Конструкторская часть	12
4.1 Реализация алгоритмов	12
4.1.1 Стандартный алгоритм	12
4.1.2 Алгоритм Виноградова	12
4.1.3 Список оптимизаций алгоритма Виноградова	12
4.2 Схемы алгоритмов	13
4.3 Оценка трудоемкости	23
4.3.1 Классический алгоритм	23
4.3.2 Алгоритм Виноградова	23
4.3.3 Оптимизированный алгоритм Винограда	23
4.4 Замер используемой памяти	24
4.5 Требования к программе	24
4.6 Выводы по конструкторской части	25

5	Технологическая часть	26
5.1	Средства реализации	26
5.2	Сведения о модулях программы	26
5.3	Тесты	36
5.4	Выводы по технологической части	38
6	Экспериментальная часть	39
6.1	Сравнение работы алгоритмов при чётных размерах матриц	39
6.2	Сравнение работы алгоритмов при нечетных размерах матриц	41
6.3	Выводы по экспериментальной части	43
	Литература	45

Введение

Для решения любой сколь угодно простой задачи можно написать программу, которая будет работать сколь угодно медленно. (Афоризм, рожденный из практики приема курсовых работ.)

Скорость выполнения программы (или производительность) зависят от многих факторов: языка программирования, способа реализации транслятора (компилятор, интерпретатор), производительности процессора. Поэтому заранее оценить производительность еще не написанной программы сложно. Но для уже разработанного алгоритма и написанной программы оценить перспективы ее использования с различными объемами данных вполне реально. Для этого и вводится понятие трудоемкости.

Трудоемкость программы (алгоритма) – это зависимость количества массовых операций (сравнения, обмены, сдвиги, повторения цикла и т.п.) от объема (размерностей) обрабатываемых данных.

Самое важное, что трудоемкость напрямую не связана со временем выполнения программы, но является мерой затрат на ее выполнение. Отметим наиболее важные свойства трудоемкости:

- трудоемкость определяется отдельно для каждого вида операций;
- трудоемкость может зависеть от входных данных. Поэтому оценка трудоемкости дается для лучшего и худшего случая, а также в среднем T_{\min} , T_{\max} , $T_{\text{ср}}$. Свойство программы – иметь различную трудоемкость для разных данных, называется чувствительностью к данным;

- на практике обычно используется грубая оценка трудоемкости, основанная на понятии скорости (степени) роста функции.

Для грубой оценки трудоемкости есть свои основания. Дело в том, что размерности исходных данных (N) меняются в программах в широких пределах: на несколько порядков при отладке программы и ее работе в реальных условиях. Поэтому для функции трудоемкости важно ее асимптотическое поведение при достаточно больших N . Говорится, что функция $T(N)$ имеет степень роста $G(N)$, если

$$\exists C, N_0 : \forall N > N_0 : C * G(N) > T(N) \quad (1)$$

, то есть начиная с некоторого N_0 функция $G(N)$ всегда превышает $T(N)$ с некоторым коэффициентом пропорциональности.

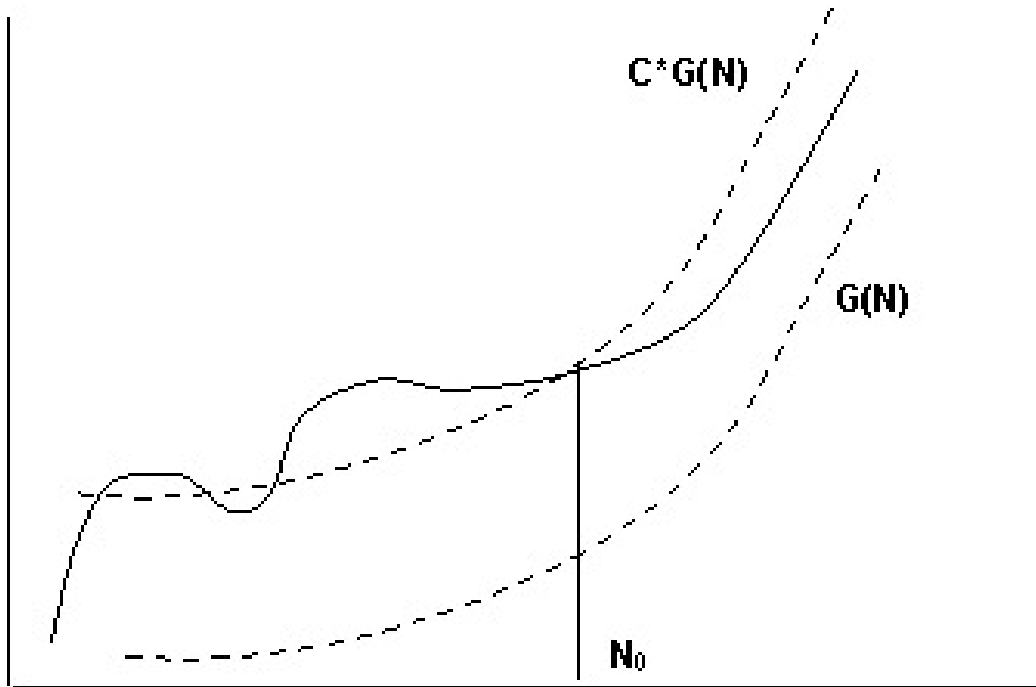


Рис. 1: Определение степени роста функции $T(N)$.

Умножение матриц — это один из базовых алгоритмов, который широко применяется в различных численных методах, и в частности в

алгоритмах машинного обучения. Многие реализации прямого и обратного распространения сигнала в сверточных слоях нейронной сети базируются на этой операции. Так порой до 90-95% всего времени, затрачиваемого на машинное обучение, приходится именно на эту операцию. Почему так происходит? Ответ кроется в очень эффективной реализации этого алгоритма для процессоров, графических ускорителей (а в последнее время и специальных ускорителей матричного умножения). Матричное умножение — один из немногих алгоритмов, которые позволяют эффективно задействовать все вычислительные ресурсы современных процессоров и графических ускорителей. Поэтому не удивительно, что многие алгоритмы стараются свести к матричному умножению — дополнительные расходы, связанные с подготовкой данных, как правило с лихвой окупаются общим ускорением алгоритмов.

Так как реализован алгоритм матричного умножения? На сколько сильно одни алгоритмы умножения матриц могут быть эффективнее других алгоритмов? В данной лабораторной работе будут рассмотрены три различных алгоритма умножения матриц: стандартный, алгоритм Виноградова и его оптимизированная версия с последующей оценкой трудоемкости данных алгоритмов.

1 | Цель

Целью данной лабораторной работы является изучение алгоритмов умножения матриц:

- стандартный;
- алгоритм Виноградова.

С последующей оценкой трудоемкости получившихся алгоритмов.

2 | Задачи

Для достижение поставленной цели необходимо решить следующие задачи:

- проанализировать и разработать существующие реализации стандартного алгоритма и алгоритма Виноградова;
- также разработать оптимизированную версию алгоритма Виноградова;
- выбрать технологии для последующих реализаций и исследования алгоритмов;
- реализовать эти алгоритмы;
- произвести тестирование корректности работы реализаций;
- произвести оценку трудоемкости этих алгоритмов;
- сравнить быстродействие реализаций;
- описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

3 | Аналитическая часть

3.1 Определение

Пусть даны две прямоугольные матрицы A и B размерности $l \times m$ и $m \times n$ соответственно:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \cdots & a_{lm} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}. \quad (3.1)$$

Тогда матрица C размерностью $l \times n$:

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \cdots & c_{ln} \end{bmatrix} \quad (3.2)$$

в которой:

$$c_{ij} = \sum_{r=1}^m a_{ir} b_{rj} \quad (i = 1, 2, \dots, l; j = 1, 2, \dots, n). \quad (3.3)$$

называется их произведением.

Операция умножения двух матриц выполнима только в том случае, если число столбцов в первом сомножителе равно числу строк во втором; в этом случае говорят, что матрицы согласованы. В частности, умножение всегда выполнимо, если оба сомножителя — квадратные матрицы одного и того же порядка.

Таким образом, из существования произведения AB вовсе не следует существование произведения BA .

3.2 Алгоритм Виноградова

Исходя из равенства 3.2, видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее. [3]

Рассмотрим два вектора U и V :

$$U = A_i = (u_1, u_2, \dots, u_n), \quad (3.4)$$

где $U = A_i$ – i -ая строка матрицы A ,
 $u_k = a_{ik}, k = 1 \dots n$ – элемент i -ой строки k -ого столбца матрицы A .

$$V = B_j = (v_1, v_2, \dots, v_n), \quad (3.5)$$

где $V = B_j$ – j -ый столбец матрицы B ,
 $v_k = b_{kj}, k = 1 \dots n$ – элемент k -ой строки j -ого столбца матрицы B .

По определению их скалярное произведение равно:

$$U \cdot V = u_1v_1 + u_2v_2 + u_3v_3 + u_4v_4. \quad (3.6)$$

Равенство 3.6 можно переписать в виде:

$$U \cdot V = (u_1 + v_2)(u_2 + v_1) + (u_3 + v_4)(u_4 + v_3) - u_1u_2 - u_3u_4 - v_1v_2 - v_3v_4. \quad (3.7)$$

В равенстве 3.6 насчитывается 4 операции умножения и 3 операции сложения, в равенстве 3.7 насчитывается 6 операций умножения и 9 операций сложения. Однако выражение $-u_1u_2 - u_3u_4$ используются повторно при умножении i -ой строки матрицы A на каждый из столбцов матрицы B , а выражение $-v_1v_2 - v_3v_4$ - при умножении j -ого столбца матрицы B на строки матрицы A . Таким образом, данные выражения можно вычислить предварительно для каждой строк и столбцов матриц для сокращения повторных вычислений. В результате повторно будут выполняться

лишь 2 операции умножения и 7 операций сложения (2 операции нужны для добавления предварительно посчитанных произведений).

3.2.1 Оптимизированный алгоритм Винограда

Для оптимизации алгоритма Винограда могут использоваться такие стратегии, как:

- предварительные вычисления повторяющихся одинаковых действий;
- использование более быстрых операций при вычислении (такие, как сдвиг битов вместо умножения или деления на 2);
- уменьшения количества повторных проверок;
- использование аналогичных конструкций, уменьшающих трудоёмкость операций (к примеру, замена сложения с 1 на инкремент).

Ниже представлен список личностей, проводивших оптимизацию алгоритма:

- в 2010 Эндрю Стотерс усовершенствовал алгоритм до $O(n^{2.374})$;
- в 2011 году Вирджиния Уильямс усовершенствовала алгоритм до $O(n^{2.3728642})$;
- в 2014 году Франсуа Ле Галль упростил метод Уильямса и получил новую улучшенную оценку $O(n^{2.3728639})$.

3.2.2 Модель вычислений

В рамках данной работы используется следующая модель вычислений:

- операции, имеющие трудоёмкость 1: $<$, $>$, $=$, $<=$, $=>$, $==$, $!=$, $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $[]$;
- оператор условного перехода имеет трудоёмкость, равную трудоёмкости операторов тела условия;

- оператор цикла `for` имеет трудоемкость:

$$F_{for} = F_{init} + F_{check} + N * (F_{body} + F_{inc} + F_{check}), \quad (3.8)$$

где F_{init} – трудоёмкость инициализации, F_{check} – трудоёмкость проверки условия, F_{inc} – трудоёмкость инкремента аргумента, F_{body} – трудоёмкость операций в теле цикла, N – число повторений. [4]

3.3 Выводы по аналитической части

В рамках данной работы были выбраны для исследования стандартный алгоритм, алгоритм Виноградова, а также оптимизированная его версия, умножения матриц. Были расписаны способы оптимизации алгоритма Винограда (3.2.1) и выбрана модель вычислений по которой будет оцениваться трудоемкость алгоритмов (3.2.2).

4 | Конструкторская часть

4.1 Реализация алгоритмов

4.1.1 Стандартный алгоритм

В стандартном алгоритме мы перемножаем каждый вектор-строку первой матрицы на каждый вектор-столбец второй матрицы. То есть в первом цикле мы проходимся по всем строкам первой матрицы, во втором цикле проходимся по всем столбцам второй матрицы, и в третьем цикле мы проходимся по всем столбцам первой/строкам второй матрицы. При этом используем формулу 3.3 для получения ответа.

4.1.2 Алгоритм Виноградова

В алгоритме Виноградова мы также перемножаем каждый вектор-строку первой матрицы на каждый вектор-столбец второй матрицы, только вместо того, чтобы делать это по отдельности, для каждого i -ого элемента: $u_i v_i$, мы считаем сумму сразу по 2 элемента: $u_i v_i + u_{i+1} v_{i+1}$. Преобразовав это выражение к: $(u_i + v_{i+1})(u_{i+1} + v_i) - u_i u_{i+1} - v_i v_{i+1}$, и посчитав $-u_i u_{i+1} - v_i v_{i+1}$ заранее (вне основного цикла), получим всего 1 операцию умножения и 2 операции сложения вместо двух операций умножения и одной операции сложения.

4.1.3 Список оптимизаций алгоритма Виноградова

Алгоритм Винограда был оптимизирован с помощью следующих модификаций:

- вычисление повторно используемых величин выполняется предварительно ($N/2$, $N - 1$, $2 * k$);
- проверка на четность/нечетность матрицы происходит до основного цикла, и в зависимости от результата выбирается один из основных циклов: для четной и нечетной матрицы (цикл вычислений для нечётных элементов включён в основной цикл, добавив дополнительные операции). Данная модификация исключает повторные проверки на нечётность N ;
- вместо обращения к ij -ому элементу массива результирующей матрицы на каждой итерации, в основном цикле используется указатель на i -ый элемент массива, чтобы во втором цикле исключить лишнее обращение к i -ому элементу массива;
- используется буфер `buf` для уменьшения количества обращений к результирующей матрице.

4.2 Схемы алгоритмов

На рисунках 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9 представлены схемы реализаций стандартного алгоритма, алгоритма Винограда и его оптимизированный вариант.

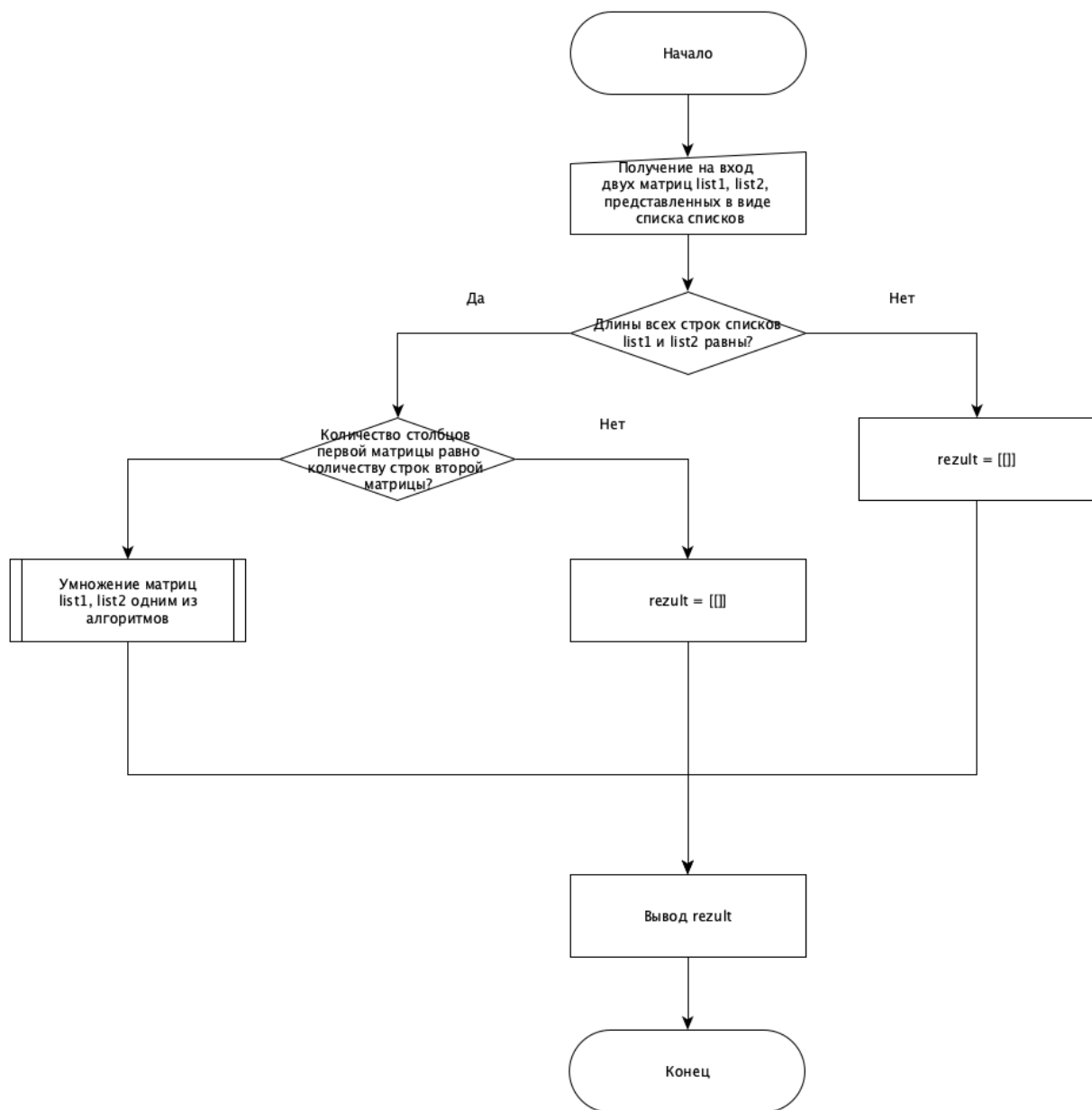


Рис. 4.1: Алгоритм умножения матриц.

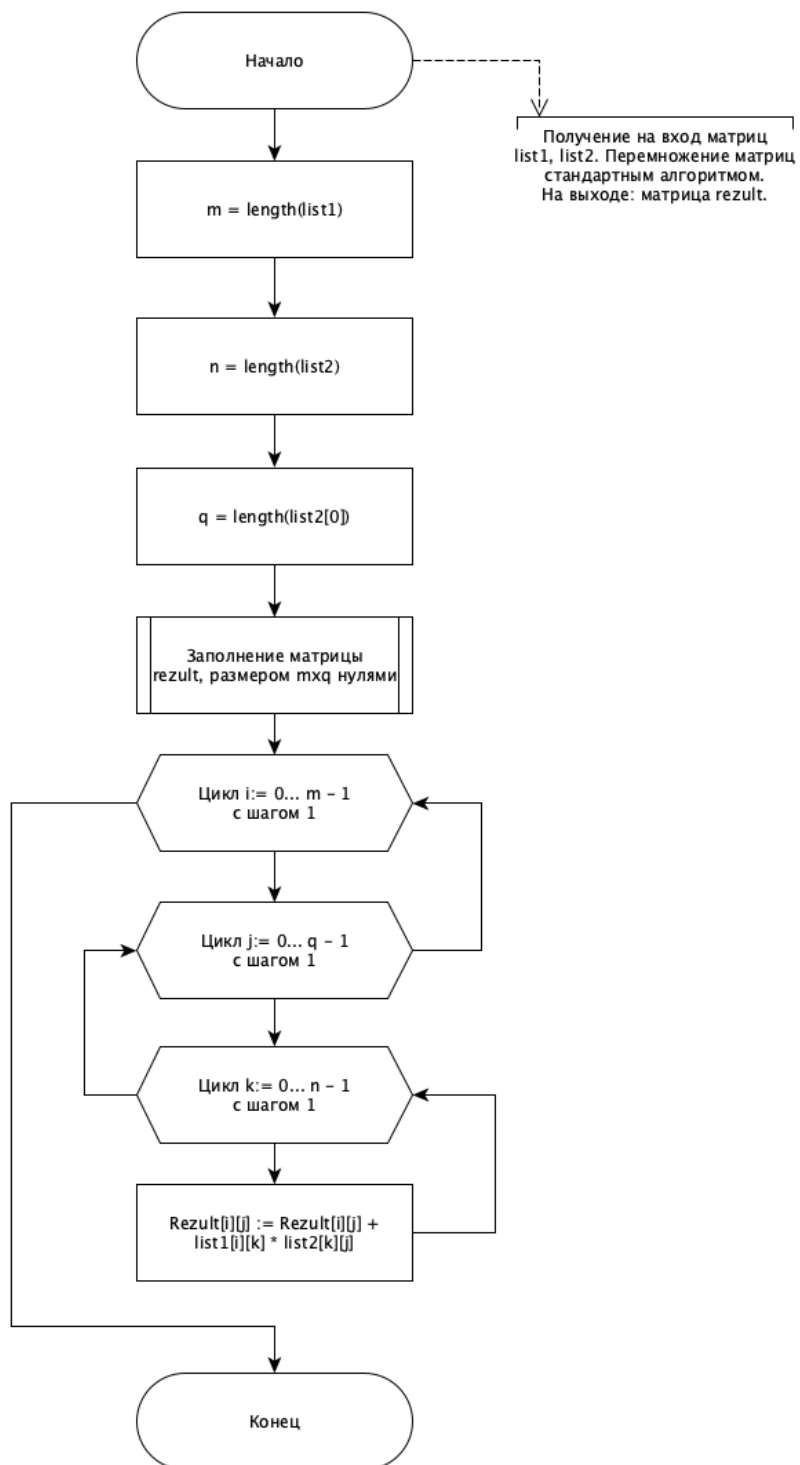


Рис. 4.2: Подпрограмма стандартного умножения матриц.

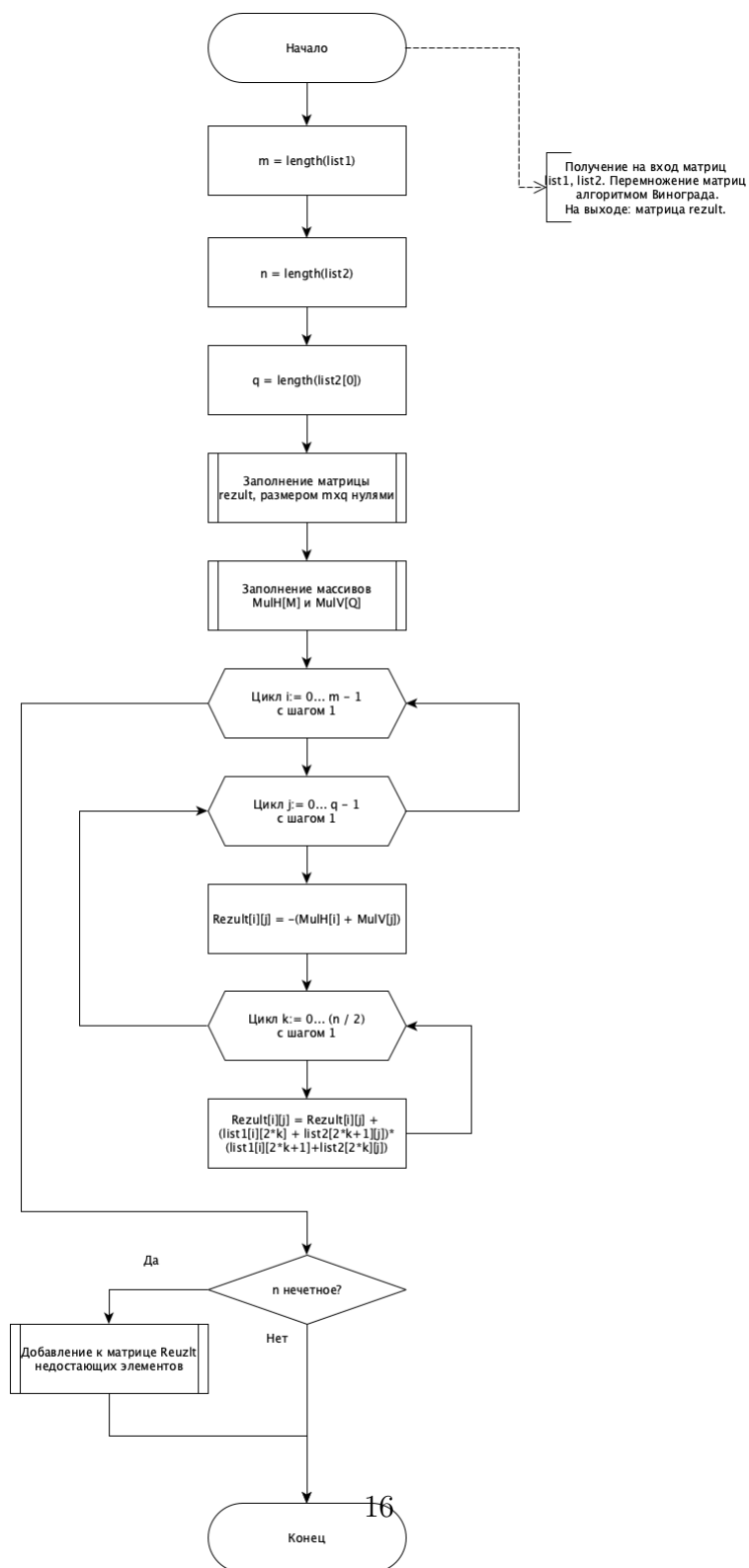


Рис. 4.3: Подпрограмма умножения матриц по Винограду.

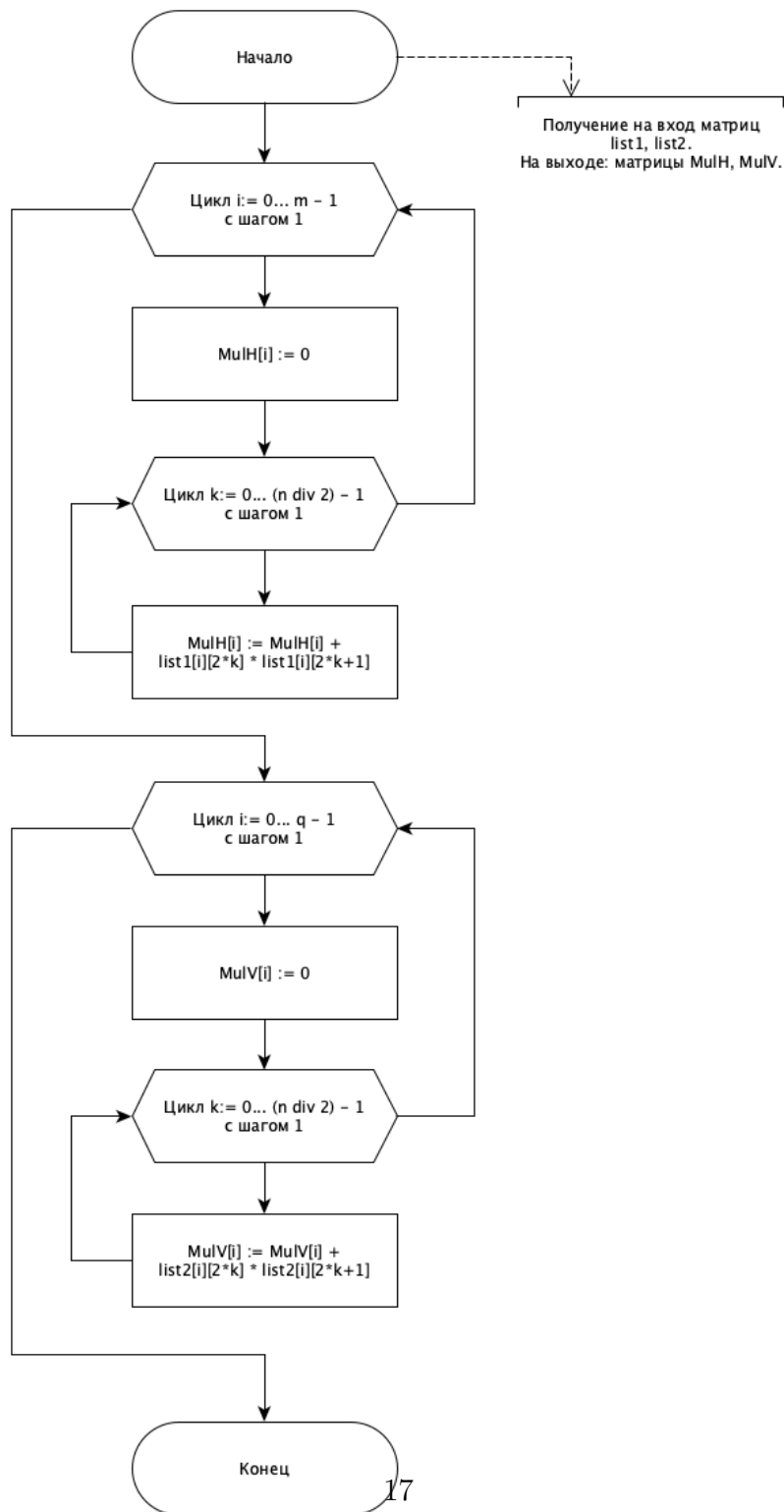


Рис. 4.4: Подпрограмма заполнения массивов $MulH[M]$ и $MulV[Q]$.

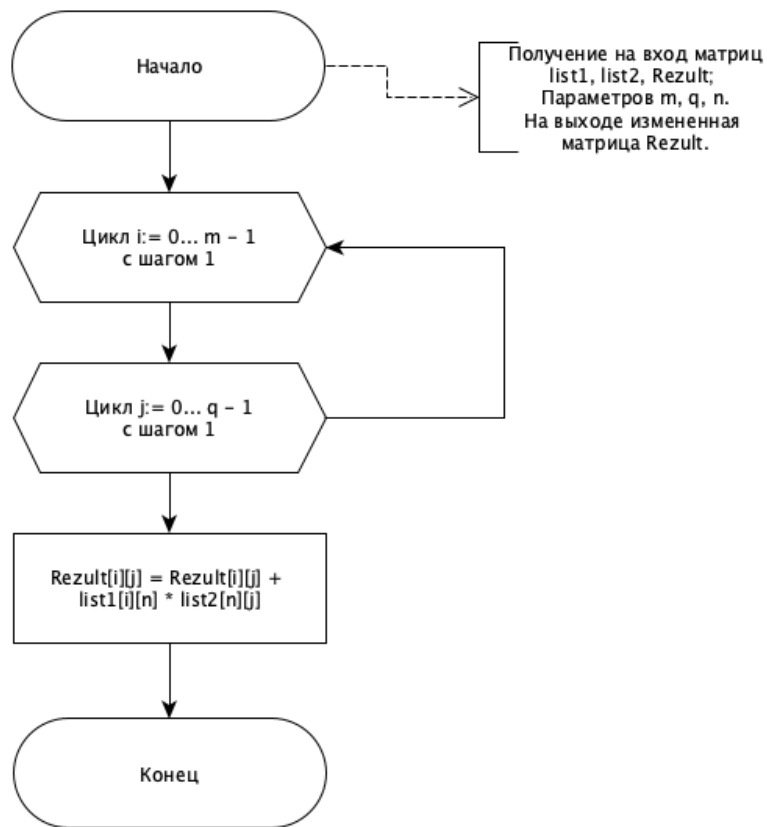


Рис. 4.5: Подпрограмма добавления к матрице Result нечетных элементов.



Рис. 4.6: Подпрограмма умножения матриц по Винограду (оптимизированный вариант).

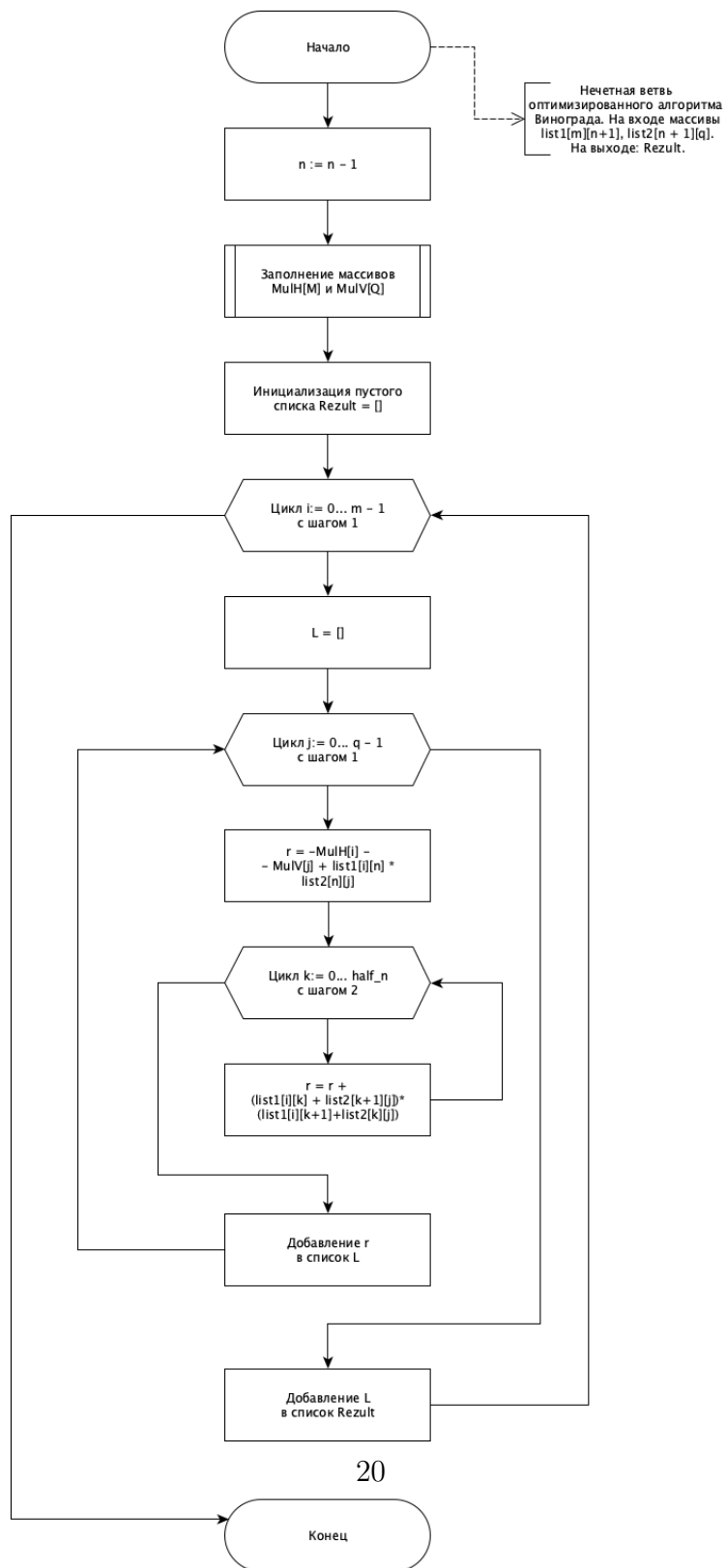


Рис. 4.7: Подпрограмма нечетного заполнения матрицы.

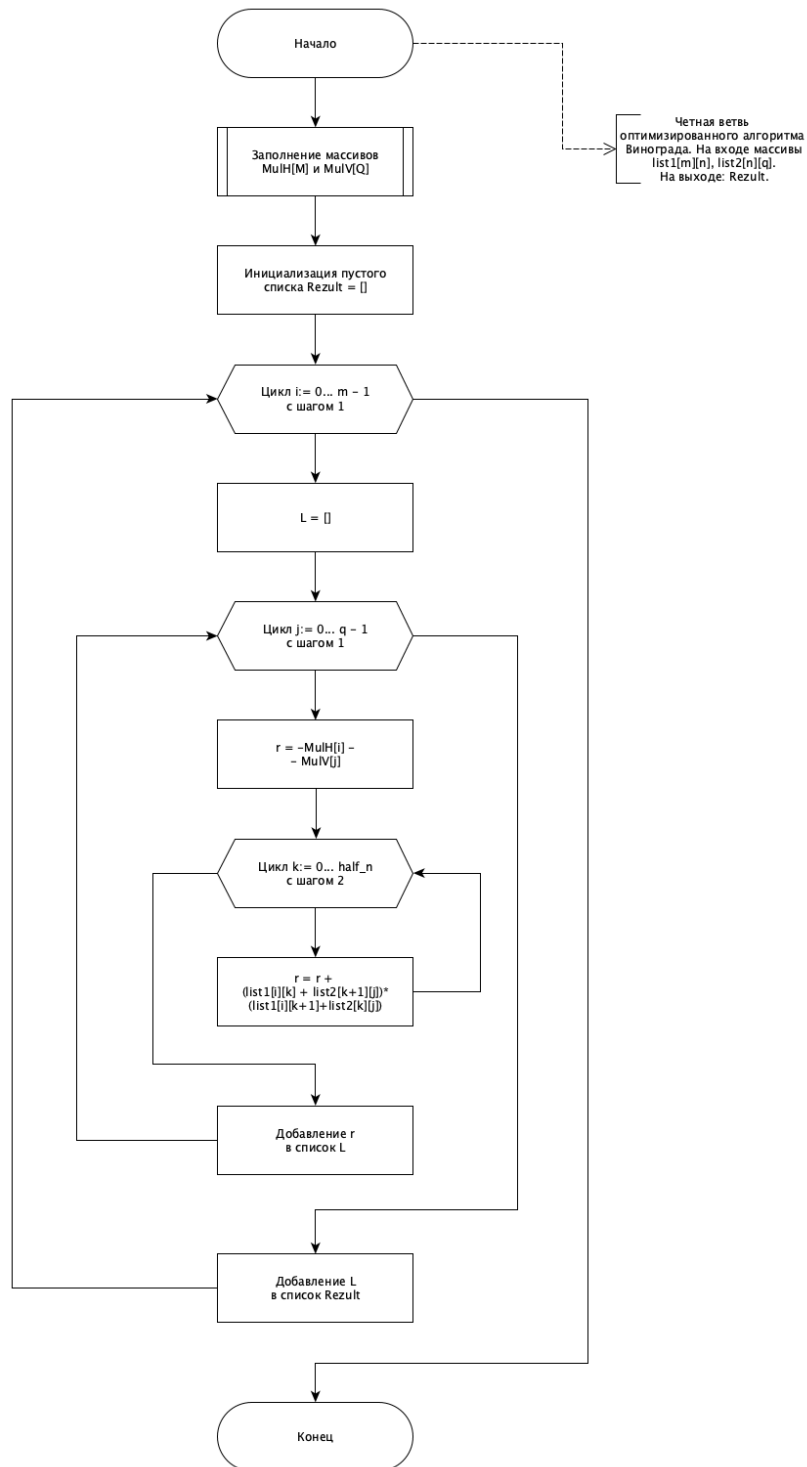


Рис. 4.8: Подпрограмма четного заполнения матрицы.

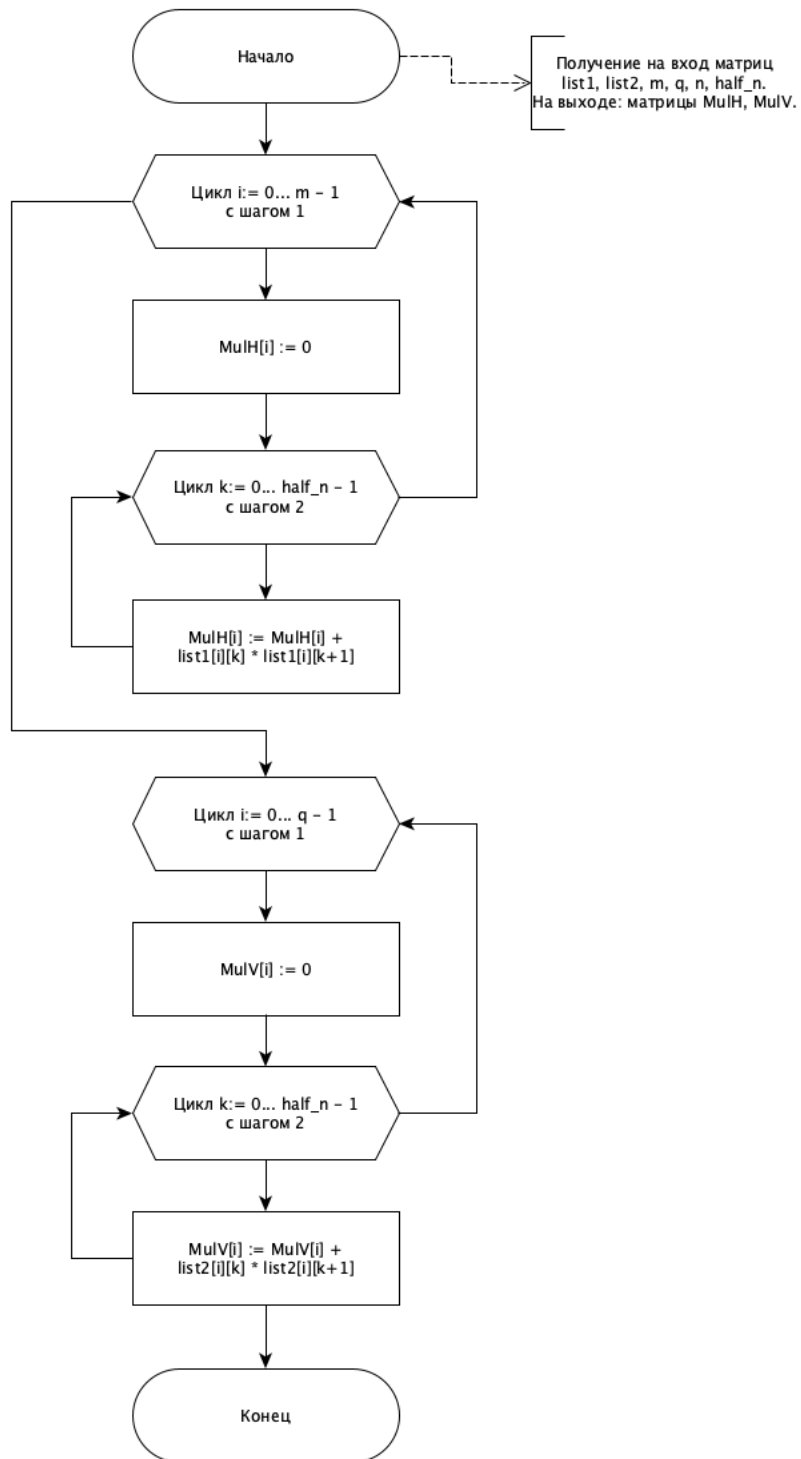


Рис. 4.9: Подпрограмма заполнения массивов $MulH[M]$ и $MulV[Q]$.

4.3 Оценка трудоемкости

4.3.1 Классический алгоритм

$$f_{classic} = 13MNQ + 4MQ + 4M + 1 \quad (4.1)$$

4.3.2 Алгоритм Виноградова

Трудоёмкости составных частей алгоритма:

- цикл создания вектора MulH: $f_I = \frac{15}{2}MN + 4M + 2$;
- цикл создания вектора MulV: $f_{II} = \frac{15}{2}NQ + 4Q + 2$;
- основной цикл: $f_{III} = 13MNQ + 4MQ + 4M + 2$;
- условный переход при чётном N : $f_{IV} = 2$;
- условный переход и цикл при нечётном N : $f_{IV} = 17MQ + 4M + 4$.

Общая трудоемкость алгоритма:

- при чётном N :

$$f_{vin} = f_I + f_{II} + f_{III} + f_{IV} = 13MNQ + \frac{15}{2}MN + \frac{15}{2}NQ + 4MQ + 8M + 4Q + 8 \quad (4.2)$$

- при нечётном N :

$$f_{vin} = f_I + f_{II} + f_{III} + f_{IV} = 13MNQ + \frac{15}{2}MN + \frac{15}{2}NQ + 21MQ + 12M + 4Q + 10 \quad (4.3)$$

4.3.3 Оптимизированный алгоритм Винограда

Трудоёмкости составных частей алгоритма:

- цикл создания вектора MulH: $f_I = \frac{11}{2}MN + 4M + 2$;
- цикл создания вектора MulV: $f_{II} = \frac{11}{2}NQ + 4Q + 2$;

- основной цикл при чётном N : $f_{III} = 10MNQ + 17MQ + 4M + 6$;
- основной цикл при нечётном N : $f_{III} = 10MNQ + 10NQ + 4M + 2$.

Общая трудоемкость алгоритма:

- при чётном N :

$$f_{opt} = f_I + f_{II} + f_{III} = 10MNQ + \frac{11}{2}MN + \frac{11}{2}NQ + 9MQ + 8M + 4Q + 6 \quad (4.4)$$

- при нечётном N :

$$f_{opt} = f_I + f_{II} + f_{III} = 10MNQ + \frac{11}{2}MN + \frac{11}{2}NQ + 16MQ + 8M + 4Q + 10 \quad (4.5)$$

4.4 Замер используемой памяти

Пусть даны две матрицы A и B размерностью $M \times N$ и размерностью $N \times Q$ соответственно и для хранения целого числа требуется 4 байта памяти.

В каждом из алгоритмов требуется хранить исходные матрицы A и B и матрицу результата умножения C , которая имеет размеры $M \times Q$. Таким образом, под хранение матриц требуется $4 \cdot (MN + NQ + MQ)$ байт памяти.

Для алгоритма Винограда и его оптимизированного варианта требуется также хранить два дополнительных вектора $MulH$ и $MulV$. Их размеры M и Q соответственно, следовательно требуется дополнительно $4 \cdot (M + Q)$ байт памяти. В итоге для алгоритмов Винограда требуется $4 \cdot (MN + NQ + MQ + M + Q)$ байт памяти. Таким образом, при больших размерах матриц (больше 100) алгоритмы Винограда будут незначительно проигрывать по памяти классическому алгоритму.

4.5 Требования к программе

- Программа должна предоставлять доступный и понятный интерфейс с выбором алгоритмов (и всех их реализаций);
- Должно быть 2 реализации алгоритма Виноградова:
 - неоптимизированная;
 - оптимизированная.
- На вход каждой функции, реализующей конкретный алгоритм, подаются две матрицы **целых** чисел;
- Пользователь должен иметь возможность вводить матрицы вручную либо выбрать пункт случайного заполнения, в случае чего ему будет предоставлен выбор длин матриц;
- Требования к выводу программы:
 - программа должна выдавать корректное перемножение двух матриц (см. Аналитическую часть);
 - при вводе пустых матриц - программа не должна аварийно завершаться, а выдавать пустую матрицу;
 - при вводе матриц, не удовлетворяющих условиям перемножения матриц (см. Аналитическую часть) - программа не должна аварийно завершаться, а выдавать пустую матрицу;

4.6 Выводы по конструкторской части

В результате проведенной работы было расписано описание алгоритмов, разработаны схемы алгоритмов, была оценена трудоемкость каждого из алгоритмов, а также расписаны оптимизации для алгоритма Винограда.

Помимо всего прочего, был произведен замер используемой памяти программой (аналитически). Были сформулированы основные требования к программе.

5 | Технологическая часть

5.1 Средства реализации

Для реализации программы был использован язык программирования **Haskell** [1]. Время работы алгоритмов было замерено с помощью функции **getCPUTime** из библиотеки **System.CPUTime**. Для отключения встроенной оптимизации компилятора был использован атрибут `-O0` компилятора `ghci`. Тестирование было организовано с помощью библиотеки **TestHUnit**. При сравнении результатов двух функций использовалась функция `randomRs` из библиотеки `System.Random`, которая генерирует случайный список целых чисел нужного размера.

5.2 Сведения о модулях программы

Программа состоит из:

- `Main.hs` - Главный файл программы, в котором располагается меню;
- `Lib.hs` - файл с подключенными модулями реализаций алгоритмов;
- `Standard.hs` - Файл с реализацией стандартного алгоритма умножения матриц;
- `Grape.hs` - Файл с реализацией алгоритма Виноградова;
- `Optimize_grape.hs` - Файл с реализацией оптимизированного алгоритма Виноградова;

- Spec.hs - Файл с модульными тестами.

Ниже приведены листинги 5.1, 5.2, 5.3 функций программы.

Листинг 5.1: Стандартный алгоритм умножения матриц

```

1
2 import Data.List
3
4
5 answer_fill :: Int -> Int -> [[Int]]
6 answer_fill 0 j = []
7 answer_fill i j = take j (repeat 0) : answer_fill (i - 1)
8     j
9
10 — c[m][q] += a[m][n] * b[n][q]
11 get_new_line_answer :: Int -> Int -> Int -> [Int] -> [Int]
12 get_new_line_answer j a b line = (take j line) ++
13     ((line !! j) + a * b) ++
14     (reverse (take (length line - j - 1) $ reverse line))
15
16
17
18 third_loop :: [Int] -> [[Int]] -> Int -> Int -> Int -> Int
19     -> [[Int]] -> [[Int]]
20 third_loop list1_i list2 i j k n answer | k == n =
21     answer
22 | otherwise =
23     (third_loop list1_i list2 i j (k + 1) n $
24     take i answer ++
25     [get_new_line_answer j
26     (list1_i !! k)
27     ((list2 !! k) !! j)
28     (answer !! i)] ++
29     (reverse (take (length answer - i - 1) $ reverse answer)))
30
31 second_loop :: [Int] -> [[Int]] -> Int -> Int -> Int -> [[
32     Int]] -> [[Int]]

```

```

31 second_loop list1_i list2 i j q answer | j == q    =
    answer
32 | otherwise =
33 second_loop list1_i list2 i (j + 1) q $
34 third_loop list1_i list2 i j 0 (length list2) answer
35
36
37 first_loop :: [[Int]] -> [[Int]] -> Int -> Int -> [[Int]]
    -> [[Int]]
38 first_loop list1 list2 i m answer | i == m    = answer
39 | otherwise =
40 first_loop list1 list2 (i + 1) m $
41 second_loop (list1 !! i) list2 i 0 (length $ head list2)
    answer
42
43
44 calc_multyp1 :: [[Int]] -> [[Int]] -> [[Int]]
45 calc_multyp1 list1 list2 = (first_loop list1 list2 0 (
    length list1)) $ answer_fill
46 (length list1) (length $ head list2)
47
48
49 check_size :: [[Int]] -> Int -> Int
50 check_size list 1 = length list
51 check_size list 2 = length $ head list
52
53
54 check_matr :: [[Int]] -> Bool
55 check_matr [] = True
56 check_matr (_:[]) = True
57 check_matr (x:xs) = if ((length x) == (length $ head xs))
58 then check_matr xs
59 else False
60
61
62 standard :: [[Int]] -> [[Int]] -> [[Int]]
63 standard list1 list2 = if (check_matr list1 && check_matr
    list2)
64 then (if (check_size list1 2) == (check_size list2 1)
65 then calc_multyp1 list1 list2

```

```

66 else [[]])
67 else [[]]

```

Листинг 5.2: Алгоритм Виноградова

```

1  import Data.List
2
3  answer_fill :: Int -> Int -> [[Int]]
4  answer_fill 0 j = []
5  answer_fill i j = take j (repeat 0) : answer_fill (i - 1)
6      j
7
8
9  mulh_loop :: Int -> Int -> Int -> [Int] -> Int -> Int
10 mulh_loop i k n a_i mulh_i | k == div n 2 = mulh_i
11 | otherwise = mulh_i + (a_i !! (2 * k)) * (a_i !! (2
12     * k + 1)) +
13 (mulh_loop i (k + 1) n a_i mulh_i)
14
15
16 calc_mulh :: Int -> Int -> Int -> [[Int]] -> [Int] -> [Int]
17 calc_mulh i m n list mulh | i == m = mulh
18 | otherwise = calc_mulh (i + 1) m n list
19 ((take i mulh) ++
20 [mulh_loop i 0 n (list !! i) (mulh !! i)] ++
21 (reverse (take (length mulh - i - 1) $ reverse mulh)))
22
23
24
25 mulv_loop :: Int -> Int -> Int -> [[Int]] -> Int -> Int
26 mulv_loop i k n a mulv_i | k == div n 2 = mulv_i
27 | otherwise = mulv_i + ((a !! (2 * k)) !! i) * ((a
28     !! (2 * k + 1)) !! i) +
29 (mulv_loop i (k + 1) n a mulv_i)
30
31
32 calc_mulv :: Int -> Int -> Int -> [[Int]] -> [Int] -> [Int]

```

```

    ]
33 calc_mulv i q n list mulv | i == q    = mulv
34 | otherwise = calc_mulv (i + 1) q n list
35 ((take i mulv) ++
36 [mulv_loop i 0 n list (mulv !! i)] ++
37 (reverse (take (length mulv - i - 1) $ reverse mulv)))
38
39
40 main_k_loop :: [[Int]] -> [[Int]] -> Int -> Int -> Int ->
    [[Int]] -> [[Int]]
41 main_k_loop a b i j k c | k == div (length b) 2 = c
42 | otherwise          = main_k_loop a b i j (k + 1) (
43 (take i c) ++
44 [
45 take j (c !! i) ++
46 (((c !! i) !! j) + (((a !! i) !! (2 * k)) + ((b !! (2 * k
    + 1)) !! j)) *
47 (((a !! i) !! (2 * k + 1)) + ((b !! (2 * k)) !! j)))) ++
48 (reverse (take (length (c !! i) - j - 1) $ reverse (c !! i
    ))))
49 ] ++
50 (reverse (take (length c - i - 1) $ reverse c))
51 )
52
53
54
55 main_q_loop :: [[Int]] -> [[Int]] -> [Int] -> [Int] -> Int
    -> Int -> [[Int]] -> [[Int]]
56 main_q_loop a b mulh mulv i j c | j == (length $ head b)
    = c
57 | otherwise          = main_q_loop a b mulh mulv i (j
    + 1) (main_k_loop a b i j 0
58 (
59 (take i c) ++
60 [
61 take j (c !! i) ++
62 [-1 * ((mulh !! i) + (mulv !! j))] ++
63 (reverse (take (length (c !! i) - j - 1) $ reverse (c !! i
    ))))
64 ] ++

```

```

65 (reverse (take (length c - i - 1) $ reverse c))
66 )
67 )
68
69
70 main_calc :: [[Int]] -> [[Int]] -> [Int] -> [Int] -> Int
71           -> [[Int]] -> [[Int]]
72 main_calc a b mulh mulv i c | i == (length a) = c
73 | otherwise = main_calc a b mulh mulv (i + 1) (
74   main_q_loop a b mulh mulv i 0 c)
75
76 nechet_q_loop :: [[Int]] -> [[Int]] -> Int -> Int -> Int
77               -> Int -> [[Int]] -> [Int]
78 nechet_q_loop a b i j n q c | j == q = (c !! i)
79 | otherwise = nechet_q_loop a b i (j + 1) n q
80 (
81   (take i c) ++
82   [(take j (c !! i)) ++
83    ((c !! i) !! j) + ((a !! i) !! (n - 1)) * ((b !! (n - 1))
84     !! j)] ++
85   (reverse (take (length (c !! i) - j - 1) $ reverse (c !! i
86     )))) ++
87   (reverse (take (length c - i - 1) $ reverse c))
88 )
89
90 nechet_m_loop :: [[Int]] -> [[Int]] -> Int -> Int -> Int
91               -> Int -> [[Int]] -> [[Int]]
92 nechet_m_loop a b i m q n c | i == m = c
93 | otherwise = nechet_m_loop a b (i + 1) m q n
94 ((take i c) ++
95  [nechet_q_loop a b i 0 n q c] ++
96  (reverse (take (length c - i - 1) $ reverse c)))
97
98 nechet_calc :: [[Int]] -> [[Int]] -> Int -> Int -> Int ->
99              [[Int]] -> [[Int]]

```



```

98 | nechet_calc a b m q n c | not (odd n) = c
99 | otherwise      = nechet_m_loop a b 0 m q n c
100
101
102
103 | calc_multyp1 :: [[Int]] -> [[Int]] -> [[Int]]
104 | calc_multyp1 list1 list2 = nechet_calc list1 list2 (length
105 |   list1)
106 |   (length $ head list2)
107 |   (length list2) $
108 |   main_calc list1 list2
109 |   (calc_mulh 0 (length list1) (length $ head list1) list1 (
110 |     take (length list1) (repeat 0)))
111 |   (calc_molv 0 (length $ head list2) (length list2) list2 (
112 |     take (length $ head list2) (repeat 0)))
113 |   0
114 |   (answer_fill (length list1) (length $ head list2))
115
116
117
118
119
120 | check_size :: [[Int]] -> Int -> Int
121 | check_size list 1 = length list
122 | check_size list 2 = length $ head list
123
124
125
126
127
128
129 | check_matr :: [[Int]] -> Bool
130 | check_matr [] = True
131 | check_matr (_:[]) = True
132 | check_matr (x:xs) = if ((length x) == (length $ head xs))
133 |   then check_matr xs
134 |   else False
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

Листинг 5.3: Оптимизированный алгоритм Виноградова

```

1  answer_fill :: Int -> Int -> [[Int]]
2  answer_fill 0 j = []
3  answer_fill i j = take j (repeat 0) : answer_fill (i - 1)
   j
4
5
6
7
8  mulh_loop :: Int -> Int -> Int -> [Int] -> Int -> Int
9  mulh_loop i k n a_i mulh_i | k >= n      = mulh_i
10 | otherwise = mulh_i + (a_i !! k) * (a_i !! (k + 1)) +
11 (mulh_loop i (k + 2) n a_i mulh_i)
12
13
14
15 calc_mulh :: Int -> Int -> Int -> [[Int]] -> [Int] -> [Int]
   ]
16 calc_mulh i m n list mulh | i == m      = mulh
17 | otherwise = calc_mulh (i + 1) m n list
18 ((take i mulh) ++
19 [mulh_loop i 0 n (list !! i) (mulh !! i)] ++
20 (reverse (take (length mulh - i - 1) $ reverse mulh)))
21
22
23
24 mulv_loop :: Int -> Int -> Int -> [[Int]] -> Int -> Int
25 mulv_loop i k n a mulv_i | k >= n      = mulv_i
26 | otherwise = mulv_i + ((a !! k) !! i) * ((a !! (k + 1))
   !! i) +
27 (mulv_loop i (k + 2) n a mulv_i)
28
29
30
31 calc_mulv :: Int -> Int -> Int -> [[Int]] -> [Int] -> [Int]
   ]
32 calc_mulv i q n list mulv | i == q      = mulv
33 | otherwise = calc_mulv (i + 1) q n list
34 ((take i mulv) ++
35 [mulv_loop i 0 n list (mulv !! i)] ++

```

```

36 (reverse (take (length mulv - i - 1) $ reverse mulv)))
37
38
39
40
41 buf :: [Int] -> [[Int]] -> Int -> Int -> Int -> Int
42 buf a_i b j k n | k >= n = 0
43 | otherwise = (((a_i !! k) + ((b !! (k + 1)) !! j)) *
44 ((a_i !! (k + 1)) + ((b !! k) !! j))) +
45 (buf a_i b j (k + 2) n)
46
47
48
49 main_q_loop :: [Int] -> [[Int]] -> Int -> Int -> Int ->
50 Int -> Int -> [Int] -> [Int]
51 main_q_loop a_i b j m q n mulh_i mulv | j == q = []
52 | otherwise = ((buf a_i b j 0 n) -
53 (mulh_i +
54 (mulv !! j))) :
55 (main_q_loop a_i b (j + 1) m q n mulh_i mulv)
56
57 main_calc :: [[Int]] -> [[Int]] -> Int -> Int -> Int ->
58 Int -> [Int] -> [Int] -> [[Int]]
59 main_calc a b i m q n mulh mulv | i == m = []
60 | otherwise = (main_q_loop (a !! i) b 0 m q n (mulh !! i)
61 mulv) :
62 (main_calc a b (i + 1) m q n mulh mulv)
63
64 buf_nechet :: [Int] -> [[Int]] -> Int -> Int -> Int -> Int
65 buf_nechet a_i b j k n_minus_one | k >= n_minus_one = 0
66 | otherwise = (((a_i !! k) + ((b !! (k + 1)) !! j))
67 *
68 ((a_i !! (k + 1)) + ((b !! k) !! j))) +
69 (buf_nechet a_i b j (k + 2) n_minus_one)
70
71 add_nechet :: [Int] -> [[Int]] -> Int -> Int -> Int

```

```

72 add_nechet a_i b j n_minus_one = (a_i !! n_minus_one) * ((
    b !! n_minus_one) !! j)
73
74
75
76 main_q_loop_nechet :: [Int] -> [[Int]] -> Int -> Int ->
    Int -> Int -> Int -> [Int] -> Int -> [Int]
77 main_q_loop_nechet a_i b j m q n mulh_i mulv n_minus_one |
    j == q    = []
78 | otherwise = ((buf_nechet a_i b j 0 n_minus_one) -
79 (mulh_i +
80 (mulv !! j)) + (add_nechet a_i b j n_minus_one)) :
81 (main_q_loop_nechet a_i b (j + 1) m q n mulh_i mulv
    n_minus_one)
82
83
84
85 main_calc_nechet :: [[Int]] -> [[Int]] -> Int -> Int ->
    Int -> Int -> [Int] -> [Int] -> Int -> [[Int]]
86 main_calc_nechet a b i m q n mulh mulv n_minus_one | i ==
    m    = []
87 | otherwise = (main_q_loop_nechet (a !! i) b 0 m q n (mulh
    !! i) mulv n_minus_one) :
88 (main_calc_nechet a b (i + 1) m q n mulh mulv n_minus_one)
89
90
91
92
93 calc_multypl :: [[Int]] -> [[Int]] -> Int -> Int -> Int ->
    [[Int]]
94 calc_multypl list1 list2 m q n = if not (odd n) then
95 (main_calc list1 list2 0 m q n
96 (calc_mulh 0 m n list1 (take m (repeat 0)))
97 (calc_mulv 0 q n list2 (take q (repeat 0))))
98 else (main_calc_nechet list1 list2 0 m q n
99 (calc_mulh 0 m (n - 1) list1 (take m (repeat 0)))
100 (calc_mulv 0 q (n - 1) list2 (take q (repeat 0)))
101 (n - 1))
102
103

```

```

104
105 check_size :: [[Int]] -> Int -> Int
106 check_size list 1 = length list
107 check_size list 2 = length $ head list
108
109 check_matr :: [[Int]] -> Bool
110 check_matr [] = True
111 check_matr (_:[]) = True
112 check_matr (x:xs) = if ((length x) == (length $ head xs))
113 then check_matr xs
114 else False
115
116
117
118 optimize_grape :: [[Int]] -> [[Int]] -> [[Int]]
119 optimize_grape list1 list2 = if (check_matr list1 &&
    check_matr list2)
120 then (if (check_size list1 2) == (check_size list2 1)
121 then calc_multipl list1 list2 (length list1)
122 (length $ head list2)
123 (length list2)
124 else [])
125 else []

```

5.3 Тесты

Для проверки корректности работы алгоритмов были подготовлены функциональные автоматические тесты, представленные в таблице 5.1.

Таблица 5.1: Функциональные тесты.

Описание теста	Матрица 1	Матрица 2	Ожидаемый результат
Пустые матрицы	$\begin{bmatrix} \end{bmatrix}$	$\begin{bmatrix} \end{bmatrix}$	$\begin{bmatrix} \end{bmatrix}$
Незаполненные матрицы	$\begin{bmatrix} 2 & 2 \\ 3 & 3 \end{bmatrix}$	$\begin{bmatrix} 1 & 5 \\ 3 & \end{bmatrix}$	$\begin{bmatrix} \end{bmatrix}$
$n1 \neq n2$	$\begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix}$	$\begin{bmatrix} 2 & 5 \end{bmatrix}$	$\begin{bmatrix} \end{bmatrix}$
n четное (4x6 6x1)	$\begin{bmatrix} 1 & 3 & 4 & 5 & 8 & 3 \\ 7 & 4 & 9 & 4 & 6 & 2 \\ 6 & 5 & 9 & 5 & 5 & 7 \\ 5 & 2 & 9 & 2 & 8 & 3 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix}$	$\begin{bmatrix} 97 \\ 100 \\ 130 \\ 102 \end{bmatrix}$
n нечетное (5x3 3x7)	$\begin{bmatrix} 1 & 7 & 9 \\ 2 & 4 & 7 \\ 4 & 2 & 4 \\ 7 & 3 & 2 \\ 5 & 8 & 4 \end{bmatrix}$	$\begin{bmatrix} 1 & 9 & 7 & 3 & 4 & 6 & 8 \\ 5 & 7 & 7 & 4 & 5 & 7 & 7 \\ 7 & 4 & 8 & 3 & 8 & 6 & 2 \end{bmatrix}$	$\begin{bmatrix} 99 & 94 & 128 & 58 & 111 & 109 & 75 \\ 71 & 74 & 98 & 43 & 84 & 82 & 58 \\ 42 & 66 & 74 & 32 & 58 & 62 & 54 \\ 36 & 92 & 86 & 39 & 59 & 75 & 81 \\ 73 & 117 & 123 & 59 & 92 & 110 & 104 \end{bmatrix}$
Обычный случай (3x3 3x6)	$\begin{bmatrix} 1 & 5 & 3 \\ 5 & 8 & 4 \\ 5 & 5 & 7 \end{bmatrix}$	$\begin{bmatrix} 4 & 4 & 3 & 4 & 7 & 3 \\ 6 & 2 & 5 & 2 & 8 & 5 \\ 8 & 5 & 2 & 1 & 2 & 3 \end{bmatrix}$	$\begin{bmatrix} 58 & 29 & 34 & 17 & 53 & 37 \\ 100 & 56 & 63 & 40 & 107 & 67 \\ 106 & 65 & 54 & 37 & 89 & 61 \end{bmatrix}$
Обычный случай №2 (1x4 4x1)	$\begin{bmatrix} 5 & 4 & 5 & 8 \end{bmatrix}$	$\begin{bmatrix} 5 \\ 8 \\ 2 \\ 2 \end{bmatrix}$	$\begin{bmatrix} 83 \end{bmatrix}$
Обычный случай №3 (6x2 2x5)	$\begin{bmatrix} 3 & 5 \\ 7 & 5 \\ 2 & 5 \\ 8 & 9 \\ 9 & 5 \\ 4 & 8 \end{bmatrix}$	$\begin{bmatrix} 3 & 4 & 7 & 8 & 2 \\ 5 & 9 & 2 & 6 & 4 \end{bmatrix}$	$\begin{bmatrix} 34 & 57 & 31 & 54 & 26 \\ 46 & 73 & 59 & 86 & 34 \\ 31 & 53 & 24 & 46 & 24 \\ 69 & 113 & 74 & 118 & 52 \\ 52 & 81 & 73 & 102 & 38 \\ 52 & 88 & 44 & 80 & 40 \end{bmatrix}$

Также были подготовлены тесты, в которых попарно сравнивались все 3 реализации функций умножения матриц.

В результате проверки реализации всех алгоритмов умножения прошли все поставленные функциональные тесты.

5.4 Выводы по технологической части

В качестве языка программирования был выбран язык программирования Haskell, а также все необходимые функции для реализации поставленных задач.

Была разработана программа, реализующая все требования, поставленные в конструкторской части. Сведения о модулях программы и листинги кода также приведены в этом разделе.

Были составлены функциональные тесты, проверяющие работу реализованных алгоритмов.

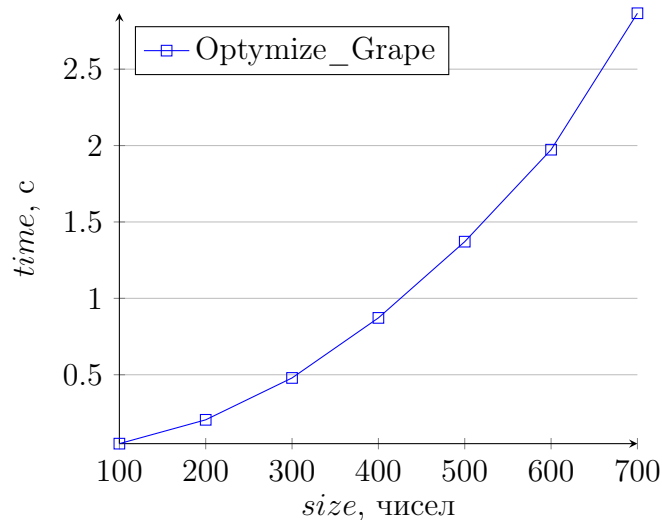
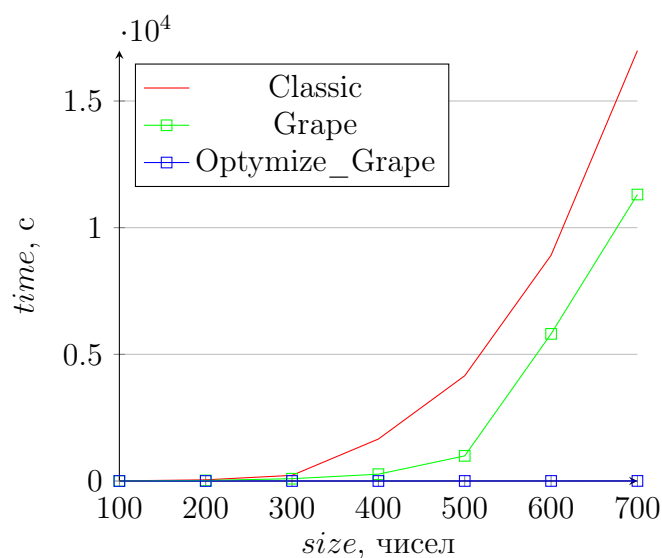
6 | Экспериментальная часть

6.1 Сравнение работы алгоритмов при чётных размерах матриц

Для сравнения времени работы алгоритмов умножения матриц были использованы квадратные матрицы размером от 100 до 1000 с шагом 100. Эксперимент для более точного результата повторялся 100 раз. Итоговый результат рассчитывался как средний из полученных результатов. Результаты измерений показаны в таблице и на рисунке

Таблица 6.1: Время работы алгоритмов в секундах.

Размер матриц	Классический, с	Алгоритм Винограда, с	Оптимизированный алгоритм Винограда, с
100	4.927	2.517	0.049
200	45.956	22.215	0.205
300	222.990	90.549	0.479
400	1656.764	268.980	0.872
500	4154.464	991.378	1.371
600	8910.944	5808.120	1.973
700	16988.810	11308.290	2.866



Из результатов эксперимента видно, что алгоритм Винограда, даже в самой плохой своей реализации, на больших размерах матриц (порядка 100x100) значительно выигрывает по сравнению с классическим алгоритмом умножения матриц (при умножении матриц 500x500 время выполнения различается больше чем в 4 раза).

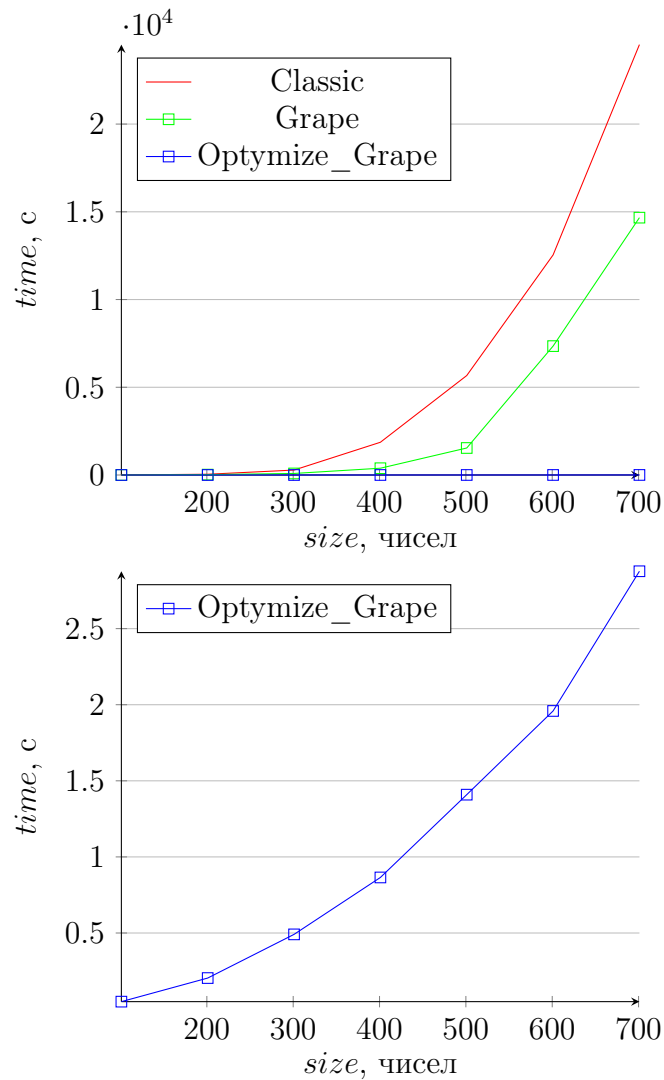
Если же говорить про оптимизированную версию алгоритма Винограда, то она затрачивает в тысячи раз меньше времени (при умножении матриц 500x500 время выполнения различается больше чем в 4000 раз) чем классический алгоритм умножения.

6.2 Сравнение работы алгоритмов при нечетных размерах матриц

Для сравнения времени работы алгоритмов умножения матриц были использованы квадратные матрицы размером от 100 до 1000 с шагом 100. Эксперимент для более точного результата повторялся 100 раз. Итоговый результат рассчитывался как средний из полученных результатов. Результаты измерений показаны в таблице и на рисунке

Таблица 6.2: Время работы алгоритмов в секундах.

Размер матриц	Классический, с	Алгоритм Винограда, с	Оптимизированный алгоритм Винограда, с
101	5.150	2.368	0.049
201	43.338	22.306	0.204
301	281.546	88.246	0.491
401	1865.899	386.807	0.865
501	5668.476	1538.992	1.409
601	12540.691	7346.204	1.959
701	24528.664	14666.842	2.877



Для случая с нечетными матрицами ситуация не меняется даже не смотря на выполнение худшего случая алгоритма Винограда: алгоритм Винограда, даже в самой плохой своей реализации, на больших размерах матриц (порядка 100×100) значительно выигрывает по сравнению с классическим алгоритмом умножения матриц (при умножении матриц 501×501 время выполнения различается больше чем в 3 раза)

Если же говорить про оптимизированную версию алгоритма Винограда, то она затрачивает в тысячи раз меньше времени (при умножении матриц 501×501 время выполнения различается больше чем в 5000 раз)

чем классический алгоритм умножения.

6.3 Выводы по экспериментальной части

В результате полученных экспериментов выяснилось, что на больших размерах матриц оба алгоритма Винограда (неоптимизированная версия и оптимизированная) выигрывают у стандартного алгоритма умножения матриц как в лучшем (при четных размерах), так и в худшем случаях (при нечетных размерах).

Заключение

В ходе лабораторной работы были изучены 2 алгоритма умножения матриц: классический алгоритм и алгоритм Винограда. Также были разработаны 2 реализации алгоритма Винограда: оптимизированная и неоптимизированная версия, и одна реализация стандартного алгоритма на языке программирования Haskell.

Сравнительный анализ алгоритмов показал, что алгоритмы Винограда, введя дополнительные векторы и увеличив тем самым расход памяти, добились уменьшения времени умножения матриц за счет уменьшения трудоемких операций: вместо 2 операций умножения, используемых в стандартном варианте, выполнялась всего одна операция умножения в алгоритме Винограда. Причем неоптимизированная версия алгоритма Винограда работает в среднем на 55% быстрее нежели стандартный алгоритм на больших размерах матриц (от 100 и больше), а оптимизированная версия работает в среднем в 2300 раз быстрее, чем стандартный алгоритм.

Такая большая разница между оптимизированной версией и неоптимизированной обусловлена не только тем списком оптимизаций, что изложен в конструкторской части, но также тем, что реализация данного алгоритма была написана на языке программирования Haskell, у которого основной структурой данных является не массив, а список. В связи с чем операция обращения к i -ому элементу списка занимает гораздо больше времени нежели обращение к голове списка (0-ому элементу) в отличие от "сишного" массива, где не имеет значения, к какому элементу по счету идет обращение, так как происходит обращение по указателю.

Литература

- [1] <https://www.haskell.org/documentation/> - Документация Haskell
- [2] Бахвалов, Н.С. Численные методы / Н.С. Бахвалов, Н.П. Жидков, Г.М. Кобельков – М.: Наука, 1987.
- [3] Jelfimova L. A new fast systolic array for modified Winograd algorithm // Proc. Sevens Int. Workshop on Parallel Processing by Cellular Automata and Array, PARCELLA-96 (Berlin, Germany, Sept. 1996). — Berlin: Akad. Verlag. — 1996.
- [4] Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р.М. Ривест: – МЦНТО, 1999.
- [5] <https://cppreference.com/> [Электронный ресурс]