Государственное образовательное учреждение
высшего профессионального образования
«Московский Государственный Технический Университет
имени Н.Э. Баумана»

Отчет
По лабораторной работе №2
По курсу «Анализ Алгоритмов»
На тему «Исследование сложности алгоритмов умножения матриц»

Кизилов Дмитрий, ИУ7-54

Москва, 2017

Оглавление

Постановка задачи

Реализовать алгоритмы умножения матриц:

1. Стандартный
2. Винограда
3. Улучшенного Винограда

Рассчитать сложность алгоритмов и провести временные эксперименты

Модель вычислений

Операнды +, -, *, /, <, >, <=, >=, ==, !=, [ ] имеют значение f = 1;
Значение для цикла f = 2 + N(...), где N - количество итераций.

Блок-схемы

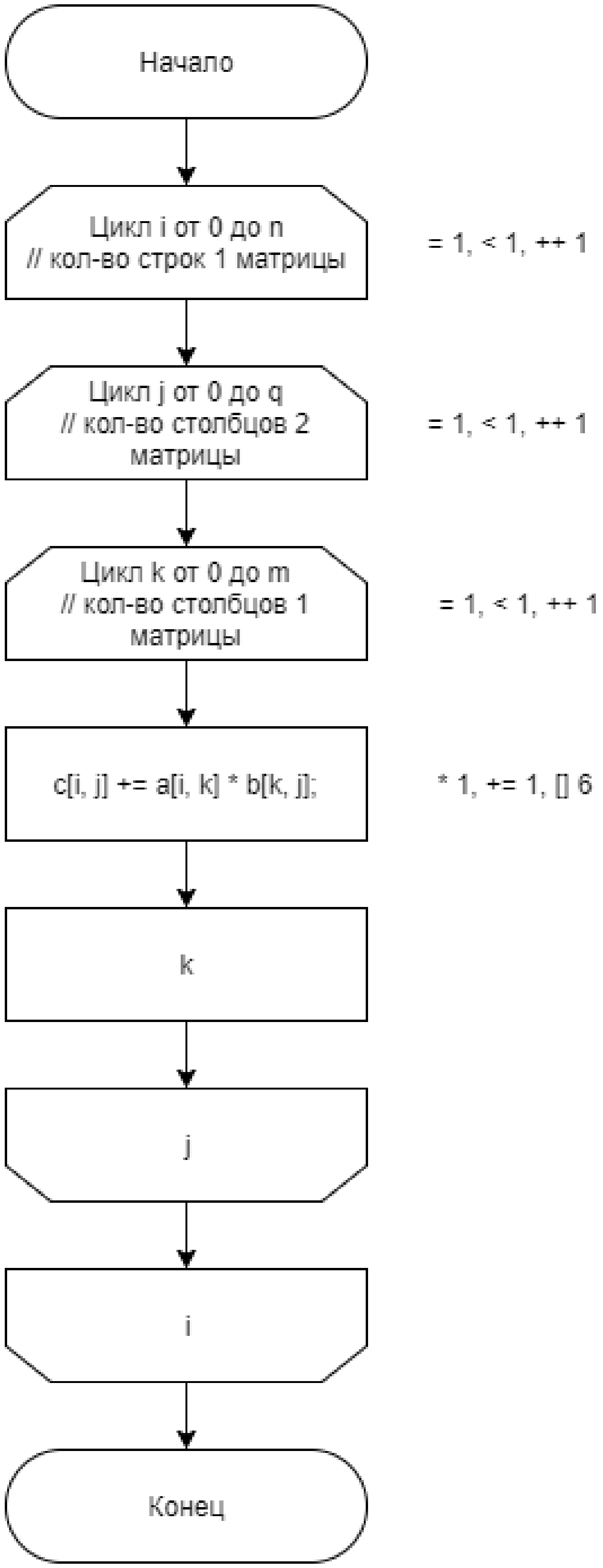

Рис. 1: Блок-схема базового алгоритма

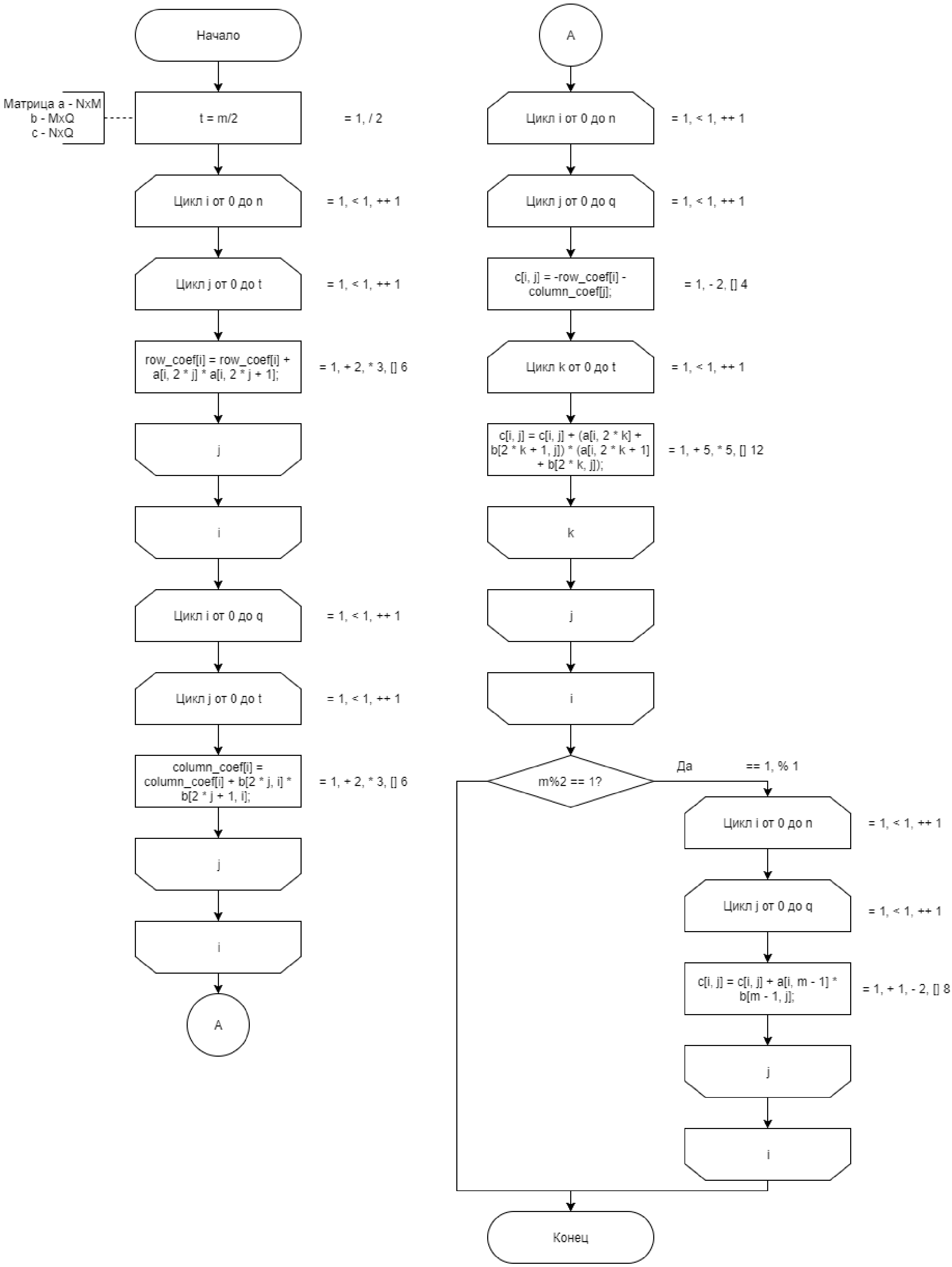

Рис. 2: Блок-схема алгоритма Винограда

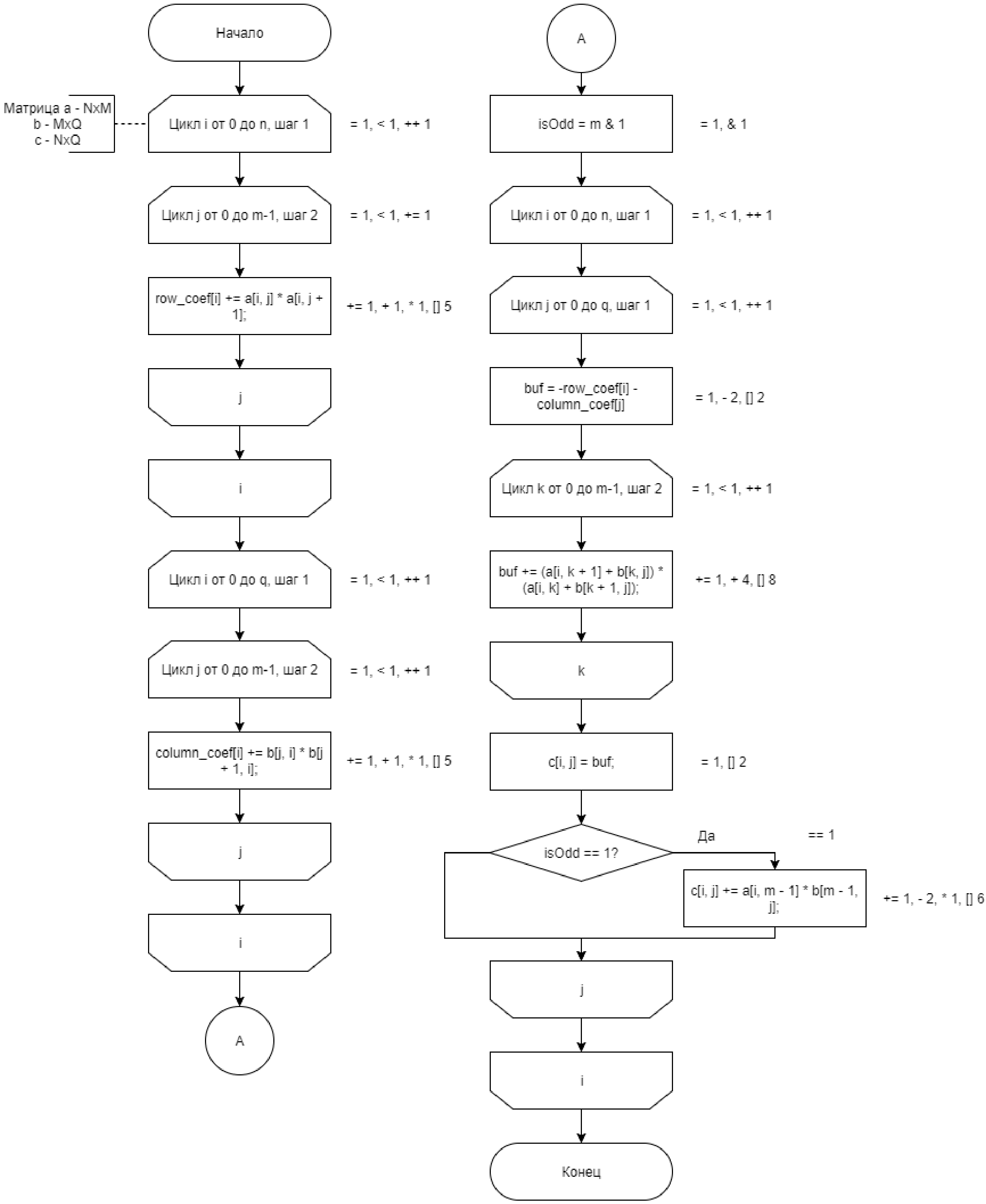

Рис. 3: Блок-схема улучшенного алгоритма Винограда

Листинг

MAIN.CPP:

```cpp
#include <iostream>
#include <fstream>
#include "mymat.cpp"

#define REP_NUM 7           // std == 7
#define REP_SIZE_NUM 10     // std == 10

using namespace std;

template <typename T>
void timeCounter(int st_size, int d_size, int rep_size_num, ofstream & file, int rep_num,
                 long double (*f)(const vector<vector<T> >&, const vector<vector<T> >&))
{
    vector<vector<T> > mat1;
    vector<vector<T> > mat2;
    long double res;

    for (int i = 0; i < rep_size_num; i++)
    {
        randFill(mat1, st_size + d_size * i, st_size + d_size * i);
        randFill(mat2, st_size + d_size * i, st_size + d_size * i);

        res = 0;
        for (int j = 0; j < rep_num; j++)
        {
            res += f(mat1, mat2);
        }
        res /= rep_num;

        cout.setf(ios::fixed);
        cout.precision(4);
        cout << st_size+i*d_size << " : " << res << endl;

        file.setf(ios::fixed);
        file.precision(4);
        file << res << "\n";
    }
    file << "\n";
}

int main(void)
{
    srand(time(0));
    ofstream file_ev("results_even.txt");
    ofstream file_odd("results_odd.txt");
```

```
    timeCounter<double>(100, 100, REP_SIZE_NUM, file_ev, REP_NUM, classicMatMult<double>);
    timeCounter<double>(101, 100, REP_SIZE_NUM, file_odd, REP_NUM, classicMatMult<double>);

    timeCounter<double>(100, 100, REP_SIZE_NUM, file_ev, REP_NUM, VinogradMatMult<double>);
    timeCounter<double>(101, 100, REP_SIZE_NUM, file_odd, REP_NUM, VinogradMatMult<double>);

    timeCounter<double>(100, 100, REP_SIZE_NUM, file_ev, REP_NUM, betterVinogradMatMult<double>);
    timeCounter<double>(101, 100, REP_SIZE_NUM, file_odd, REP_NUM, betterVinogradMatMult<double>);

    file_ev.close();
    file_odd.close();

    return 0;
}
```

MYMAT.CPP:

```
#include "mymat.h"

template <typename T>
long double classicMatMult(const vector<vector<T> >& mat1, const vector<vector<T> >& mat2)
{
    vector<vector<T> > ans;
    vector<T> tmp;
    int n = mat1.size();
    int m = mat2.size();
    int q = mat2[0].size();

    tmp.resize(q, 0);
    ans.resize(n, tmp);

    unsigned long int start = clock();

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < q; j++)
        {
            for (int k = 0; k < m; k++)
            {
                ans[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
```

```

    return double(clock() - start)/double(CLOCKS_PER_SEC)*1000;
}

template <typename T>
long double VinogradMatMult(const vector<vector<T> >& mat1, const vector<vector<T> >& mat2)
{
    vector<vector<T> > ans;
    vector<T> tmp;

    int n = mat1.size();
    int m = mat2.size();
    int q = mat2[0].size();

    tmp.resize(q, 0);
    ans.resize(n, tmp);

    vector<T> row_factor;
    vector<T> col_factor;

    row_factor.resize(n, 0);
    col_factor.resize(q, 0);

    int d = m/2;

    unsigned long int start = clock();

    // Row Factor calc
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < d; j++)
        {
            row_factor[i] += mat1[i][2*j] * mat1[i][2*j+1];
        }
    }

    // Col Factor calc
    for (int i = 0; i < q; i++)
    {
        for (int j = 0; j < d; j++)
        {
            col_factor[i] += mat2[2*j][i] * mat2[2*j+1][i];
        }
    }

    // Ans calc
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < q; j++)
```

```
            {
                ans[i][j] = - row_factor[i] - col_factor[j];
                for (int k = 0; k < d; k++)
                {
                    ans[i][j] += (mat1[i][2*k] + mat2[2*k+1][j]) * (mat1[i][2*k+1] + mat2[2*k][j]);
                }
            }
        }

        // For odd matrix
        if (m % 2)
        {
            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < q; j++)
                {
                    ans[i][j] += mat1[i][m-1] * mat2[m-1][j];
                }
            }
        }

        return double(clock() - start)/double(CLOCKS_PER_SEC)*1000;
}

template <typename T>
long double betterVinogradMatMult(const vector<vector<T> >& mat1, const vector<vector<T> >& mat2)
{
        vector<vector<T> > ans;
        vector<T> tmp;

        int n = mat1.size();
        int m = mat2.size();
        int q = mat2[0].size();

        tmp.resize(q, 0);
        ans.resize(n, tmp);

        vector<T> row_factor;
        vector<T> col_factor;

        row_factor.resize(n, 0);
        col_factor.resize(q, 0);

        int d = m-1;

        unsigned long int start = clock();

        // Row Factor calc
```

```
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < d; j += 2)
        {
            row_factor[i] += mat1[i][j] * mat1[i][j+1];
        }
    }

    // Col Factor calc
    for (int i = 0; i < q; i++)
    {
        for (int j = 0; j < d; j += 2)
        {
            col_factor[i] += mat2[j][i] * mat2[j+1][i];
        }
    }

    // Ans calc
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < q; j++)
        {
            T buf = -row_factor[i] - col_factor[j];
            for (int k = 0; k < d; k += 2)
            {
                buf += (mat1[i][k] + mat2[k+1][j]) * (mat1[i][k+1] +
                    mat2[k][j]);
            }
            ans[i][j] = buf;
        }
    }

    // For odd matrix
    if (m % 2)
    {
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < q; j++)
            {
                ans[i][j] += mat1[i][m-1] * mat2[m-1][j];
            }
        }
    }

    return double(clock() - start)/double(CLOCKS_PER_SEC)*1000;
}


template <typename T>
void randFill(vector<vector<T> >& mat, const int row, const int col)
```

```
{
    matClr(mat);
    vector<T> tmp;

    for (int i = 0; i < row; i++)
    {
        tmp.clear();

        for (int j = 0; j < col; j++)
        {
            tmp.push_back(rand());
        }

        mat.push_back(tmp);
    }
}

template <typename T>
void matClr(vector<vector<T> >& mat)
{
    for (auto &x : mat)
    {
        x.clear();
    }
    mat.clear();
}
```

Временные эксперименты

Измерения проводились для квадратных вещественных матриц

Размеры матриц	Обычное	Виноград	Улучшенный Виноград
100x100	24.6549	14.0676	11.3303
200x200	170.8249	137.3591	119.8764
300x300	488.6610	429.6541	355.9341
400x400	1358.2967	1082.4930	935.0479
500x500	3132.0469	2562.7789	2248.1820
600x600	5765.4314	4467.8990	3802.2509
700x700	8566.0461	7090.2327	6045.3397
800x800	12530.9540	10513.3806	9130.0609
900x900	18801.5250	15415.6954	13241.0050
1000x1000	26042.9380	24677.6349	26579.2120

Замеры времени в милисекундах (среднее из 7 замеров)

Рис. 4: Сравнение времени при четном кол-ве элементов

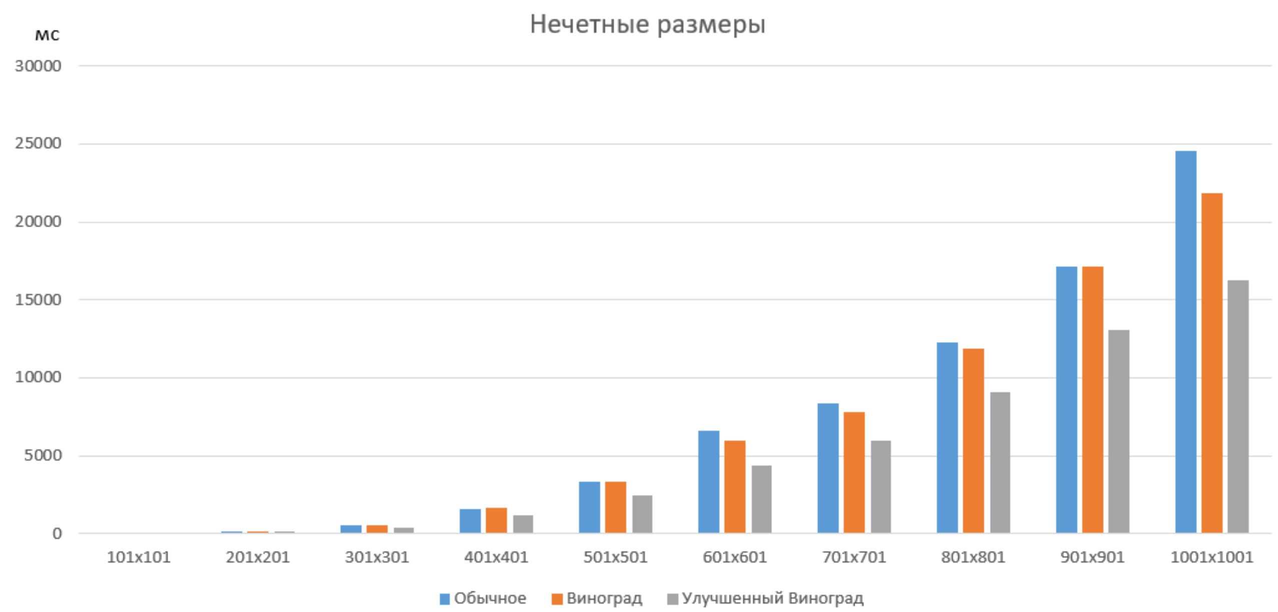

Рис. 5: Сравнение времени при нечетном кол-ве элементов

Теоретическая оценка

а) Обычное умножение:
$f = 2 + N * (2 + 2 + Q(2 + 2 + M(2 + 8))) = 2 + 4N + 4NQ + 10MNQ \sim O(n^3)$
б) Умножение алгоритмом Винограда:

- Переменная t: $f = 2$
- row coef: $f = 2 + N(2 + 2 + 2 + M/2(2 + 12)) = 2 + 6N + 7MN$
- column coef: $f = 2 + Q(2 + 2 + 2 + M/2(2 + 12)) = 2 + 6Q + 7MQ$
- Вычисление результирующей матрицы:
  $f = 2 + N(2 + 2 + Q(2 + 7 + 2 + M/2(2 + 23))) = 2 + 4N + 11QN + 25MNQ/2$
- Вычисления последней строки: $f = 2 + 2 + N(2 + 2 + Q(2 + 13)) = 4 + 4N + 15NQ$

Лучший случай (размер матрицы четный):
$f = 2 + 2 + 6N + 7MN + 2 + 6Q + 7MQ + 2 + 4N + 11QN + 25MNQ/2 =$
$8 + 10N + 7MN + 6Q + 7QM + 11QN + 25QMN/2 \sim O(n^3)$
Худший случай (размер матрицы нечетный):
$f = 2 + 2 + 6N + 7MN + 2 + 6Q + 7MQ + 2 + 4N + 11QN + 25MNQ/2 + 4 + 4N + 15NQ =$
$12 + 14N + 7MN + 6Q + 7QM + 26QN + 25QMN/2 \sim O(n^3)$
в) Умножение улучшенным алгоритмом Винограда:

- Переменная t: $f = 2$
- row coef: $f = 2 + N(2 + 2 + 2 + (M - 1)/2(2 + 8)) = 5MN + N + 2$
- column coef: $f = 2 + Q(2 + 2 + 2 + (M - 1)/2(2 + 8)) = 5MQ + Q + 2$
- Переменная isOdd: $f = 2$
- Вычисление результирующей матрицы:
  $f = 2 + N(2 + 2 + Q(2 + 5 + 2 + (M - 1)/2(2 + 14) + 3[+1 + 10])) =$
  $2 + 4N + 8MNQ + [5NQ|14NQ]$

Лучший случай (размер матрицы четный):
$f = 2 + 5MN + N + 2 + 5MQ + Q + 2 + 2 + 2 + 4N + 8MNQ + 5NQ =$
$10 + Q + 5N + 5MN + 5MQ + 5NQ + 8MNQ \sim O(n^3)$
Худший случай (размер матрицы нечетный):
$f = 2 + 2 - 4N + 10MN + 2 - 4Q + 10MQ + 2 + 2 + 4N + 8MNQ + 13NQ =$
$10 + Q + 5N + 5MN + 5MQ + 13NQ + 8MNQ \sim O(n^3)$

Выводы

Сравнение алгоритмов по времени подтверждает теоретические замеры времени.
Алгоритмы работают примерно одинаково, что подтверждает их оценка трудоемкости порядка $O(n^3)$.
После умножения матриц размерами примерно 600x600 алгоритм Винограда работает быстрее обычного алгоритма умножения матриц постоянно.

Заключение

В ходе лабораторной работы были реализованы 3 алгоритма умножения матриц: классический, алгоритм Винограда и улучшенным алгоритмом Винограда. Были получены навыки работы с матрицами и контейнером std::vector в C++, а так же с LaTeX. Изучен подход к вычислению сложности алгоритмов.