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 «ФУНДАМЕНТАЛЬНЫЕ НАУКИ»

КАФЕДРА «ПРИКЛАДНАЯ МАТЕМАТИКА»

Лабораторная работа № 1

по дисциплине «Типы и структуры данных»

Тема Умножение мартиц

Студент Лямин И.С.

Группа ФН12-31Б

Преподаватели Волкова Л.Л.

Москва, 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В лабораторной работе будут разобраны и реализованы алгоритмы перемножения произвольных матриц(отличных рамеров), стандартным алгоритмом, алгоритмом Винограда и оптимизированным алгоритмом Винограда.

Во всей работе подразумевается, что lh(left height) – высота левой матрицы, lw(left wide) – ширина левой матрицы, rh(right height) – высота правой матрицы, lh(right wide) – ширина правой матрицы.

Цель работы – выполнить оценку ресурсной эффективности алгоритмов перемножения мартиц и их реализации.

Для достижения поставленной цели требуется выполнить следующие задачи.

1. Описать математическую основу обычного алгоритма, алгоритма Винограда и оптимизированного алгоритма Винограда перемножения стриц,
2. Описать модель вычисления,
3. Реализовать программу на основе этих алгоритмов.
4. Выполнить оценку трудоёмкости реализации алгоритмов,
5. Реализовать разработанные алгоритмы в программном обеспечении с двумя режимами работы – одиночного расчёта и массированного замера процессорного времени,
6. Выполнить замеры процессорного времени выполнения реализации каждого алгоритма в зависимости от размера матриц,
7. Выполнить сравнительный анализ рассчитанных трудоёмкостей и результатов замера процессорного времени,

1 Аналитическая часть

1.1 Стандартный алгоритм

Стандартный алгоритм: представляет собой поочерёдное перемножение i-ой вектор-строки первой мартицы на j-ый вектор-столбец второй матрицы, таким образом получим элемнт [i, j] из матрицы. Формула для этого метода перемножения выглядит следующим образом:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1l} \\ b_{21} & b_{22} & \dots & b_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{ml} \end{pmatrix}$$

$$A * B_{i,j} = \sum_{k=1}^{m}(a_{ik} \cdot b_{kj}) \tag{1}$$

1.2 Алгоритм Винограда

Алгоритм Винограда — алгоритм направлен на уменьшение количества операций умножения помещением часто используемых произведений в буффер и использованием вычисленных значений при необходимой итерации. Рассмотрим элемент c_{ij} из матрицы C:

$$c_{ij} = \sum_{k=1}^{m/2}((a_{ik\cdot 2} + b_{k\cdot 2+1j}) \cdot (a_{ik\cdot 2+1} + b_{k\cdot 2j}) - (a_{ik\cdot 2} \cdot a_{ik\cdot 2+1}) - (b_{ik\cdot 2} \cdot b_{ik\cdot 2+1})) \tag{2}$$

На этом тождестве основывется алгоритм Винограда. Все произведения вида $a_{i,k} * a_{i,k+1}$, будут всегда вычитаться при получении элемента i-ой строки новой матрицы и аналогично j-ого столбца. Следовательно можно создать два массива с элементами которые используются по одному при каждой итерации вычисления значения элемента матрицы $c_{i,j}$.

$$rows[j] = \sum_{i=0}^{m/2}(a_{ji\cdot 2} \cdot a_{ji\cdot 2+1}), j = 1, 2, 3, ..., n, \tag{3}$$

$$columns[j] = \sum_{i=0}^{n/2}(b_{i\cdot 2j} \cdot b_{i\cdot 2+1j}), j = 1, 2, 3, ..., l \tag{4}$$

где m - количество стлобцов первой матрицы,
n - количество строк в первой матрице, l - количество столбцов во второй матрице

Итоговая формула вычисления значений элемента строки i столбца j матрицы C имеет вид:

$$c_{ij} = (\sum_{k=1}^{m/2} (a_{ik\cdot 2} + b_{k\cdot 2+1j}) \cdot (a_{ik\cdot 2+1} + b_{k\cdot 2j})) - row[i] - columns[j] \tag{5}$$

И для случая когда m нечётное:

$$c_{ij} = (\sum_{k=1}^{(m-1)/2} (a_{ik\cdot 2}+b_{k\cdot 2+1j})\cdot(a_{ik\cdot 2+1}+b_{k\cdot 2j}))-row[i]-columns[j]+a_{i(m/2)}\cdot b_{(m/2)j} \tag{6}$$

1.3 Оптимизированный алгоритм Винограда

Оптимизированный Алгоритм Винограда — оптимизация моего варианта состоит в замене операции умножения на двоичный сдвиг:

$$c_{ij} = (\sum_{k=1}^{(m)/2} (a_{ik<<1} + b_{k<<1+1j}) \cdot (a_{ik<<1+1} + b_{k<<1j})) - row[i] - columns[j] \tag{7}$$

$$\begin{gathered} c_{ij} = (\sum_{k=1}^{(m-1)/2} (a_{ik<<1} + b_{k<<1+1j}) \cdot (a_{ik<<1+1} + b_{k<<1j})) \\ -row[i] - columns[j] + a_{i(m/2)} \cdot b_{(m/2)j} \end{gathered} \tag{8}$$

формула (7) для случая чётного количество строк в первой матрице и формула (8) для нечётного случая.

Второй частью оптимизации было объединение операции нахождения массивов $rows$ и $columns$. Реализовано следующим образом, $lim = max(l_w, r_h)$, где l_w-ширина левой матрицы, а r_h-высота правой. Пременная lim показывает до какой итерации дойтёт цикл в котором будут перемножаться i и $i+1$ элементы из соответствующий строк и столбцов. И в момент когда либо строки в первой матрице, либо столбцы во второй закончатся, то перемножение элементов строки или соответственно столбца прекратится, а процесс получения второго массива продолжится до момента пока количество интераци не достигнет значения lim. Это будет реализованно в коде, также как и вынос первой итерации.

2 Конструкторская часть

2.1 Описание алгоритмов

На следующих рисунках блок-схем представлены соответствующие алгоритмы: 1 - стандартный алгоритм, 4 - Алгоритм Винограда, и оптимизированный алгоритм Винограда 5. Также перед описанием алгоритма Винограда и оптимизированного алгоритма Винограда опишем вспомогательные функции $prepair\_arrays$ рисунок 2 и $optimzed\_prepair\_arrays$ рисунок 3

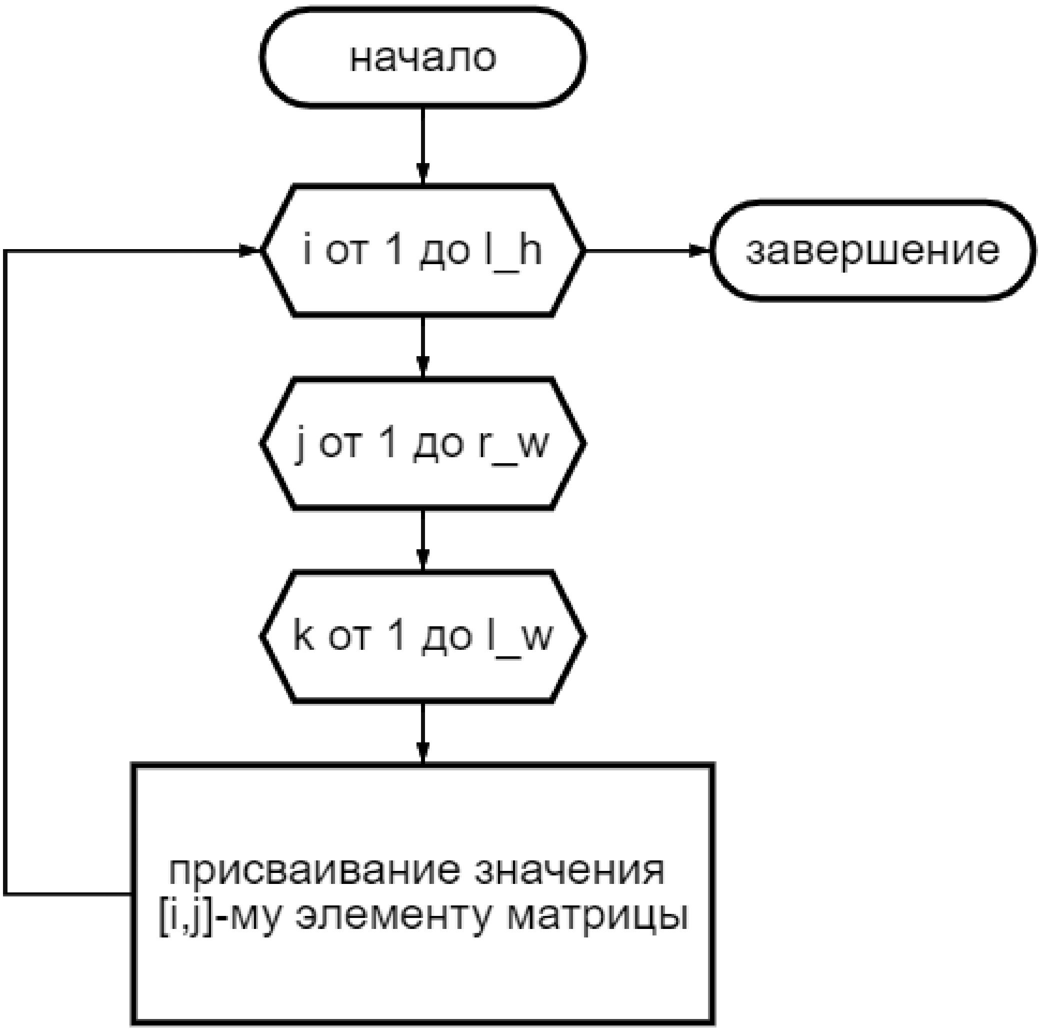

Рисунок 1 — Схема стандартного алгоритма

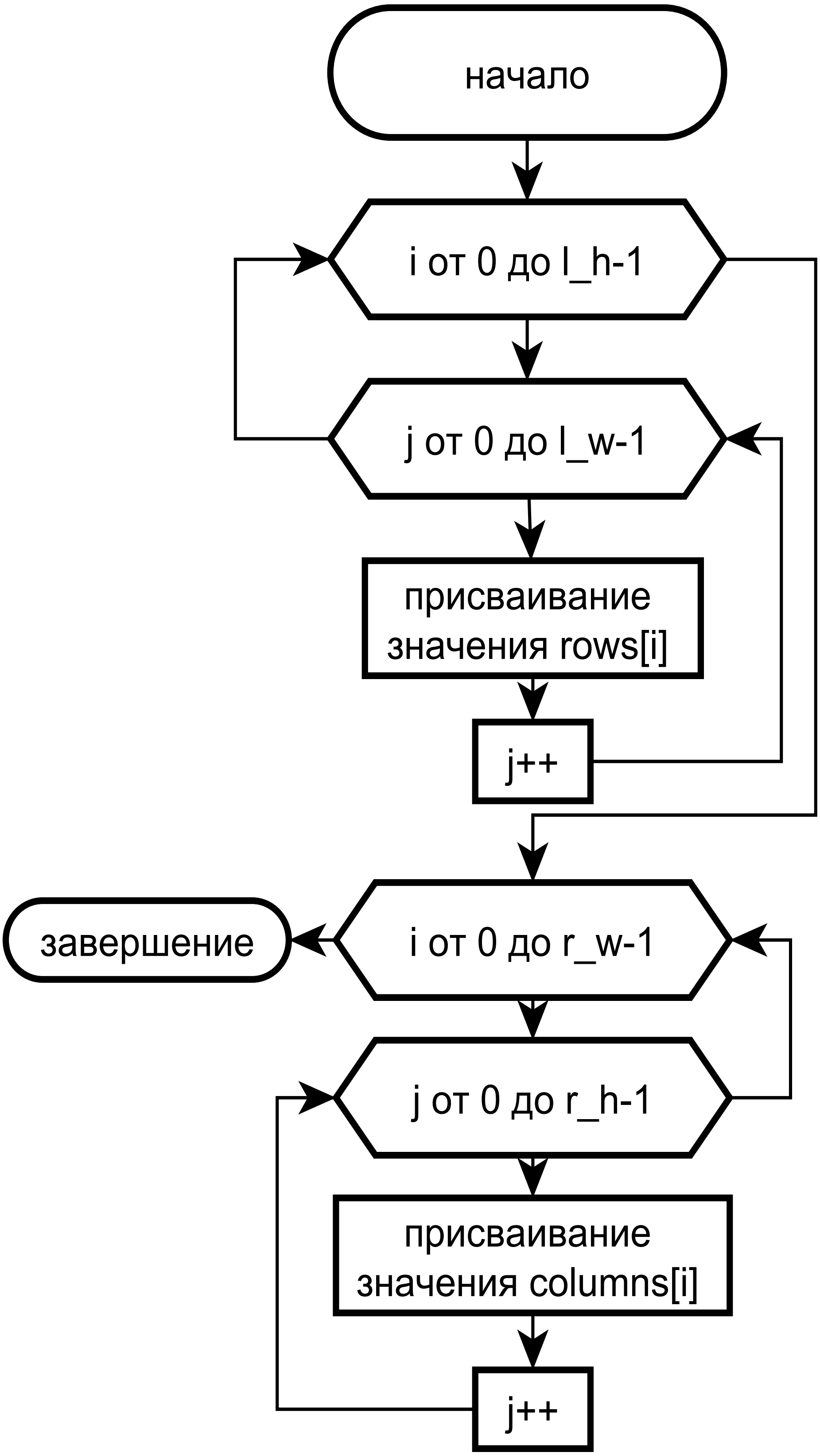

Рисунок 2 — PrepairArrays

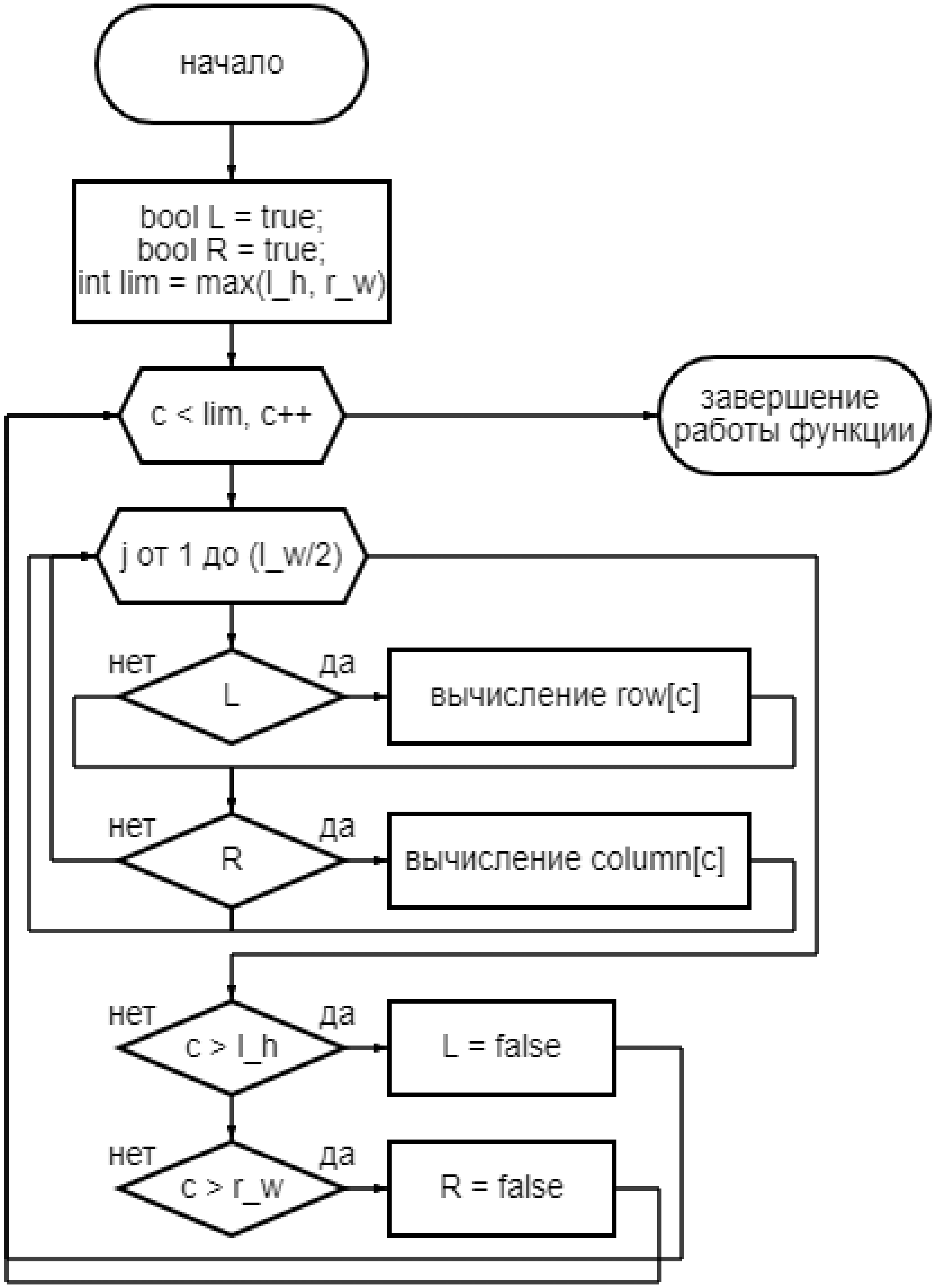

Рисунок 3 — OptimizedPrepairArrays

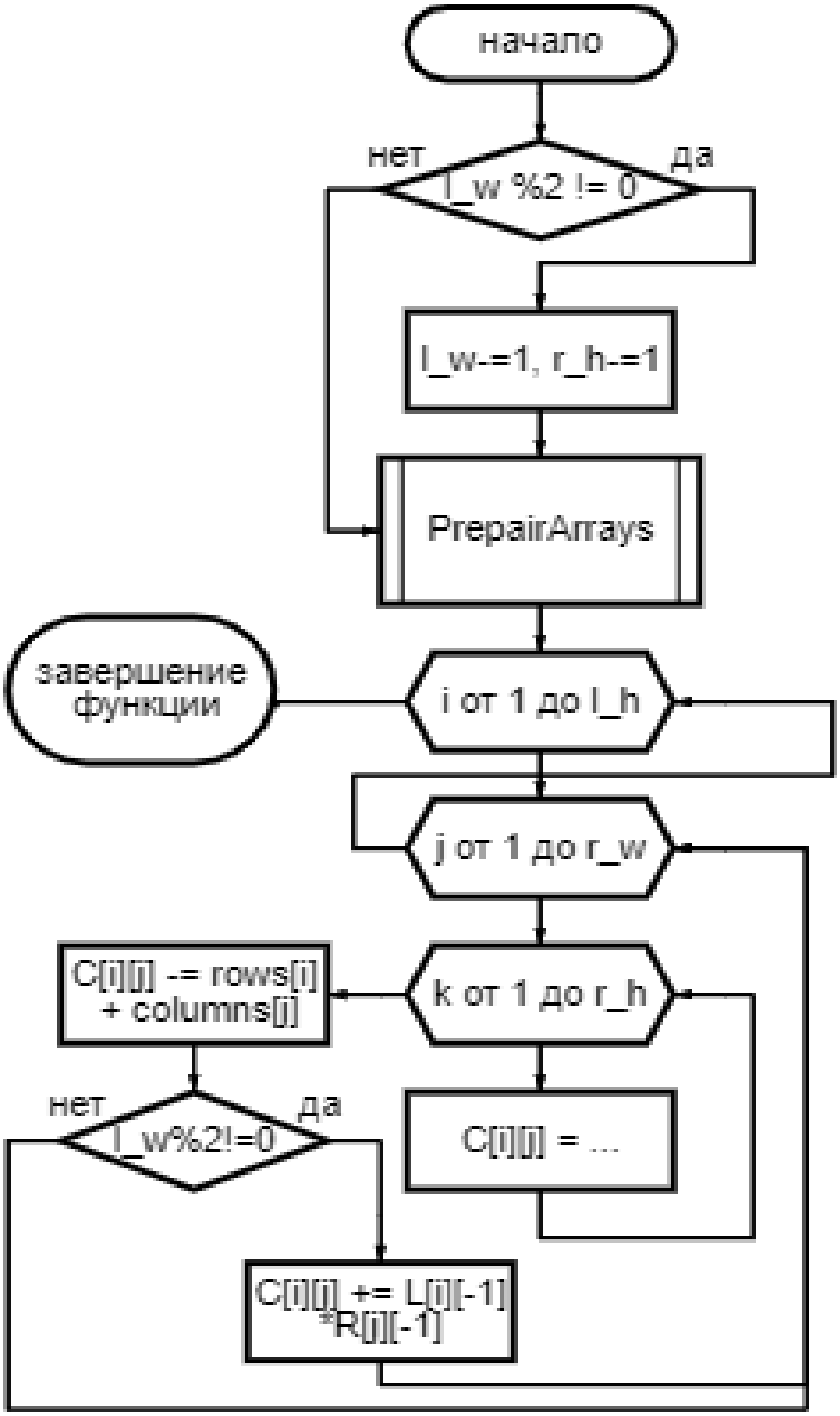

Рисунок 4 — Алгоритм Винограда

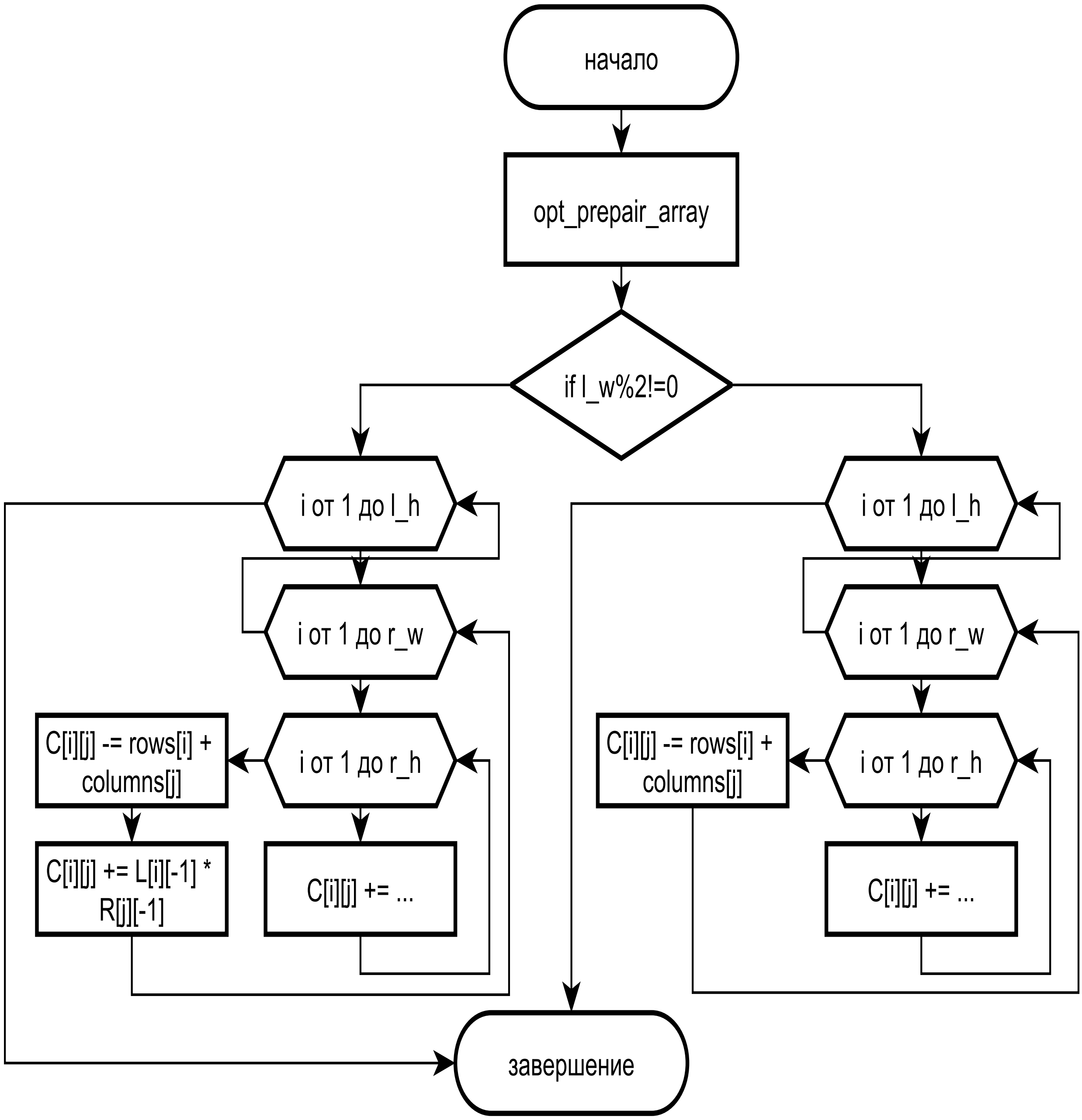

Рисунок 5 — Оптимизированный Алгоритм Винограда

2.2 Анализ сложностей

Обычный(лучший случай, остальные случаи):

$$\left[\begin{array}{c} 0, lw \neq rh \\ 2 + 4 \cdot lh + 4 \cdot rw \cdot lh + 14 \cdot rw \cdot lw \cdot lh \end{array}\right.$$

Стандартный алгоритм подготовки массивов rows и columns(prep_arr)

$$\left[\begin{array}{c} 0, lw \neq rh \\ 2 + 4 \cdot lh + 12 \cdot rw \cdot lh \end{array}\right.$$

Виноград(лучший случай, чётные матрицы, остальные случаи):

$$\left[\begin{array}{c} 0, lw \neq rh \\ 4 + 6 \cdot lh + 12 \cdot lw + 8 \cdot rw \cdot lh + 27/2 \cdot rw \cdot lh \cdot rh \\ 4 + 6 \cdot lh + 12 \cdot lw + 8 \cdot rw \cdot lh + 9rw \cdot lh + 27/2 \cdot rw \cdot lh \cdot rh \end{array}\right.$$

Оптимизированный алгоритм подготовки массивов rows columns(opt_prep_arr)

$$\left[\begin{array}{c} 0, lw \neq rh \\ 2 + lh + rw + 19 * max + 2 * max * lw + 9/2 * lw * rh + 9/2 * lw * rw \end{array}\right.$$

Где max = max(lh, rw)

Оптимизированный Виноград(лучший случай, чётные матрицы, остальные случаи):

$$\left[\begin{array}{c} 0, lw \neq rh \\ 6 + OptPrepArr + 4 * lh + 10 * rw * lh + 23/2 * rw * rh * lh \\ 8 + OptPrepArr + 4 * lh + 10 * rw * lh + 7 * rw * lh + 23/2 * rw * rh * lh \end{array}\right.$$

Стандартный алгоритм будет работать дольше всего в связи с тем, что коэфицент при rw*rh*lh (что является самой трудозатратной частью алгоритма при достаточно больших матрицах) в этом алгоритме больше чем в других (27/2 - Виноград и 23/2 оптимизированный Виноград) остальная часть играет меньшую роль в значении трудозатратности алгоритма.

3 Технологическая часть

3.1 Выбор средств реализации

Для программной реализации алгоритма использовалась среда разработки Visual Studio, язык программирования, на котором была выполнена реализации алгоритмов, — C++. Исследование проводилось на ноутбуке (64-разрядная операционная система, процессор x64, частота процессора 3.10 Ггц, оперативная память 16 ГБ)

Для замера времени использовалась функция now() библиотеки chrono[2].

3.2 Реализация алгоритмов

В листинге 1 можно увидеть программную реализацию описанных алгоритмов.

Листинг 1 — Программная реализация

```
template<class dtype>
Mass<dtype> multiplication(Mass<dtype>& array_l, Mass<dtype>& array_r) {
        Mass<dtype> new_ar(array_l.get_height(), array_r.get_wide());
        Timer x;
        x.start_time();
        for (int i = 0; i < array_l.get_height(); i++) {
                for (int j = 0; j < array_r.get_wide(); j++) {
                        for (int k = 0; k < array_l.get_wide(); k++) {
                                new_ar[i][j] += array_l[i][k] * array_r[k][j];
                        }
                }
        }
        x.stop_time();
        x.get_time();
        return new_ar;
}

///////////////////vinograd//////////////////

template<class dtype>
void prepair_arrays(dtype* array_rows, dtype* array_columns, Mass<dtype>& array_l, Mass<dtype>& array_r, int l_h, int l_w, int r_h, int r_w) {

        for (int i = 0; i < l_h; i++) {
                array_rows[i] = 0;
                for (int j = 0; j < l_w; j++) {
                        array_rows[i] += array_l[i][j] * array_l[i][j + 1];
```

```
			j++;
		}
	}
	array_rows[l_h] = -1;
	for (int i = 0; i < r_w; i++) {
		array_columns[i] = 0;
		for (int j = 0; j < r_h; j++) {
			array_columns[i] += array_r[j][i] * array_r[j + 1][i];
			j++;
		}
	}
	array_columns[r_w] = -1;
}

template<class dtype>
Mass<dtype> vinograd_multiplication(Mass<dtype>& array_l, Mass<dtype>& array_r) {
	Mass<dtype> new_ar(array_l.get_height(), array_r.get_wide());
	dtype fix_rows[FIX_ARR_SIZE];
	dtype fix_columns[FIX_ARR_SIZE];

	Timer x;
	x.start_time();

	BOOL A = TRUE;
	int l_h = array_l.get_height();
	int l_w = array_l.get_wide();
	int r_h = array_r.get_height();
	int r_w = array_r.get_wide();

	if ((array_l.get_wide() % 2) != 0) {
		A = FALSE;
		l_w -= 1;
		r_h -= 1;
	}


	prepair_arrays(fix_rows, fix_columns, array_l, array_r, l_h, l_w, r_h, r_w);

	for (int i = 0; i < l_h; i++) {
		for (int j = 0; j < r_w; j++) {
			for (int k = 0; k < (r_h); k++) {
				new_ar[i][j] += (array_l[i][k] + array_r[(k) + 1][j]) * (array_l[i][(k) + 1] + array_r[(k)][j]);
				k++;
			}
			if (!(A)) {
				new_ar[i][j] += array_l[i][l_w] * array_r[r_h][j];
			}
```

```
                                new_ar[i][j] -= (fix_rows[i] + fix_columns[j]);
                        }
                }

                x.stop_time();
                x.get_time();

                return new_ar;
}

//////////////////optimized_vin//////////////////

template<class dtype>
void opt_prepair_arrays(dtype* array_rows, dtype* array_columns, Mass<dtype>& array_l, Mass<dtype>& array_r, int l_h, int l_w, int r_h, int r_w) {
        int lim = max(l_h, r_w);
        int counter = 0;
        bool L = true;
        bool R = true;
        while (counter < lim) {
                if (L) {
                        array_rows[counter] = 0;
                }
                if (R) {
                        array_columns[counter] = 0;
                }
                if ((l_w != 1) && (l_h != 1)) {
                        array_rows[counter] += array_l[counter][0] * array_l[counter][1];
                        array_columns[counter] += array_r[0][counter] * array_r[1][counter];
                }
                for (int j = 1; j < (l_w / 2); j++) {
                        if (L) {
                                array_rows[counter] += array_l[counter][j << 1] * array_l[counter][(j << 1) + 1];
                        }
                        if (R) {
                                array_columns[counter] += array_r[j << 1][counter] * array_r[(j << 1) + 1][counter];
                        }
                }
                counter++;
                if (counter >= l_h) L = false;
                if (counter >= r_w) R = false;
        }
        array_rows[l_h] = -1;
        array_columns[r_w] = -1;
}
```

```

template<class dtype>
Mass<dtype> optimized_vinograd_multiplication(Mass<dtype>& array_l, Mass<dtype>& array_r) {
        Mass<dtype> new_ar(array_l.get_height(), array_r.get_wide());
        dtype fix_rows[FIX_ARR_SIZE];
        dtype fix_columns[FIX_ARR_SIZE];

        Timer x;
        x.start_time();

        bool A = true;
        int l_h = array_l.get_height();
        int l_w = array_l.get_wide();
        int r_h = array_r.get_height();
        int r_w = array_r.get_wide();


        if ((array_l.get_wide() % 2) != 0) {
                A = FALSE;
                l_w -= 1;
                r_h -= 1;
        }

        opt_prepair_arrays(fix_rows, fix_columns, array_l, array_r, l_h, l_w, r_h, r_w);
        if (!(A)) {
                for (int i = 0; i < l_h; i++) {
                        for (int j = 0; j < r_w; j++) {
                                for (int k = 0; k < (r_h / 2); k++) {
                                        new_ar[i][j] += (array_l[i][k << 1] + array_r[(k << 1) + 1][j]) * (array_l[i][(k << 1) + 1] + array_r[(k << 1)][j]);
                                }
                                new_ar[i][j] -= (fix_rows[i] + fix_columns[j] - array_l[i][l_w] * array_r[r_h][j]);
                        }
                }
        }
        else {
                for (int i = 0; i < l_h; i++) {
                        for (int j = 0; j < r_w; j++) {
                                for (int k = 0; k < (r_h / 2); k++) {
                                        new_ar[i][j] += (array_l[i][k << 1] + array_r[(k << 1) + 1][j]) * (array_l[i][(k << 1) + 1] + array_r[(k << 1)][j]);
                                }
                                new_ar[i][j] -= (fix_rows[i] + fix_columns[j]);
                        }
                }
        }
```

```
        x.stop_time();
        x.get_time();

        return new_ar;
}

//////////////////MODES//////////////////

void MODE_1() {
        int hei, wid;
        cout << "write height and wide for first" << "\n";
        cin >> hei;
        cin >> wid;
        Mass<int> A(hei, wid);
        Mass<int> C;
        cout << "fill the first matrix" << "\n";
        A.fill_arr(1);
        A.print();
        cout << "write height and wide for second" << "\n";
        cin >> hei;
        cin >> wid;
        Mass<int> B(hei, wid);
        cout << "fill the second matrix" << "\n";
        B.fill_arr(1);
        cout << "Ordinary:\n";
        C = multiplication(A, B);
        C.print();
        C = vinograd_multiplication(A, B);
        cout << "Vinograd:\n";
        C.print();
        cout << "Optimized Vingrad:\n";
        C = optimized_vinograd_multiplication(A, B);
        C.print();
}


void MODE_2() {
        srand(time(0));
        int size;

        for (int i = 0; i < 10; i++) {
                size = 10 + 10 * i;
                cout << "SIZE: " << size << endl;
                Mass<int> A(size, size);
                Mass<int> B(size, size);
                Mass<int> C(size, size);
                for (int r = 0; r < size; r++) {
                        for (int c = 0; c < size; c++) {
                                A[r][c] = get_num();
                                B[r][c] = get_num();
                        }
```

```
                }
                cout << "-----A*B----- ";
                C = multiplication(A, B);
                cout << "-----A*B_vin----- ";
                C = vinograd_multiplication(A, B);
                cout << "-----A*B_opt_vin----- ";
                C = optimized_vinograd_multiplication(A, B);
        }
}
```

3.3 Тестирование программы

В таблицах 1 и 2 представлены описания тестов по методологии чёрного ящика, все тесты пройдены успешно Все тесты пройдены успешно.

Таблица 1 — Асимптотические сложности реализаций алгоритмов

	Описание теста	**Входные данные**	**Ожидаемый результат**	**Полученный результат**
1	проверка на обработку не валидных данных	1 2 3 3 3 1	оповещение о некорректности данных и запрос новых	оповещение о некорректности данных и запрос новых
2	проверка на обработку нулевой матрицы	0 1	оповещение программы и запрос новых данных	оповещение программы и запрос новых данных
3	умножение квадратных матриц	2 2 1 2 3 4 2 2 5 6 7 8	19 22 43 50	19 22 43 50

Таблица 2 — Асимптотические сложности реализаций алгоритмов

4	умножение не квадратных матриц	1 2 1 2 2 3 5 6 7 8 9 10	21 24 27 47 54 61	21 24 27 47 54 61
5	умножение векторов	1 2 1 2 2 1 5 6	17	17

4 Исследовательская часть

4.1 Графики зависимостей процессорного времени

В результате работы был проведён массированный замер эффективности реализаций алгоритмов. В результате этого замера, составлен график зависимости процессорного времени от размера перемножаемых матриц. График изображён на рисунке 6.

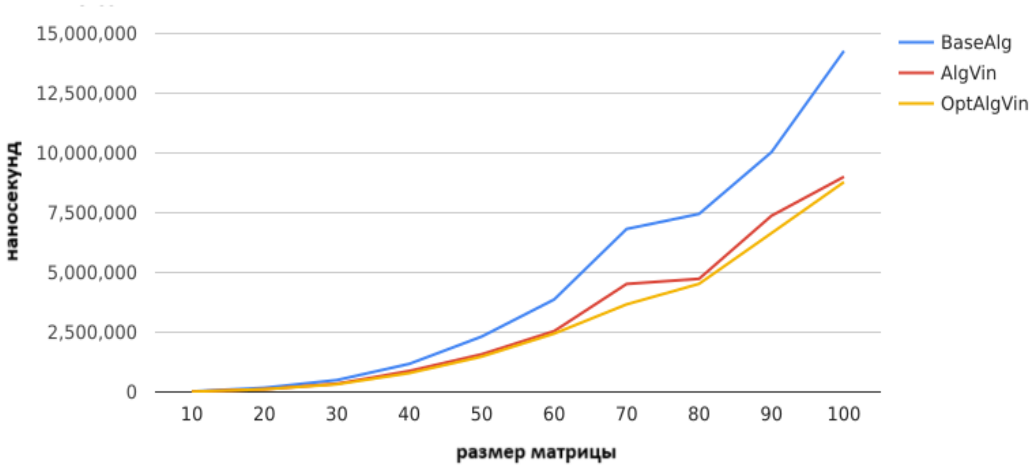

Рисунок 6 — Визуализация зависимости процессорного времени выполнения алгоритмов от размеров матриц

Из этого можно сделать вывод, что реализация алгоритма Винограда и оптимизированного алгоритма Винограда затрачивают меньше процессорного времени для расчёта сответствующих матриц. Но они используют большее количество памяти, так как помимо трёх двумерных массивов (матриц) должны хранить ещё и два массива со значаниями часто используемых произведений .

ЗАКЛЮЧЕНИЕ

В результате лабораторной работы была достигнута цель – рассмотрены и разобраны три алгоритма перемножения матриц (стандартный, Виноград, оптимизированный Виноград).
Были выполнены все поставленные задач.

1. Описана математическая основу обычного алгоритма, алгоритма Винограда и оптимизированного алгоритма Винограда перемножения стриц,

2. Описана модель вычисления,

3. Реализована программа на основе этих алгоритмов,

4. Выполнена оценка трудоёмкости реализации алгоритмов,

5. Реализованы разработанные алгоритмы в программном обеспечении с 2-мя режимами работы – одиночного расчёта и массированного замера процессорного времени,

6. Выполнены замеры процессорного времени выполнения реализации каждого алгоритма в зависимости от размера матриц,

7. Выполнен сравнительный анализ рассчитанных трудоёмкостей и результатов замера процессорного времени,

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ульянов М. В. Ресурсно эффективные компьютерные алгоритмы. Москва 2007. ФИЗМАТЛИТ. 376 стр.

2. Microsoft. GetProcessTimes function. [Электронный ресурс] - URL:https://learn.microsoft.com/ru-ru/cpp/standard-library/processthreadsapi/chrono

   (дата обращения: 20.09.2024).

3. AlgoLib. [Электронный ресурс] - URL: https://www.algolib.narod/Math/Matrix

   (дата обращения: 20.09.2024).