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 Алгоритмы сортировки

Студент Ивахненко Д.А.

Группа ИУ7-56Б

Преподаватель Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

Алгоритм сортировки — это алгоритм для упорядочивания эле ментов в массиве каких-либо данных. Другими словами, это перегруппи ровка данных по какому-либо ключу. По некоторым источникам, именно программа сортировки стала первой программой для вычислительных машин.

В основном алгоритмы сортировки являются не конечной целью, а промежуточной - отсортированные данные намного легче обрабаты вать. Например, чтобы удалить дупликаты в массиве, будет разумно сначала отсортировать массив. Также намного легче производить поиск какой-либо информации в заранее отсортированном массиве. Ассимпто тическая сложность при поиске в отсортированном массиве составляет O(log(n)) против O(n) в произвольном массиве данных.

Каждый алгоритм сортировки имеет свои особенности реализа ции, однако в конечном итоге эту задачу можно разбить на следующие шаги:

1. Сравнение пары элементов массива. Для этого нужно указать способ сравнения (бинарное отношение) для двух элементов, например, учесть природу данных или задать ключ в случае, если записи в массиве имеют несколько полей;

2. Перестановка двух элементов, которые могут располагаться как последовательно друг за другом, так и находиться в разных частях массива;

3. Алгоритм должен завершать свою работу, когда массив становится полностью упорядоченным.

Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти. Обычно асимптотическая сложность алгоритмов сортировок массивов размерностью n лежит в диапазоне от $O(n)$ до $O(n^2)$

Целью работы явлется анализ эффективности алгоритмов сортировки. Для достижения поставленной цели необходимо выполнить следующие задачи: Для достижения поставленной цели необходимо выполнить следующие задачи:

- реализовать три алгоритма сортировки: пузрьком, вставками и выбором;

- 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

В данном разделе будут описаны ключевые идеи каждого из выбранных алгоритмов сортировки[1].

1.1 Алгоритмы

Сортировка пузырьком является одним из простейших алгоритмов сортировки данных. Пусть имеется массив для сортировки размерностью N . В классической версии массив обходится слева направо. Алгоритм состоит из повторяющихся проходов по этому массиву - за каждый про ход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется перестановка элементов так, чтобы пара стала упорядоченной. Проходы по массиву повторяются 1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. В результате каждого прохода по внутреннему циклу определяет ся очередной максимум подмассива, этот максимум помещается в конец рядом с предыдущим наибольшим элементом, вычисленным на преды дущем проходе, а наименьший элемент перемещается ближе к началу массива («всплывает» до нужной позиции, как пузырёк в воде — отсюда и название алгоритма).

1.2 Алгоритм Винограда для перемножения матриц

Если рассмотреть результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее [?].

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

скалярное произведение равно:

$$V \cdot W = v_1 w_1 + v_2 w_2 + v_3 w_3 + v_4 w_4 \tag{1.1}$$

что эквивалентно

$$V \cdot W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1 v_2 - v_3 v_4 - w_1 w_2 - w_3 w_4. \tag{1.2}$$

На первый взгляд неочевидно, в чем преимущество выражения **??**, поскольку оно требует вычисления большего количества операций, чем классическое **??**: вместо четырёх умножений – шесть, а вместо трёх сложений – десять. Заметим, что выражение в правой части формулы **??** 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 рассчеты.

2 Конструкторская часть

2.1 Разработка алгоритмов

На рисунке ?? приведена схема классического алгоритма для перемножения матриц.

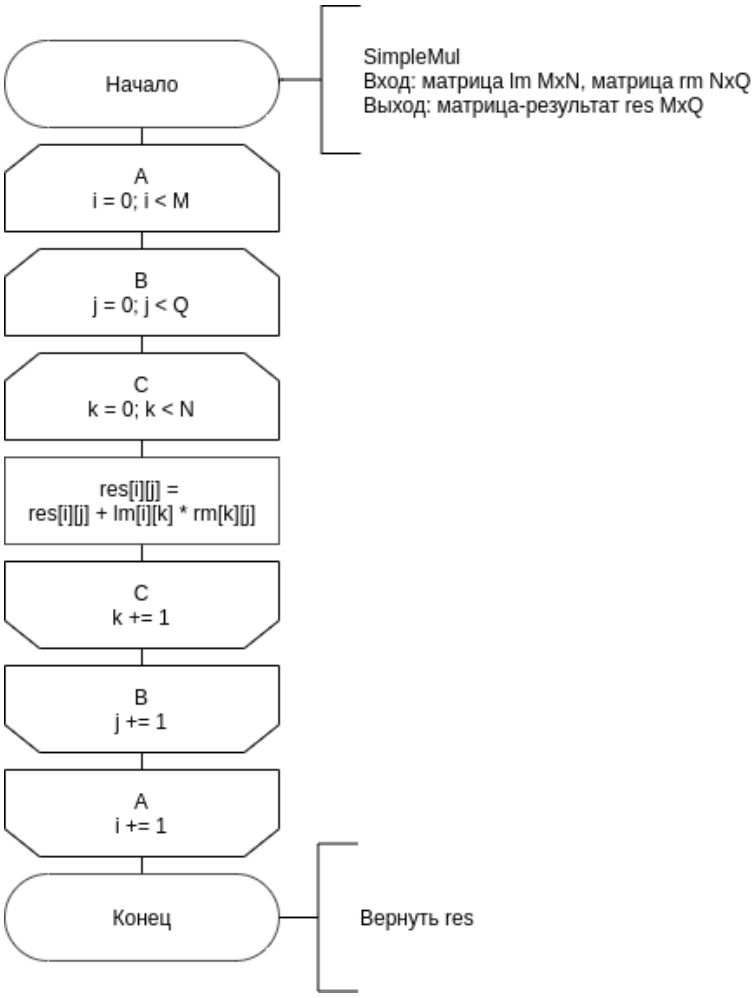

Рисунок 2.1 – Схема классического алгоритма для перемножения матриц

На рисунках ?? – ?? приведены схемы алгоритма Винограда для перемножения матриц.

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

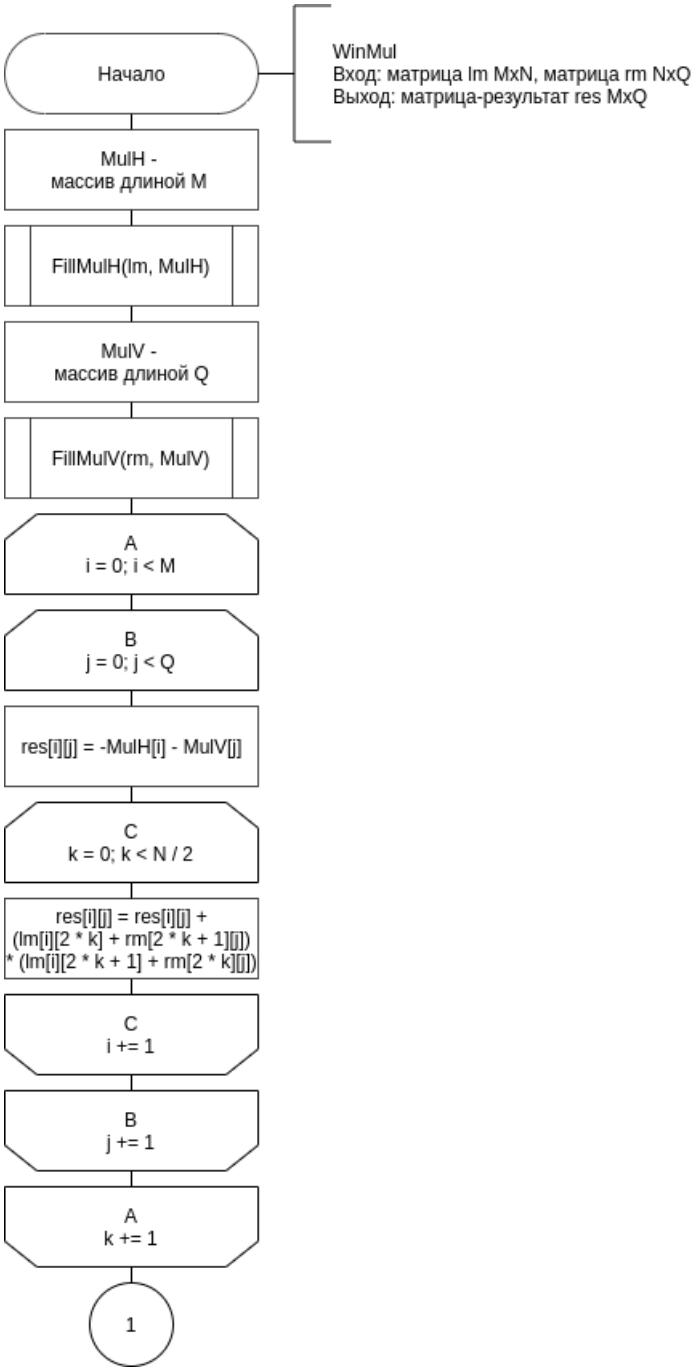

Рисунок 2.2 – Схема алгоритма Винограда для перемножения матриц

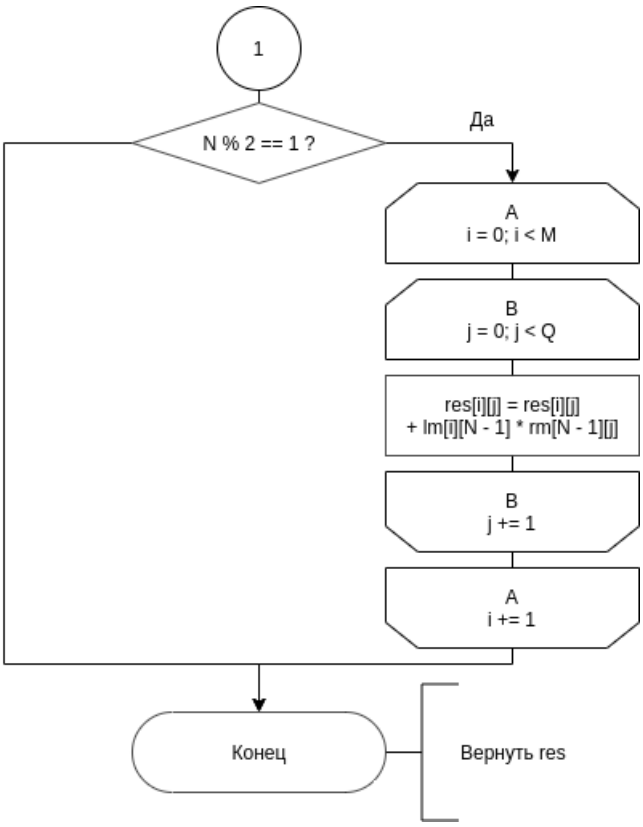

Рисунок 2.3 – Продолжение схемы алгоритма Винограда для перемножения матриц

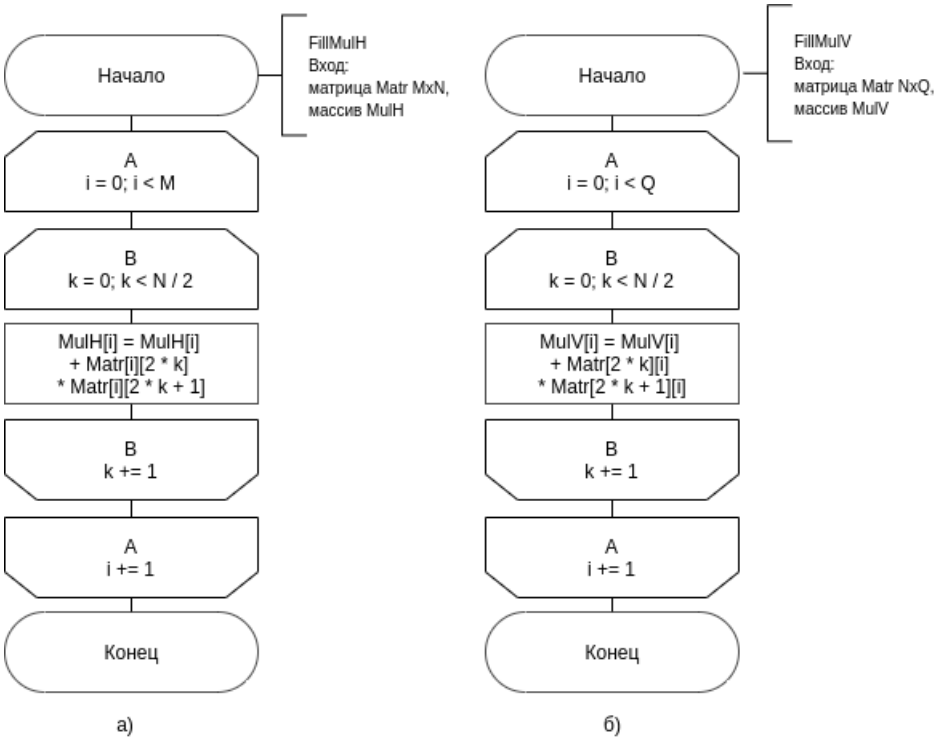

Рисунок 2.4 – Схемы алгоритмов предварительных вычислений в алгоритме Винограда. Заполнение векторов MulH (а) и MulV (б)

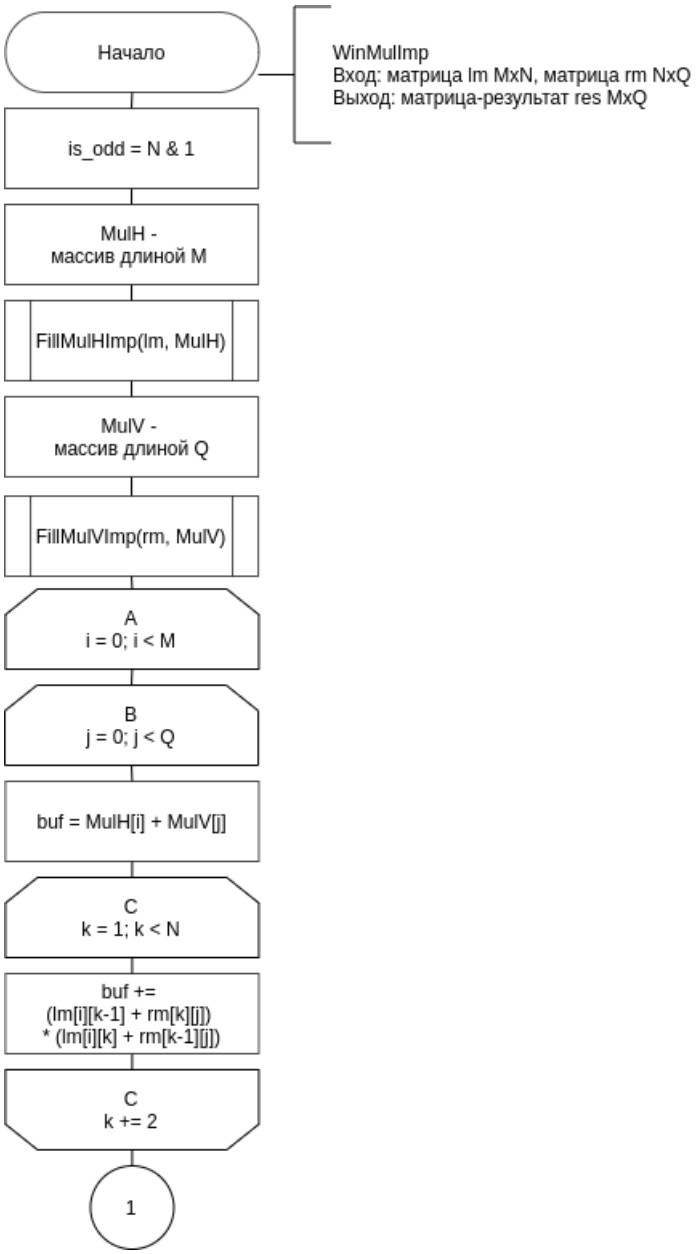

Рисунок 2.5 – Схема оптимизированного алгоритма Винограда для перемножения матриц

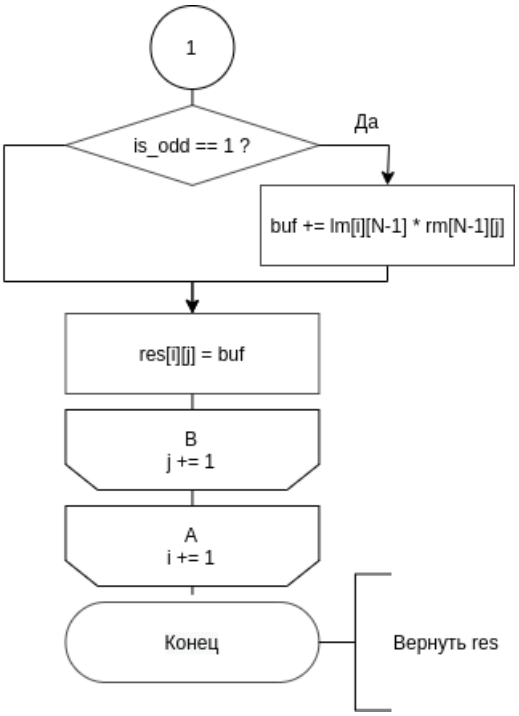

Рисунок 2.6 – Продолжение схемы оптимизированного алгоритма Винограда для перемножения матриц

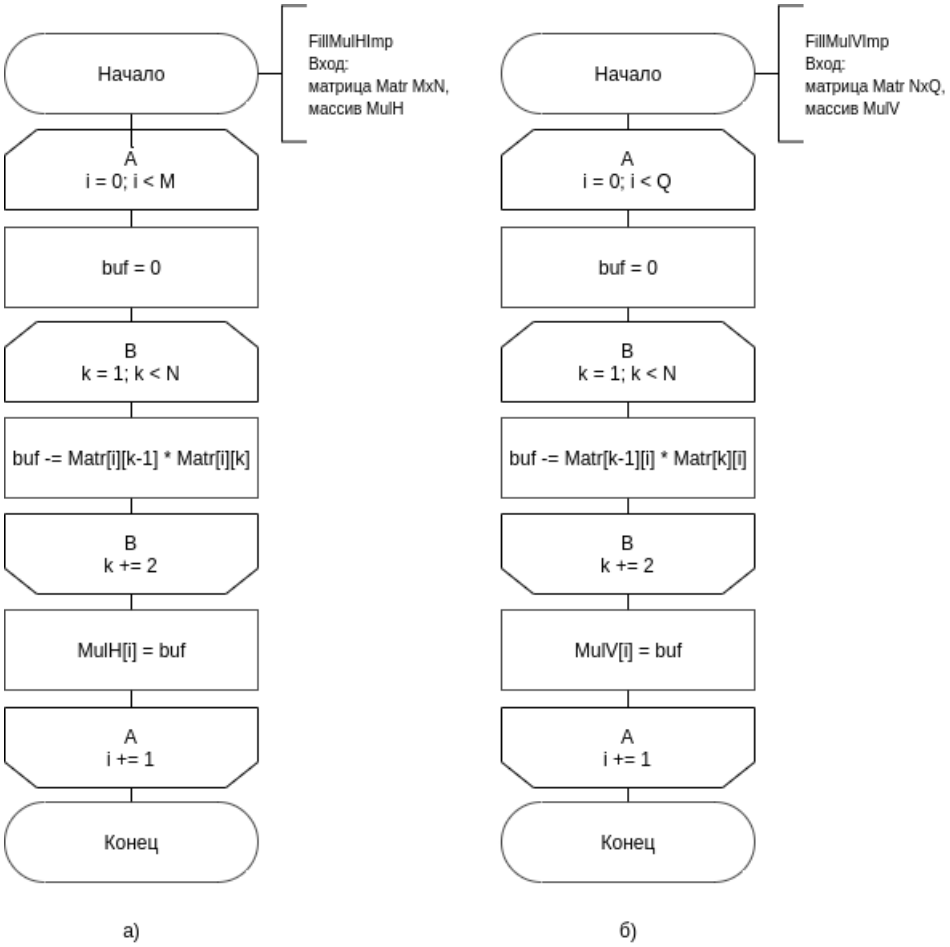

Рисунок 2.7 – Схемы алгоритмов предварительных вычислений в опитмизированном алгоритме Винограда. Заполнение MulH (а) и MulV (б)

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 (**??**) имеют единичную трудоемкость.

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

   Пусть операции из (**??**) имеют трудоемкость 2.

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

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

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

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

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

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 (**??**):

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

2.3.2 Алгоритм Винограда

Трудоёмкость алгоритма Винограда для перемножения матриц складывается из трудоемкостей:

1. Создания векторов $MulH$ и $MulV$ (**??**):

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

2. Заполнения вектора $MulH$ (**??**):

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

3. Заполнения вектора $MulV$ (аналогично **??**):

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

4. Цикла заполнения матрицы для чётных размеров (**??**):

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

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

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

Суммарная трудоёмкость:

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

Итого, для худшего случая (нечётный размер матриц):

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

Для лучшего случая (чётный размер матриц):

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

2.3.3 Оптимизированный Алгоритм Винограда

Трудоёмкость оптимизированного алгоритма Винограда для перемножения матриц складывается из трудоемкостей:

1. Создания векторов $MulH$ и $MulV$ (**??**):

$$f_{create} = f_{MulH} + f_{MulV} \tag{2.14}$$

2. Заполнения вектора $MulH$ (**??**):

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

3. Заполнения вектора $MulV$ (аналогично **??**):

$$f_{MulV} = 5QN + 7Q + 2 \tag{2.16}$$

4. Цикла заполнения матрицы для чётных размеров (**??**):

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

5. Определения четности N (**??**):

$$f_{odd-check} = 2; \tag{2.18}$$

Суммарная трудоёмкость:

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

Итого, для худшего случая (нечётный размер матриц):

$$f_{win\_imp-worst\_case} = \frac{17}{2}MNQ + 5(MN + QN) + 21MQ + 9 + 7Q + 8; \tag{2.20}$$

Для лучшего случая (чётный размер матриц):

$$f_{win\_imp-best\_case} = \frac{17}{2}MNQ + 5(MN + QN) + 12MQ + 9 + 7Q + 8; \tag{2.21}$$

2.4 Вывод

В данном разделе были рассмотрены схемы алгоритмов умножения матриц, введена модель вычисления трудоёмкости, рассчитаны трудоёмкости алгоритмов. Из формул **??**, **??**, **??** 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 исользованием 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[?], 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`[?], как наиболее популярная IDE для Python3.

3.3 Функциональное тестирование

При разработке функционалых 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 квадртаными с четным размером;
- обе матрицы являются квадртаными с нечетным размером;
- произвольные прямоугольные матрицы корректных размеров.

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

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 **??**–**??**.

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[?] Linux x86_64;
- память: 16 GiB;
- процессор: AMD Ryzen™ 7 4700U[?].

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

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

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

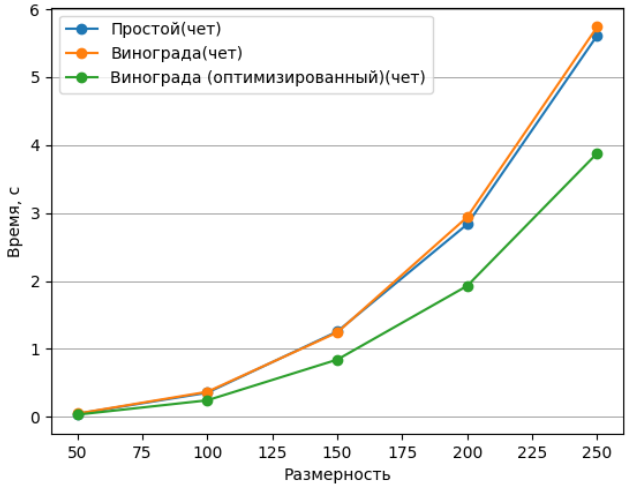

Рисунок 4.1 – Зависимость времени работы алгоритмов при чётных размерах матрицы

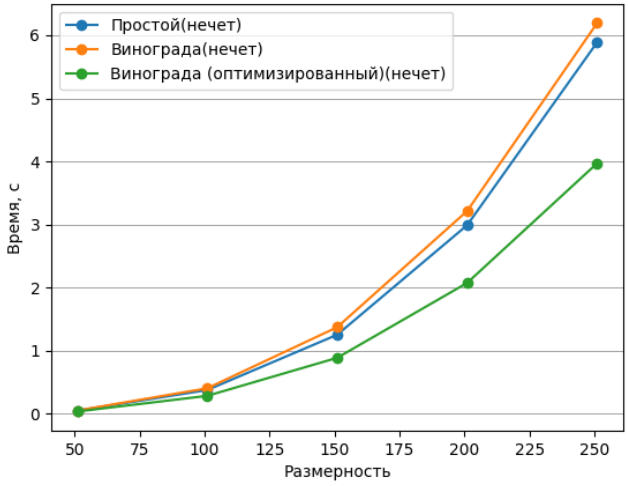

Рисунок 4.2 – Зависимость времени работы алгоритмов при нечётных размерах матрицы

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.

А Листинги

Листинг А.1 – Функция классического перемножения матриц

```
    def simple_mul(lm: MatrixInt, rm: MatrixInt) -> MatrixInt:
        m, n, q = lm.m, lm.n, rm.n
        res = MatrixInt(m, q)
        i = 0
        while i < m:
            j = 0
            while j < q:
                k = 0
                while k < n:
                    res[i][j] += lm[i][k] * rm[k][j]
                    k += 1
                j += 1
            i += 1
        return res
```

Листинг А.2 – Функция для перемножения матриц алгоритмом Винограда

```
def win_mul(lm: MatrixInt, rm: MatrixInt) -> MatrixInt:
    m, n, q = lm.m, lm.n, rm.n

    mul_h, mul_v = VectorInt(m), VectorInt(q)
    fill_mul_h(mul_h, lm)
    fill_mul_v(mul_v, rm)

    res = MatrixInt(m, q)
    i = 0
    while i < m:
        j = 0
        while j < q:
            k = 0
            res[i][j] = - mul_h[i] - mul_v[j]
            while k < n // 2:
                res[i][j] = res[i][j] \
                            + (lm[i][2 * k] + rm[2 * k + 1][j]) \
                            * (lm[i][2 * k + 1] + rm[2 * k][j])
                k += 1
            j += 1
        i += 1

    if n % 2 == 1:
        i = 0
        while i < m:
            j = 0
            while j < q:
```

```
                res[i][j] = res[i][j] + lm[i][n - 1] * rm[n - 1][j]
                j += 1
            i += 1
    return res


def fill_mul_h(mul_h: VectorInt, matrix: MatrixInt) -> None:
    m, n = matrix.m, matrix.n

    i = 0
    while i < m:
        k = 0
        while k < n // 2:
            mul_h[i] = mul_h[i] + matrix[i][2 * k] * matrix[i][2 * k + 1]
            k += 1
        i += 1


def fill_mul_v(mul_v: VectorInt, matrix: MatrixInt) -> None:
    n, q = matrix.m, matrix.n

    i = 0
    while i < q:
        k = 0
        while k < n // 2:
            mul_v[i] = mul_v[i] + matrix[2 * k][i] * matrix[2 * k + 1][i]
            k += 1
        i += 1
```

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

```
def win_mul_imp(lm: MatrixInt, rm: MatrixInt) -> MatrixInt:
    m, n, q = lm.m, lm.n, rm.n

    mul_h = VectorInt(m)
    fill_mul_h_imp(mul_h, lm)

    mul_v = VectorInt(q)
    fill_mul_v_imp(mul_v, rm)

    is_odd = n & 1
    res = MatrixInt(m, q)
    i = 0
    while i < m:
        j = 0
        while j < q:
            buf, k = mul_h[i] + mul_v[j], 1
```

```
            while k < n:
                buf += (lm[i][k - 1] + rm[k][j]) * (lm[i][k] + rm[k - 1][j])
                k += 2
            if is_odd == 1:
                buf += lm[i][n - 1] * rm[n - 1][j]

            res[i][j] = buf
            j += 1
        i += 1
return res


def fill_mul_h_imp(mul_h: VectorInt, matrix: MatrixInt) -> None:
    m, n = matrix.m, matrix.n

    i = 0
    while i < m:
        buf, k = 0, 1
        while k < n:
            buf -= matrix[i][k - 1] * matrix[i][k]
            k += 2
        mul_h[i] = buf
        i += 1


def fill_mul_v_imp(mul_v: VectorInt, matrix: MatrixInt) -> None:
    n, q = matrix.m, matrix.n

    i = 0
    while i < q:
        buf, k = 0, 1
        while k < n:
            buf -= matrix[k - 1][i] * matrix[k][i]
            k += 2
        mul_v[i] = buf
        i += 1
```