Федеральное государственное бюджетное образовательное учреждение «Сибирский государственный университет телекоммуникаций и информатики»

Кафедра ПМиК

Расчётно-графическое задание

по дисциплине «Теория сложности вычислительных процессов и структур»

Вариант 4

Выполнил: студент 3 курса

ИВТ, гр. ИП-813

Бурдуковский Илья Александрович

Проверил:

ст. п. Разинкина Татьяна Эдуардовна

Новосибирск 2020

Содержание

[Постановка задачи 2](#_Toc28037657)

[Применяемые методы и их характеристики 3](#_Toc28037658)

[Обычный алгоритм умножения матриц 3](#_Toc28037659)

[Быстрый алгоритм умножения матриц 3](#_Toc28037660)

[Программная реализация 5](#_Toc28037661)

[Результаты работы программы 6](#_Toc28037662)

[Листинг 7](#_Toc28037663)

# Постановка задачи

Написать 2 программы, вычисляющие произведение двух матриц

A[i,j]=(-1)\*\*(i+j), B[i,j]=i-j, i,j=1..100.

Использовать алгоритмы быстрого и обычного умножения. Сравнить трудоемкости двух алгоритмов.

# Применяемые методы и их характеристики

## Обычный алгоритм умножения матриц

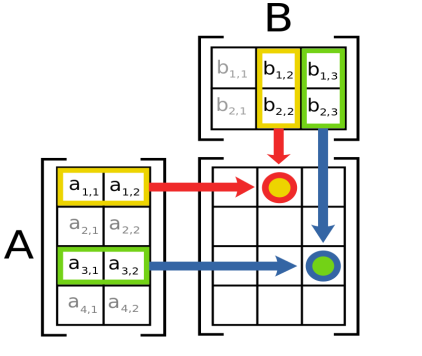
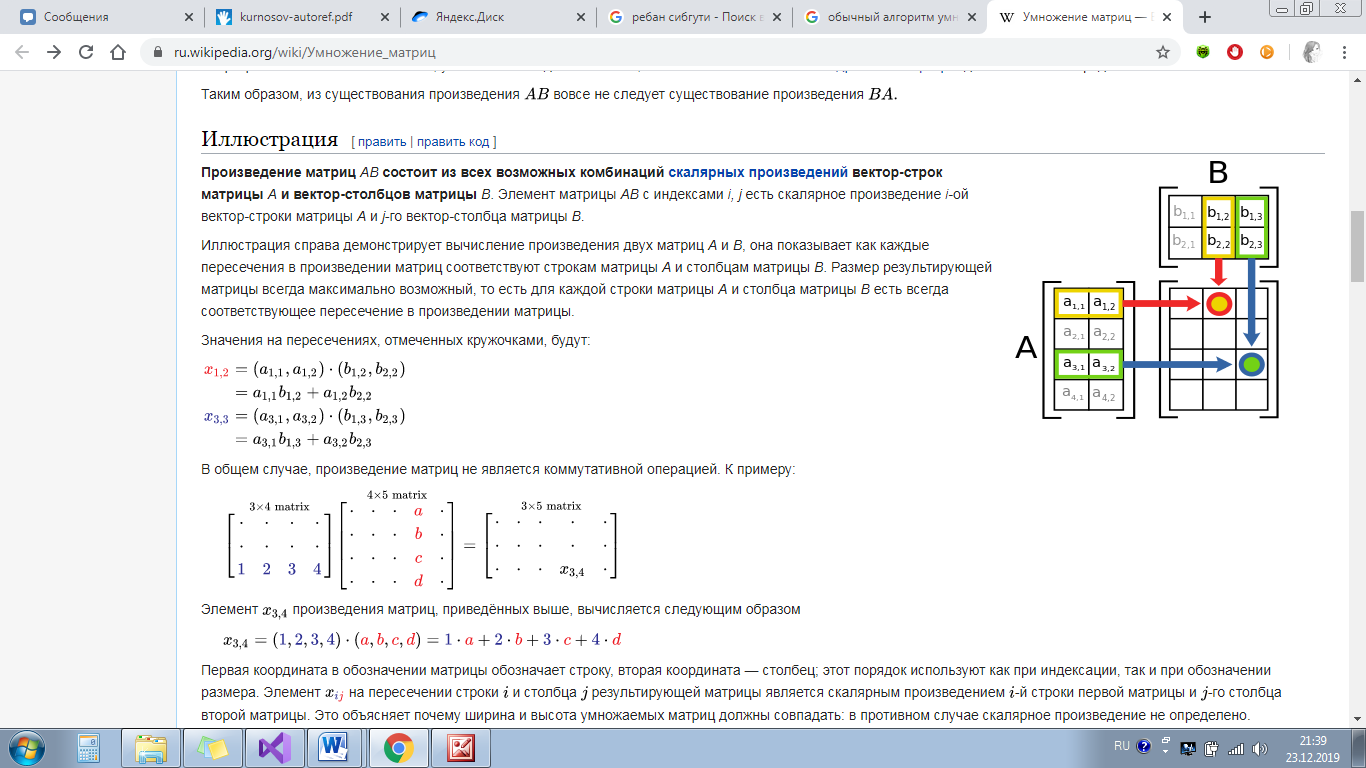
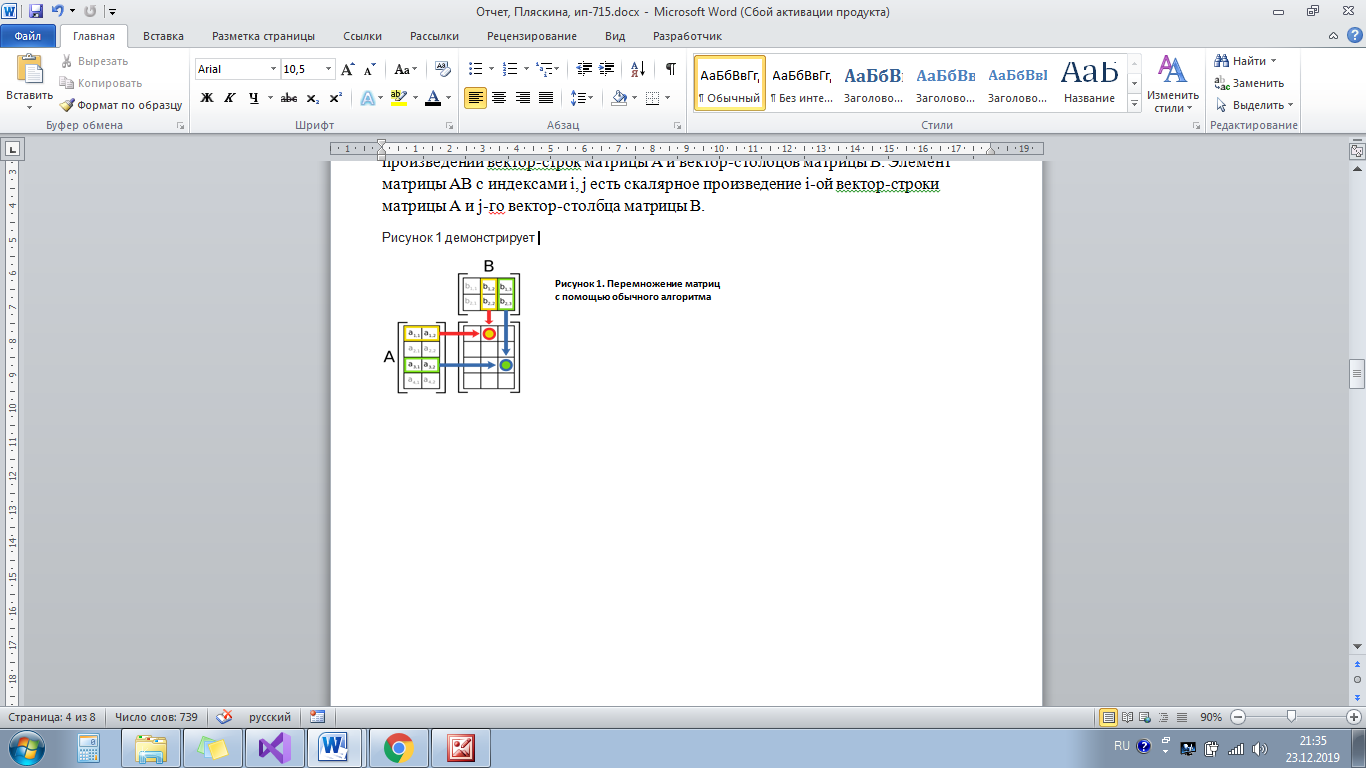
Произведение матриц AB состоит из всех возможных комбинаций [скалярных произведений](https://ru.wikipedia.org/wiki/%D0%A1%D0%BA%D0%B0%D0%BB%D1%8F%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5) вектор-строк матрицы A и вектор-столбцов матрицы B. Элемент матрицы AB с индексами i, j есть скалярное произведение i-ой вектор-строки матрицы A и j-го вектор-столбца матрицы B.

Рисунок 1 демонстрирует вычисление произведения двух матриц *A* и *B*, она показывает как каждые пересечения в произведении матриц соответствуют строкам матрицы *A* и столбцам матрицы *B*. Размер результирующей матрицы всегда максимально возможный, то есть для каждой строки матрицы *A* и столбца матрицы *B* есть всегда соответствующее пересечение в произведении матрицы.

Значения на пересечениях, отмеченных кружочками, будут:



Первая координата в обозначении матрицы обозначает строку, вторая координата — столбец; этот порядок используют как при индексации, так и при обозначении размера. Элемент xi, j на пересечении строки i и столбца j результирующей матрицы является скалярным произведением i-й строки первой матрицы и j-го столбца второй матрицы. Это объясняет почему ширина и высота умножаемых матриц должны совпадать: в противном случае скалярное произведение не определено.

## Быстрый алгоритм умножения матриц

Оценим трудоемкость обычного умножения двух матриц *n×n*. Трудоемкость будет иметь порядок *n3,*  т.к. в матрице-результате перемножения будет *n×n = n2* элементов и каждый из них вычисляется за *n* операций попарного умножения и сложения.

Попробуем применить ту же идею, что и быстрого умножения чисел, при перемножении двух матриц *n×n*. Разделим каждую из них на 4 матрицы вдвое меньшего размера:

Итак, при применении обычных формул блочного произведения матриц получаем рекуррентную формулу:

трудоемкость перемножения матриц вдвое меньшего размера.

трудоемкость сложений.

Соответственно получаем *T(n) =* , то есть трудоемкость такая же, как и при обычном умножении.

Однако существуют формулы Штрассена для блочного умножения матриц. В этих формулах будет не 8, а 7 попарных умножений матриц размера .

Формула Штрассена:

*M1 = (A2 – A4)(B3 + B4)*

*M2 = (A1 + A4)(B1 + B4)*

*M3 = (A1 – A3)(B1 + B2)*

*M4 = (A1 + A2)B4*

*M5 = A1(B2 – B4)*

***(2.11)***

*M6 = A4 (B3 – B1)*

*M7 = (A3 + A4)B1*

*C1 = M1 +M2 – M4 + M6*

*C2 = M4 + M5*

*C3 = M6 +M7*

*C4 = M2 – M3 + M5 – M7*

7 умножений и 18 сложений и вычитаний в этих формулах.

Получаем

# Программная реализация

Для реализации работы программы были разработаны следующие функции:

**void Matrix\_Initialization(long int\*\* A, long int\*\* B, int n) -**инициализирует матрицы A и B размером n A[i,j]=(-1)\*\*(i+j), B[i,j]=i-j, i,j=1..n.

**long int\*\* Simple\_multiplication\_matrix(long int\*\* A, long int\*\* B, int n)** – возвращает результат умножения матриц A и B размерностью n с помощью обычного алгоритма.

**long int\*\* add(long int\*\* A, long int\*\* B, int n, int k)** – возвращает результат сложения матриц А и B, размерностью n, если k=1, или вычитания B из A, если к=-1.

**long int\*\* Fast\_multiplication\_matrix(long int\*\*A, long int\*\*B, int n)** – возвращает результат умножения матриц A и B размерностью n с помощью быстрого алгоритма.

**void Print\_matrix(long int\*\* matrix, int n)** – выводит матрицу matrix с размерностью n

# Листинг

#include <iostream>

#include <iomanip>

#include <math.h>

using namespace std;

int oper\_add = 0;

int oper\_mult = 0;

int n;

void Matrix\_Initialization(long int\*\*, long int\*\*, int);

long int\*\* Simple\_multiplication\_matrix(long int\*\*, long int\*\*, int);

long int\*\* add(long int\*\*, long int\*\*, int, int);

long int\*\* Fast\_multiplication\_matrix(long int\*\*, long int\*\*, int);

void Print\_matrix(long int\*\*, int);

int main()

{

setlocale(LC\_ALL, "Russian");

cout << "Размер матриц: ";

cin >> n;

long int\*\* A = new long int\* [n];

long int\*\* B = new long int\* [n];

long int\*\* C = new long int\* [n];

Matrix\_Initialization(A, B, n);

cout << "\nМатрица A:" << endl;

Print\_matrix(A, n);

cout << "\nМатрица B:" << endl;

Print\_matrix(B, n);

cout << endl;

cout << "Обычное умножение:";

C = Simple\_multiplication\_matrix(A, B, n);

cout << "\nМатрица C:" << endl;

Print\_matrix(C, n);

cout << "Трудоемкость: " << oper\_add + oper\_mult << "\nСложений: " << oper\_add << "\nУмножений: " << oper\_mult << endl << endl;

oper\_add = 0;

oper\_mult = 0;

cout << "Быстрое умножение:";

C = Fast\_multiplication\_matrix(A, B, n);

cout << "\nМатрица C:" << endl;

Print\_matrix(C, n);

cout << "Трудоемкость: " << oper\_add + oper\_mult << "\nСложений: " << oper\_add << "\nУмножений: " << oper\_mult << endl << endl;

system("pause");

return 0;

}

void Matrix\_Initialization(long int\*\* A, long int\*\* B, int n) {

for (int i = 0; i < n; i++) {

A[i] = new long int[n];

B[i] = new long int[n];

}

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

A[i][j] = pow(-1,i + j);

B[i][j] = i - j;

}

}

}

long int\*\* Simple\_multiplication\_matrix(long int\*\* A, long int\*\* B, int n) {

long int\*\* C = new long int\* [n];

for (int i = 0; i < n; i++) {

C[i] = new long int[n] {};

for (int j = 0; j < n; j++)

C[i][j] = 0;

}

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

for (int k = 0; k < n; k++) {

C[i][j] = C[i][j] + A[i][k] \* B[k][j];

oper\_add++;

oper\_mult++;

}

}

}

return C;

}

long int\*\* add(long int\*\* A, long int\*\* B, int n, int k) {

long int\*\* C = new long int\* [n];

for (int i = 0; i < n; i++) {

C[i] = new long int[n] {};

for (int j = 0; j < n; j++) {

C[i][j] = A[i][j] + k \* B[i][j];

oper\_add++;

}

}

return C;

}

long int\*\* Fast\_multiplication\_matrix(long int\*\*A, long int\*\*B, int n) {

int h = n / 2;

long int\*\* C = new long\* [n];

for (int i = 0; i < n; i++) {

C[i] = new long int[n] {};

}

if(n % 2 ==1 ){

return Simple\_multiplication\_matrix(A,B,n);

}

else {

/\*

cout << endl << endl;

Print\_matrix(A, n);

cout << endl << endl;

Print\_matrix(B, n);

cout << endl << endl;\*/

long int\*\*\* Ak = new long int\*\* [4];

long int\*\*\* Bk = new long int\*\* [4];

long int\*\*\* Ck = new long int\*\* [4];

long int\*\*\* M = new long int\*\* [7];

long int\*\* rA = new long int\* [n];

long int\*\* rB = new long int\* [n];

for (int i = 0; i < n; i++) {

rA[i] = A[i] + h;

rB[i] = B[i] + h;

}

Ak[0] = A;

Ak[1] = rA;

Ak[2] = A + h;

Ak[3] = rA + h;

Bk[0] = B;

Bk[1] = rB;

Bk[2] = B + h;

Bk[3] = rB + h;

M[0] = Fast\_multiplication\_matrix(add(Ak[1], Ak[3], h, -1), add(Bk[2], Bk[3], h, 1), h);

M[1] = Fast\_multiplication\_matrix(add(Ak[0], Ak[3], h, 1), add(Bk[0], Bk[3], h, 1), h);

M[2] = Fast\_multiplication\_matrix(add(Ak[0], Ak[2], h, -1), add(Bk[0], Bk[1], h, 1), h);

M[3] = Fast\_multiplication\_matrix(add(Ak[0], Ak[1], h, 1), Bk[3], h);

M[4] = Fast\_multiplication\_matrix(Ak[0], add(Bk[1], Bk[3], h, -1), h);

M[5] = Fast\_multiplication\_matrix(Ak[3], add(Bk[2], Bk[0], h, -1), h);

M[6] = Fast\_multiplication\_matrix(add(Ak[2], Ak[3], h, 1), Bk[0], h);

Ck[0] = add(add(add(M[0], M[1], h, 1), M[3], h, -1), M[5], h, 1);

Ck[1] = add(M[3], M[4], h, 1);

Ck[2] = add(M[5], M[6], h, 1);

Ck[3] = add(add(add(M[1], M[2], h, -1), M[4], h, 1), M[6], h, -1);

for (int i = 0; i < h; i++) {

for (int j = 0; j < h; j++) {

C[i][j] = Ck[0][i][j];

C[i][j + h] = Ck[1][i][j];

C[i + h][j] = Ck[2][i][j];

C[i + h][j + h] = Ck[3][i][j];

}

}

return C;

}

}

void Print\_matrix(long int\*\* matrix, int n) {

int otstup = 3;

if (n <= 15) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

cout << setw(otstup) << matrix[i][j] << " ";

}

cout << endl;

}

}

else {

int col = 5;

for (int i = 0; i < col; i++) {

for (int j = 0; j < col; j++) {

cout << setw(otstup) << matrix[i][j] << " ";

}

cout << setw(otstup) << "..." << " ";

cout << setw(otstup) << matrix[i][n - 1] << endl;

}

for (int j = 0; j < col + 2; j++) {

cout << setw(otstup) << "..." << " ";

} cout << endl;

for (int j = 0; j < col; j++) {

cout << setw(otstup) << matrix[n - 1][j] << " ";

}

cout << setw(otstup) << "..." << " ";

cout << setw(otstup) << matrix[n - 1][n - 1] << endl;

}

}

# Результаты работы программы

