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» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

О Т Ч Е Т

по лабораторной работе № 2

Название: Трудоёмкость алгоритмов умножения матриц

Дисциплина: Анализ алгоритмов

Студент ИУ7-52Б (Группа) ______ (Подпись, дата) В.А. Иванов (И.О. Фамилия)

Преподаватель ______ (Подпись, дата) ______ (И.О. Фамилия)

Москва, 2020

Оглавление

Введение

Трудоёмкость алгоритма - это зависимость стоимости операций от линейного размера входа[1].

Модель вычислений трудоёмкости имеет следующие оценки:

- Оценка стоимости базовых операций. Операции $=$, $+$, - и т.д. имеют стоиость 1.

- Оценка циклов. Включает в себя стоимость тела цикла, сравнения и инкремента.

- Оценка условного оператора if. Производится оценка обоих случаев.

Оценка характера трудоёмкости даётся по наиболее быстрорастущему слагаемому. Такая оценка играет важную роль в разработке и анализе алгоритмов, так как позволяет судить об оптимальности использования алгритма при тех или иных входных данных.

В данной лабораторной оценивается трудоёмкость классического алгоритма умножения матриц и алгоритма Винограда.

1. Аналитическая часть

Целью лабораторной работы является оценка трудоёмкости алгоритма умножения матриц и получение практического навыка оптимизации алгоритмов.

Выделены следующие задачи лабораторной работы:

- математическое описание операции умножения матриц;
- описание и реализация алгоритмов умножения матриц;
- описание применённых к алгоритму Винограда способов оптимизации;
- проведение замеров процессорного времени работы алгоритмов при различных размерах матриц (серия экспериментов для чётного размера и для нечётного);
- оценка трудоёмкости алгоритов;
- проведение сравнительного анализа алгоритмов на основании экспериментов.

Умножение матриц - операция над матрицами $[MxN]$ и $B[NxQ]$. Результатом операции является матрица C размерами $M * Q$, в которой элемент $c_{i,j}$ задаётся формулой

$$c_{i,j} = \sum_{k=1}^{N} (a_{i,k} \cdot b_{k,j}) \tag{1.1}$$

2. Конструкторская часть

Рассмотрим и произведём вычисление трудоёмкости для классического алгоритма и алгоритма Винограда для умножения матриц $[MxN]$ и $B[NxQ]$

2.1. Классический алгоритм умножения

Данный алгоритм непосредственно использует вышеприведённую формулу. Для вычисления каждого элемента матрицы С совершается циклический обход k элементов из таблиц A и B.

Схема алгоритма приведена на рисунке 2.1.

2.2. Алгоритм Винограда

Цель алгоритма заключается в сокращении доли умножений в самом трудоёмком участке кода. Основная идея заключается в следующем.

Пусть u, v - элементы матриц A, B соотв., участвующие в вычислении значения элемента матрицы С. Тогда данный элемент вычисляется как $u_1v_1 + u_2v_2 + u_3v_3 + u_4v_4$. Такое выражение можно представить как $(u_1 + v_2)(v_1 + u_2) + (u_3 + v_4)(v_3 + u_4) - u_1u_2 - u_3u_4 - v_1v_2 - v_3v_3)$. В этом выражении вычитаемые можно вычислить однократно и применить их для всех столбцов и строк, где они используются. Таким образом можно снизить трудоёмкость алгоритма за счёт снижения количества операций.

В случае, если матрица нечётный размер N, требуется производить дополнительные вычисления для крайних строк и столбцов. Таким образом, алгоритм наиболее эффективен в случае матриц, у которых N является чётным.

Схема алгоритма приведена на рисунках 2.2. и 2.3.

2.3. Требования к программному обеспечению

Для полноценной проверки и оценки алгоритмов необходимо выполнить следующее.

1. Обеспечить возможность консольного ввода двух матриц и выбора алгоритма для умножения. Программа должна вывести результирующую матрцу.

2. Реализовать функцию замера процессорного времени, затраченного функцией. Для этого также создать возможность ввода размера матрицы, на которых будет выполнен замер.

2.4. Заготовки тестов

При проверке алгоритмов необходимо будет использовать следующие классы тестов:

- матрицы размером 1x1;
- две или одна пустая матрица;
- квадратные матрицы;
- чётный и нечётный размер N;

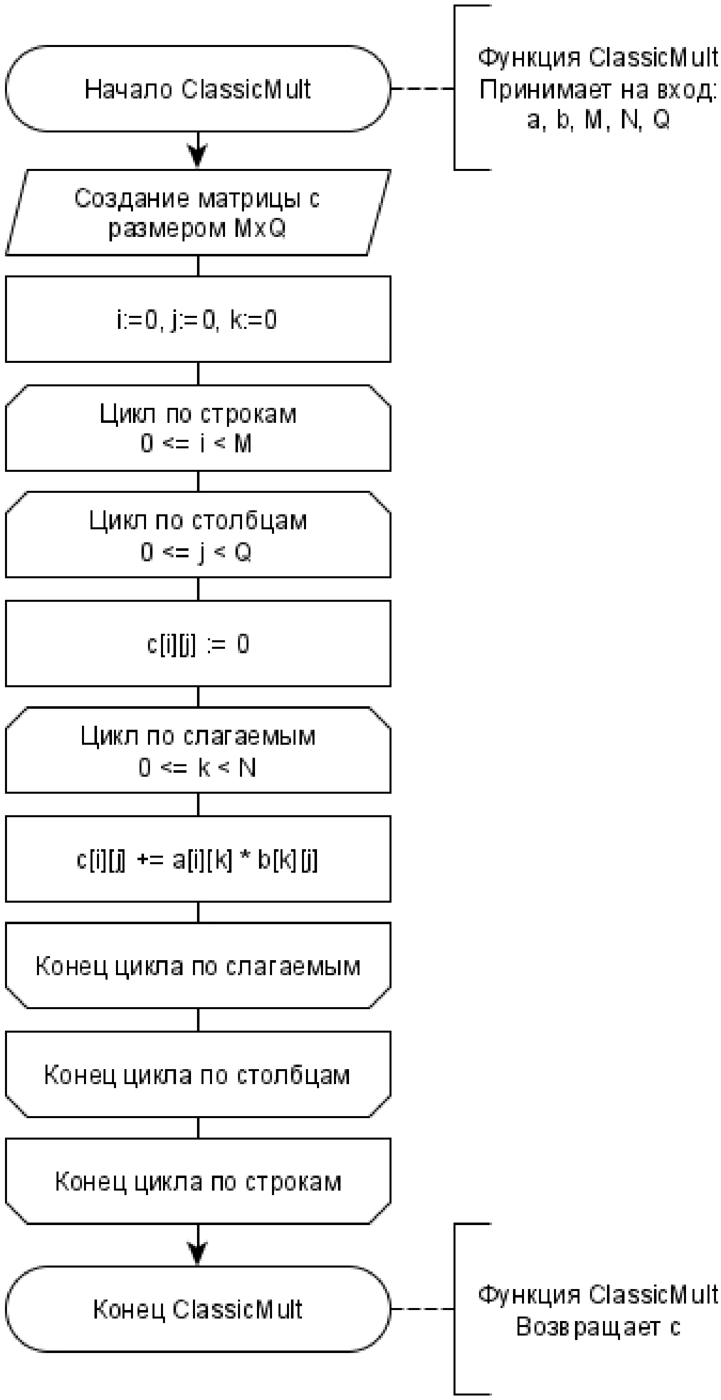

Рис. 2.1: Классический алгоритм умножения матриц

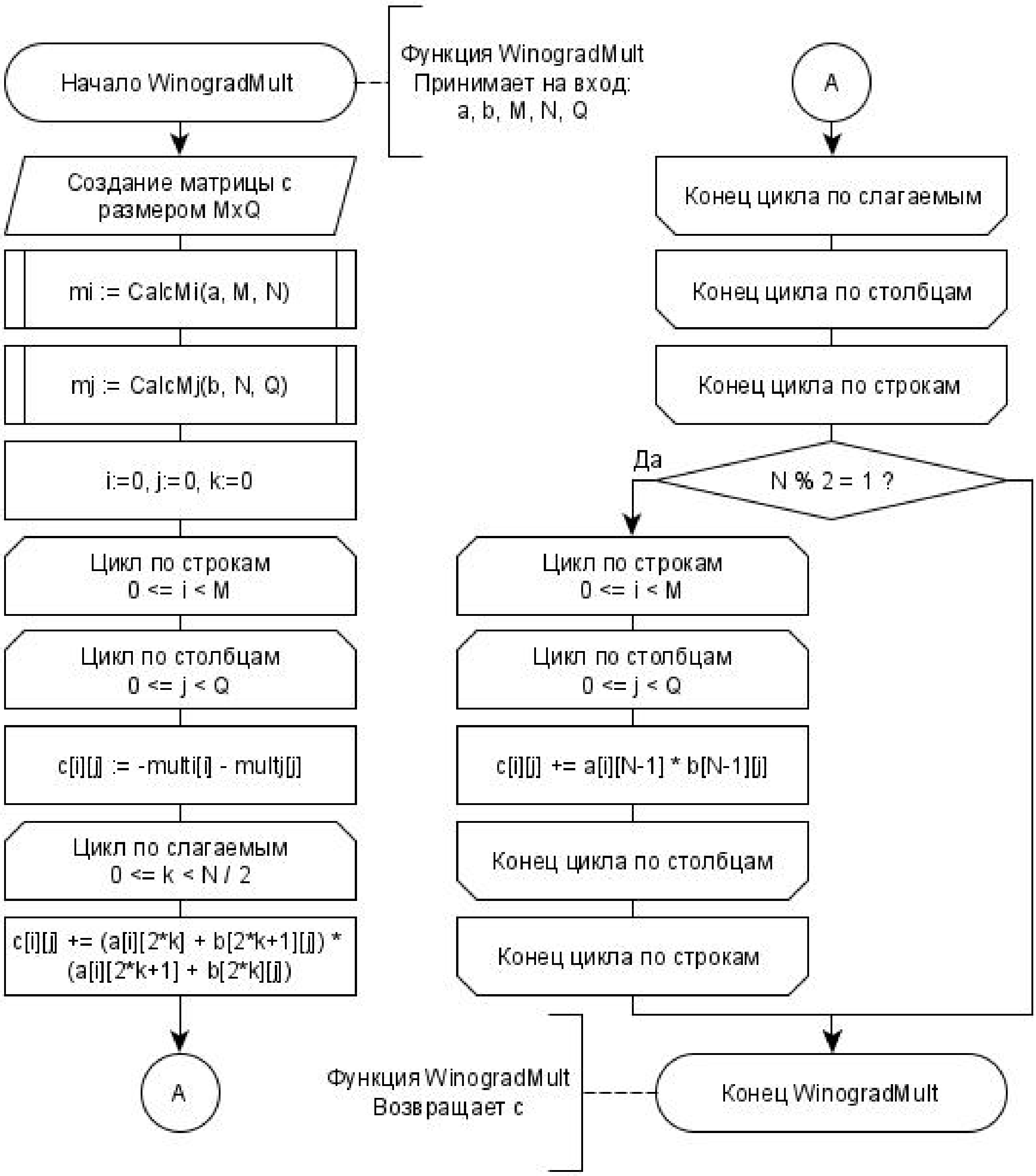

Рис. 2.2: Алгоритм Винограда

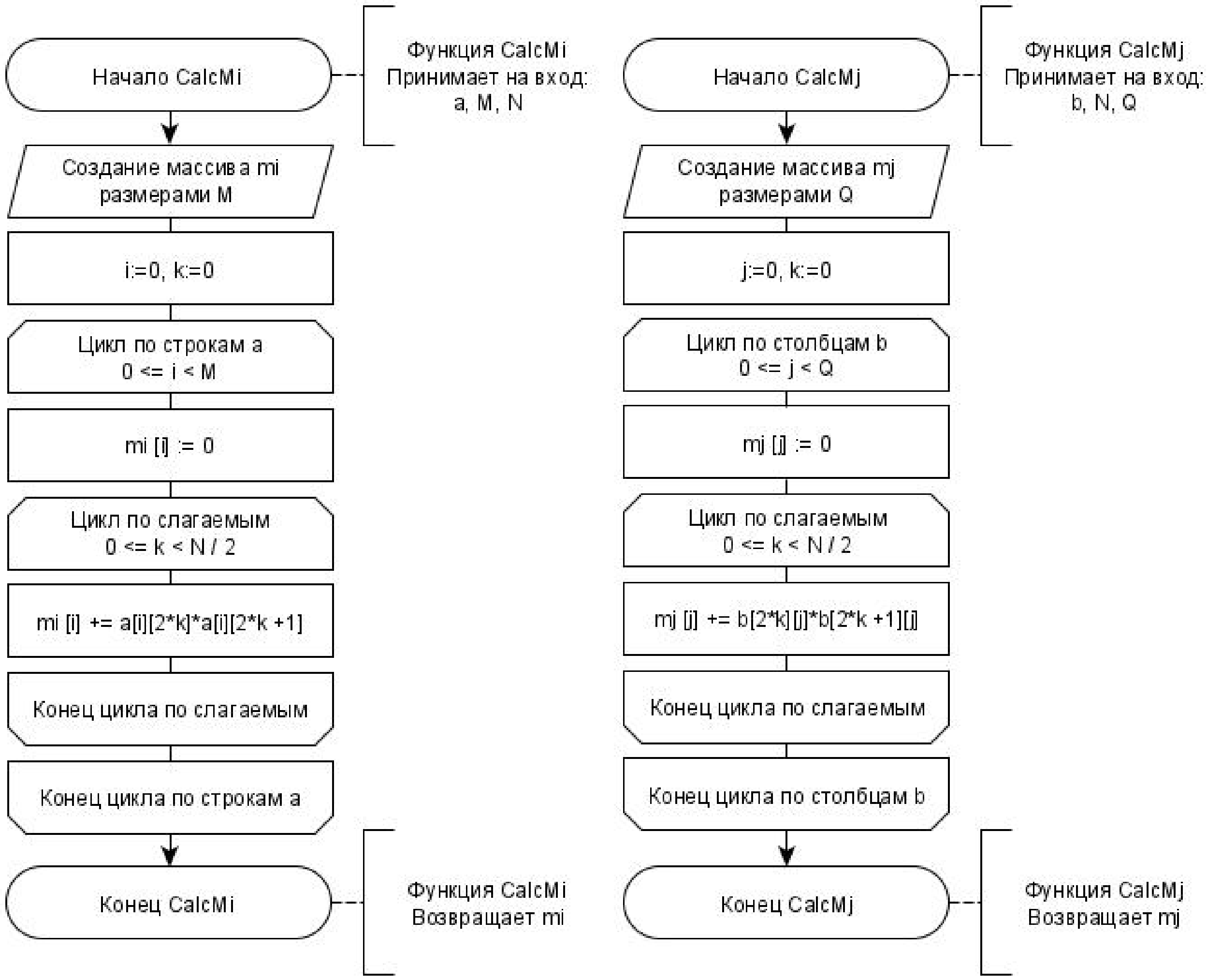

Рис. 2.3: Алгоритм Винограда, функции вычисления вспомогательных массивов

3. Технологическая часть

3.1. Выбор языка программирования

В качестве языка программирования был выбран C++, так как имеется опыт работы с ним, и с библиотеками, позволяющими провести исследование и тестирование программы. Также в языке имеются средства для отключения оптимизации компилятора.

3.2. Листинг кода

Реализация алгоритмов умножения матриц представлена на листингах 3.1-3.2.

Листинг 3.1: Функция умножения матриц классическим алгоритмом.

```
#include "classic.h"
#pragma optimize( "", off )
mat_t classic_mult(mat_t a, mat_t b, int m, int n, int q)
{
  mat_t c = create_mat(m, q);

  for (int i = 0; i < m; i++)
    for (int j = 0; j < q; j++)
    {
      c[i][j] = 0;
      for (int k = 0; k < n; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
  return c;
}
#pragma optimize( "", on )
```

Листинг 3.2: Функция умножения матриц алгоритмом Винограда.

```
#include "winograd.h"

#pragma optimize( "", off )
```

```

arr_t calc_mi(mat_t a, int m, int n)
{
  arr_t mi = create_arr(m);
  for (int i = 0; i < m; i++)
  {
    mi[i] = 0;
    for (int k = 0; k < n / 2; k++)
      mi[i] += a[i][2*k] * a[i][2*k + 1];
  }
  return mi;
}
arr_t calc_mj(mat_t b, int n, int q)
{
  arr_t mj = create_arr(q);
  for (int j = 0; j < q; j++)
  {
    mj[j] = 0;
    for (int k = 0; k < n / 2; k++)
      mj[j] += b[2*k][j] * b[2*k + 1][j];
  }
  return mj;
}
mat_t winograd_mult(mat_t a, mat_t b, int m, int n, int q)
{
  mat_t c = create_mat(m, q);
  arr_t mi = calc_mi(a, m, n);
  arr_t mj = calc_mj(b, n, q);
  for (int i = 0; i < m; i++)
    for (int j = 0; j < q; j++)
    {
      c[i][j] = -(mi[i] + mj[j]);
      for (int k = 0; k < n / 2; k++)
        c[i][j] += (a[i][2*k] + b[2*k + 1][j]) *
```

```
                (a[i][2*k + 1] + b[2*k][j]);
    }
  if (n % 2)
    for (int i = 0; i < m; i++)
      for (int j = 0; j < q; j++)
        c[i][j] += a[i][n-1] * b[n-1][j];
  return c;
}

#pragma optimize( "", on )
```

Листинг 3.3: Оптимизированая функция умножения матриц алгоритмом Винограда.

```
#include "winograd.h"

#pragma optimize( "", off )

arr_t calc_mj(mat_t b, int n, int q)
{
  arr_t mj = create_arr(q);
  for (int j = 0; j < q; j++)
  {
    double mjj = 0;
    for (int k = 1; k < n; k += 2)
      mjj += b[k][j] * b[k - 1][j];
    mj[j] = mjj;
  }
  return mj;
}

mat_t winograd_mult(mat_t a, mat_t b, int m, int n, int q)
{
  mat_t c = create_mat(m, q);
```

```
  arr_t mj = calc_mj(b, n, q);

  for (int i = 0; i < m; i++)
  {
    double mi_i = 0;
    for (int k = 1; k < n; k += 2)
      mi_i += a[i][k] * a[i][k - 1];

    for (int j = 0; j < q; j++)
    {
      double cij = -(mi_i + mj[j]);
      int k = 1;
      int k1 = 0;
      for (; k < n; k += 2, k1 += 2)
        cij += (a[i][k] + b[k1][j]) * (a[i][k1] + b[k][j]);
      c[i][j] = cij;
    }
  }

  if (n % 2)
  {
    int n_minus1 = n - 1;
    for (int i = 0; i < m; i++)
      for (int j = 0; j < q; j++)
        c[i][j] += a[i][n_minus1] * b[n_minus1][j];
  }

  free_arr(&mj);
  return c;
}

#pragma optimize( "", on )
```

3.3. Использованые оптимизации

Для уменьшения трудоёмкости алгоритма Винограда над его исходной версией были проделаны следующие оптимизации:

- При вычислении ячеек C, промежеточный результат записывается в временную переменную, которая после получения результата переносится в матрцу C. Таким образом снижено количество операций высчитывания адреса.
- Цикл по слагаемым (с переменной k) изменён на аналогичный с шагом 2. Таким образом, внутри цикла не требуется производить умножение k на 2.
- Также введена переменная k1, равная k-1. Она, как и k, увеличивается на каждом проходе цикла на 2, и таким образом вместо 2 операций (k-1) за цикл остаётся одна.
- Вычисление массива multi заменено на высчитывание значения mi = multi[i] внутри общего цикла. Таким образом удалось избавиться от накладных расходов для цикла и от выделения и освобождения памяти под массив.

3.4. Результаты тестирования

Для тестирования написанных функций был создан отдельный файл с вышеописаными классами тестов. Тестирование функций проводилось за счёт сравнения результов двух функций.

Состав тестов приведён в листинге 3.4.

Листинг 3.4: Модульные тесты

```
#include "tests.h"
// Сравнение результата умножения разными способами
```

```
bool _cmp_funcs(mat_t a, mat_t b, int m, int n, int q)
{
    mat_t c1 = classic_mult(a, b, m, n, q);
    mat_t c2 = winograd_mult(a, b, m, n, q);
    bool flag = cmp_matrix(c1, c2, m, q);
    free_mat(&c1, m, q);
    free_mat(&c2, m, q);
    return flag;
}

// Матрицы с размером 1x1
void _size_one_test()
{
    mat_t a = create_mat(1, 1);
    mat_t b = create_mat(1, 1);

    a[0][0] = 0;
    b[0][0] = 1;
    if (!_cmp_funcs(a, b, 1, 1, 1))
    {
        std::cout << __FUNCTION__ << " - FAILED\n";
        return;
    }

    a[0][0] = 3;
    b[0][0] = 4;
    if (!_cmp_funcs(a, b, 1, 1, 1))
    {
        std::cout << __FUNCTION__ << " - FAILED\n";
        return;
    }

    free_mat(&a, 1, 1);
    free_mat(&b, 1, 1);
```

```
    std::cout << __FUNCTION__ << " - OK\n";
}
// Нулевые матрицы
void _void_test()
{
    mat_t a = random_matrix(3, 2);
    mat_t b = void_matrix(2, 1);
    if (!_cmp_funcs(a, b, 3, 2, 1))
    {
        std::cout << __FUNCTION__ << " - FAILED\n";
        return;
    }
    free_mat(&a, 3, 2);
    a = void_matrix(3, 2);
    if (!_cmp_funcs(a, b, 3, 2, 1))
    {
        std::cout << __FUNCTION__ << " - FAILED\n";
        return;
    }
    free_mat(&a, 3, 2);
    free_mat(&b, 2, 1);
    std::cout << __FUNCTION__ << " - OK\n";
}
// Квадратные матрицы
void _square_test()
{
    mat_t a = random_matrix(4, 4);
    mat_t b = random_matrix(4, 4);

    if (!_cmp_funcs(a, b, 4, 4, 4))
    {
        std::cout << __FUNCTION__ << " - FAILED\n";
        return;
```

```
    }

    free_mat(&a, 4, 4);
    free_mat(&b, 4, 4);
    std::cout << __FUNCTION__ << " - OK\n";
}
// Матрицы нечётного размера
void _odd_test()
{
    mat_t a = random_matrix(5, 3);
    mat_t b = random_matrix(3, 7);

    if (!_cmp_funcs(a, b, 5, 3, 7))
    {
        std::cout << __FUNCTION__ << " - FAILED\n";
        return;
    }

    free_mat(&a, 5, 3);
    free_mat(&b, 3, 7);
    std::cout << __FUNCTION__ << " - OK\n";
}

void run_tests()
{
    _size_one_test();
    _void_test();
    _square_test();
    _odd_test();
}
```

Все тесты пройдены успешно.

3.5. Оценка трудоёмкости

Произведём оценку трудоёмкости алгоритов. Будем считать, что умножаются матрицы $A[M * N]$ и $B[N * Q]$

Классический алгоритм умножения.

$f_{cls} = 2 + M \cdot (4 + Q \cdot (4 + 3 + N \cdot (2 + 2 + 1 + 2 + 1 + 2)))$

$f_{cls} = 2 + 4M + 7QM + 10MNQ$

Оптимизированный алгоритм умножения Винограда.

$$f_{win} = (2+Q\cdot(2+1+2+(N/2)\cdot(2+2+1+2+1+3)+1))+2+M\cdot(2+1+2+(N/2)\cdot(2+1+2+1+3)+2+Q\cdot(2+4+3+(N/2)\cdot(3+1+5+1+5))) + 1 + \frac{num}{den}\begin{cases} 0, & \text{л.с.} \\ 2 + 2 + M \cdot (2 + 2 + Q \cdot (2 + 3 + 2 + 1 + 2)), & \text{х.с.} \end{cases}$$

$$f_{win} = 5 + 6Q + 5.5NQ + 7M + 4.5MN + 9MQ + 7.5MNQ + \begin{cases} 0, & \text{л.с.} \\ 4 + 4M + 10MQ, & \text{х.с.} \end{cases}$$

3.6. Оценка времени

Для замера процессорного времени исполнения функции используется функция QueryPerformanceCounter библиотеки windows.h[2]. Проведение измерений производится в функции, приведённой в листинге 3.5.

Листинг 3.5: Функция замера процессорного времени работы функции

```
void test_time(mat_t(*f)(mat_t, mat_t, int, int, int), int n)
{
  cout << "\Размерn матрицы: " << n << endl;
```

```
  mat_t a = random_matrix(n, n);
  mat_t b = random_matrix(n, n);
  mat_t c;
  int count = 0;
  start_counter();
  while (get_counter() < 3.0 * 1000) {
    c = f(a, b, n, n, n);
    free_mat(&c, n, n);
    count++;
  }
  double t = get_counter() / 1000;
  cout << "Выполнено " << count << " операций за " << t << "
   секунд" << endl;
  cout << "Время: " << t / count << endl;
  free_mat(&a, n, n);
  free_mat(&b, n, n);
}
```

4. Исследовательская часть

4.1. План экспериментов

Измерения процессорного времени проводятся на квадратных матрица. Содержание матриц сгенерировано случайным образом. Ввиду разного поведения алгоритма Винограда для чётных и нечётных размерностей, время работы изучается двумя сериями экспериментов с размерностями матриц:

1. 50, 100, 200, 400, 800;

2. 51, 101, 201, 401, 801.

Для повышения точности, каждый замер производится пять раз, за результат берётся среднее арифметическое.

4.2. Результат экспериментов

По результатам измерений процессорного времени можно составить таблицу 4.1 и таблицу 4.2

Таблица 4.1: Чётная размерность матриц. Результат измерений процессорного времени (в секундах)

	50	100	200	400	800
Классический	$5.1 \cdot 10^{-4}$	$4.2 \cdot 10^{-3}$	0.037	0.32	3.54
Виноград	$3.2 \cdot 10^{-4}$	$2.7 \cdot 10^{-3}$	0.023	0.20	2.31

Таблица 4.2: Нечётная размерность матриц. Результат измерений процессорного времени (в секундах)

	51	101	201	401	801
Классический	$5.0 \cdot 10^{-4}$	$4.1 \cdot 10^{-3}$	0.034	0.35	3.48
Виноград	$3.3 \cdot 10^{-4}$	$2.5 \cdot 10^{-3}$	0.023	0.21	2.27

Эксперименты проводились на компьютере с характеристиками:

- ОС - Windows 10, 64 бит;
- Процессор - Intel Core i7 8550U (1800 МГц);
- Объем ОЗУ: 8 ГБ.

4.3. Сравнительный анализ

По результатам экспериментов можно заключить следующее.

- Алгоритм Винограда затрачивает меньше времени, чем классический алгоритм умножения на всех исследованных размерах матриц.
- Существенных различий в процессорном времени при умножении матриц чётных и нечётных размеров у алгоритмп Винограда не выявлено.
- При увеличении размера матриц в 2 раза, наблюдается рост затраченного процессорного времени для обоих алгоритмов примерно в 8-10 раз, что соответсвует расчётам их трудоёмкости.

Заключение

В ходе лабораторной работы достигнута поставленная цель: оценка трудоёмкости алгоритма умножения матриц и получение практического навыка оптимизации алгоритмов. Решены все задачи работы.

Были изучены и описаны понятия трудоёмкости и операции умножения матриц. Также были описаны и реализованы алгоритмы умножения матриц. Был оптимизирован алгоритм Винограда. Проведены замеры процессорного времени работы каждого алгоритма при различных размерах матриц (в том числе чётных и нечётных), оценена трудоёмкость. На основании оценок и экспериментов проведён сравнительный анализ.

Список литературы

1. Трудоёмкость программ [Электронный ресурс] Режим доступа: http://ermak.cs.nstu.ru/cprog/html/041.htm , свободный (дата обращения: 27.09.2020).

2. QueryPerformanceCounter function [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter, свободный (дата обращения: 28.09.2020).