



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 курсу "Анализ алгоритмов"

Тема Умножение матриц. Алгоритм Винограда

Студент Ивахненко Д.А.

Группа ИУ7-56Б

Преподаватель Волкова Л.Л.

Оглавление

Введение	2
1 Аналитическая часть	3
1.1 Классический алгоритм перемножения матриц	3
1.2 Алгоритм Винограда для перемножения матриц	3
1.3 Вывод	4
2 Конструкторская часть	5
2.1 Разработка алгоритмов	5
2.2 Модель вычислений	11
2.3 Трудоемкость алгоритмов	11
2.3.1 Стандартный алгоритм перемножения матриц	11
2.3.2 Алгоритм Винограда	12
2.3.3 Оптимизированный Алгоритм Винограда	13
2.4 Вывод	15
3 Технологическая часть	16
3.1 Требования к ПО	16
3.2 Средства реализации	16
3.3 Функциональное тестирование	16
3.4 Реализация алгоритмов	17
3.5 Вывод	18
4 Исследовательская часть	19
4.1 Технические характеристики	19
4.2 Проведение эксперимента	19
4.3 Вывод	21
Заключение	22
Литература	23
А Листинги	24

Введение

Целью работы является исследование алгоритма Винограда для перемножения матриц.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- реализовать классический алгоритм перемножения матриц;
- реализовать алгоритм Винограда перемножения матриц и оптимизировать его;
- определить модель вычислений трудоемкости;
- на основе теоретических расчетов и выбранной модели вычислений провести сравнительный анализ трудоёмкости алгоритмов;
- экспериментально провести сравнительный анализ производительности алгоритмов.

1 Аналитическая часть

1.1 Классический алгоритм перемножения матриц

Пусть даны две прямоугольные матрицы, с размерностями $l \times m$ и $m \times n$ соответственно:

$$A_{lm} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \dots & a_{lm} \end{pmatrix}, \quad B_{mn} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}, \quad (1.1)$$

тогда матрица C

$$C_{ln} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \dots & c_{ln} \end{pmatrix}, \quad (1.2)$$

где

$$c_{ij} = \sum_{r=1}^m a_{ir} b_{rj} \quad (i = \overline{1, l}; j = \overline{1, n}) \quad (1.3)$$

называется их произведением[1].

Для вычисления произведения двух матриц, каждая строка первой матрицы почленно умножается на каждый столбец второй, затем подсчитывается сумма таких произведений и записывается в соответствующую ячейку результирующей матрицы.

1.2 Алгоритм Винограда для перемножения матриц

Если рассмотреть результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующей строки первой матрицы и соответствующего столбца второй матрицы.

ющих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее [2].

Рассмотрим два вектора $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно:

$$V \cdot W = v_1w_1 + v_2w_2 + v_3w_3 + v_4w_4 \quad (1.4)$$

что эквивалентно

$$V \cdot W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1v_2 - v_3v_4 - w_1w_2 - w_3w_4. \quad (1.5)$$

На первый взгляд неочевидно, в чем преимущество выражения 1.5, поскольку оно требует вычисления большего количества операций, чем классическое 1.4: вместо четырёх умножений – шесть, а вместо трёх сложений – десять. Заметим, что выражение в правой части формулы 1.5 допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй, что позволит для каждого элемента выполнять лишь два умножения и пять сложений. Из-за того, что операция сложения быстрее операции умножения в ЭВМ, на практике алгоритм должен работать быстрее стандартного.

1.3 Вывод

В данном разделе был рассмотрены классический алгоритм перемножения матриц и алгоритм Винограда, позволяющий сократить количество операций умножения, а также использовать предварительные расчеты.

2 Конструкторская часть

2.1 Разработка алгоритмов

На рисунке 2.1 приведена схема классического алгоритма для перемножения матриц.

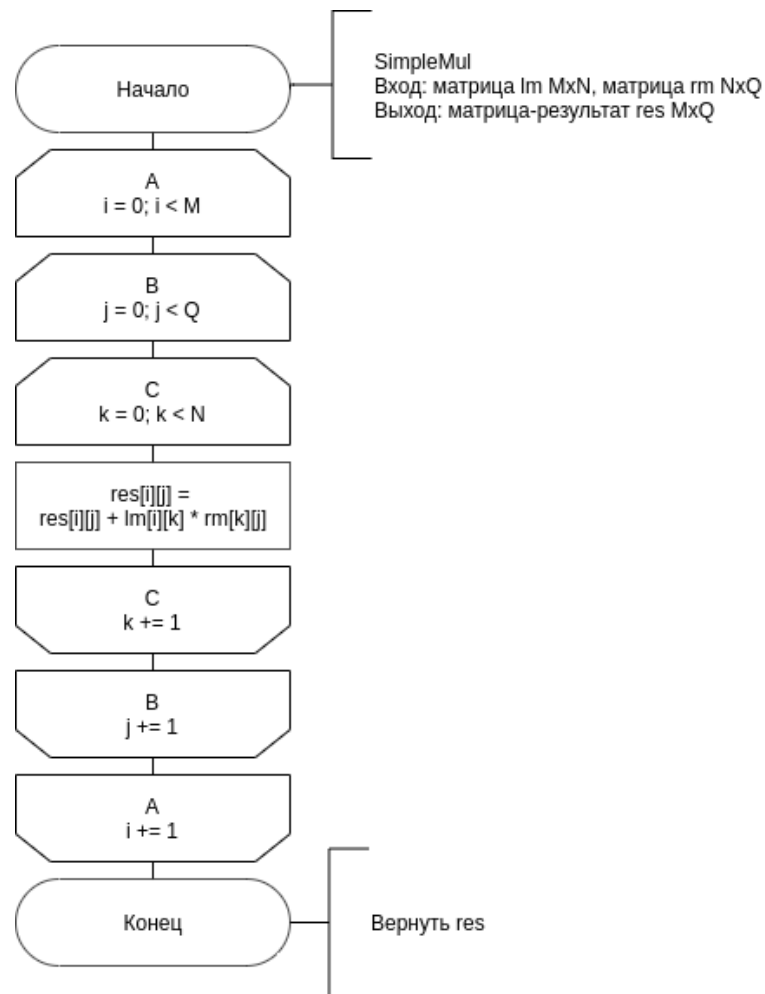


Рисунок 2.1 – Схема классического алгоритма для перемножения матриц

На рисунках 2.2 – 2.4 приведены схемы алгоритма Винограда для перемножения матриц.

На рисунках 2.5 – 2.7 приведены схемы алгоритма оптимизированного алгоритма Винограда для перемножения матриц.

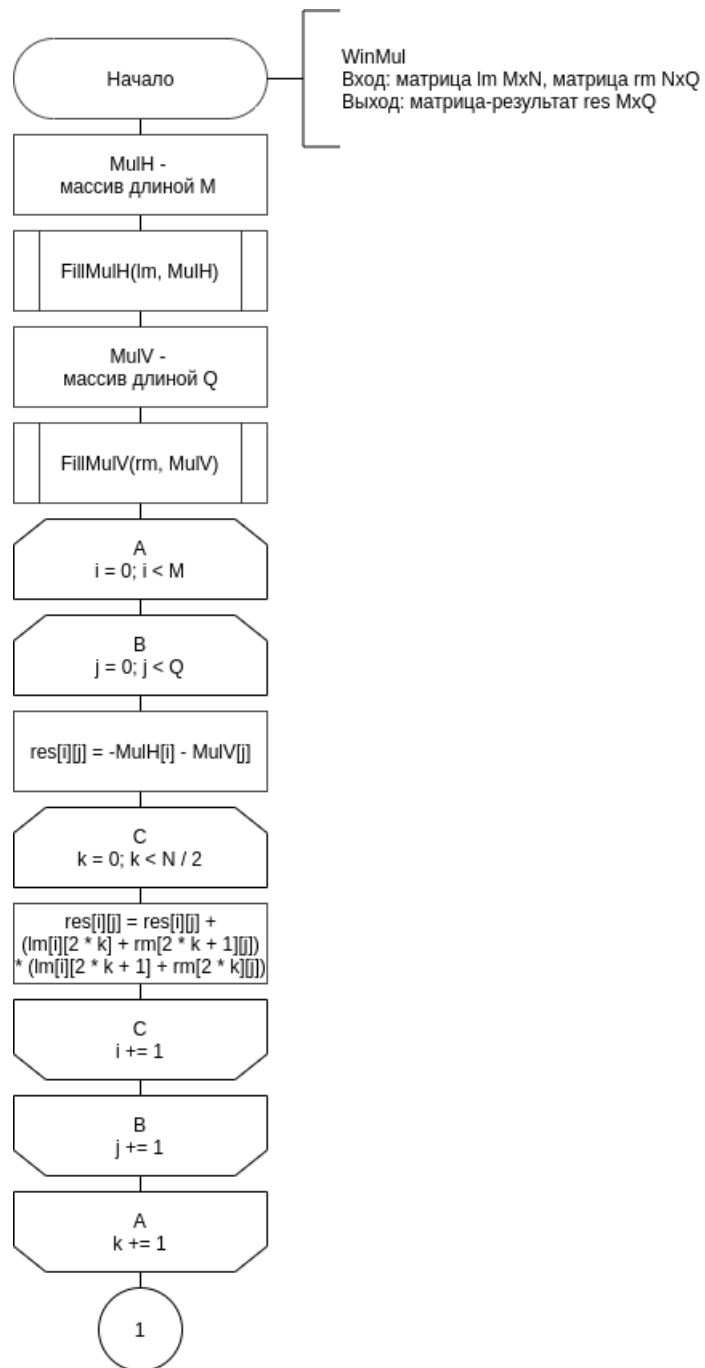


Рисунок 2.2 – Схема алгоритма Винограда для перемножения матриц

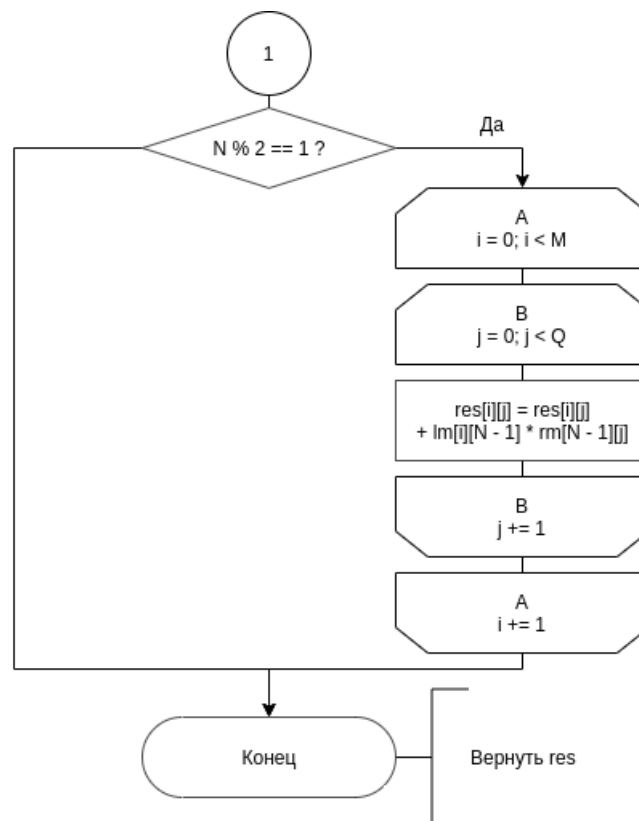


Рисунок 2.3 – Продолжение схемы алгоритма Винограда для перемножения матриц

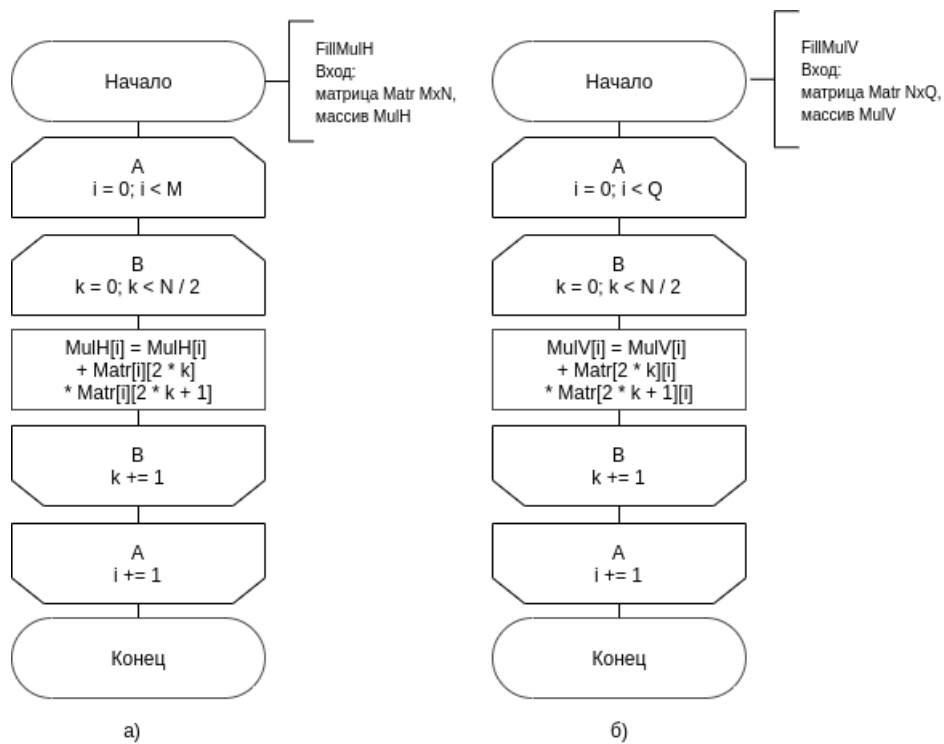


Рисунок 2.4 – Схемы алгоритмов предварительных вычислений в алгоритме Винограда. Заполнение векторов MulH (а) и MulV (б)

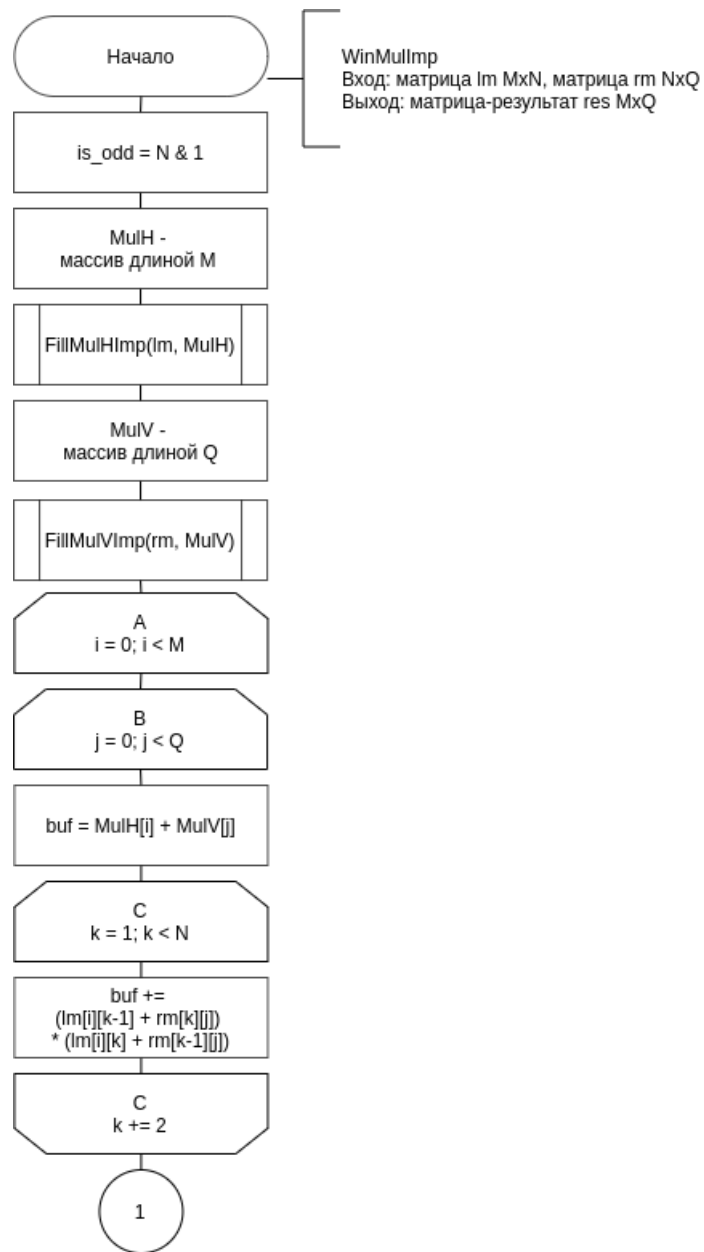


Рисунок 2.5 – Схема оптимизированного алгоритма Винограда для перемножения матриц

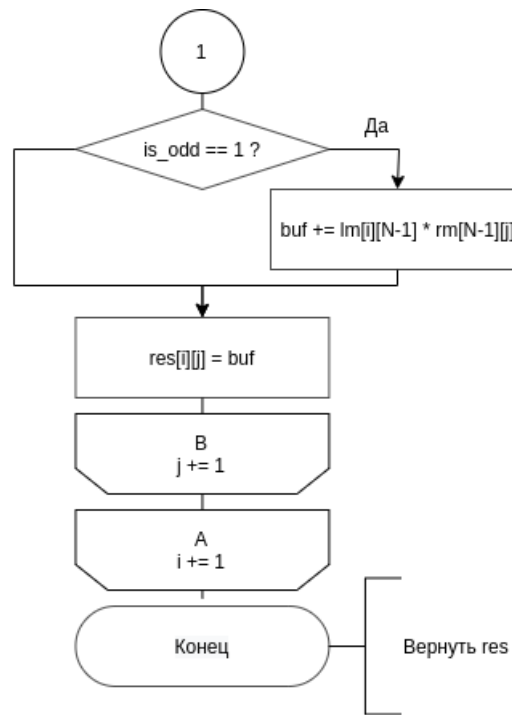


Рисунок 2.6 – Продолжение схемы оптимизированного алгоритма Винограда для перемножения матриц

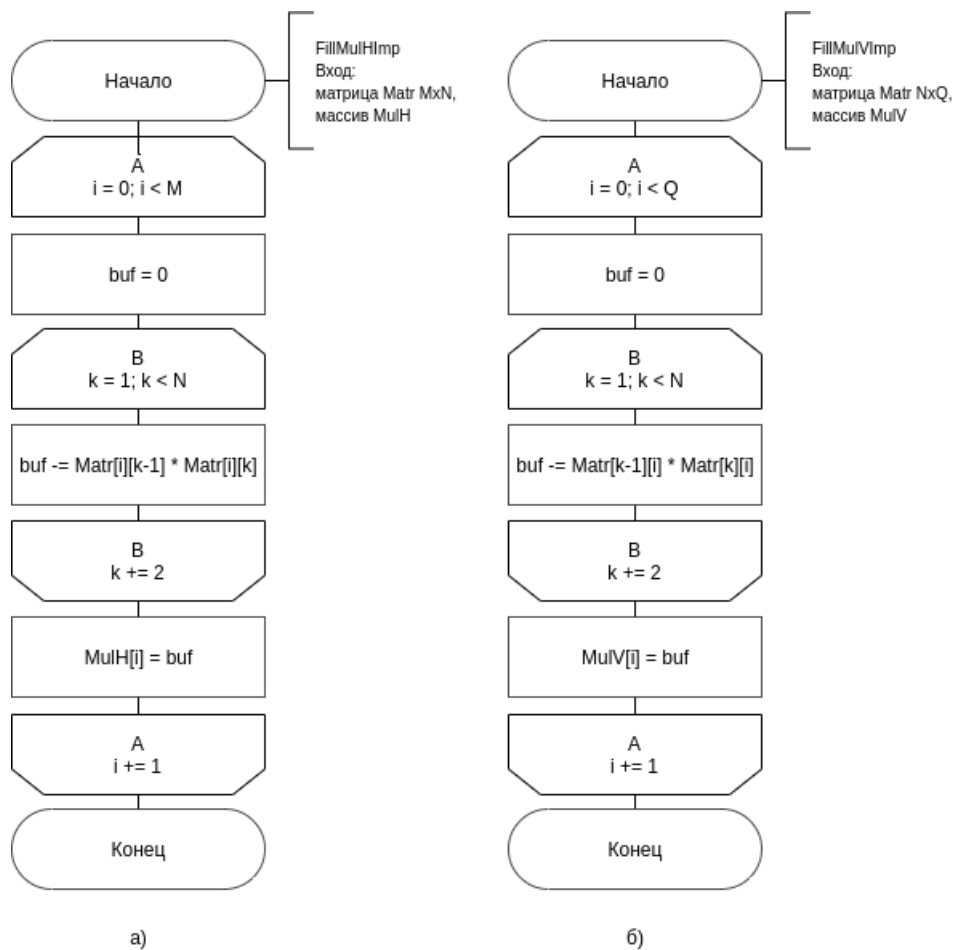


Рисунок 2.7 – Схемы алгоритмов предварительных вычислений в оптимизированном алгоритме Винограда. Заполнение MulH (а) и MulV (б)

Заметим, что для алгоритма Винограда худшим случаем являются матрицы нечетной размерности, а лучшим – четной, поскольку в таком случае отпадает необходимость в последнем цикле.

Данный алгоритм может быть улучшен путем использования следующих оптимизаций:

- использование конструкции $a += x$ вместо $a = a + x$;
- накопление результата в буфер, а затем помещение буфера в ячейку;
- замена в циклах $k < N/2$, $k += 1$ на $k < N$, $k += 2$;
- предварительное вычисление констант таких как $is_odd = N \% 2$;
- замена операции взятия остатка по модулю два ($N \% 2$) на битовое умножение на 1 ($N \& 1$);
- объединение основного внешнего цикла с последним.

2.2 Модель вычислений

Для последующего вычисления трудоемкости введём модель вычислений:

1. Трудоемкость базовых операций:

Пусть операции из (2.1) имеют единичную трудоемкость.

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

Пусть операции из (2.2) имеют трудоемкость 2.

$$/, *, \% \quad (2.2)$$

2. Трудоемкость условного оператора “if условие then A else B” рассчитывается, как (2.3).

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

3. Трудоемкость цикла рассчитывается по С-подобной модели, то есть “for (i = 0; i < N; i += 1)”, как (2.4).

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

4. Трудоемкость вызова функции равна 0.

2.3 Трудоемкость алгоритмов

2.3.1 Стандартный алгоритм перемножения матриц

Трудоёмкость стандартного алгоритма перемножения матриц складывается из трудоемкостей:

- внешнего цикла по $i \in [1..M]$, трудоёмкость которого: $f = 2 + M \cdot (2 + f_{body})$;
- цикла по $j \in [1..Q]$, трудоёмкость которого: $f = 2 + Q \cdot (2 + f_{body})$;
- скалярного умножения двух векторов - цикл по $k \in [1..N]$, трудоёмкость которого: $f = 2 + 11N$.

Трудоёмкость стандартного алгоритма равна трудоёмкости внешнего цикла, можно вычислить ее, подставив циклы тела (2.5):

$$f_{base} = 2 + M \cdot (4 + Q \cdot (4 + 11N)) = 2 + 4M + 4MQ + 11MQN \approx 11MQN \quad (2.5)$$

2.3.2 Алгоритм Винограда

Трудоёмкость алгоритма Винограда для перемножения матриц складывается из трудоемкостей:

1. Создания векторов $MulH$ и $MulV$ (2.6):

$$f_{create} = f_{MulH} + f_{MulV} \quad (2.6)$$

2. Заполнения вектора $MulH$ (2.7):

$$f_{MulH} = 2 + M(2 + 4 + \frac{N}{2}(4 + 6 + 2 + 1 + 2 \cdot 3)) = \frac{19}{2}MN + 6M + 2 \quad (2.7)$$

3. Заполнения вектора $MulV$ (аналогично 2.7):

$$f_{MulV} = \frac{19}{2}QN + 6Q + 2; \quad (2.8)$$

4. Цикла заполнения матрицы для чётных размеров (2.9):

$$f_{cycle} = 2 + M(4 + Q(13 + \frac{N}{2} \cdot 32)) = 16MNQ + 13MQ + 4M + 2 \quad (2.9)$$

5. Цикла, для дополнения умножения суммой последних нечётных строки и столбца, если общий размер нечётный (2.10):

$$f_{last} = 3 + \begin{cases} 0, & \text{если } N - \text{четно} \\ 13MQ + 4M + 2, & \text{иначе.} \end{cases} \quad (2.10)$$

Суммарная трудоёмкость:

$$\begin{aligned} f_{win-common} = f_{create} + f_{cycle} + f_{last} = 16MNQ + 13MQ + \frac{19}{2}(MN + QN) + \\ + 10M + 6Q + 9 + \begin{cases} 0, & \text{если } N - \text{четно} \\ 13MQ + 4M + 2, & \text{иначе.} \end{cases} \end{aligned} \quad (2.11)$$

Итого, для худшего случая (нечётный размер матриц):

$$f_{win-worst_case} = 16MNQ + 26MQ + \frac{19}{2}(MN + QN) + 14M + 6Q + 11; \quad (2.12)$$

Для лучшего случая (чётный размер матриц):

$$f_{win-best_case} = 16MNQ + 13MQ + \frac{19}{2}(MN + QN) + 10M + 6Q + 9; \quad (2.13)$$

2.3.3 Оптимизированный Алгоритм Винограда

Трудоёмкость оптимизированного алгоритма Винограда для перемножения матриц складывается из трудоемкостей:

1. Создания векторов $MulH$ и $MulV$ (2.14):

$$f_{create} = f_{MulH} + f_{MulV} \quad (2.14)$$

2. Заполнения вектора $MulH$ (2.15):

$$f_{MulH} = 2 + M(2 + 1 + 2 + \frac{N}{2}(2 + 4 + 2 + 1 + 1) + 2) = 5MN + 7M + 2 \quad (2.15)$$

3. Заполнения вектора $MulV$ (аналогично 2.15):

$$f_{MulV} = 5QN + 7Q + 2 \quad (2.16)$$

4. Цикла заполнения матрицы для чётных размеров (2.17):

$$\begin{aligned} f_{cycle} &= 2 + M(4 + Q(8 + \frac{17N}{2}) + 4 + \begin{cases} 0, & \text{если } N - \text{чётно} \\ 9, & \text{иначе.} \end{cases}) = \\ &= \frac{17}{2}MNQ + MQ \cdot \begin{cases} 12, & \text{если } N - \text{чётно} \\ 21, & \text{иначе.} \end{cases} + 4M + 2 \end{aligned} \quad (2.17)$$

5. Определения четности N (2.18):

$$f_{odd-check} = 2; \quad (2.18)$$

Суммарная трудоёмкость:

$$\begin{aligned} f_{win_imp-common} &= f_{create} + f_{cycle} + f_{odd-check} = \frac{17}{2}MNQ + 5(MN + QN) + \\ &+ 9M + 7Q + 8 + \begin{cases} 12MQ & \text{если } N - \text{чётно} \\ 21MQ & \text{иначе.} \end{cases} \end{aligned} \quad (2.19)$$

Итого, для худшего случая (нечётный размер матриц):

$$f_{win_imp-worst_case} = \frac{17}{2}MNQ + 5(MN + QN) + 21MQ + 9 + 7Q + 8; \quad (2.20)$$

Для лучшего случая (чётный размер матриц):

$$f_{win_imp-best_case} = \frac{17}{2}MNQ + 5(MN + QN) + 12MQ + 9 + 7Q + 8; \quad (2.21)$$

2.4 Вывод

В данном разделе были рассмотрены схемы алгоритмов умножения матриц, введена модель вычисления трудоёмкости, рассчитаны трудоёмкости алгоритмов. Из формул 2.5, 2.13, 2.21 видно, что все три алгоритма имеют зависимость от размерности матриц, однако оптимизированный алгоритм Винограда имеет наилучший коэффициент перед слагаемым MQN , равный 8, из чего следует, что он должен быть наиболее эффективным.

3 Технологическая часть

3.1 Требования к ПО

К программе предъявляется ряд требований:

- предоставить возможность работы в двух режимах: проведения эксперимента и ручного тестирования;
- в режиме ручного тестирования предоставить пользователю как самостоятельно осуществлять ввод двух целочисленных матриц, которые необходимо перемножить, так и использовать автозаполнение матриц заданных размерностей псевдослучайными числами от 0 до 99;
- выходными данными программы в ручном режиме являются три матрицы - результаты перемножения матриц с использованием классического алгоритма, алгоритма Винограда и оптимизированного алгоритма Винограда;
- в режиме проведения эксперимента выходными данными является результат эксперимента в текстовом виде, а также графики в виде файла с изображением.

3.2 Средства реализации

Для реализации алгоритмов перемножения матриц был выбран язык программирования Python3[3], что обусловлено простотой и скоростью написания программ, а также наличием встроенных библиотек для построения графиков функций и тестирования. В качестве среды разработки был выбран PyCharm[4], как наиболее популярная IDE для Python3.

3.3 Функциональное тестирование

При разработке функциональных тестов были выделены следующие классы эквивалентности:

- хотя бы один из размеров матрицы неположительный;
- кол-во столбцов первой матрицы не совпадает с кол-вом строк второй;
- хотя бы одна матрица вырождена в вектор-строку или вектор-столбец;
- обе матрицы размером 1 x 1;
- обе матрицы являются квадратами с четным размером;
- обе матрицы являются квадратами с нечетным размером;
- произвольные прямоугольные матрицы корректных размеров.

В соответствии с данными классами эквивалентности были разработаны тесты, представленные в таблице 3.1.

Таблица 3.1 – Тестирование функций

Размеры M1	Размеры M2	M1	M2	Результат
−1 x 2	—	—	—	Ошибка
2 x 3	2 x 2	—	—	Ошибка
1 x 2	2 x 2	$\begin{pmatrix} 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 2 & 2 \end{pmatrix}$
1 x 1	1 x 1	$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 3 \end{pmatrix}$	$\begin{pmatrix} 6 \end{pmatrix}$
2 x 2	2 x 2	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$
3 x 3	3 x 3	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 3 & 3 \end{pmatrix}$
2 x 3	3 x 2	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 3 & 3 \\ 3 & 3 \end{pmatrix}$

3.4 Реализация алгоритмов

Реализации классического алгоритма перемножения матриц, алгоритма Винограда, а также оптимизированного алгоритма Винограда соответственно представлены в листингах А.1–А.3.

3.5 Вывод

В данном разделе были выделены классы эквивалентности для операции перемножения матриц, на основе которых разработаны функциональные тесты для ПО, также были реализованы сами алгоритмы перемножения матриц на языке Python3.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялся эксперимент:

- операционная система: Ubuntu[5] Linux x86_64;
- память: 16 GiB;
- процессор: AMD Ryzen™ 7 4700U[6].

4.2 Проведение эксперимента

Эксперимент проводился на квадратных матрицах четного и нечетного размеров из диапазона $[50; 250]$ с шагом 50, в ячейках которых находились псевдослучайные числа из диапазона $[0; 99]$.

При помощи встроенного модуля `timeit`[7] языка Python3 было произведено 50 замеров для каждого размера матриц, после чего определено среднее время выполнения алгоритма.

Результаты замеров приведены в таблицах 4.1–4.2. На основании результатов замеров построены графики зависимости времени выполнения алгоритма от размеров матрицы. Данные графики представлены на рисунках 4.1 и 4.2.

Таблица 4.1 – Таблица замеров времени выполнения алгоритмов в секундах для четных размеров

Размер	Простой	Виноград	Виноград (оптимиз.)
50	0.044	0.047	0.031
100	0.354	0.367	0.242
150	1.254	1.242	0.841
200	2.834	2.940	1.927
250	5.613	5.740	3.880

Таблица 4.2 – Таблица замеров времени выполнения алгоритмов в секундах для нечетных размеров

Размерность	Простой	Виноград	Виноград (оптимиз.)
51	0.051	0.053	0.038
101	0.378	0.407	0.286
151	1.255	1.374	0.889
201	2.991	3.216	2.077
251	5.889	6.194	3.972

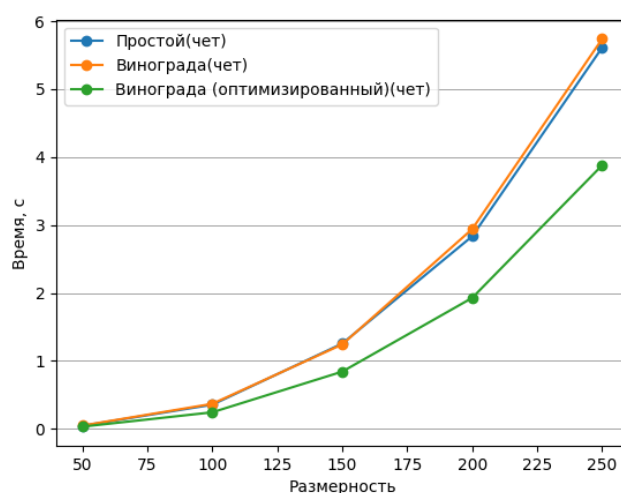


Рисунок 4.1 – Зависимость времени работы алгоритмов при чётных размерах матрицы

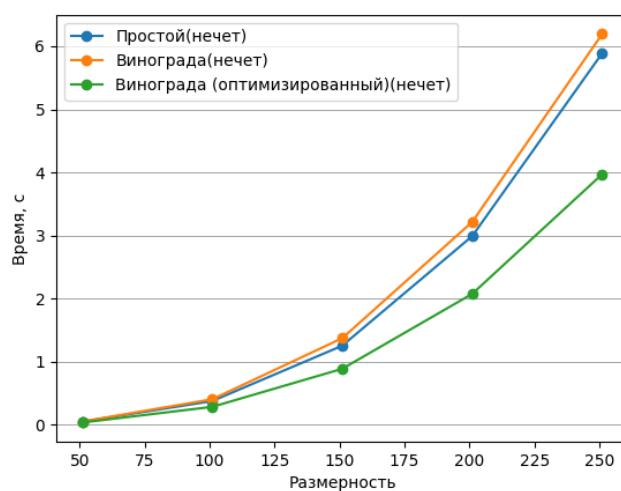


Рисунок 4.2 – Зависимость времени работы алгоритмов при нечётных размерах матрицы

4.3 Вывод

Оптимизированный алгоритм Винограда для перемножения матриц работает в ≈ 1.5 раза **быстрее** классического алгоритма перемножения матриц при четных размерностях и ≈ 1.4 раза – нечетных. В то же время, неоптимизированный алгоритм Винограда работает в ≈ 1.03 раза **медленнее** классического алгоритма перемножения матриц при четных размерностях и ≈ 1.07 раза – нечетных.

Заключение

В ходе выполнения лабораторной работы была проделана следующая работа:

- реализованы классический алгоритм перемножения матриц и алгоритм Винограда;
- реализован оптимизированный алгоритм Винограда;
- рассчитана трудоемкость для каждого алгоритма;
- проведен анализ временных характеристик экспериментально;
- сделаны выводы по поводу эффективности всех реализованных алгоритмов.

Литература

- [1] Бахвалов Н.С. Численные методы. Наука, 1987.
- [2] Кормен Т. Алгоритмы: построение и анализ. МЦНТО, 1999.
- [3] Python 3.9.7 documentation [Электронный ресурс]. Режим доступа: <https://docs.python.org/3/> (дата обращения: 28.09.2021).
- [4] PyCharm: IDE для профессиональной разработки на Python [Электронный ресурс]. Режим доступа: <https://www.jetbrains.com/ru-ru/pycharm/> (дата обращения: 28.09.2021).
- [5] Ubuntu 20.04.3 LTS [Электронный ресурс]. Режим доступа: <https://ubuntu.com/download/desktop> (дата обращения: 28.09.2020).
- [6] Процессор AMD Ryzen™ 7 4700U [Электронный ресурс]. Режим доступа: <https://www.amd.com/ru/products/apu/amd-ryzen-7-4700u> (дата обращения: 28.09.2021).
- [7] timeit — Measure execution time of small code snippets [Электронный ресурс]. Режим доступа: <https://docs.python.org/3/library/timeit.html> (дата обращения: 28.09.2021).

А Листинги

Листинг А.1 – Функция классического перемножения матриц

```
1  def simple_mul(lm: MatrixInt, rm: MatrixInt) -> MatrixInt:
2      m, n, q = lm.m, lm.n, rm.n
3      res = MatrixInt(m, q)
4      i = 0
5      while i < m:
6          j = 0
7          while j < q:
8              k = 0
9              while k < n:
10                 res[i][j] += lm[i][k] * rm[k][j]
11                 k += 1
12             j += 1
13         i += 1
14     return res
```

Листинг А.2 – Функция для перемножения матриц алгоритмом Винограда

```
1  def win_mul(lm: MatrixInt, rm: MatrixInt) -> MatrixInt:
2      m, n, q = lm.m, lm.n, rm.n
3
4      mul_h, mul_v = VectorInt(m), VectorInt(q)
5      fill_mul_h(mul_h, lm)
6      fill_mul_v(mul_v, rm)
7
8      res = MatrixInt(m, q)
9      i = 0
10     while i < m:
11         j = 0
12         while j < q:
13             k = 0
14             res[i][j] = - mul_h[i] - mul_v[j]
15             while k < n // 2:
16                 res[i][j] = res[i][j] \
17                     + (lm[i][2 * k] + rm[2 * k + 1][j]) \
18                     * (lm[i][2 * k + 1] + rm[2 * k][j])
19                 k += 1
20             j += 1
21         i += 1
22
23     if n % 2 == 1:
24         i = 0
25         while i < m:
26             j = 0
27             while j < q:
```

```

28         res[i][j] = res[i][j] + lm[i][n - 1] * rm[n - 1][j]
29         j += 1
30     i += 1
31     return res
32
33
34 def fill_mul_h(mul_h: VectorInt, matrix: MatrixInt) -> None:
35     m, n = matrix.m, matrix.n
36
37     i = 0
38     while i < m:
39         k = 0
40         while k < n // 2:
41             mul_h[i] = mul_h[i] + matrix[i][2 * k] * matrix[i][2 * k + 1]
42             k += 1
43         i += 1
44
45
46 def fill_mul_v(mul_v: VectorInt, matrix: MatrixInt) -> None:
47     n, q = matrix.m, matrix.n
48
49     i = 0
50     while i < q:
51         k = 0
52         while k < n // 2:
53             mul_v[i] = mul_v[i] + matrix[2 * k][i] * matrix[2 * k + 1][i]
54             k += 1
55         i += 1

```

Листинг А.3 – Функция для перемножения матриц оптимизированным алгоритмом Винограда

```

1 def win_mul_imp(lm: MatrixInt, rm: MatrixInt) -> MatrixInt:
2     m, n, q = lm.m, lm.n, rm.n
3
4     mul_h = VectorInt(m)
5     fill_mul_h_imp(mul_h, lm)
6
7     mul_v = VectorInt(q)
8     fill_mul_v_imp(mul_v, rm)
9
10    is_odd = n & 1
11    res = MatrixInt(m, q)
12    i = 0
13    while i < m:
14        j = 0
15        while j < q:
16            buf, k = mul_h[i] + mul_v[j], 1

```

```

17         while k < n:
18             buf += (lm[i][k - 1] + rm[k][j]) * (lm[i][k] + rm[k - 1][j])
19             k += 2
20         if is_odd == 1:
21             buf += lm[i][n - 1] * rm[n - 1][j]
22
23         res[i][j] = buf
24         j += 1
25     i += 1
26 return res
27
28
29 def fill_mul_h_imp(mul_h: VectorInt, matrix: MatrixInt) -> None:
30     m, n = matrix.m, matrix.n
31
32     i = 0
33     while i < m:
34         buf, k = 0, 1
35         while k < n:
36             buf -= matrix[i][k - 1] * matrix[i][k]
37             k += 2
38         mul_h[i] = buf
39         i += 1
40
41
42 def fill_mul_v_imp(mul_v: VectorInt, matrix: MatrixInt) -> None:
43     n, q = matrix.m, matrix.n
44
45     i = 0
46     while i < q:
47         buf, k = 0, 1
48         while k < n:
49             buf -= matrix[k - 1][i] * matrix[k][i]
50             k += 2
51         mul_v[i] = buf
52         i += 1

```