**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 №** 3

**Название:** Алгоритмы сортировки

**Дисциплина:** Анализ алгоритмов

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

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

Москва, 2020

# Оглавление

## Введение

Сортировка - это процесс упорядочения некоторого множества элементов, на котором определены отношения порядка[1].

Задача сортировки множества данных является одной из самых часто встречающихся при разработке программ. Также существует и достаточное количество алгоритмов, решающих данную задачу. Однако, "лучшего"алгоритма подходящего для решения любых задач сортировки наиболее оптимальным образом не существует. Поэтому задача изучения алгоритмов упорядочения до сих пор остаётся актуальной.

В данной лабораторной изучается и оценивается три алгоритма сортировки:

- сортировка пузырьком;
- сортировка расчёской;
- сортировка слиянием.

# 1. Аналитическая часть

Целью лабораторной работы является оценка трудоёмкости алгоритмов сортировки.

Выделены следующие задачи лабораторной работы:

- описание операции сортировки;
- описание и реализация алгоритмов сортировки;
- проведение замеров процессорного времени работы алгоритмов при различных размерах массивов;
- оценка трудоёмкости алгоритов;
- проведение сравнительного анализа алгоритмов на основании экспериментов.

Сортировка массива по неубыванию - операция над массивом $arr[N]$, в результате которой в нём начинает выполняется условие:

$$arr[i+1] >= arr[i], i \in [0, N-2] \tag{1.1}$$

Аналогично формулируется определение для сортировки по невозрастанию. В случае, если в массиве нет равных элементов, также возможно применить операции сортироки по возрастанию и убыванию.

# 2. Конструкторская часть

Рассмотрим вышеупомянутые алгоритмы сортировки. Для удобства изложения сути алгоритмов, будем рассматривать сортировку по неубыванию. Алгоритмы сортировки других порядков могут быть получены заменой условия сравнения.

## 2.1. Сортировка пузырьком

Алгоритм сортировки пузырьком основывается на следующем действии. Массив проссматривается от 0 до N-2 элемента, и в случае, если текущий элемент массива больше следующего, они меняются местами. Таким образом, после первого прохода в конце массива окажется максимальный элемент, после второго - два максимальных, и так далее до полного упорядочивания массива.

Схема алгоритма приведена на рисунке 2.1.

## 2.2. Сортировка расчёской

Алгоритм является модификацией вышеописаного алгоритма. Основной идеей является то, что "почти упорядоченый"массив можно отсоритровать быстрее, чем неупорядоченый вовсе. Поэтому, сначала сортируются множества элементов массива расположеных друг от друга на расстоянии $Step$. Следующим шагом сортировка происходит по $\frac{Step}{2}$ и т.д. до шага в 1. Сортировка подмассивов происходит по алгоритму пузырька.

Первоначально $Step = N/\alpha$, где $\alpha$ - фактор уменьшения. Наиболее оптимальное значение этого фактора равно $\frac{1}{1 - e^{-\phi}} \approx 1.247$, где $\phi$ - золотое сечение. Следующий шаг считается как $Step/\alpha$

Схема алгоритма приведена на рисунке 2.2.

## 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;

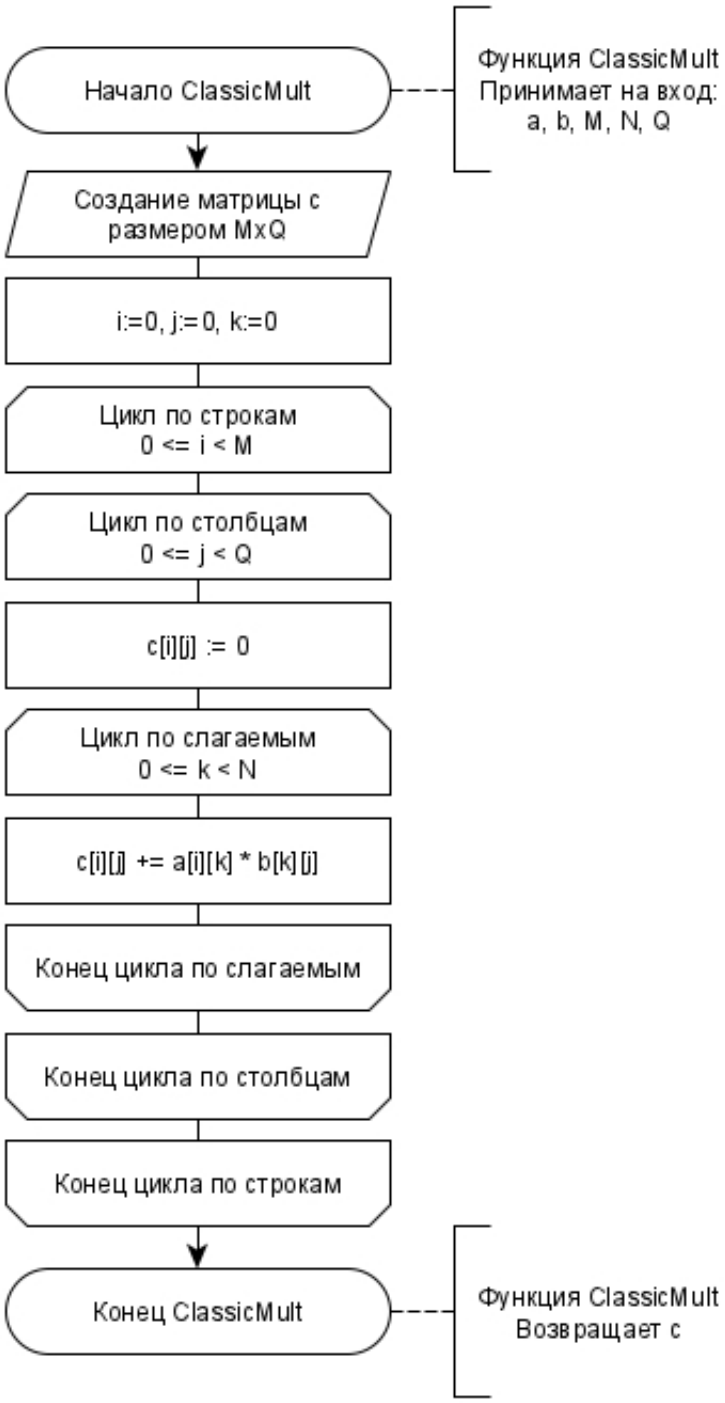


Рис. 2.1: Сортировка пузырьком

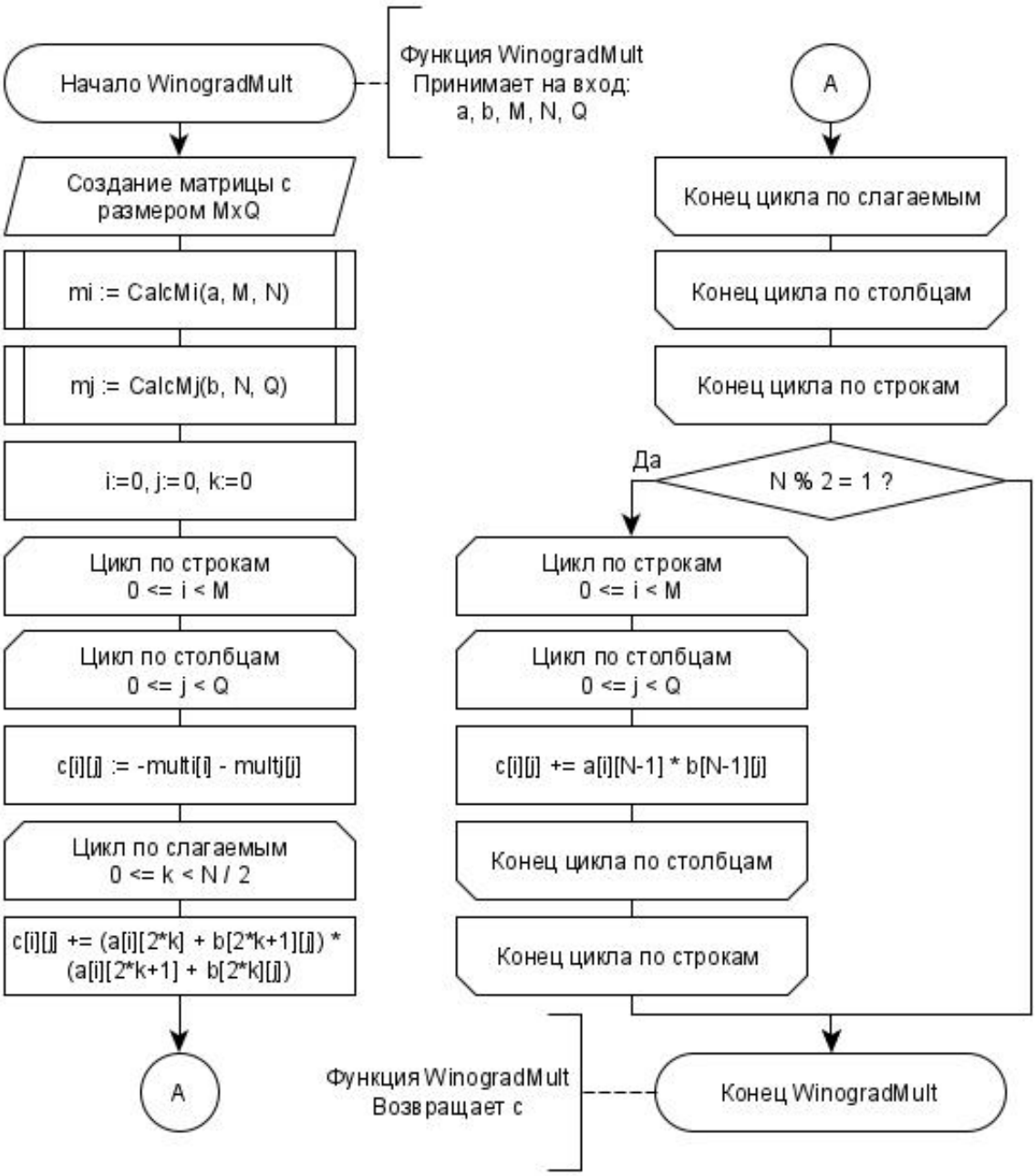


Рис. 2.2: Алгоритм Винограда

# 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 )
```

```
4
5  arr_t calc_mi(mat_t a, int m, int n)
6  {
7    arr_t mi = create_arr(m);
8    for (int i = 0; i < m; i++)
9    {
10     mi[i] = 0;
11     for (int k = 0; k < n / 2; k++)
12       mi[i] += a[i][2*k] * a[i][2*k + 1];
13   }
14   return mi;
15 }
16 arr_t calc_mj(mat_t b, int n, int q)
17 {
18   arr_t mj = create_arr(q);
19   for (int j = 0; j < q; j++)
20   {
21     mj[j] = 0;
22     for (int k = 0; k < n / 2; k++)
23       mj[j] += b[2*k][j] * b[2*k + 1][j];
24   }
25   return mj;
26 }
27 mat_t winograd_mult(mat_t a, mat_t b, int m, int n, int q)
28 {
29   mat_t c = create_mat(m, q);
30   arr_t mi = calc_mi(a, m, n);
31   arr_t mj = calc_mj(b, n, q);
32   for (int i = 0; i < m; i++)
33     for (int j = 0; j < q; j++)
34     {
35       c[i][j] = -(mi[i] + mj[j]);
36       for (int k = 0; k < n / 2; k++)
37         c[i][j] +=  (a[i][2*k] + b[2*k + 1][j]) *
```

```
38                (a[i][2*k + 1] + b[2*k][j]);
39      }
40    if (n % 2)
41      for (int i = 0; i < m; i++)
42        for (int j = 0; j < q; j++)
43          c[i][j] += a[i][n-1] * b[n-1][j];
44    return c;
45  }
46
47  #pragma optimize( "", on )
```

Листинг 3.3: Оптимизированая функция умножения матриц алгоритмом Винограда.

```
1   #include "winograd.h"
2
3   #pragma optimize( "", off )
4
5   arr_t calc_mj(mat_t b, int n, int q)
6   {
7     arr_t mj = create_arr(q);
8     for (int j = 0; j < q; j++)
9     {
10      double mjj = 0;
11      for (int k = 1; k < n; k += 2)
12        mjj += b[k][j] * b[k - 1][j];
13      mj[j] = mjj;
14    }
15    return mj;
16  }
17
18  mat_t winograd_mult(mat_t a, mat_t b, int m, int n, int q)
19  {
20    mat_t c = create_mat(m, q);
```

```
21    arr_t mj = calc_mj(b, n, q);
22
23    for (int i = 0; i < m; i++)
24    {
25      double mi_i = 0;
26      for (int k = 1; k < n; k += 2)
27        mi_i += a[i][k] * a[i][k - 1];
28
29      for (int j = 0; j < q; j++)
30      {
31        double cij = -(mi_i + mj[j]);
32        int k = 1;
33        int k1 = 0;
34        for (; k < n; k += 2, k1 += 2)
35          cij +=  (a[i][k] + b[k1][j]) * (a[i][k1] + b[k][j]);
36        c[i][j] = cij;
37      }
38    }
39
40    if (n % 2)
41    {
42      int n_minus1 = n - 1;
43      for (int i = 0; i < m; i++)
44        for (int j = 0; j < q; j++)
45          c[i][j] += a[i][n_minus1] * b[n_minus1][j];
46    }
47
48    free_arr(&mj);
49    return c;
50  }
51
52  #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

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

```
3  bool _cmp_funcs(mat_t a, mat_t b, int m, int n, int q)
4  {
5    mat_t c1 = classic_mult(a, b, m, n, q);
6    mat_t c2 = winograd_mult(a, b, m, n, q);
7    bool flag = cmp_matrix(c1, c2, m, q);
8    free_mat(&c1, m, q);
9    free_mat(&c2, m, q);
10   return flag;
11 }
12
13 // Матрицы с размером 1x1
14 void _size_one_test()
15 {
16   mat_t a = create_mat(1, 1);
17   mat_t b = create_mat(1, 1);
18
19   a[0][0] = 0;
20   b[0][0] = 1;
21   if (!_cmp_funcs(a, b, 1, 1, 1))
22   {
23     std::cout << __FUNCTION__ << " - FAILED\n";
24     return;
25   }
26
27   a[0][0] = 3;
28   b[0][0] = 4;
29   if (!_cmp_funcs(a, b, 1, 1, 1))
30   {
31     std::cout << __FUNCTION__ << " - FAILED\n";
32     return;
33   }
34
35   free_mat(&a, 1, 1);
36   free_mat(&b, 1, 1);
```

```
37
38    std::cout << __FUNCTION__ << " - OK\n";
39 }
40 // Нулевые матрицы
41 void _void_test()
42 {
43    mat_t a = random_matrix(3, 2);
44    mat_t b = void_matrix(2, 1);
45    if (!_cmp_funcs(a, b, 3, 2, 1))
46    {
47       std::cout << __FUNCTION__ << " - FAILED\n";
48       return;
49    }
50    free_mat(&a, 3, 2);
51    a = void_matrix(3, 2);
52    if (!_cmp_funcs(a, b, 3, 2, 1))
53    {
54       std::cout << __FUNCTION__ << " - FAILED\n";
55       return;
56    }
57    free_mat(&a, 3, 2);
58    free_mat(&b, 2, 1);
59    std::cout << __FUNCTION__ << " - OK\n";
60 }
61 // Квадратные матрицы
62 void _square_test()
63 {
64    mat_t a = random_matrix(4, 4);
65    mat_t b = random_matrix(4, 4);
66
67    if (!_cmp_funcs(a, b, 4, 4, 4))
68    {
69       std::cout << __FUNCTION__ << " - FAILED\n";
70       return;
```

```
71    }
72
73    free_mat(&a, 4, 4);
74    free_mat(&b, 4, 4);
75    std::cout << __FUNCTION__ << " - OK\n";
76  }
77  // Матрицы нечётного размера
78  void _odd_test()
79  {
80    mat_t a = random_matrix(5, 3);
81    mat_t b = random_matrix(3, 7);
82
83    if (!_cmp_funcs(a, b, 5, 3, 7))
84    {
85      std::cout << __FUNCTION__ << " - FAILED\n";
86      return;
87    }
88
89    free_mat(&a, 5, 3);
90    free_mat(&b, 3, 7);
91    std::cout << __FUNCTION__ << " - OK\n";
92  }
93
94  void run_tests()
95  {
96    _size_one_test();
97    _void_test();
98    _square_test();
99    _odd_test();
100 }
```

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.**

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

$f_{cls} = 2 + 2M + 5QM + 10MNQ$

**Алгоритм умножения Винограда.**

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

$f_{win} = 5 + 4Q + 4.5NQ + 7M + 4MN + 8MQ + 7.5MNQ + \begin{cases} 0, & \text{л.с.} \\ 4 + 2M + 10MQ, & \text{х.с.} \end{cases}$

## 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

```
1 void test_time(mat_t(*f)(mat_t, mat_t, int, int, int), int n)
2 {
3   cout << "\Размерn матрицы: " << n << endl;
```

```
4   mat_t a = random_matrix(n, n);
5   mat_t b = random_matrix(n, n);
6   mat_t c;
7   int count = 0;
8   start_counter();
9   while (get_counter() < 3.0 * 1000) {
10    c = f(a, b, n, n, n);
11    free_mat(&c, n, n);
12    count++;
13  }
14  double t = get_counter() / 1000;
15  cout << "Выполнено " << count << " операций за " << t << "
     секунд" << endl;
16  cout << "Время: " << t / count << endl;
17  free_mat(&a, n, n);
18  free_mat(&b, n, n);
19 }
```

# Исследовательская часть

## 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.

## 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.

## 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. Методы сортировки: метод. указания к лаб. работе по дисциплине «Информационные технологии» для студентов направления подготовки бакалавра 210400 «Радиотехника» дневной формы обучения / НГТУ; Сост.: Е.Н.Приблудова, С.Б.Сидоров. Н.Новгород, 2012, 11 с. (дата обращения: 30.09.2020).

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).