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		Е.В. Брянская
	(Группа)	(Подпись, дата)	(И.О. Фамилия)
Преподаватель			Л.Л. Волкова
		(Подпись, дата)	(И.О. Фамилия)

Москва, 2020

Оглавление

Введение

В этой лабораторной работе будет оцениваться **трудоёмкость алгоритмов умножения матриц**.

Трудоёмкость алгоритма - это зависимость стоимости операций от линейного(ых) размера(ов) входа(ов) [1].

Модель вычислений трудоёмкости должна учитывать:

1) стоимость базовых операций. К ним относятся: =, +, -, *, /, ==, !=, $<$, $<=$, $>$, $>=$, %, +=, -=, *=, /=, [ ], $< <$, $> >$. Каждая из операций имеет стоимость равную 1.

2) оценку цикла. Она складывается из стоимости тела, инкремента и сравнения.

3) оценку условного оператора if. Положим, что стоимость перехода к одной из веток равной 0. В таком случае, общая стоимость складывается из подсчета условия и рассмотрения худшего и лучшего случаев.

Оценка характера трудоёмкости даётся по наиболее быстрорастущему слагаемому.

1. Аналитическая часть

Цель данной работы – оценить трудоёмкость алгоритмов умножения матриц и получить практический навык оптимизации алгоритмов.

Для достижения поставленной цели необходимо решить ряд следующих **задач**:

1) дать математическое описание;
2) описать алгоритмы умножения матриц;
3) дать теоретическую оценку трудоёмкости алгоритмов;
4) реализовать эти алгоритмы ;
5) провести замеры процессорного времени работы алгоритмов на материале серии экспериментов;
6) провести сравнительный анализ алгоритмов.

Умножение осуществляется над матрицами $A[M \times N]$ и $B[N \times Q]$. Число столбцов первой матрицы должно совпадать с числом строк второй, а таком случае можно осуществлять умножение. Результатом является матрица $C[M \times Q]$, в которой число строк столько же, сколько в первой, а столбцов, столько же, сколько во второй.

В основе **стандартного алгоритма** умножения матриц лежит формула (1.1).

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

Существует и другой алгоритм умножения - **алгоритм Винограда**. Обозначим строку $A_{i,*}$ как $\overrightarrow{u}$, $B_{*,j}$ как $\overrightarrow{v}$. Пусть $u = (u_1, u_2, u_3, u_4)$ и $v = (v_1, v_2, v_3, v_4)$, тогда их произведение равно согласно формуле 1.2.

$$u \cdot v = u_1 \cdot v_1 + u_2 \cdot v_2 + u_3 \cdot v_3 + u_4 \cdot v_4 \tag{1.2}$$

Выражение (1.2) можно преобразовать в формулу 1.3.

$$u \cdot v = (u_1 + v_2) \cdot (u_2 + v_1) + (u_3 + v_1) \cdot (u_4 + v_3) - u_1 \cdot u_2 - u_3 \cdot u_4 - v_1 \cdot v_2 - v_3 \cdot v_4 \tag{1.3}$$

Причём, при нечётном значении N нужно учесть ещё одно слагаемое $u_5 \cdot v_5$.

Алгоритм Винограда основывается на раздельной работе со слагаемыми из выражения (1.3).

2. Конструкторская часть

Рассмотрим и оценим работу алгоритмов на матрицах $A[M \times N]$ и $B[N \times Q]$.

2.1. Стандартный алгоритм умножения матриц

В основе этого алгоритма лежит формула (1.1). То есть для вычисления произведения двух матриц, каждая строка первой матрицы почленно умножается на каждый столбец второй, и затем подсчитывается сумма таких произведений, и полученный результат записывается в соответствующую ячейку результурующей матрицы.

Схема алгоритма представлена на Рис.2.1.

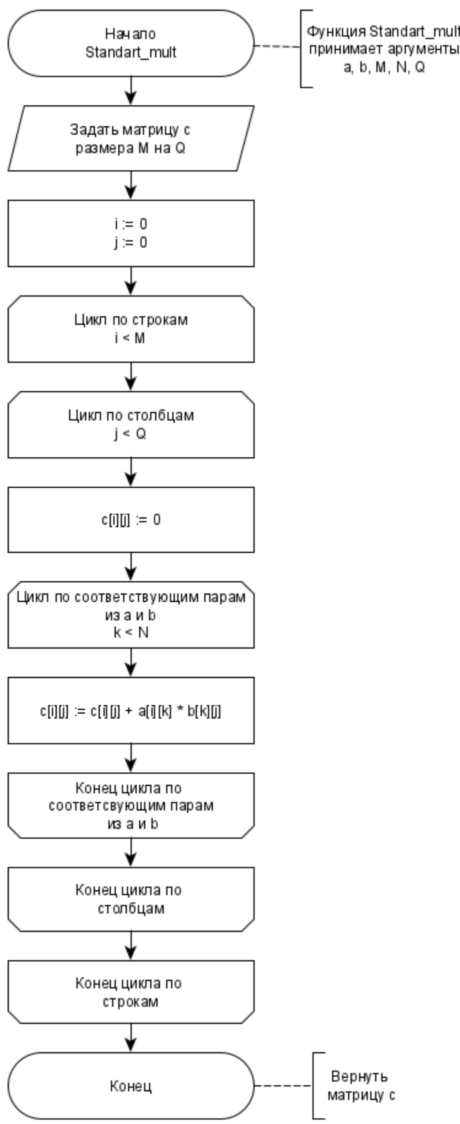

Рис. 2.1 — Стандартный алгоритм умножения матриц

2.2. Алгоритм Винограда

Цель данного алгоритма - сократить долю умножений в самом тяжёлом, затратном участке кода. Для этого используется формула (1.3).

Некоторые из слагаемых можно вычислить заранее и использовать повторно для каждой строки первой матрицы и для каждого столбца второй. Таким образом, трудоёмкость алгоритма уменьшается за счёт сокращения количества производимых операций.

В этом алгоритме важно учитывать, что при нечётном значении N, необходимо вычислять дополнительное слагаемое $u_N \cdot v_N$.

Схема алгоритма представлена на Рис.2.2.

2.3. Алгоритм Винограда (оптимизированный)

Алгоритм призван уменьшить трудоёмкость алгоритма, чтобы это сделать были использованы оптимизации:

1) видоизменён цикл по k, изменён шаг и условие. Таким образом, ушла необходимость в целочисленном делении, и в теле цикла не требуется больше умножать k на 2 каждый раз.

2) введена вспомогательная переменная buf, в которую записывается промежуточнее значение соответсвующей ячейки матрицы, и затем, конечный результат переносится в саму матрицу. Тем самым, уменьшается количество обращений к элементам матрицы, находящимся по конкретному адресу.

3) заранее высчитываются некоторые значения, например, n - 1, которые далее используюся во вложенных циклах.

4) используется дополнительная переменная t = k - 1, чтобы сократить число подсчетов этого значения на каждом шаге цикла.

5) объединён цикл 3 и 4, что позволило избежать ещё одного вложенного цикла.

Схема алгоритма представлена на Рис.2.3.

2.4. Требования к ПО

Для корректной работы алгоритмов и проведения тестов необходимо выполнить.

1) Обеспечить возможность ввода двух матриц через консоль и выбора алгоритма для умножения.
2) Вывести, в случае ввода размеров матриц, не удовлетворяющих главному условию, соответствующее сообщение. Программа не должна аварийно завершаться.
3) Рассчитать искомую матрицу и вывести её на экран.
4) Реализовать функцию замера процессорного времени, которое выбранный метод затрачивает на вычисление результата. Вывести результаты замеров на экран.

2.5. Заготовки тестов

При проверке на корректность работы реализованных функций необходимо провести следующие тесты:

- умножение матриц размером 1×1;
- квадратные матрицы;
- прямоугольные матрицы;
- чётное и нечётное значение N.

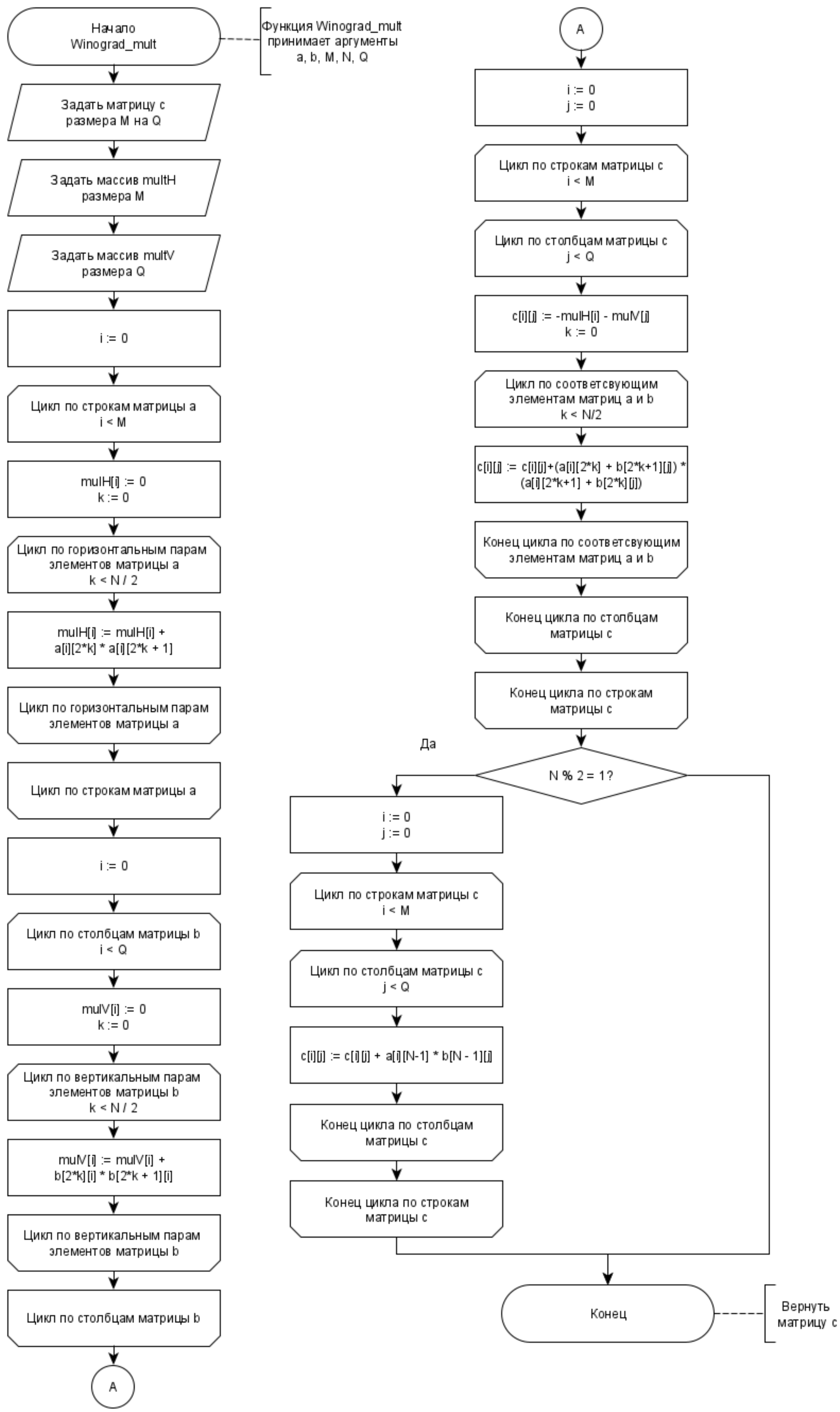

Рис. 2.2 — Алгоритм Винограда

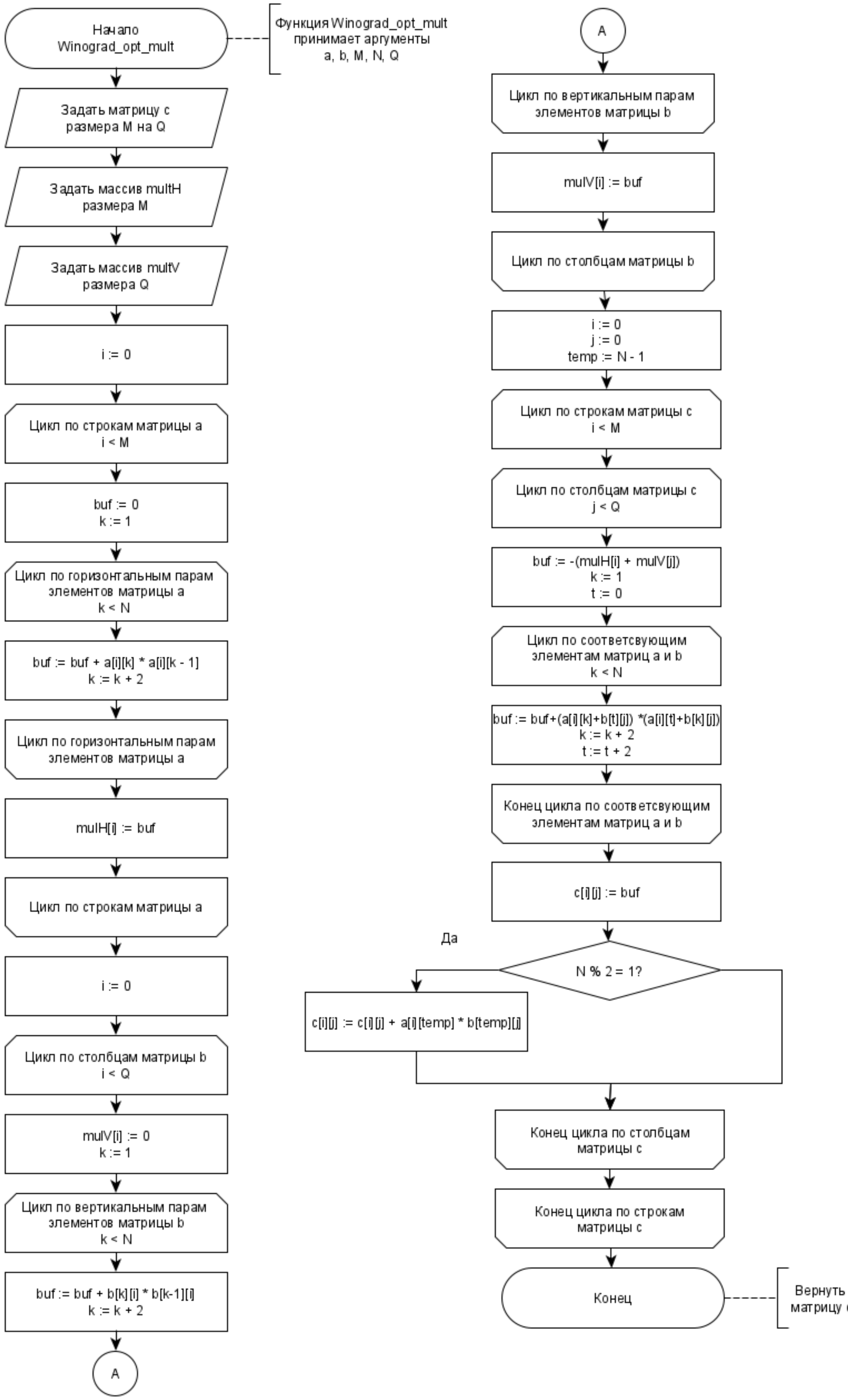

Рис. 2.3 — Оптимизированный алгоритм Винограда

3. Технологическая часть

3.1. Выбранный язык программирования

Для выполнения этой лабораторной работы был выбран язык программирования C++, так как есть большой навык работы с ним и с подключаемыми библиотеками, которые также использовались для проведения тестирования и замеров.

Использованная среда разработки - Visual Studio [3].

3.2. Листинг кода

Ниже представлены Листиги 3.1 - 3.3 функций, реализующих алгоритмы поиска расстояний.

Листинг 3.1 — Стандартный алгоритм умножения матриц

```
matrix_t standart_mult(matrix_t a, matrix_t b, int m, int n, int q)
{
  matrix_t c = create_matrix(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;
}
```

Листинг 3.2 — Алгоритм Винограда

```
matrix_t winograd_mult(matrix_t a, matrix_t b, int m, int n, int q)
{
  arr_t mulH = create_array(m);
  arr_t mulV = create_array(q);
  matrix_t c = create_matrix(m, q);

  for (int i = 0; i < m; i++)
  {
```

```
    mulH[i] = 0;
    for (int k = 0; k < n / 2; k++)
      mulH[i] = mulH[i] + a[i][2 * k] * a[i][2 * k + 1];
  }

  for (int i = 0; i < q; i++)
  {
    mulV[i] = 0;
    for (int k = 0; k < n / 2; k++)

      mulV[i] = mulV[i] + b[2 * k][i] * b[2 * k + 1][i];
  }

  for (int i = 0; i < m; i++)
    for (int j = 0; j < q; j++)
    {
      c[i][j] = -mulH[i] - mulV[j];
      for (int k = 0; k < n / 2; k++)
        c[i][j] = 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] = c[i][j] + a[i][n - 1] * b[n - 1][j];

  return c;
}
```

Листинг 3.3 — Оптимизированный алгоритм Винограда

```
matrix_t winograd_mult(matrix_t a, matrix_t b, int m, int n, int q)
{
  arr_t mulH = create_array(m);
  arr_t mulV = create_array(q);
  double buf;

  matrix_t c = create_matrix(m, q);

  for (int i = 0; i < m; i++)
```

```
  {
    buf = 0;
    for (int k = 1; k < n; k += 2)
      buf += a[i][k] * a[i][k - 1];
    mulH[i] = buf;
  }

  for (int i = 0; i < q; i++)
  {
    buf = 0;
    for (int k = 1; k < n; k += 2)
      buf += b[k][i] * b[k - 1][i];
    mulV[i] = buf;
  }

  int temp = n - 1;
  int is_odd = n % 2;

  for (int i = 0; i < m; i++)
    for (int j = 0; j < q; j++)
    {
      buf = -(mulH[i] + mulV[j]);
      for (int k = 1, t = 0; k < n; k += 2, t +=2)
        buf += (a[i][k] + b[t][j]) * (a[i][t] + b[k][j]);
      c[i][j] = buf;

      if (is_odd)
        c[i][j] += a[i][temp] * b[temp][j];
    }

  return c;
}
```

3.3. Результаты тестов

Для тестирования были написаны функции, проверяющие, согласно заготовкам выше, случаи. Выводы о корректности работы делаются на основе сравнения результатов.

Все тесты пройдены успешно. Сами тесты представлены ниже (Листинг 3.4).

Листинг 3.4 — Тесты

```
bool mult_cmp(matrix_t a, matrix_t b, int m, int n, int q)
{
  matrix_t c1 = standart_mult(a, b, m, n, q);
  matrix_t c2 = winograd_mult(a, b, m, n, q);

  bool res = cmp_matrix(c1, c2, m, q);

  free_matrix(&c1, m, q);
  free_matrix(&c2, m, q);

  return res;
}

// Матрицы размером 1 x 1
void test_size_1_1()
{
  int n = 1;

  matrix_t a = create_matrix(n, n);
  matrix_t b = create_matrix(n, n);

  a[0][0] = 15;
  b[0][0] = -7;

  if (!mult_cmp(a, b, n, n, n))
  {
    cout << endl << __FUNCTION__ << " FAILED" << endl;
    free_matrix(&a, n, n);
    free_matrix(&b, n, n);
    return;
  }

  free_matrix(&a, n, n);
  free_matrix(&b, n, n);

  cout << endl << __FUNCTION__ << " OK" << endl;
}

// Квадратные матрицы
```

```
void test_square_matr()
{
  int n[] = { 2, 6, 10 };

  for (int i = 0; i < sizeof(n) / sizeof(n[0]); i++)
  {
    matrix_t a = random_fill_matrix(n[i], n[i]);
    matrix_t b = random_fill_matrix(n[i], n[i]);

    if (!mult_cmp(a, b, n[i], n[i], n[i]))
    {
      cout << endl << __FUNCTION__ << " FAILED" << endl;
      free_matrix(&a, n[i], n[i]);
      free_matrix(&b, n[i], n[i]);
      return;
    }

    free_matrix(&a, n[i], n[i]);
    free_matrix(&b, n[i], n[i]);

    cout << endl << __FUNCTION__ << " OK" << endl;
  }
}

// Прямоугольные матрицы
void test_rectangulat_matr()
{
  int m[] = { 2, 6, 10 };
  int n[] = { 1, 4, 7 };
  int q[] = { 3, 4, 8 };

  for (int i = 0; i < sizeof(n) / sizeof(n[0]); i++)
  {
    matrix_t a = random_fill_matrix(m[i], n[i]);
    matrix_t b = random_fill_matrix(n[i], q[i]);

    if (!mult_cmp(a, b, m[i], n[i], q[i]))
    {
      cout << endl << __FUNCTION__ << " FAILED" << endl;
      free_matrix(&a, m[i], n[i]);
```

```
      free_matrix(&b, n[i], q[i]);
      return;
    }
    free_matrix(&a, m[i], n[i]);
    free_matrix(&b, n[i], q[i]);

    cout << endl << __FUNCTION__ << " OK" << endl;
  }
}

// Матрицы с чётным размером
void test_even_size()
{
  int m[] = { 2, 4 };
  int n[] = { 6, 2 };
  int q[] = { 2, 8 };

  for (int i = 0; i < sizeof(n) / sizeof(n[0]); i++)
  {
    matrix_t a = random_fill_matrix(m[i], n[i]);
    matrix_t b = random_fill_matrix(n[i], q[i]);

    if (!mult_cmp(a, b, m[i], n[i], q[i]))
    {
      cout << endl << __FUNCTION__ << " FAILED" << endl;
      free_matrix(&a, m[i], n[i]);
      free_matrix(&b, n[i], q[i]);
      return;
    }

    free_matrix(&a, m[i], n[i]);
    free_matrix(&b, n[i], q[i]);

    cout << endl << __FUNCTION__ << " OK" << endl;
  }
}

// Матрицы с нечётным размером
void test_odd_size()
{
```

```
  int m[] = { 3, 3 };
  int n[] = { 3, 1 };
  int q[] = { 5, 7 };

  for (int i = 0; i < sizeof(n) / sizeof(n[0]); i++)
  {
    matrix_t a = random_fill_matrix(m[i], n[i]);
    matrix_t b = random_fill_matrix(n[i], q[i]);

    if (!mult_cmp(a, b, m[i], n[i], q[i]))
    {
      cout << endl << __FUNCTION__ << " FAILED" << endl;
      free_matrix(&a, m[i], n[i]);
      free_matrix(&b, n[i], q[i]);
      return;
    }

    free_matrix(&a, m[i], n[i]);
    free_matrix(&b, n[i], q[i]);

    cout << endl << __FUNCTION__ << " OK" << endl;
  }
}

void run_tests()
{
  test_size_1_1();
  test_square_matr();
  test_rectangulat_matr();
  test_even_size();
  test_odd_size();
}
```

3.4. Оценка трудоёмкости

Произведём оценку трудоёмкости приведённых алгоритмов. Рассмотрим умножение матриц $A[M \times N]$ и $B[N \times Q]$.

Стандартный алгоритм

$f_{st} = 2 + M(2 + 2 + Q(3 + 2 + 2 + N(2 + 1 + 2 + 2 + 1 + 2)))$
$f_{st} = 2 + 4M + 7MQ + 10MNQ$

Алгоритм Винограда (неоптимизированный)

$$f_w = 2 + M(2 + 3 + 2 + \frac{N}{2}(12 + 3)) + 2 + Q(2 + 3 + 2 + \frac{N}{2}(12 + 3)) + 2 + M(2 + 2 + Q(7 + 2 + 3 + \frac{N}{2}(23 + 3))) + 1 + \left[\begin{array}{ll} 0, & \text{л.с.} \\ 2 + M(2 + 2 + Q(13 + 2)), & \text{х.с.} \end{array}\right.$$

$$f_w = 7 + 11M + 7Q + \frac{15}{2}MN + \frac{15}{2}NQ + 12MQ + 13MNQ + \left[\begin{array}{ll} 0, & \text{л.с.} \\ 2 + 4M + 15MQ, & \text{х.с.} \end{array}\right.$$

Алгоритм Винограда (оптимизированный)

$$f_{wop} = 2 + M(2 + 1 + 2 + \frac{N}{2}(7 + 2) + 2) + 2 + Q(2 + 1 + 2 + \frac{N}{2}(7 + 2) + 2) + 2 + 2 + M(2 + 2 + Q(2 + 4 + 3 + \frac{N}{2}(3 + 12) + 3 + 1 + \left[\begin{array}{ll} 0, & \text{л.с.} \\ 8, & \text{х.с.} \end{array}\right.))$$

$$f_{wop} = 8 + 11M + 7Q + 4.5MN + 4.5NQ + 13MQ + 7.5MNQ + \left[\begin{array}{ll} 0, & \text{л.с.} \\ 8MQ, & \text{х.с.} \end{array}\right.$$

3.5. Оценка времени

Процессорное время измеряется с помощью функции QueryPerformanceCounter библиотеки windows.h [2]. Осуществление замеров показано ниже (Листинг 3.5).

Листинг 3.5 — Замеры процессорного времени

```
void test_time(matrix_t(*f)(matrix_t, matrix_t, int, int, int), int n)
{
  matrix_t a = random_fill_matrix(n, n);
  matrix_t b = random_fill_matrix(n, n);
  matrix_t c;

  int num = 0;
  start_measuring();

  while (get_measured() < 3 * 1000)
  {
    c = f(a, b, n, n, n);
    free_matrix(&c, n, n);
    num++;
  }

```

```
  double t = get_measured() / 1000;
  cout << "Выполнено " << num << " операций за " << t << " секунд" << endl;
  cout << "Время: " << t / num << endl;

  free_matrix(&a, n, n);
  free_matrix(&b, n, n);
}

void test_range(vector<int> &n)
{
  for (int key : n)
  {
    cout << endl << endl << "Размер тестируемых матриц: " << key << "x" << key << endl;

    cout << endl << "———Standart———" << endl;
    test_time(standart_mult, key);
    cout << endl << "———Winograd———" << endl;
    test_time(winograd_mult, key);
    cout << endl << "———Winograd(improved)———" << endl;
    test_time(winograd_opt_mult, key);
  }
}
```

4. Исследовательская часть

Характеристики ПО

При проведении замеров времени использовался компьютер, имеющий следующие характеристики:

- ОС - Windows 10 Pro
- Процессор - Inter Core i7 10510U (1800 МГц)
- Объём ОЗУ - 16 Гб

Измерения

Для проведения замеров процессорного времени использовались квадратные матрицы. Их содержимое генерируется случайным образом. Было проведено две серии экспериментов, ориентированных на выявление чувствительности алгоритмов к чётным и нечётным значениям N.

1) $50, 100, 200, 300, 400, 500, 600, 700$
2) $51, 101, 201, 301, 401, 501, 601, 701$

Каждый замер проводится 5 раз для получения более точного среднего результата.

В таблице 4.1 и таблице 4.2 представлены результаты замеров процессорного времени работы реализаций алгоритмов (в сек).

Таблица 4.1 — Результаты измерений (чётная размерность)

Размер n / Алгоритм	50	100	200	300	400	500	600	700
Стандартный	$5.1 * 10^{-4}$	$4.1 * 10^{-3}$	0.037	0.133	0.322	0.777	1.08	1.445
Виноград	$3.5 * 10^{-4}$	$2.9 * 10^{-3}$	0.027	0.096	0.237	0.559	0.727	1.226
Виноград (оптимизированный)	$3.4 * 10^{-4}$	$2.5 * 10^{-3}$	0.024	0.084	0.207	0.474	0.601	1.101

Таблица 4.2 — Результаты измерений (нечётная размерность)

Размер n / Алгоритм	51	101	201	301	401	501	601	701
Стандартный	$5.4 * 10^{-4}$	$4.2 * 10^{-3}$	0.037	0.133	0.329	0.838	0.982	1.628
Виноград	$4.2 * 10^{-4}$	0.003	0.028	0.098	0.240	0.6	0.852	1.465
Виноград (оптимизированный)	$3.5 * 10^{-4}$	$2.5 * 10^{-3}$	0.025	0.082	0.206	0.512	0.705	1.052

Согласно полученным данным можно сделать следующие **выводы**.

1) Оптимизированный алгоритм Винограда осуществляет умножение матриц быстрее, чем два других сравниваемых алгоритма.

2) Оптимизированный алгоритм Винограда показывает результаты, примерно на 10%, чем неоптимизированный, что подтверждает разницу в рассчётах трудоёмкости обоих алгоритмов.

3) Результаты на матрицах с чётным и нечётным размером N отличаются на предложенном множестве значений, но не существенно.

4) Наихудшие результаты измерений показал стандартный алгоритм умножения.

Заключение

В ходе лабораторной работы была достигнута поставленная цель, а именно, оценена трудоёмкость алгоритмов умножения матриц и рассмотрены возможные оптимизации алгоритма Винограда.

В процессе выполнения были решены все задачи. Описаны все рассматриваемые алгоритмы, дана теоретическая оценка трудоёмкости каждого. Все проработанные алгоритмы реализованы, кроме того, были проведены замеры процессорного времени работы на материале серии экспериментов и проведён сравнительный анализ, сделаны выводы:

1) оптимизированный алгоритм Винограда, как и ожидалось, выполняет умножение матриц быстрее, чем два других сравниваемых алгоритма;

2) оптимизированный алгоритм Винограда осуществляет умножение мтариц быстрее, чем неоптимизированный, что подтверждает разницу в рассчётах трудоёмкости обоих алгоритмов;

3) результаты отличаются несущественно на матрицах с чётным и нечётным размером N;

4) наихудшие результаты измерений времени показал стандартный алгоритм умножения.

Список литературы

1. Трудоёмкость алгоритмов и временные оценки [Электронный ресурс]. Режим доступа: http://techn.sstu.ru/kafedri/подразделения/1/MetMat/shaturn/theoralg/5.htm, свободный (дата обращения: 29.09.20).

2. QueryPerformanceCounter function [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter, свободный (дата обращения: 01.10.2020).

3. Документация по Visual Studio 2019 [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/visualstudio/windows/?view=vs-2019, свободный (дата обращения: 01.10.2020)