```
#ifndef CBIOINFCPP_H
#define CBIOINFCPP_H

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <set>
#include <algorithm>
#include <queue>
#include <map>
#include <cmath>
#include <stack>
#include <limits.h>
#include <float.h>


int FastaRead (std::ifstream & fin, std::vector <std::string> & IndexS,
std::vector <std::string> & DataS)

// Чтение строк FASTA из файла. Возвращает 0, если кол-во индексов строк равно кол-ву самих строк, самая первая строка является индексом (не начинается с ">") и в процессе считывания не встретились 2 индекса подряд. Иначе вернет -1.

// Reads FASTA dataset from file. Returns 0 if the number of indexes of strings = number of strings, the first string in dataset is index (starts with ">") and in dataset there is no 2 indexes one-by-one without a data string in between. Otherwise returns -1.


{
IndexS.clear();
// Сюда будут записываться индексы (обозначения) строк
// Here indexes of strings will be contained

DataS.clear();
// Сюда будут записываться сами строки
// Here data strings will be contained

std::string TempS = "";

int q = -1;

char f = 'd';
// флаг f = 'd'(data) или 'i' (индекс) - для исключения ситуации когда подряд идут 2 индекса
```

```
// flag f may be set as 'd' (data) or 'i' (index) - to prevent code-after-code
situation.


while (!fin.eof())
    {
       TempS.clear();
       getline (fin, TempS);
       if (TempS[0] == '>')
        {
           if (f == 'i') {IndexS.clear(); DataS.clear(); return -1;}
           q++;
           f = 'i';
           DataS.push_back("");
           TempS.erase(0,1); // Вырезание начального символа ">"  cutting symbol
">" from string's code
           IndexS.push_back(TempS);
        }
        else
        {
            if (q==-1) return -1;
            if (TempS.length() !=0) {DataS[q] = DataS[q] + TempS; f = 'd';};
        }

   }

        if (DataS.size()!=IndexS.size()) {IndexS.clear(); DataS.clear(); return -
1;}

return 0;
}

void StringsRead (std::ifstream & fin, std::vector <std::string> & DataS) // reads
all strings from file to vector DataS

{
   std::string TempS = "";
    while (!fin.eof())
    {
        getline (fin, TempS);
        if (TempS.length()!=0) DataS.push_back(TempS);
        TempS.clear();
}

}

int MatrixSet (std::vector <std::vector <double>> & B, const int NLines, const int
NColumns, const double i)
// Создает матрицу NLines x NColumns и заполняет значением i (double). Возвращает
-1 если число строк или столбцов неположительно
// Sets (resets) matrix NLines x NColumns filled value "i" (double). Returns -1 if
NLines or NColumns <=0

{

    if ((NLines<=0) || (NColumns<=0)) return -1;

    B.resize (NLines);
    for (unsigned int row = 0; (row< NLines); row++)
    {
        B [row].resize(NColumns);
        for (unsigned int column = 0; column < NColumns; column++)
            {
                B [row] [column] = i;
```

```
            }

    }
    return 0;
}


int MatrixSet (std::vector <std::vector <int>> & B, const int NLines, const int
NColumns, const int i)
// Создает матрицу NLines x NColumns и заполняет значением i (int). Возвращает -1
если число строк или столбцов неположительно
// Sets (resets) matrix NLines x NColumns filled value "i" (int). Returns -1 if
NLines or NColumns <=0

{

    if ((NLines<=0) || (NColumns<=0)) return -1;

    B.resize (NLines);
    for (unsigned int row = 0; (row< NLines); row++)
    {
        B [row].resize(NColumns);
        for (unsigned int column = 0; column < NColumns; column++)
            {
                B [row] [column] = i;
            }
    }
    return 0;
}



int MatrixSet (std::vector <std::vector <long long int>> & B, const int NLines,
const int NColumns, const long long int i)
// Создает матрицу NLines x NColumns и заполняет значением i (long long int).
Возвращает -1 если число строк или столбцов неположительно
// Sets (resets) matrix NLines x NColumns filled value "i" (long long int).
Returns -1 if NLines or NColumns <=0

{

    if ((NLines<=0) || (NColumns<=0)) return -1;

    B.resize (NLines);
    for (unsigned int row = 0; (row< NLines); row++)
    {
        B [row].resize(NColumns);
        for (unsigned int column = 0; column < NColumns; column++)
            {
                B [row] [column] = i;
            }
    }
    return 0;
}



int MatrixSet (std::vector <std::vector <long double>> & B, const int NLines,
const int NColumns, const long double i)
// Создает матрицу NLines x NColumns и заполняет значением i (long double).
Возвращает -1 если число строк или столбцов неположительно
// Sets (resets) matrix NLines x NColumns filled value "i" (long double). Returns
-1 if NLines or NColumns <=0
```

```
{

    if ((NLines<=0) || (NColumns<=0)) return -1;

    B.resize (NLines);
    for (unsigned int row = 0; (row< NLines); row++)
    {
        B [row].resize(NColumns);
        for (unsigned int column = 0; column < NColumns; column++)
            {
                B [row] [column] = i;
            }

    }
    return 0;
}



int MatrixCout (std::vector <std::vector <int>> & B, char g = ' ')
// Вывод матрицы (int) на экран через пробелы. Возвращает -1, если матрица не
содержит строк/ столбцов
// Если строки матрицы разной длины, то "остатки" длины до максимальной при выводе
заполняются символом g
// "Couts" Matrix (int) to screen. Returns -1 if the Matrix is empty.
// The Matrix may have lines of different length. In this case the "missing"
values to the end for the "shorter lines" are filled with the char g.

{

    int NLines, NColumns;
    NLines = B.size();
    if (NLines==0) return -1;
    NColumns = B[0].size();

    for (int i=0;i<NLines; i++)
        if (B[i].size()>NColumns) NColumns=B[i].size();

    if (NColumns==0) return -1;



    for (unsigned int row = 0; (row< NLines); row++)
    {

        //for (unsigned int column = 0; column < NColumns; column++)
        for (unsigned int column = 0; column < B[row].size(); column++)
            {
                std::cout<< B [row] [column]<< " ";
            }

        for (unsigned int column = B[row].size(); column < NColumns; column++)
            {
                std::cout<< g << " ";
            }


        std::cout<< std::endl;
    }
    return 0;
}
```

```
int MatrixCout (std::vector <std::vector <long long int>> & B, char g = ' ')
// Вывод матрицы (long long int) на экран через пробелы. Возвращает -1, если
матрица не содержит строк/ столбцов
// Если строки матрицы разной длины, то "остатки" длины до максимальной при выводе
заполняются символом g
// "Couts" Matrix (long long int) to screen. Returns -1 if the Matrix is empty.
// The Matrix may have lines of different length. In this case the "missing"
values to the end for the "shorter lines" are filled with the char g.

{

    int NLines, NColumns;
    NLines = B.size();
    if (NLines==0) return -1;
    NColumns = B[0].size();

    for (int i=0;i<NLines; i++)
        if (B[i].size()>NColumns) NColumns=B[i].size();

    if (NColumns==0) return -1;



    for (unsigned int row = 0; (row< NLines); row++)
    {

        //for (unsigned int column = 0; column < NColumns; column++)
        for (unsigned int column = 0; column < B[row].size(); column++)
            {
                std::cout<< B [row] [column]<< " ";
            }

        for (unsigned int column = B[row].size(); column < NColumns; column++)
            {
                std::cout<< g << " ";
            }


        std::cout<< std::endl;
    }
    return 0;
}


int MatrixCout (std::vector <std::vector <double>> & B, unsigned int prec = 4,
char g = ' ', bool scientifique = false)
// Вывод матрицы (double) на экран через пробелы. Возвращает -1, если матрица не
содержит строк/ столбцов
// Если строки матрицы разной длины, то "остатки" длины до максимальной при выводе
заполняются символом g
// Вывод чисел проводится с заданной точностью prec, если bool scientifique ==
false. Если bool scientifique == true, вывод производится в экспоненциальной
форме.
// "Couts" Matrix (double) to screen. Returns -1 if the Matrix is empty.
// The Matrix may have lines of different length. In this case the "missing"
values to the end for the "shorter lines" are filled with the char g.
// If bool scientifique == false, the precision will be set as prec; if bool
scientifique == true, scientific notation will be applied.
{

    int NLines, NColumns;
    NLines = B.size();
```

```
    if (NLines==0) return -1;
    NColumns = B[0].size();

    for (int i=0;i<NLines; i++)
        if (B[i].size()>NColumns) NColumns=B[i].size();

    if (NColumns==0) return -1;


    if (!scientifique)
   {
    std::cout.precision(prec);

    for (unsigned int row = 0; (row< NLines); row++)
    {

        //for (unsigned int column = 0; column < NColumns; column++)
        for (unsigned int column = 0; column < B[row].size(); column++)
            {
                std::cout<< std::fixed<< B [row] [column]<< " ";
            }

        for (unsigned int column = B[row].size(); column < NColumns; column++)
            {
                std::cout<< g << " ";
            }


        std::cout<< std::endl;
    }

    }


    if (scientifique)
    {


        for (unsigned int row = 0; (row< NLines); row++)
        {

            //for (unsigned int column = 0; column < NColumns; column++)
            for (unsigned int column = 0; column < B[row].size(); column++)
                {
                    std::cout<< std::scientific << B [row] [column]<< " ";
                }

            std::cout.unsetf(std::ios::scientific);

            for (unsigned int column = B[row].size(); column < NColumns; column++)
                {
                    std::cout<< g << " ";
                }


            std::cout<< std::endl;
        }

    }


    return 0;
}
```

```
int MatrixCout (std::vector <std::vector <long double>> & B, unsigned int prec =
4, char g = ' ', bool scientifique = false)
// Вывод матрицы (long double) на экран через пробелы. Возвращает -1, если матрица
не содержит строк/ столбцов
// Если строки матрицы разной длины, то "остатки" длины до максимальной при выводе
заполняются символом g
// Вывод чисел проводится с заданной точностью prec, если bool scientifique ==
false. Если bool scientifique == true, вывод производится в экспоненциальной
форме.
// "Couts" Matrix (long double) to screen. Returns -1 if the Matrix is empty.
// The Matrix may have lines of different length. In this case the "missing"
values to the end for the "shorter lines" are filled with the char g.
// If bool scientifique == false, the precision will be set as prec; if bool
scientifique == true, scientific notation will be applied.
{

    int NLines, NColumns;
    NLines = B.size();
    if (NLines==0) return -1;
    NColumns = B[0].size();

    for (int i=0;i<NLines; i++)
        if (B[i].size()>NColumns) NColumns=B[i].size();

    if (NColumns==0) return -1;


    if (!scientifique)
   {
    std::cout.precision(prec);

    for (unsigned int row = 0; (row< NLines); row++)
    {

        //for (unsigned int column = 0; column < NColumns; column++)
        for (unsigned int column = 0; column < B[row].size(); column++)
            {
                std::cout<< std::fixed<< B [row] [column]<< " ";
            }

        for (unsigned int column = B[row].size(); column < NColumns; column++)
            {
                std::cout<< g << " ";
            }


        std::cout<< std::endl;
    }

    }


    if (scientifique)
    {


        for (unsigned int row = 0; (row< NLines); row++)
        {

            //for (unsigned int column = 0; column < NColumns; column++)
            for (unsigned int column = 0; column < B[row].size(); column++)
                {
                    std::cout<< std::scientific<< B [row] [column]<< " ";
```

```
                }

            std::cout.unsetf(std::ios::scientific);

            for (unsigned int column = B[row].size(); column < NColumns; column++)
                {
                    std::cout<< g << " ";
                }


            std::cout<< std::endl;
        }

    }


    return 0;
}


int MatrixFout (std::vector <std::vector <double>> & B, std::ofstream & fout,
unsigned int prec = 4, char g = ' ', bool scientifique = false)
// Вывод матрицы (double) в файл через пробелы. Возвращает -1, если матрица не
содержит строк/ столбцов
// Если строки матрицы разной длины, то "остатки" длины до максимальной при выводе
заполняются символом g
// Вывод чисел проводится с заданной точностью prec, если bool scientifique ==
false. Если bool scientifique == true, вывод производится в экспоненциальной
форме.
// "Fouts" Matrix (double) to file. Returns -1 if the Matrix is empty.
// The Matrix may have lines of different length. In this case the "missing"
values to the end for the "shorter lines" are filled with the char g.
// If bool scientifique == false, the precision will be set as prec; if bool
scientifique == true, scientific notation will be applied.
{

    int NLines, NColumns;
    NLines = B.size();
    if (NLines==0) return -1;
    NColumns = B[0].size();

    for (int i=0;i<NLines; i++)
        if (B[i].size()>NColumns) NColumns=B[i].size();

    if (NColumns==0) return -1;


    if (!scientifique)
    {
    fout.precision(prec);

    for (unsigned int row = 0; (row< NLines); row++)
    {

        //for (unsigned int column = 0; column < NColumns; column++)
        for (unsigned int column = 0; column < B[row].size(); column++)
            {
                fout<< std::fixed<< B [row] [column]<< " ";
            }

        for (unsigned int column = B[row].size(); column < NColumns; column++)
            {
                fout<< g << " ";
            }
```

```
        fout<< std::endl;
    }
    }


    if (scientifique)
    {


    for (unsigned int row = 0; (row< NLines); row++)
    {

        //for (unsigned int column = 0; column < NColumns; column++)
        for (unsigned int column = 0; column < B[row].size(); column++)
            {
                fout<< std::scientific<< B [row] [column]<< " ";
            }

        fout.unsetf(std::ios::scientific);
        for (unsigned int column = B[row].size(); column < NColumns; column++)
            {
                fout<< g << " ";
            }


        fout<< std::endl;
    }


    }



    return 0;
}


int MatrixFout (std::vector <std::vector <long double>> & B, std::ofstream & fout,
unsigned int prec = 4, char g = ' ', bool scientifique = false)
// Вывод матрицы (long double) в файл через пробелы. Возвращает -1, если матрица
не содержит строк/ столбцов
// Если строки матрицы разной длины, то "остатки" длины до максимальной при выводе
заполняются символом g
// Вывод чисел проводится с заданной точностью prec, если bool scientifique ==
false. Если bool scientifique == true, вывод производится в экспоненциальной
форме.
// "Fouts" Matrix (long double) to file. Returns -1 if the Matrix is empty.
// The Matrix may have lines of different length. In this case the "missing"
values to the end for the "shorter lines" are filled with the char g.
// If bool scientifique == false, the precision will be set as prec; if bool
scientifique == true, scientific notation will be applied.
{

    int NLines, NColumns;
    NLines = B.size();
    if (NLines==0) return -1;
    NColumns = B[0].size();

    for (int i=0;i<NLines; i++)
        if (B[i].size()>NColumns) NColumns=B[i].size();

    if (NColumns==0) return -1;
```

```
    if (!scientifique)
    {
    fout.precision(prec);

    for (unsigned int row = 0; (row< NLines); row++)
    {

        //for (unsigned int column = 0; column < NColumns; column++)
        for (unsigned int column = 0; column < B[row].size(); column++)
            {
                fout<< std::fixed<< B [row] [column]<< " ";
            }

        for (unsigned int column = B[row].size(); column < NColumns; column++)
            {
                fout<< g << " ";
            }


        fout<< std::endl;
    }
    }


    if (scientifique)
    {


    for (unsigned int row = 0; (row< NLines); row++)
    {

        //for (unsigned int column = 0; column < NColumns; column++)
        for (unsigned int column = 0; column < B[row].size(); column++)
            {
                fout<< std::scientific<< B [row] [column]<< " ";
            }

        fout.unsetf(std::ios::scientific);
        for (unsigned int column = B[row].size(); column < NColumns; column++)
            {
                fout<< g << " ";
            }


        fout<< std::endl;
    }


    }



    return 0;
}



int MatrixFout (std::vector <std::vector <int>> & B, std::ofstream & fout, char g
= ' ')
// Вывод матрицы (int) в файл через пробелы. Возвращает -1, если матрица не
содержит строк/ столбцов
```

```
// Если строки матрицы разной длины, то "остатки" длины до максимальной при выводе заполняются символом g
// "Fouts" Matrix (int) to file. Returns -1 if the Matrix is empty.
// The Matrix may have lines of different length. In this case the "missing" values to the end for the "shorter lines" are filled with the char g.
{

    int NLines, NColumns;
    NLines = B.size();
    if (NLines==0) return -1;
    NColumns = B[0].size();

    for (int i=0;i<NLines; i++)
        if (B[i].size()>NColumns) NColumns=B[i].size();

    if (NColumns==0) return -1;


    for (unsigned int row = 0; (row< NLines); row++)
    {

        //for (unsigned int column = 0; column < NColumns; column++)
        for (unsigned int column = 0; column < B[row].size(); column++)
            {
                fout<< B [row] [column]<< " ";
            }

        for (unsigned int column = B[row].size(); column < NColumns; column++)
            {
                fout<< g << " ";
            }


        fout<< std::endl;
    }
    return 0;
}



int MatrixFout (std::vector <std::vector <long long int>> & B, std::ofstream & fout, char g = ' ')
// Вывод матрицы (long long int) в файл через пробелы. Возвращает -1, если матрица не содержит строк/ столбцов
// Если строки матрицы разной длины, то "остатки" длины до максимальной при выводе заполняются символом g
// "Fouts" Matrix (long long int) to file. Returns -1 if the Matrix is empty.
// The Matrix may have lines of different length. In this case the "missing" values to the end for the "shorter lines" are filled with the char g.
{

    int NLines, NColumns;
    NLines = B.size();
    if (NLines==0) return -1;
    NColumns = B[0].size();

    for (int i=0;i<NLines; i++)
        if (B[i].size()>NColumns) NColumns=B[i].size();

    if (NColumns==0) return -1;


    for (unsigned int row = 0; (row< NLines); row++)
    {
```

```
        //for (unsigned int column = 0; column < NColumns; column++)
        for (unsigned int column = 0; column < B[row].size(); column++)
            {
                fout<< B [row] [column]<< " ";
            }

        for (unsigned int column = B[row].size(); column < NColumns; column++)
            {
                fout<< g << " ";
            }


        fout<< std::endl;
    }
    return 0;
}


int FindIn (std::vector <int> &D, int a, int step = 1, int start = 0)
{
// Возвращает индекс первого найденного элемента (int), совпадающего с искомым
(a), поиск ведется с позиции start, шаг поиска = step, если не нашли такого
элемента - возвращаем -1. Если переданное значение step <1, то step присваивается
значение 1. Если переданное значение start <0, то start присваивается значение 1.
// Returns index in vector (int) of the first element = a. Search starts from
index "start" with step = "step". If no such element found the function returns 0.
If step<1 step will be set as 1. If start<0 start will be set as 0.


    if (step <1) step = 1; if (start <0) start = 0;
    for (unsigned int y = start; y<D.size(); y=y+step)
    {
        if (D[y] == a) return y;
    }
    return -1;
}


int FindIn (std::vector <long long int> &D, long long int a, int step = 1, int
start = 0)
{
// Возвращает индекс первого найденного элемента (long long int), совпадающего с
искомым (a), поиск ведется с позиции start, шаг поиска = step, если не нашли
такого элемента - возвращаем -1. Если переданное значение step <1, то step
присваивается значение 1. Если переданное значение start <0, то start
присваивается значение 1.
// Returns index in vector (long long int) of the first element = a. Search starts
from index "start" with step = "step". If no such element found the function
returns 0. If step<1 step will be set as 1. If start<0 start will be set as 0.

    if (step <1) step = 1; if (start <0) start = 0;
    for (unsigned int y = start; y<D.size(); y=y+step)
    {
        if (D[y] == a) return y;
    }
    return -1;
}


int FindIn (std::vector <double> &D, double a, int step = 1, int start = 0)
{
// Возвращает индекс первого найденного элемента (double), совпадающего с искомым
(a), поиск ведется с позиции start, шаг поиска = step, если не нашли такого
```

```
элемента - возвращаем -1. Если переданное значение step <1, то step присваивается
значение 1. Если переданное значение start <0, то start присваивается значение 1.
// Да, прямое сравнение чисел double не совсем корректно и это нужно принимать во
внимание, но в ряде случаев функция может быть полезна.
// Для сравнения с заданной точностью см. вариант функции ниже.
// Returns index in vector (double) of the first element = a. Search starts from
index "start" with step = "step". If no such element found the function returns 0.
If step<1 step will be set as 1. If start<0 start will be set as 0.
// Yes, operation like (a==b) may be not correct for doubles. But this function
may be considered as an useful one in some cases.
//  The following version of the function finds the first element, that differs
from "a" less than "d".

    if (step <1) step = 1; if (start <0) start = 0;
    for (unsigned int y = start; y<D.size(); y=y+step)
    {
        if ((D[y]-a)==0.0) return y;
    }
    return -1;
}



int FindIn (std::vector <double> &D, double a, double d, int step = 1, int start =
0)
{
// Возвращает индекс первого найденного элемента (double), совпадающего с искомым
(a) с точностью до d, поиск ведется с позиции start, шаг поиска = step, если не
нашли такого элемента - возвращает -1. Если переданное значение step <1, то step
присваивается значение 1. Если переданное значение start <0, то start
присваивается значение 1.
// Returns index in vector (double) of the first element, that differs from "a"
less than nonnegative double "d".
// Search starts from index "start" with step = "step". If no such element found
the function returns 0. If step<1 step will be set as 1. If start<0 start will be
set as 0.

    if (step <1) step = 1; if (start <0) start = 0;
    for (unsigned int y = start; y<D.size(); y=y+step)
    {
        if ( (std::abs(D[y] - a))<d) return y;
    }
    return -1;
}


int FindIn (std::vector <long double> &D, long double a, int step = 1, int start =
0)
{
// Возвращает индекс первого найденного элемента (long double), совпадающего с
искомым (a), поиск ведется с позиции start, шаг поиска = step, если не нашли
такого элемента - возвращает -1. Если переданное значение step <1, то step
присваивается значение 1. Если переданное значение start <0, то start
присваивается значение 1.
// Да, прямое сравнение чисел long double не совсем корректно и это нужно
принимать во внимание, но в ряде случаев функция может быть полезна. Для сравнения
с заданной точностью см. вариант функции ниже.
// Returns index in vector (long double) of the first element = a. Search starts
from index "start" with step = "step". If no such element found the function
returns 0. If step<1 step will be set as 1. If start<0 start will be set as 0.
// Yes, operation like (a==b) may be not correct for doubles. But this function
may be considered as an useful one in some cases. The following version of the
function finds the first element, that differs from "a" less than "d".
```

```
    if (step <1) step = 1; if (start <0) start = 0;
    for (unsigned int y = start; y<D.size(); y=y+step)
    {
        if ((D[y]-a)==0.0) return y;
    }
    return -1;
}


int FindIn (std::vector <long double> &D, long double a, long double d, int step =
1, int start = 0)
{
// Возвращает индекс первого найденного элемента (long double), совпадающего с
искомым (a) с точностью до d, поиск ведется с позиции start, шаг поиска = step,
если не нашли такого элемента - возвращаем -1. Если переданное значение step <1,
то step присваивается значение 1. Если переданное значение start <0, то start
присваивается значение 1.
// Returns index in vector (long double) of the first element, that differs from
"a" less than nonnegative long double "d".
// Search starts from index "start" with step = "step". If no such element found
the function returns 0. If step<1 step will be set as 1. If start<0 start will be
set as 0.

    if (step <1) step = 1; if (start <0) start = 0;
    for (unsigned int y = start; y<D.size(); y=y+step)
    {
        if ( (std::abs(D[y] - a))<d) return y;
    }
    return -1;
}



int FindIn (std::vector <std::string> &D, std::string a, int step = 1, int start =
0)
{
// Возвращает индекс первого найденного элемента (string), совпадающего с искомым
(a), поиск ведется с позиции start, шаг поиска = step, если не нашли такого
элемента - возвращаем -1. Если переданное значение step <1, то step присваивается
значение 1. Если переданное значение start <0, то start присваивается значение 1.
// Returns index in vector (string) of the first element = a. Search starts from
index "start" with step = "step". If no such element found the function returns 0.
If step<1 step will be set as 1. If start<0 start will be set as 0.

    if (step <1) step = 1; if (start <0) start = 0;
    for (unsigned int y = start; y<D.size(); y=y+step)
    {
        if (D[y] == a) return y;
    }
    return -1;
}


int VectorCout (const std::vector <int> &P)
// Вывод вектора (int) на экран через пробелы
// "Couts" vector (int) to screen. Returns -1 if the vector is empty

{

    if (P.size()==0) return -1;

    for (int i=0; i<P.size();i++)
        std::cout<< P[i]<<" ";
    std::cout<< std::endl;
```

```
    return 0;
}

int VectorFout (const std::vector <int> &P, std::ofstream &fout)
// Вывод вектора (int) в файл через пробелы. Возвращает -1, если вектор - пустой
// "Fouts" vector (int) to file. Returns -1 if the vector is empty
{

     if (P.size()==0) return -1;

    for (int i=0; i<P.size();i++)
        fout<< P[i]<<" ";
    fout<< std::endl;
    return 0;
}



int VectorCout (const std::vector <long long int> &P)
// Вывод вектора (long long int) на экран через пробелы
// "Couts" vector (long long int) to screen. Returns -1 if the vector is empty

{

    if (P.size()==0) return -1;

    for (int i=0; i<P.size();i++)
        std::cout<< P[i]<<" ";
    std::cout<< std::endl;

    return 0;
}

int VectorFout (const std::vector <long long int> &P, std::ofstream &fout)
// Вывод вектора (long long int) в файл через пробелы. Возвращает -1, если вектор
- пустой
// "Fouts" vector (long long int) to file. Returns -1 if the vector is empty
{

     if (P.size()==0) return -1;

    for (int i=0; i<P.size();i++)
        fout<< P[i]<<" ";
    fout<< std::endl;
    return 0;
}



int VectorCout (const std::vector <double> &P, unsigned int prec = 4, bool
scientifique = false)
// Вывод вектора (double) на экран через пробелы. Возвращает -1, если вектор -
пустой
// Вывод чисел проводится с заданной точностью prec, если bool scientifique ==
false. Если bool scientifique == true, вывод производится в экспоненциальной
форме.
// "Couts" vector (double) to screen. Returns -1 if the vector is empty
// If bool scientifique == false, the precision will be set as prec; if bool
scientifique == true, scientific notation will be applied.
{
    if (P.size()==0) return -1;
```

```
    if (!scientifique)
    {

    std::cout.precision(prec);
    for (int i=0; i<P.size();i++)
        std::cout<< std::fixed<< P[i]<<" ";
    }

    if (scientifique)
    {


    for (int i=0; i<P.size();i++)
        std::cout<< std::scientific<< P[i]<<" ";

    std::cout.unsetf(std::ios::scientific);

    }


    std::cout<< std::endl;
    return 0;
}

int VectorFout (const std::vector <double> &P, std::ofstream &fout, unsigned int
prec = 4, bool scientifique = false)
// Вывод вектора (double) в файл через пробелы. Возвращает -1, если вектор -
пустой
// Вывод чисел проводится с заданной точностью prec, если bool scientifique ==
false. Если bool scientifique == true, вывод производится в экспоненциальной
форме.
// "Fouts" vector (double) to file. Returns -1 if the vector is empty
// If bool scientifique == false, the precision will be set as prec; if bool
scientifique == true, scientific notation will be applied.
{

    if (P.size()==0) return -1;

    if (!scientifique)
    {
    fout.precision(prec);
    for (int i=0; i<P.size();i++)
        fout<< std::fixed<< P[i]<<" ";
    }

    if (scientifique)
    {


    for (int i=0; i<P.size();i++)
        fout<< std::scientific<< P[i]<<" ";

    fout.unsetf(std::ios::scientific);
    }

    fout<< std::endl;
    return 0;
}




int VectorCout (const std::vector <long double> &P, unsigned int prec = 4, bool
scientifique = false)
```

```
// Вывод вектора (long double) на экран через пробелы. Возвращает -1, если вектор
- пустой
// Вывод чисел проводится с заданной точностью prec, если bool scientifique ==
false. Если bool scientifique == true, вывод производится в экспоненциальной
форме.
// "Couts" vector (long double) to screen. Returns -1 if the vector is empty
// If bool scientifique == false, the precision will be set as prec; if bool
scientifique == true, scientific notation will be applied.
{
    if (P.size()==0) return -1;

    if (!scientifique)
    {

    std::cout.precision(prec);
    for (int i=0; i<P.size();i++)
        std::cout<< std::fixed<< P[i]<<" ";
    }

    if (scientifique)
    {


    for (int i=0; i<P.size();i++)
        std::cout<< std::scientific<< P[i]<<" ";

    std::cout.unsetf(std::ios::scientific);
    }

    std::cout<< std::endl;
    return 0;
}

int VectorFout (const std::vector <long double> &P, std::ofstream &fout, unsigned
int prec = 4, bool scientifique = false)
// Вывод вектора (long double) в файл через пробелы. Возвращает -1, если вектор -
пустой
// Вывод чисел проводится с заданной точностью prec, если bool scientifique ==
false. Если bool scientifique == true, вывод производится в экспоненциальной
форме.
// "Fouts" vector (long double) to file. Returns -1 if the vector is empty
// If bool scientifique == false, the precision will be set as prec; if bool
scientifique == true, scientific notation will be applied.
{

    if (P.size()==0) return -1;

    if (!scientifique)
    {
    fout.precision(prec);
    for (int i=0; i<P.size();i++)
        fout<< std::fixed<< P[i]<<" ";
    }

    if (scientifique)
    {


    for (int i=0; i<P.size();i++)
        fout<< std::scientific<< P[i]<<" ";


    fout.unsetf(std::ios::scientific);
    }
```

```
    fout<< std::endl;
    return 0;
}



int VectorCout (const std::vector <std::string> & P)
// Вывод вектора (string) через Enter на экран. Возвращает -1, если вектор - пустой
// "Couts" vector (string) to screen. Returns -1 if the vector is empty
{
    if (P.size()==0) return -1;


    for (int i=0; i<P.size();i++)
        std::cout<< P[i]<<std::endl;
    std::cout<< std::endl;
    return 0;
}

int VectorFout (const std::vector <std::string> & P, std::ofstream &fout)
// Вывод вектора (string) через Enter в файл. Возвращает -1, если вектор - пустой
// "Fouts" vector (string) to file. Returns -1 if the vector is empty
{
    if (P.size()==0) return -1;


    for (int i=0; i<P.size();i++)
        fout<< P[i]<<std::endl;
    fout<< std::endl;
    return 0;
}



int PairVectorCout (const std::pair < std::vector<int>, std::vector<double>> & P, unsigned int prec = 4, bool scientifique = false)
// Модификация функции VectorCout (см. выше).
// Modification of the function VectorCout (see it above) for not-integer (double) weights of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an "Adjacency vector" without weights. But weights are set in the second one.
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the first vector has its weight set as i-th element in the second one.


{

    if ((P.first).size()==0) return -1;
    if ((P.second).size()==0) return -1;
    if (  (P.first).size()!=((P.second).size())*2 ) return -1;

    if (!scientifique)
    {
    std::cout.precision(prec);

    for (int i=0; i<(P.second).size();i++)
    {
        std::cout<< (P.first)[2*i]<<" ";
        std::cout<< (P.first)[2*i+1]<<" ";
        std::cout<< std::fixed<<(P.second)[i]<<" ";
```

```
    }
    std::cout<< std::endl;

    }

    if (scientifique)
        {

                for (int i=0; i<(P.second).size();i++)
                {
                    std::cout<< (P.first)[2*i]<<" ";
                    std::cout<< (P.first)[2*i+1]<<" ";
                    std::cout<< std::scientific<<(P.second)[i]<<" ";
                    std::cout.unsetf(std::ios::scientific);

                }


                std::cout<< std::endl;


        }



    return 0;
}

int PairVectorFout (const std::pair < std::vector<int>, std::vector<double>> & P,
std::ofstream &fout, unsigned int prec = 4, bool scientifique = false)
// Модификация функции VectorFout (см. выше).
// Modification of the function VectorFout (see it above) for not-integer (double)
weights of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an
"Adjacency vector" without weights. But weights are set in the second one.
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the
first vector has its weight set as i-th element in the second one.

{

    if ((P.first).size()==0) return -1;
    if ((P.second).size()==0) return -1;
    if (  (P.first).size()!=((P.second).size())*2 ) return -1;

    if (!scientifique)
    {
    fout.precision(prec);

    for (int i=0; i<(P.second).size();i++)
    {
        fout<< (P.first)[2*i]<<" ";
        fout<< (P.first)[2*i+1]<<" ";
        fout<< std::fixed<<(P.second)[i]<<" ";

    }
    fout<< std::endl;

    }

    if (scientifique)
        {

                for (int i=0; i<(P.second).size();i++)
```

```
                {
                    fout<< (P.first)[2*i]<<" ";
                    fout<< (P.first)[2*i+1]<<" ";
                    fout<< std::scientific<<(P.second)[i]<<" ";
                    fout.unsetf(std::ios::scientific);

                }


                fout<< std::endl;


        }



    return 0;
}



int GraphCout (const std::vector <int> &P, const bool weighted)
// Вывод графа, заданного вектором смежности, на экран: каждое ребро выводится в
новой строке. Граф - невзвешенный, либо веса ребер целочисленны.
// "Couts" a graph that is set by Adjacency vector A to screen: one edge in one
line. Parameter "weighted" sets if the graph A is weighted or no.
// Returns -1 if input data is not correct. Otherwise returns 0.

{

    if (P.size()==0) return -1; // checking for input data correctness
    if ( (P.size())%(2+weighted)!=0 ) return -1; // checking for input data
correctness

    for (int i=0; i<P.size();i=i+2)
        std::cout<< P[i]<<' '<<P[i+1]<<std::endl;
    std::cout<< std::endl;

    return 0;
}


int GraphFout (const std::vector <int> &P, const bool weighted, std::ofstream
&fout)
// Вывод графа, заданного вектором смежности, в файл: каждое ребро выводится в
новой строке. Граф - невзвешенный, либо веса ребер целочисленны.
// "Fouts" a graph that is set by Adjacency vector A to file: one edge in one
line. Parameter "weighted" sets if the graph A is weighted or no.
// Returns -1 if input data is not correct. Otherwise returns 0.

{

    if ( (P.size())%(2+weighted)!=0 ) return -1; // checking for input data
correctness

    if (P.size()==0) return -1;  // checking for input data correctness

    for (int i=0; i<P.size();i=i+2)
        fout<< P[i]<<' '<<P[i+1]<<std::endl;
    fout<< std::endl;

    return 0;
}
```

```
int GraphCout (const std::pair < std::vector<int>, std::vector<double>> & P,
unsigned int prec = 4, bool scientifique = false)
// Модификация функции GraphCout (см. выше) для взвешенных графов с
нецелочисленными весами ребер.
// Modification of the function GraphCout (see it above) for not-integer (double)
weights of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an
"Adjacency vector" without weights. But weights are set in the second one.
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the
first vector has its weight set as i-th element in the second one.


{

    if ((P.first).size()==0) return -1;
    if ((P.second).size()==0) return -1;
    if (  (P.first).size()!=((P.second).size())*2 ) return -1;

    if (!scientifique)
    {
    std::cout.precision(prec);

    for (int i=0; i<(P.second).size();i++)
    {
        std::cout<< (P.first)[2*i]<<" ";
        std::cout<< (P.first)[2*i+1]<<" ";
        std::cout<< std::fixed<<(P.second)[i]<<" ";
        std::cout<< std::endl;
    }
    std::cout<< std::endl;

    }

    if (scientifique)
        {

                for (int i=0; i<(P.second).size();i++)
                {
                    std::cout<< (P.first)[2*i]<<" ";
                    std::cout<< (P.first)[2*i+1]<<" ";
                    std::cout<< std::scientific<<(P.second)[i]<<" ";
                    std::cout.unsetf(std::ios::scientific);
                    std::cout<< std::endl;
                }


                std::cout<< std::endl;


        }



    return 0;
}


int GraphFout (const std::pair < std::vector<int>, std::vector<double>> & P,
std::ofstream &fout, unsigned int prec = 4, bool scientifique = false)
// Модификация функции GraphFout (см. выше) для взвешенных графов с
нецелочисленными весами ребер.
```

```
// Modification of the function GraphFout (see it above) for not-integer (double)
weights of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an
"Adjacency vector" without weights. But weights are set in the second one.
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the
first vector has its weight set as i-th element in the second one.


{

    if ((P.first).size()==0) return -1;
    if ((P.second).size()==0) return -1;
    if (  (P.first).size()!=((P.second).size())*2 ) return -1;

    if (!scientifique)
    {
    fout.precision(prec);

    for (int i=0; i<(P.second).size();i++)
    {
        fout<< (P.first)[2*i]<<" ";
        fout<< (P.first)[2*i+1]<<" ";
        fout<< std::fixed<<(P.second)[i]<<" ";
        fout<< std::endl;
    }
    fout<< std::endl;

    }

    if (scientifique)
        {

                for (int i=0; i<(P.second).size();i++)
                {
                    fout<< (P.first)[2*i]<<" ";
                    fout<< (P.first)[2*i+1]<<" ";
                    fout<< std::scientific<<(P.second)[i]<<" ";
                    fout.unsetf(std::ios::scientific);
                    fout<< std::endl;
                }


                fout<< std::endl;


        }



    return 0;
}









int SwapInVector (std::vector <int> & A1, unsigned int f, unsigned int l)
{
    // swaps 2 elements in vector (int). Returns -1 if some index out of vector's
range or vector is empty
```

```
    // Замена элементов в векторе (int), возвращает -1 если хоть один из
запрашиваемых индексов выходит за размер вектора либо если вектор пустой


    if ((A1.size()==0) || (f>=A1.size()) || (l>=A1.size())) return -1;

    int t = A1[f];
    A1[f] = A1[l];
    A1[l] = t;
    return 0;

}

int SwapInVector (std::vector <long long int> & A1, unsigned int f, unsigned int
l)
{
    // swaps 2 elements in vector (long long int). Returns -1 if some index out of
vector's range or vector is empty
    // Замена элементов в векторе (long long int), возвращает -1 если хоть один из
запрашиваемых индексов выходит за размер вектора либо если вектор пустой


    if ((A1.size()==0) || (f>=A1.size()) || (l>=A1.size())) return -1;

    long long int t = A1[f];
    A1[f] = A1[l];
    A1[l] = t;
    return 0;

}

int SwapInVector (std::vector <double> & A1, unsigned int f, unsigned int l)
{
    // swaps 2 elements in vector (double). Returns -1 if some index out of
vector's range or vector is empty
    // Замена элементов в векторе (double), возвращает -1 если хоть один из
запрашиваемых индексов выходит за размер вектора либо если вектор пустой


    if ((A1.size()==0) || (f>=A1.size()) || (l>=A1.size())) return -1;

    double t = A1[f];
    A1[f] = A1[l];
    A1[l] = t;
    return 0;

}


int SwapInVector (std::vector <long double> & A1, unsigned int f, unsigned int l)
{
    // swaps 2 elements in vector (long double). Returns -1 if some index out of
vector's range or vector is empty
    // Замена элементов в векторе (long double), возвращает -1 если хоть один из
запрашиваемых индексов выходит за размер вектора либо если вектор пустой


    if ((A1.size()==0) || (f>=A1.size()) || (l>=A1.size())) return -1;

    long double t = A1[f];
    A1[f] = A1[l];
    A1[l] = t;
    return 0;
```

```
}


int SwapInVector (std::vector <std::string> & A1, unsigned int f, unsigned int l)
{
    // swaps 2 elements in vector (string). Returns -1 if some index out of
vector's range or vector is empty
    // Замена элементов в векторе (string), возвращает -1 если хоть один из
запрашиваемых индексов выходит за размер вектора либо если вектор пустой


    if ((A1.size()==0) || (f>=A1.size()) || (l>=A1.size())) return -1;

    std::string t = A1[f];
    A1[f] = A1[l];
    A1[l] = t;
    return 0;

}


int HmDist (const std::string &s1, const std::string &s2)
{
    // Counts Hamming Distance; returns -1 if any string is empty or they have
different length.
    // Считает Hamming Distance, возвращает -1 если строки разной длины либо хоть
одна пустая.

    if ((s1.length()==0) || (s2.length()==0) || (s1.length()!=s2.length())) return
-1;
    int q = 0;
    for (unsigned int i=0; i<s1.length(); i++)
    {if (s1[i] != s2[i]) q++;}
    return q;
}

int RComplDNA (const std::string& s, std::string & sr)
{
    // generates reverse complement of string s as string sr, returns -1 and empty
string sr if string s is empty or it is not DNA

     sr = s;

     if (s.length()==0) return -1;

     for (unsigned int i = 0; i<s.length(); i++)
     {
      if (s [s.length()-i-1] == 'A') {sr [i]  = 'T'; continue;}
      if (s [s.length()-i-1] == 'T') {sr [i]  = 'A'; continue;}
      if (s [s.length()-i-1] == 'G') {sr [i]  = 'C'; continue;}
      if (s [s.length()-i-1] == 'C') {sr [i]  = 'G'; continue;}

      sr.clear();
      return -1;
     }

    return 0;
}


int RComplRNA (const std::string& s, std::string & sr)
{
    // generates reverse complement of string s as string sr, returns -1 and empty
string sr if string s is empty or it is not RNA
```

```
     sr = s;

     if (s.length()==0) return -1;

     for (unsigned int i = 0; i<s.length(); i++)
     {
      if (s [s.length()-i-1] == 'A') {sr [i]  = 'U'; continue;}
      if (s [s.length()-i-1] == 'U') {sr [i]  = 'A'; continue;}
      if (s [s.length()-i-1] == 'G') {sr [i]  = 'C'; continue;}
      if (s [s.length()-i-1] == 'C') {sr [i]  = 'G'; continue;}

      sr.clear();
      return -1;
     }

    return 0;
}


std::string rp (const std::string& s)
{
    // generates reverse complement of DNA without any checking of input data
correctness

     std::string sr = s;
     for (unsigned int i = 0; i<s.length(); i++)
     {
      if (s [s.length()-i-1] == 'A') sr [i]  = 'T';
      if (s [s.length()-i-1] == 'T') sr [i]  = 'A';
      if (s [s.length()-i-1] == 'G') sr [i]  = 'C';
      if (s [s.length()-i-1] == 'C') sr [i]  = 'G';
     }
    return sr;
}


std::string rpr (const std::string& s)
{
    // generates reverse complement of RNA without any checking of input data
correctness

     std::string sr = s;
     for (unsigned int i = 0; i<s.length(); i++)
     {
      if (s [s.length()-i-1] == 'A') sr [i]  = 'U';
      if (s [s.length()-i-1] == 'U') sr [i]  = 'A';
      if (s [s.length()-i-1] == 'G') sr [i]  = 'C';
      if (s [s.length()-i-1] == 'C') sr [i]  = 'G';
     }
    return sr;
}



double gcDRNA (const std::string &s)
{

    // Counts DNA/RNA GC-content; in case any symbol not DNA/RNA-nucleotide or
string s is empty returns -1.0.

    if (s.length()==0) return -1.0;

    int count = 0;
```

```
    for (int i = 0; i< s.length(); i++)
    {
        if ((s[i]=='G') || (s[i]=='C')) {count++; continue;}
        if ((s[i]=='A') || (s[i]=='T') || (s[i]=='U')) {continue;}
        return -1.0;
    }


    return 1.0*count/s.length();
}


int RNAfromDNA (const std::string &s, std::string & sr)
{
     // generates RNA from DNA, returns -1 and empty string sr if the input string
s is empty or it is not DNA

    if (s.length()==0) {sr.clear(); return -1;}
    sr = s;
    for (int i = 0; i<s.length(); i++)
    {
        if (sr [i] == 'T') {sr [i]  = 'U'; continue;}
        if (sr [i] == 'A') {continue;}
        if (sr [i] == 'G') {continue;}
        if (sr [i] == 'C') {continue;}

        sr.clear();
        return -1;
    }

return 0;

}


int DNAfromRNA (const std::string &s, std::string & sr)
{
     // generates DNA from RNA, returns -1 and empty string sr if the input string
s is empty or it is not RNA

    if (s.length()==0) {sr.clear(); return -1;}
    sr = s;
    for (int i = 0; i<s.length(); i++)
    {
        if (sr [i] == 'U') {sr [i]  = 'T'; continue;}
        if (sr [i] == 'A') {continue;}
        if (sr [i] == 'G') {continue;}
        if (sr [i] == 'C') {continue;}

        sr.clear();
        return -1;
    }

return 0;

}


std::string RNAg (const std::string &s)
{
// generates RNA from DNA without checking of data correctness

    std::string sr = s;
```

```
    for (int i = 0; i<s.length(); i++)
        {
        sr [i]= s [i];
        if (sr [i] == 'T') sr [i]  = 'U';
    }

return sr;
}


std::string DNAg (const std::string &s)
{
// generates RNA from DNA without checking of data correctness

    std::string sr = s;
    for (int i = 0; i<s.length(); i++)
        {
        sr [i]= s [i];
        if (sr [i] == 'U') sr [i]  = 'T';
    }

return sr;
}



std::string StrToCircular (const std::string& s, int tail = INT_MAX)
// Находит и возвращает кратчайшую "круговую" ("скрученную в кольцо") строку для
заданной (т.е. с максимальным "нахлестом" конца строки на начало).
// Для строк длиной менее 3 символов возвращает ее же (считается, что в этом
случае "нахлест" невозможен.
// Возможно задать параметр tail - величнину максимального "нахлеста". Значения
tail<=0 в расчет не принимаются.

//  Returns a circular string of minimal length of a string s; if length of s <3
returns s itself.
//  One may set "tail" i.e. maximal overlap "tail-to-beginning" of the string s
(nonpositive values of "tail" will be ignored).


{
    if (s.length()<3) return s;
    if (tail<=0) tail = INT_MAX;

    int N = s.length()/2;
    if (tail<s.length()/2) N=tail;

    std::string sr = s;
    for (int i = N; i>0; i--)
        if (s.substr(0, i) == s.substr(s.length() - i, i))
        {
            sr = s.substr(0, s.length() - i);
            break;
        }

    return sr;
}



int TandemRepeatsFinding (const std::string &s, std::vector <int> &Result, int
MaxWordLength = 4, int limit = 5)
```

```
// Функция находит все тандемные повторы в строке s, сформированные j-мерами длинной от 1 по MaxWordLength символов и имеющие длину не менее limit
// MaxWordLength должно быть в границах от 2 по 6. limit должен быть больше MaxWordLength (иначе limit будет присвоено значение MaxWordLength+1).
// Результат возвращается в векторе Result, где на четных позициях, отсчитываемых с нуля - позиции начала тандемного повтора в s (символы в s также нумеруются с нуля), а на нечетных - длина повтора
// Возвращает 0 в случае успеха. В случае некорректных данных (длина s < MaxWordLength, MaxWordLength не в диапазоне [2; 6]) возвращает пустой Result и -1.

// Finds all tandem repeats in the string s as follows:
// the repeat has its lenght >= limits, the repeat should be formed by j-mers: j in [1; MaxWordLength], MaxWordLength in [2; 6]
// limit should be more than MaxWordLength (if no the limit's value will be set as MaxWordLength+1)
// Returns 0 and vector Result: every even position of Result contains the starting position of tandem repeat in the s (0-based indexing)
// and every odd position of Result contain the lenghts of the repeat that have its starting position as previous element in the Result.
// Returns -1 and empty Result if input data is incorrect.


{
    Result.clear();
    if (limit <= MaxWordLength ) limit = MaxWordLength+1;
    if ((MaxWordLength <2) || (MaxWordLength >6) ) return -1;
    if (s.length()<MaxWordLength) return -1;


    std::string Alph = "ACGT";
    int WordLenght;
    int AlphLenght = Alph.length();

    std::string SA = "A";
    std::string SC = "C";
    std::string SG = "G";
    std::string ST = "T";

    for (int q=0; q<MaxWordLength; q++)
    {
        SA = SA + "A";
        SC = SC + "C";
        ST = ST + "T";
        SG = SG + "G";
    }

    std::string TempS;

    char out [10];   // an auxiliary array to make j-mers

    std::vector <int> l; // an auxiliary array to form j-mers (to do so let's do 4-digit numbers: one digit corresponds to one and only one symbol from the string Alph)

    std::vector <int> P; // an auxiliary array to contain starting positions of j-mers in the string s
    int y, t;
    int f=-1;
    int count;
    int b, e;
```

```
    // First of all lets see all j-mers: j in [2; MaxWordLength], but except those
are formed by only one and the same symbol (i.e. except those like "AAAA", "TTT",
etc)

    for (int j=MaxWordLength; j>1; j--)  // let's generate j-mers: j in [2;
MaxWordLength]
    {

        SA.pop_back(); // j-mers that are formed
        SC.pop_back();  // by only one repeating symbol
        SG.pop_back();  // (i.e. j-mers like "AAAA", "CC", "TTT, etc)
        ST.pop_back(); // will be ignored here.


        WordLenght = j;
        long long int nn = (long long int) (pow (double(AlphLenght), WordLenght));
//nn - total number of j-mers

        l.clear();
        l.resize(j, 0);

        for (int g=0; g<WordLenght; g++)
        {

            out [g] = Alph [0];  // setting starting j-mer

        }


        for (long long int ii=0; ii<nn; ii++)
        {
             TempS = "";


            for (int u=0; u< WordLenght; u++)
            {

                TempS= TempS + out [u]; //forming the next j-mer
            }

            // we have TempS as j-mer now


            if ((TempS==SA) || (TempS==SC) || (TempS==SG) || (TempS==ST) ) goto
label1;  // if j-mer is formed by only one repeating symbol - it is not for
working with it here

            P.clear();
            y=s.find(TempS, 0);
            while (y!=-1) // We' ll write to P every starting positions of the j-
mer as a substring of the string s
            {
                P.push_back(y);
                y = s.find(TempS, y+1);

            }


            count = WordLenght; //count will contain the lenght of tandem repeat
            for (int q=1; q<P.size(); q++)
            {
                if (P[q]-P[q-1]==WordLenght)  //if so - the tandem repeat
continues
                {
```

```
                    count = count + WordLenght;
                    t=P[q];
                    continue;
                }

                if (count>= limit) // if the tandem repeat has its lenght >= limit
(as limits is the minimal lenght of repeats to be found)
                {
                    b = t+WordLenght-count; //the starting position in the string
s of the repeat
                    e = count; // and its lenght

                    // cheking if the repeat is contained in any already found or
it contains any of them itself
                    f=1;
                    for (int w=0; w<Result.size(); w=w+2)
                    {
                        if ( (Result[w]<=b)&& ((Result[w]+Result[w+1])>=(e+b))  )
                        {f=-1; break;} // it is contained in some one found before

                        if ( (Result[w]>=b)&& ((Result[w]+Result[w+1])<=(e+b))  )
// in this case we eliminate the one contained in the new found repeat
                        {Result.erase(Result.begin()+w);
Result.erase(Result.begin()+w);}

                    }

                    if (f==1)
                    {
                    Result.push_back(t+WordLenght-count); // writing to P the
starting pos
                    Result.push_back(count); // and the lenght of the new repeat
found
                    }



                }

                count = WordLenght;
            }

            if (count>= limit) // another iteration for the last repeat in the s
            {
                b = t+WordLenght-count;
                e = count;
                f=1;
                for (int w=0; w<Result.size(); w=w+2)
                {
                    if ( (Result[w]<=b)&& ((Result[w]+Result[w+1])>=(e+b))  )
                    {f=-1; break;}

                    if ( (Result[w]>=b)&& ((Result[w]+Result[w+1])<=(e+b))  )
                    {Result.erase(Result.begin()+w);
Result.erase(Result.begin()+w);}

                }



                if (f==1)
                {
                Result.push_back(t+WordLenght-count);
                Result.push_back(count);
```

```
                }
            }



    label1: ;
        l [WordLenght-1]++;  //увеличение на 1 кода последней буквы // the digit
that corresponds to the last symbol of j-mer is to be increased

        if (l [WordLenght-1] == AlphLenght) // if so - we must recalculate the
senior digits too
         {
           int r = WordLenght-1;

           while (r>0)
             {l [r] = 0;
              r--;
              l [r] ++;
              if (l[r] < AlphLenght) break;
             }
         }

        for (int t = 0; t<WordLenght; t++)
              out [t] = Alph [(l[t])];


        }

    }  // End of observing all j-mers: j in [2; MaxWordLength], but except those
are formed by only one and the same symbol (i.e. except those like "AAAA", "TTT",
etc)


    // Now lets find substrings formed by only one repeating symbol and that have
their lenght not less than "limits"
    // The algorithm is like above, but we are looking for substrings formed by by
only one repeating symbol.
    //In doing so we count their lenght in the "count" and if (count>= limit) - we
have found another one repeat


    count =1;

    for (int q=1; q<s.length(); q++)
    {

        if (s[q]==s[q-1])
        {
            count++;
            t=q;
            continue;
        }

        if (count>= limit)
        {
            Result.push_back(t+1-count);
            Result.push_back(count);
        }

    count=1;
    }

    if (count>= limit)
```

```
    {
        Result.push_back(t+WordLenght-count);
        Result.push_back(count);
    }

    return 0;
}



void GMapCodonRNA (std::map <std::string, std::string> & MapCodon)
{
    //Generates codon table for RNA in the map MapCodon ("$" means stop codon).
    // MapCodon format: Codon -> Amino acid.


    MapCodon.clear();

    MapCodon =
    {
        {"UUU", "F"},
        {"CUU", "L"},
        {"AUU", "I"},
        {"GUU", "V"},
        {"UUC", "F"},
        {"CUC", "L"},
        {"AUC", "I"},
        {"GUC", "V"},
        {"UUA", "L"},
        {"CUA", "L"},
        {"AUA", "I"},
        {"GUA", "V"},
        {"UUG", "L"},
        {"CUG", "L"},
        {"AUG", "M"},
        {"GUG", "V"},
        {"UCU", "S"},
        {"CCU", "P"},
        {"ACU", "T"},
        {"GCU", "A"},
        {"UCC", "S"},
        {"CCC", "P"},
        {"ACC", "T"},
        {"GCC", "A"},
        {"UCA", "S"},
        {"CCA", "P"},
        {"ACA", "T"},
        {"GCA", "A"},
        {"UCG", "S"},
        {"CCG", "P"},
        {"ACG", "T"},
        {"GCG", "A"},
        {"UAU", "Y"},
        {"CAU", "H"},
        {"AAU", "N"},
        {"GAU", "D"},
        {"UAC", "Y"},
        {"CAC", "H"},
        {"AAC", "N"},
        {"GAC", "D"},
        {"UAA", "$"},
        {"CAA", "Q"},
        {"AAA", "K"},
```

```
        {"GAA", "E"},
        {"UAG", "$"},
        {"CAG", "Q"},
        {"AAG", "K"},
        {"GAG", "E"},
        {"UGU", "C"},
        {"CGU", "R"},
        {"AGU", "S"},
        {"GGU", "G"},
        {"UGC", "C"},
        {"CGC", "R"},
        {"AGC", "S"},
        {"GGC", "G"},
        {"UGA", "$"},
        {"CGA", "R"},
        {"AGA", "R"},
        {"GGA", "G"},
        {"UGG", "W"},
        {"CGG", "R"},
        {"AGG", "R"},
        {"GGG", "G"}
    };

}


void GMapCodonRNA_A (std::map <std::string, std::vector<std::string>> & MapCodon)
//Generates codon table for RNA in the map MapCodon ("$" means stop codon).
// MapCodon format: Amino acid -> vector of relevant codons.
{

    MapCodon.clear();

    MapCodon =
        {
            {"F", {"UUU", "UUC"}},
            {"L", {"CUU", "CUC", "CUA", "CUG", "UUG", "UUA"}},
            {"I", {"AUU", "AUC", "AUA"}},
            {"V", {"GUU", "GUC", "GUA", "GUG"}},
            {"M", {"AUG"}},
            {"S", {"UCU", "UCC", "UCA", "UCG", "AGU", "AGC"}},
            {"P", {"CCU", "CCC", "CCA", "CCG"}},
            {"T", {"ACU", "ACC", "ACA", "ACG"}},
            {"A", {"GCU", "GCC", "GCA", "GCG"}},
            {"Y", {"UAU", "UAC"}},
            {"H", {"CAU", "CAC"}},
            {"N", {"AAU", "AAC"}},
            {"D", {"GAU", "GAC"}},
            {"$", {"UAA", "UAG", "UGA"}},
            {"Q", {"CAA", "CAG"}},
            {"K", {"AAA", "AAG"}},
            {"E", {"GAA", "GAG"}},
            {"C", {"UGU", "UGC"}},
            {"R", {"CGU", "CGC", "CGA", "AGA", "CGG", "AGG"}},
            {"G", {"GGU", "GGC", "GGA", "GGG"}},
            {"W", {"UGG"}},
         };


}


void GMapMonoisotopicMassTableLD (std::map <char, long double> & MassTable)
{
```

```
    //Generates Monoisotopic mass table in the map (long double)

    MassTable.clear();


MassTable =
  {
    {'A', 71.03711},
    {'C', 103.00919},
    {'D', 115.02694},
    {'E', 129.04259},
    {'F', 147.06841},
    {'G', 57.02146},
    {'H', 137.05891},
    {'I', 113.08406},
    {'K', 128.09496},
    {'L', 113.08406},
    {'M', 131.04049},
    {'N', 114.04293},
    {'P', 97.05276},
    {'Q', 128.05858},
    {'R', 156.10111},
    {'S', 87.03203},
    {'T', 101.04768},
    {'V', 99.06841},
    {'W', 186.07931},
    {'Y', 163.06333}

  };
}



int GPFM (std::vector <std::string> &s, std::vector <std::vector <int>> & B, const
std::string &Alph)
{

    // Генерирует позиционную матрицу частот B по набору исходных строк s и
алфавиту Alph (содержит последовательность символов алфавита);
    // Последовательность строк в матрице B соответствует последовательности
символов в строке Alph (т.е. последовательности символов алфавита).
    // в случае если в наборе менее 2х строк или они имеют неодинаковую длину или
в алфавите менее 2 букв или хоть одна из строк содержит хоть один символ не из
алфавита,
    // или же если алфавит содержит дублирующиеся символы - возвращается -1  и
пустая матрица B (в случае успеха возвращается 0).

    // Generates position frequency matrix (PFM) B upon an array of strings s and
given Alphabet (Alphabet is set via string Alph that contains the sequence of its
symbols);
    // Ordering of the rows in B corresponds to sequence of symbols in Alph.
    // If s contains 1 or 0 items or strings have not equal length or even the
only string contains symbol that not belongs to Alphabet
    // or if there are any identical symbols in the Alphabet - returns -1 and
empty B.



    B.clear();
    if ((s.size()<=1) || (Alph.length()<2)) return -1; // checking that s contains
more than 2 stings and Alph consists of >=2 symbols.


    int lstring = (s[0]).length();
```

```
    int lvector = s.size();


    for (int i=1; i<s.size(); i++)  // checking that all strings in s have equal
length
        if (s[i].length()!=lstring)
        {
            return -1;
        }

    std::set <char> T; // проверка что нет дублирующихся символов в алфавите
    T.clear(); // Testing if there are any identical symbols in the Alphabet
    for  (int y=0; y<Alph.length(); y++)
    {
        T.insert(Alph[y]);
        if ((T.size()-1)!=y)
            return -1;

    }
    T.clear();

    MatrixSet (B, Alph.length(), lstring, 0);

    for (int i = 0; i<lstring; i++)
    {
        for (int j = 0; j<lvector; j++)
        {
            for  (int y=0; y<Alph.length(); y++)
                 if (((s[j])[i]) == Alph[y]) {B[y][i]++; goto l1;}



                B.clear(); // Если хоть один символ не из алфавита - очищаем
матрицу и возвращаем -1
                return -1; // even one symbol doesn't belong to Alph - matrix B to
be cleared and -1 to return

            l1:;
        }

    }

    return 0;

}




double PDist (const std::string& s1, const std::string& s2) // counts p-distance
without checking of the input data correctness
{
    double r;
    int q = 0;
    for (unsigned int i=0; i<s1.length(); i++)
    {if (s1[i] != s2[i]) q++;}
    r = double(q)/s1.length();
    return r;
}


int GDistanceMatrix (std::vector <std::string> &s, std::vector <std::vector
<double>> & B)
```

```
{
// Генерирует матрицу расстояний "B" по набору исходных строк s; в случае если в наборе менее 2х строк или они имеют неодинаковую длину - возвращается -1 (в случае успеха - 0).
// Generates DistanceMatrix "B" upon array of strings s; if s contains 1 or 0 items or strings have not equal length returns -1 and empty B.


    B.clear();
    if (s.size()<=1) return -1; // checking that s contains more than 2 stings

    int lstring = (s[0]).length();
    int lvector = s.size();


    for (int i=1; i<s.size(); i++)  // checking that all strings in s have equal length
        if (s[i].length()!=lstring)
        {
            return -1;
        }


    MatrixSet (B, lvector, lvector, 0);

    for (int i = 0; i<lvector; i++)
    for (int j = 0; j<lvector; j++)
    {
        B [i][j] = PDist(s[i], s[j]);
    }

    return 0;
}


int EditDist (const std::string &s1, const std::string &s2)
// Рассчитывает редакционное расстояние (расстояние Левенштейна) между строками, принимает на вход даже пустые. Цена каждой операции = 1
// Computes Edit Distance (Levenshtein distance) between two strings (strings may be empty too).

{
int n = s1.length()+1;
int m = s2.length()+1;

if ((n==1)&&(m==1)) return 0;

if (n==1) return (m-1);
if (m==1) return (n-1);

std::vector <std::vector <int>> B (n);  // Generating pre-matrix for computing distance filled by zeros.
for (unsigned int row = 0; (row< n); row++)
{
    B [row].resize(m);
    for (unsigned int column = 0; (column < m); column++)
    {
        B [row] [column] = 0;

    }

}
```

```
// Filling the matrix
int w=1;
for (unsigned int i = 0; (i< n); i++)
    for (unsigned int j = 0; (j< m); j++)
        if (j==0) B[i][j] = i;
        else if (i==0) B[i][j] = j;
        else
        {
            w = 1;
            if (s1[i-1] == s2[j-1])   // нужны элементы строк i, j по порядку, но
т.к. нумеруем с нуля символы строк - поправка на -1
            //Note that symbols of strings have 0-based indexing. So we have "-1 -
correction" here.
                w = 0;
            B[i][j] = std::min (1+B[i-1][j], 1+B[i][j-1]);
            if ((w+B[i-1][j-1]) < B[i][j]) B[i][j] = (w+B[i-1][j-1]);
        }


return B[n-1][m-1];

}




void EDistForFindMR (const std::string &s1, const std::string &s2, const int D,
const int L, int l, int b, std::set <std::pair <int, int>> &Result)

// Вспомогательная функция для FindMutatedRepeatsED (см. ниже, приводится
следующей).
// An auxiliary function for FindMutatedRepeatsED, see its info for details
(below, the following one).

{
int n = s1.length()+1;
int m = s2.length()+1;



std::vector <std::vector <int>> B (n);  // Generating pre-matrix for computing
edit distance. It is filled by zeros by default.
MatrixSet(B, n, m, 0);



// Filling the matrix
int w=1;
for (unsigned int i = 0; (i< n); i++)
    for (unsigned int j = 0; (j< m); j++)
        if (j==0) B[i][j] = i;
        else if (i==0) B[i][j] = j;
        else
        {
            w = 1;
            if (s1[i-1] == s2[j-1])   // нужны элементы строк i, j по порядку, но
т.к. нумеруем с нуля символы строк - поправка на -1
            //Note that symbols of strings have 0-based indexing. So we have "-1 -
correction" here.
                w = 0;
            B[i][j] = std::min (1+B[i-1][j], 1+B[i][j-1]);
            if ((w+B[i-1][j-1]) < B[i][j]) B[i][j] = (w+B[i-1][j-1]);
        }
```

```
int c1 = n-1;
int c2 = m-1;

while (l>=L-D)
{
    if (B[c1][c2]<=D) // The matrix B contains Edit Distance between s2 and not only s1, but for all prefixes of s1 too (the previous row at the same coloumn for the substring formed by deleting the last symbol etc).
    {


        Result.insert(std::pair<int, int>(b-1, l)); // So if Edit Distance <=D and the length of prefix not less than L-D, this prefix is the one of required. So let's insert its data to Result.

    }
    l--;
    c1--;

}


}



int FindMutatedRepeatsED (std::string &StrShort, std::string &StrLong, int D, std::set <std::pair <int, int>> &Result)
// Функция находит все подстроки для строки StrLong, редакционное расстояние которых до StrShort не превышает D. При этом принимается, что "штраф" за пропуск и несовпадение символов  = 1.
// Результат возвращается в set <std::pair <int, int>> Result, где первое число в паре - номер позиции начала подстроки в StrLong (счет позиций идет с 0), а второе - длина подстроки (пары не отсортированы).
// Если исходные данные некорректны - возвращается -1 и пустой Result;, в случае успеха возвращается 0.
// Идея реализованного алгоритма:
// (1) Найти все начала таких подстрок в StrLong.
// Для этого обе строки реверсируются, затем StrShort "выравнивается" на StrLong по обычным правилам для нахождения редакционного расстояния, но с тем отличием,
// что суммарный начальный пропуск по StrLong не "штрафуется" (начать можно с любой позиции в длинной строке без "штрафа"). Найденные начальные позиции нумеруются с 1. Затем строки реверсируются обратно.
// (2) Для каждой позиции вычисляется максимально возможная длина искомой подстроки, которая не может быть более длины StrShort плюс D, и при этом не может выходить за границу StrLong.
// Пояснение. Длины искомых подстрок не могут отличаться от длины StrShort более чем на D в ту и другую сторону, т.к. редакционное расстояние не превышает D, а цена пропуска = 1.
// (3) Если такая максимально возможная длина есть и составляет не менее длины StrShort минус D, то для соотвествующей подстроки (обозначим как TempS) и StrShort осуществляем стандартный Edit Distance Alignment с помощью вспомогательной функции EDistForFindMR.
// И в выстраиваемой для этих целей матрице будут значения Edit Distance не только между StrShort и TempS, но и (!) укороченным с конца подстрокам TempS (для этого берем значения в матрице не только по последней строке (TempS "откладывается" вниз), но и по предшествующим.
// Если для каждого такого префикса строки TempS (при условии, что его длина удовлетворяет пояснению к шагу (2)) значение Edit Distance не превышает D - фиксируем в set Result его начальную позицию (в нумерации от 0) и длину.
```

```
// Функция возвращает 0 и заполненный Result в случае успеха и -1 и пустой Result
в случае некорректности исходных данных (любая из строк пуста или StrShort длиннее
StrLong или длина StrShort не превосходит D)


// The function finds all the substrings of a string StrLong, that have Edit
Distance to a string StrShort <= D. Gap and mistmatch penalties are set as "1"
here.
// If dataset is correct returns 0 and set <std::pair <int, int>> Result, that
contains pairs of integers: first one is a start position of a required substring
in StrLong (0-based indexing)  and the second one is its length.
// If dataset is not correct (any string is empty of StrShort is longer than
StrLong or StrShort's length <= D) returns -1 and empty Result.
// The algorithm idea is:
// (1) to find all start positions of such substrings. To do so we should reverse
both strings and then do Edit Distance Alignment but with no gap penalty at the
beginning: The required substring may start at every position of the longer string
so here are no penalty fo gapping at start.
// (2) For each start position the maximal possible length for the required
substring (<= StrShort.length+D, but within StrLong).
// Note that the required substrings may have length <= StrShort.length+D and >=
StrShort.length-D because gap penalty = 1.
// (3) If such maximal possible length meets this condition, let a string TempS be
a substring of StrLong of this length (TempS starts from relevant start position
in StrLong).
// And then let's do Edit Distance Alignment between TempS and StrShort in order
to find prefixes of TempS, that require the statement of problem to be solved
here.


{
    Result.clear();

    if (StrShort.length()>StrLong.length()) return -1; // Приверка корректности
исходных данных  // checking for input tata correctness
    if ((StrShort.length()==0) || (StrLong.length()==0))return -1; // Приверка
корректности исходных данных  // checking for input tata correctness
    if (D<0) return -1; // Приверка корректности исходных данных  // checking for
input tata correctness
    if (StrShort.length()<=D) return -1; // Приверка корректности исходных данных
// checking for input tata correctness


    std::string TempS = "";


    const int gapP = -1; //gap penalty
    const int mismP = -1; //mismatch penalty

std::set <int>StartPositions; //Here the start positions of required substrings of
StrLong will be contained (their numbering starts from "1" for this set).
// Здесь будем хранить номера начал искомых строк (здесь допущение: символы в
строке нумеруются с 1)




int L = StrShort.length();
StartPositions.clear();


//Preparing a matrix for alignment
unsigned int n = StrShort.length()+1;
unsigned int m = StrLong.length()+1;
```

```
std::vector <std::vector <int>> B (n);


int count0;
int count1;

int w;
int mt, dt, in;



MatrixSet(B, n, m, 0);


  for (unsigned int i = 0; (i< n); i++)  // Поправка: начать можно с любой позиции в длинной строке без "штрафа"
      B[i][0] = i*gapP;                  // The required substring may start at every position of the longer string so here are no penalty fo gapping at start

  // End of preparing a matrix for alignment

  reverse(StrShort.begin(),StrShort.end()); // Переворачиваем строки, чтобы найти все начальные позиции искомых подстрок в StrLong
reverse(StrLong.begin(),StrLong.end()); // the strings should be reversed now in order to find all start positions of required substrings of StrLong

for (unsigned int i = 1; (i< n); i++)
    for (unsigned int j = 1; (j< m); j++)

        {
            w=mismP;
            if(StrShort[i-1] == StrLong[j-1]) w=0;
            mt = B[i-1][j-1] + w;
            dt = B[i-1][j] + gapP;
            in = B[i][j-1] + gapP;

            B[i][j] = mt;
            if (B[i][j]<std::max(dt, in)) B[i][j] = std::max(dt, in);
        }

reverse(StrShort.begin(),StrShort.end());  // теперь переворачиваем строки назад
reverse(StrLong.begin(),StrLong.end()); // now the strings are reversed back


for (int i=0;i<m;i++)
    if(B[n-1][i]*(-1)<=D) // Условие того, что в данной точке будет начало одной или нескольких искомых подстрок (с поправкой на реверс)
                          // This means that here will be the beginning for one or more required substrings (reverse adjusted)
    {
        StartPositions.insert(StrLong.length()-i+1);  // Computing the starting position ("1"-based indexing)

    }


int l;




for (auto it = StartPositions.begin(); it!=StartPositions.end(); it++)
```

```
    // Для каждой найденной точки начала подстврок вычисляем l - максимально возможную длину искомой подстроки, которая не может быть более длины короткой строки плюс D, и при этом не может выходить за границу длинной строки.
    // Let's compute "l" for every starting position. "l" means the maximal possible length for the required substring (<= StrShort.length+D, but within StrLong).
{

    l=-1;
    if ((*it+L+D-1)<=StrLong.length()) l=L+D;
    else if (*it+L-D-1<=StrLong.length()) l=StrLong.length()-*it+1;



    if (l>0) // If so - lets do Edit Distance Alignment between TempS and StrShort in order to find prefixes of TempS, that require the statement of problem to be solved by the function FindMutatedRepeatsED.
    {
        TempS = StrLong.substr(*it-1, l);
        EDistForFindMR(TempS, StrShort, D, L, l, *it, Result);
    }


}


return 0;

}




int PartitionOfNumber (std::vector <std::vector <int>> &B, int n)
// Генерирует разбиения числа на слагаемые для чисел больше 0 (иначе вернет -1). Результат генерируется в векторе векторов B.
//Generates partitions of int n (i.e. representing n as a sum of positive integers) in B. If n<=0 returns empty B and "-1"

{
    B.clear();

    if (n<=0) return -1;
    int t,y;
    std::vector <int> T (n, 1);

    while (true)
    {
        B.push_back(T);

        if (T[0]==n) break;
        if (T.size()==1) break;
        t=T[T.size()-2];
        for (int i=T.size()-2; i>=0; i--)
        {
            if (T[i]>t) {y=i+1; break;}

            y=i;
        }



        T[y]++;
```

```
        T[T.size()-1]--;
        if (T[T.size()-1]==0) T.pop_back();

        t=T.size();
        for (int z= y+1; z<t; z++)
        {
            if (T[z]==1) continue;
            T[z]--;
            T.push_back(1);
            z--;
        }


    }

    return 0;
}


int PartitionOfNumberL (std::vector <std::vector <int>> &B, int n, int l=-1)
// Генерирует разбиения числа на слагаемые для чисел больше 0 (иначе вернет -1).
Результат генерируется в векторе векторов B. Расширенная версия:
// можно задать длину разбиения l. Если l>0, то возвращаются только разбиения
длиной l. При этом более короткие разбиения "добиваются справа" нулями.
//Generates partitions of int n (i.e. representing n as a sum of positive
integers) in B. Extended version: one may set l>0 as a length of patritions (i.e.
number of summands).
// In this case "0" will be added to the end of the shorter partitions. If n<=0
returns empty B and "-1"


{
    B.clear();

    if (n<=0) return -1;
    int t,y;
    std::vector <int> T (n, 1); // starting partition has only ones as summands

    if (l<=0) l=n;  // If l<=0 (as set by default) partitions will have n summands

    while (true)
    {


        if (T.size()<=l)  // Testing if the length of patrition is <=l
            B.push_back(T); // and adding it if it is
        if (T.size()<l) // adding zeros if needed
            for (int e=0; e<l-T.size(); e++)
                B[B.size()-1].push_back(0);


        if (T[0]==n) break; // the last partition is the number n itself
        //if (T.size()==1) break;
        t=T[T.size()-2];
        for (int i=T.size()-2; i>=0; i--)
        {
            if (T[i]>t) {y=i+1; break;}

            y=i;
        }
```

```
        T[y]++; // transferring 1 to this summand (y-summand)
        T[T.size()-1]--; //from this summand
        if (T[T.size()-1]==0) T.pop_back();  // deleting it if became 0

        t=T.size();
        for (int z= y+1; z<t; z++) // now representing all the summands after y-
summand as a sum of ones only
        {
            if (T[z]==1) continue;
            T[z]--;
            T.push_back(1);
            z--;
        }


    }

    return 0;
}



bool CompStrDLO (const std::string & s1, const std::string & s2)  //Comparing
function for arranging an array (vector) of strings in descending length order /
Компаратор для сортировки строк по убыванию длин
{

  return s1.length() > s2.length();

}


std::string ShortSuperstringGr (std::vector <std::string> DataS)
// Generates shortest superstring of an array (vector) of strings DataS via
implementing greedy algorithm. In doing so, every string that is a substring of
any another one of DataS is to be excluded.
// DataS is copied (not linked) here as it will be changed here.
// Returns empty string if DataS is empty or all strings of DataS are empty.
// Применен "жадный алгоритм" поиска наименьшей надстроки. При этом из
рассмотрения исключаются строки, являющиеся подстроками других строк DataS.
// Исходные данные DataS копируются, а не привязываются по ссылке, т.к. DataS
будет изменяться в процессе работы функции
// Возвращается пустая строка, если DataS - пустой или содержит только пустые
строки.


{

    if (DataS.size()==0) return "";

    for (int y=0; y<DataS.size(); y++)  // deleting empty strings
    {
        if (DataS[y]=="")
        {
         DataS.erase(DataS.begin()+y);
         y--;
        }
    }

    if (DataS.size()==0) return "";
    if (DataS.size()==1) return DataS[0];

    std::sort (DataS.begin(), DataS.end(), CompStrDLO);
```

```
    for (int z=0; z<DataS.size()-1;z++)      // deleting every string that is a
substring of any another one
        {
            for (int zz=z+1; zz<DataS.size();zz++)
                if (DataS[z].find(DataS[zz])!=-1)
                {
                    DataS.erase(DataS.begin()+zz);
                    zz--;
                }
        }


    if (DataS.size()==1) return DataS[0];


    std::vector <std::vector <int>> Arrow;   // A matrix to contain overlap value
between strings i and j
    Arrow.clear();
    MatrixSet (Arrow, DataS.size(), DataS.size(), 0); // Заготовка матрицы величин
"перекрытий" между строками i и j

    // и ее заполнение величинами перекрытий: конца строки i и начала строки j
    // Filling the matrix Arrow with overlap value of end of i th and begin of j-
th strings

    int npos=0;
    int l1, l2;

       for (unsigned int i = 0; (i<DataS.size()); i++)
            for (unsigned int j = 0; (j<DataS.size()); j++)
            {
                l1 = DataS[i].length() ;
                l2 = DataS[j].length() ;
                npos=0;
                if (l1 > l2) npos = (l1-l2);

                if (npos==0)
                {
                     for (int y=0; y<l1; y++)
                        if ((DataS[i].substr(y, l1-y) == DataS[j].substr(0, l1-y))
&& (i!=j))
                        {Arrow [i] [j] = (l1-y); break;}

                }

                if (npos>0)
                {
                     for (int y=0; y<l2; y++)
                        if ((DataS[i].substr(npos+y, l1-y-npos) ==
DataS[j].substr(0, l1-y-npos)) && (i!=j))
                        {Arrow [i] [j] = (l1-y-npos); break;}

                }

            }
       //конец создания заполненной матрицы перекрытий
       //end of Arrow filling


        std::string Result = ""; //here result string will be
        std::string TempS;

       int mx = -1;
```

```
        int b, e;

        for (int t = 0; t<DataS.size()-1; t++)
        {
            TempS.clear();

            mx = 0;
            for (unsigned int i = 0; (i<DataS.size()); i++)    // ищем максимум mx
в 2-мерном массиве и его индексы b & e
                 for (unsigned int j = 0; (j<DataS.size()); j++)  // searching for
maximum of mx (i.e. overlap value), for such maximum indexes of overlapping
strings in DataS will be b and e.
                     if ((i!=j) && (Arrow [i] [j] >= mx)) {mx = Arrow [i] [j]; b =
i; e=j;}


            TempS = DataS[b].substr(0, DataS[b].length() - mx) + DataS[e]; //это -
склейка соотвествующих строк /Glueing of overlapping strings

            DataS[b] = TempS; //теперь ее пишем вместо b-й строки, дальше
перестроим наш двумерный массив / and this new glued string shold be write instead
of b-th string

            for (unsigned int ii = 0; (ii<DataS.size()); ii++)
                 Arrow [ii] [e] = -1;  //"обнуляем" столбец e - в строку e слева не
войдет больше ни одна строка / switching off e-th coloumn


            for (unsigned int jjj = 0; (jjj<DataS.size()); jjj++)
                 {Arrow [b] [jjj] = Arrow [e] [jjj];  //строку  e - в строку b,
строку e обнулить (теперь выходит все из b) / copying row e to b, switching off e-
th row
                 Arrow [e] [jjj] = -1;}

        }

        Result = TempS;

        return Result;

}
```

```
int TrieMake (std::vector <std::string> &DataS, std::vector <int> & Trie)
// Trie constructing upon vector of strings DataS
// Построение префиксного дерева Trie по массиву строк DataS

{
    Trie.clear(); // Here Trie will be contained as a number of triplets of
integers (a = Trie [3i], b = Trie [3i+1], c = Trie [3i+2], i = 0, 1, ...). Each
triplet means an edge a->b marked with symbol (char) c. Vertices in the Tree are
numerated starting with "1".
    // Здесь будет само дерево в виде набора триплетов чисел. Первые два задают
ребро графа, а третье - соотвествующий символ (букву). Вершины графа нумеруются с
1.

    if (DataS.size()<2) return -1; // Function works for at least 2 strings


    std::string TempS = "";
```

```
    sort (DataS.begin(), DataS.end(), CompStrDLO);

    int last = 0;
    int ind, l;


    for (int i=0;i<DataS[0].length();i++)   // формирование графа по 1 строке / Pushing to Trie the first (the longest) string from dataset
     {
         Trie.push_back(i+1);
         Trie.push_back(i+2);
         Trie.push_back( (int)DataS[0][i]  );
         last = (i+2);  // сохраняем номер последней (наибольшей) вершины / number of the very last vertex added (i.e. the maximal vertex number at any time)
     }




     for (int j=1;j<DataS.size();j++)  // добавляем остальные строки  / Pushing all the rest strings to Trie
     {


         TempS = DataS[j];
         ind = 1; // the number of vertex from which we are searching Trie for symbols matching (when matching is found "ind" will be the number of relevant sinr-vertex)/ с которой вершины ищем совпадающие символы (начинаем всегда с первой, затем, после добавления, берем вершину-сток из соответствующего ребра)
         l = 0; // counter of added symbols of every string being processed / счетчик добавленных символов по очередной строке

         lq: ;

         for (int q = 0; q<Trie.size(); q=q+3)  // searching Trie for symbols matching
         {
             if ((Trie[q]==ind)&&(Trie[q+2]==TempS[l]))
             {
                 ind = Trie[q+1];
                 l++;
                 if (l>=TempS.length()) // if the number of already added symbols = string's length, this means its adding to Trie is completed
                     goto lj;
                 goto lq; // if no - let us observe Trie from begining after adding another symbol

             }

         }

         for (int w = l; w<TempS.length(); w++) // if we haven't add all symbols of adding string at the previous step (searching Trie for symbols matching) - lets add the rest now
         {
             Trie.push_back(ind);
             Trie.push_back(last+1);
             ind = last+1;
             last = ind;
             Trie.push_back( (int)TempS[w]);

         }
```

```
         lj: ;
      }

   return 0;
}



void Num (std::string & Numbers, std::vector <double> & A)
{
    // перегон строки с числами <double> в массив (вектор) A
    // converts string of numbers <double> (separated by spaces) to a vector of
numbers

    A.clear();

    int q = 0;  // удаление лишних пробелов если есть / deletind doubled spaces
    while (Numbers.find ("  ", q) != -1)
    {
        q = Numbers.find ("  ", q);
        Numbers.erase(q, 1);
    }

    while (Numbers[0] == ' ')  // deleting spaces from the very beginning (string
must start from a number)
    {Numbers.erase(0, 1);}


    while (Numbers[Numbers.length()-1] == ' ')  // deleting spaces from the end
    {Numbers.erase((Numbers.length()-1), 1);}


    std::string TempS = "";
    int b=0; //начало каждого числа в строке  / The start position of a number
    int e=0; // конец числа в строке / the end position
    double r;  //сюда писать само число / a variable to contain a number
    while (Numbers.find (" ", b) != -1)  //число - до следующего пробела
    {
        e = Numbers.find (" ", b)-1;
        TempS = Numbers.substr(b, e-b+1);
         r = atof(TempS.c_str());
         A.push_back(r);

        b = e+2;
        TempS.clear();
    }

    TempS = Numbers.substr(b, Numbers.length()-b);  // еще одна итерация - от
последнего пробела до конца строки / the last iteration - up to the string's end
    r = atof(TempS.c_str());
    A.push_back(r);
    TempS.clear();
}



void Num (std::string & Numbers, std::vector <int> & A)
{
    // перегон строки с числами int в массив (вектор) A
    // converts string of numbers <int> (separated by spaces) to a vector of
numbers
```

```
    A.clear();

    int q = 0;  // удаление лишних пробелов / deleting doubled spaces
    while (Numbers.find ("  ", q) != -1)
    {
        q = Numbers.find ("  ", q);
        Numbers.erase(q, 1);
    }

    while (Numbers[0] == ' ')  // deleting spaces from the very beginning (string
must start from a number)
    {Numbers.erase(0, 1);}


    while (Numbers[Numbers.length()-1] == ' ')  // deleting spaces from the end
    {Numbers.erase((Numbers.length()-1), 1);}

    std::string TempS = "";
    int b=0; //начало каждого числа в строке  / The start position of a number
    int e=0; // конец числа в строке / the end position
    int r;  //сюда писать само число / a variable to contain a number
    while (Numbers.find (" ", b) != -1)  //число - до следующего пробела
    {
        e = Numbers.find (" ", b)-1;
        TempS = Numbers.substr(b, e-b+1);
         r = atoi(TempS.c_str());
         A.push_back(r);

        b = e+2;
        TempS.clear();
    }

    TempS = Numbers.substr(b, Numbers.length()-b);  // еще одна итерация - от
последнего пробела до конца строки / the last iteration - up to the string's end
    r = atoi(TempS.c_str());
    A.push_back(r);
    TempS.clear();
}



int Num (std::string & Numbers, int &a1,int &a2, double &a3)
// Вспомогательная функция: перегон строки, содержащей 3 числа, разделенных
пробелами (пара целых, задающих вершины ребра и одного double) соответственно в
int a1,int a2, double a3. Числа должны быть разделены пробелами, а более ничего
строка содержать не должна.
// Возвращает -1 если выявлена ошибка исходных данных (нет 3х "кандидатов в
числа").
// При этом проверка на то, что конвертируемая в число подстрока содержит лишь
цифры и десятичный разделитель, в данной версии функции НЕ проводится.
// Converts a string to 3 numbers (2 integers and 1 double; they should be
separated by spaces in the string and the string should not contain any other
symbols) to int &1,int a2, double a3.
// Returns -1 if input data is incorrect (no 3 "candidates to numbers" are found).
// But note that here is NO checking if a substring to be converted to a number
contains digits and decimal point only.


{

    int q = 0;  // удаление лишних пробелов если есть / deletind doubled spaces
    while (Numbers.find ("  ", q) != -1)
    {
```

```
        q = Numbers.find ("  ", q);
        Numbers.erase(q, 1);
    }

    while (Numbers[0] == ' ')  // deleting spaces from the very beginning (string
must start from a number)
    {Numbers.erase(0, 1);}


    while (Numbers[Numbers.length()-1] == ' ')  // deleting spaces from the end
    {Numbers.erase((Numbers.length()-1), 1);}


    std::string TempS = "";
    int b=0; //начало каждого числа в строке  / The start position of a number
    int e=0; // конец числа в строке / the end position

    e = Numbers.find (" ", b)-1; //reading the first int - i.e. the first vertex
of an edge
    if (e<0) return -1;
    TempS = Numbers.substr(b, e-b+1);

    a1 = atoi(TempS.c_str());
    b = e+2;TempS.clear();


    e = Numbers.find (" ", b)-1; //reading the second int - i.e. the second vertex
of an edge
    if (e<0) return -1;
    TempS = Numbers.substr(b, e-b+1);

    a2 = atoi(TempS.c_str());
    b = e+2;TempS.clear();


    if (b>=Numbers.length()) return -1; // in this case string does not contain
the 3th number

    e = Numbers.find (" ", b)-1;

    if (e<0) //There are no more " "
    {
        TempS = Numbers.substr(b, Numbers.length()-b);
        a3 = atof(TempS.c_str());
    }


    return 0;
}




int UWGraphRead (std::ifstream & fin, std::vector <int> & A)
// Чтение невзвешенного графа в вектор смежности.
// Назовем вектором смежности для взвешенного графа упорядоченный набор (массив)
четного кол-ва чисел (a[2i], a[2i+1],... / i нумеруется с 0 /),
// где каждая пара чисел a[2i], a[2i+1] задает ребро графа между вершинами a[2i] и
a[2i+1] ("список ребер в строку").
// Данный формат не содержит информации, является ли граф ориентированным или нет
(возможны оба варианта). При использовании формата для орграфа считается, что
ребро направлено из a[2i] в a[2i+1].
// Предполагается считывание из файла, содержащего список ребер (каждое ребро -
отдельная строка)
```

```
// Возвращает -1 и пустой вектор A, если полученный вектор смежности пустой или же при считывании очередного ребра считано не 2 элемента (числа)

// Reads Edge list to "Adjacency vector" of unweighted graph. Let "Adjacency vector" of unweighted graph be a data structure,
// that contains array of integers such as a[2i], a[2i+1],... / 0-basing indexing in array /.
// So such array contains even number of elements. Every pair a[2i], a[2i+1] means an edge between vertex a[2i] and a[2i+1] (~ "Edge list as one String").
// This format don't identify the graph as directed or undirected (both cases may be). If the graph is considered as directed, its edges should be considered as a[2i] -> a[2i+1].
// Input file should be in edge list format, every edge in new line.
// Returns -1 and empty "Adjacency vector" A if any line contains number of elements that !=2.


{
    std::string TempS = "";
    A.clear();
    std::vector <int> B1;

    while (!fin.eof())
    {
        B1.clear();
        getline (fin, TempS);
        if (TempS.length()!=0)
        {
             Num(TempS, B1);
             if (B1.size()!=2) {A.clear(); return -1;}
             A.push_back(B1[0]);
             A.push_back(B1[1]);
        }
    }

    if (A.size()==0) return -1;

    return 0;
}




int WGraphRead (std::ifstream & fin, std::vector <int> & A)
// Чтение взвешенного графа в вектор смежности. Назовем вектором смежности для взвешенного графа упорядоченный набор (массив) чисел (a[3i], a[3i+1], a[3i+2],... / i нумеруется с 0 /), где каждая тройка чисел a[3i], a[3i+1] задает ребро графа между вершинами a[3i] и a[3i+1], а a[3i+2] есть вес этого ребра, ("список ребер в строку").
// Рассматриваемый формат не содержит информации, является граф ориентированным или нет (возможны оба варианта). При использовании формата для орграфа считается, что ребро направлено из a[3i] в a[3i+1].
// Данная структура данных занимает меньше памяти, чем матрица смежности, и может быть удобна для решения ряда задач.
// Предполагается считывание из файла, содержащего список смежности (каждое ребро - отдельная строка)
// Возвращает -1 и пустой вектор A, , если полученный вектор смежности пустой или же при считывании очередного ребра считано не 3 элемента (числа)

// Reads Edges list to "Adjacency vector" of weighted graph. Let "Adjacency vector" of weighted graph be a data structure, that contains array of integers such as a[3i], a[3i+1], a[3i+2],... / 0-basing indexing in array /.
```

```
// So such array contains 3n number of elements. Every pair a[3i], a[3i+1] means
an edge between vertex a[3i] and a[3i+1] with weight a[3i+2]("Edge list as one
String").
// This format don't identify the graph as directed or undirected (both cases may
be). If the graph is considered as directed, its edges should be considered as
a[3i] -> a[3i+1].
// Input file should be in edge list format, every edge in new line.
// Returns -1 and empty "Adjacency vector" A if any line contains number of
elements of any line that !=3.


{
    std::string TempS = "";
    A.clear();
    std::vector <int> B1;

    while (!fin.eof())
    {
        B1.clear();
        getline (fin, TempS);
        if (TempS.length()!=0)
        {
             Num(TempS, B1);
             if (B1.size()!=3) {A.clear(); return -1;}
             A.push_back(B1[0]);
             A.push_back(B1[1]);
             A.push_back(B1[2]);
        }
    }

    if (A.size()==0) return -1;

    return 0;
}



int WGraphRead (std::ifstream & fin, std::pair < std::vector<int>,
std::vector<double>> & A)
// Модификация функции WGraphRead (см. выше) для случая нецелочисленных весов
ребер (double).
// Чтение проводится в пару векторов pair < std::vector<int>, std::vector<double>>
& A, где первый вектор является вектором смежности считываемого графа без указания
весов,
// а второй вектор содержит соотвествующие веса. Соотвественно для ребра
задаваемого парой вершин под индексами 2*i, 2*i+1 первого вектора вес будет равен
элементу под индексом i второго вектора.

// Modification of the function WGraphRead (see it above) for not-integer (double)
weihgts of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an
"Adjacency vector" without weights. But weigths are set in the second one.
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the
first vector has its weight set as i-th element in the second one.


{
    std::string TempS = "";
    (A.first).clear();
    (A.second).clear();

   int a1, a2;
   double a3;
```

```
    while (!fin.eof())
    {
        getline (fin, TempS);
        if (TempS.length()!=0)
        {
             if (Num(TempS, a1, a2, a3)==-1)
             {
                 (A.first).clear();
                 (A.second).clear();
                 return -1;
             }

             if (Num(TempS, a1, a2, a3)==0)
             {
                (A.first).push_back(a1);
                (A.first).push_back(a2);
                (A.second).push_back(a3);


             }
        }
    }

    if ((A.first).size()==0) return -1;

    return 0;
}
```

```
int RangeVGraph (std::vector <int> & A, int & mx, int & mn, const bool weighted,
bool IgnoreWeighted = false)
//Finds max (i.e. mx) and min (i.e. mn) value of numbers that assigned to vertices
// Graph must be set as "Adjacency vector", bool "weighted" sets if the graph is
weighted or no.
// If (IgnoreWeighted = true) the function looks at every element in A without any
dataset checking
{

    mn = INT_MAX;
    mx = INT_MIN;

    if (A.size()==0) return -1;

    if (IgnoreWeighted == false)
    {
    if ( (A.size())%(2+weighted)!=0 ) return -1;

    for (int q=0; q<A.size()-1-weighted; q= q+2+weighted)
    {
        if (A[q]>mx) mx = A[q];
        if (A[q+1]>mx) mx = A[q+1];

        if (A[q]<mn) mn = A[q];
        if (A[q+1]<mn) mn = A[q+1];
    }

    }

    if (IgnoreWeighted == true)
```

```
    {
        for (int q=0; q<A.size(); q++)
        {
            if (A[q]>mx) mx = A[q];
            if (A[q]<mn) mn = A[q];
        }

    }

    return 0;
}



int RenumVGraph (std::vector <int> & A, const int d, const bool weighted, bool
IgnoreWeighted = false)
//Renumerates vertices adding d-parameter (d may be non-negative or negative) /
Перенумеровывавает вершины графа: прибавляет величину d (может быть положительной
и отрицательной)
// Graph must be set as "Adjacency vector", bool "weighted" sets if the graph is
weighted or no.
// If (IgnoreWeighted = true) the function adds d to every element in A without
any dataset checking


{

    if (A.size()==0) return -1;

    if (IgnoreWeighted == false)
    {
        if ( (A.size())%(2+weighted)!=0 ) return -1;

        for (int q=0; q<A.size()-1-weighted; q= q+2+weighted)
        {
            A[q] = A[q]+d;
            A[q+1] = A[q+1]+d;
        }
    }

    if (IgnoreWeighted == true)
    {
        for (int q=0; q<A.size(); q++)
        {
            A[q] = A[q]+d;

        }

    }



    return 0;
}



int AdjVector2AdjMatrix (std::vector <int> & A, std::vector <std::vector <int>>
&B, const bool weighted, const bool directed)
//Converts "Adjacency vector" to "Adjacency matrix".
// bool "weighted" sets if the graph is weighted or no. bool "directed" sets if
the graph is directed or no.
```

```
// In case of multiple edges for a weighted graph only the last edge will be
written to Adjacency matrix, others will be lost.
// Loops for undirected unweighted graph counts as 2 edges
// In this function zero-value of any item of Adjacency matrix means no edge both
for unweighted and weighted graph

{

    B.clear();
    if (A.size()==0) return -1;
    if ( (A.size())%(2+weighted)!=0 ) return -1;

    int mx, mn;

    RangeVGraph (A, mx, mn, weighted);

    if (mn<0) return -1;

    MatrixSet(B, mx+1, mx+1, 0);

    for (int q=0; q<A.size()-1-weighted; q= q+2+weighted)
    {
        if ( (weighted==false) && (directed==true) ) B[(A[q])][(A[q+1])]++;
        if ( (weighted==true) && (directed==true) ) B[(A[q])][(A[q+1])]= A[q+2];

        if ( (weighted==false) && (directed==false) ) {B[(A[q])][(A[q+1])]++;
B[(A[q+1])][(A[q])]++;}
        if ( (weighted==true) && (directed==false) ) {B[(A[q])][(A[q+1])]= A[q+2];
B[(A[q+1])][(A[q])]= A[q+2];}

    }
    return 0;
}



int AdjVector2AdjMatrix (std::pair < std::vector<int>, std::vector<double>> & A,
std::vector <std::vector <double>> &B, const bool directed)
// Modification of the function AdjVector2AdjMatrx (see it above) for not-integer
(double) weights of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an
"Adjacency vector" without weights. But weights are set in the second one.
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the
first vector has its weight set as i-th element in the second one.
// Note that undirected graph may have only zeros lower than the Main diagonal of
its Adjacency matrix here

{
    B.clear();
    if ((A.first).size()==0) return -1;
    if ((A.second).size()==0) return -1;
    if (  (A.first).size()!=((A.second).size())*2 ) return -1;

    int mx, mn;

    RangeVGraph (A.first, mx, mn, false);

    if (mn<0) return -1;

    MatrixSet(B, mx+1, mx+1, 0.0);

    for (unsigned int q=0; q<(A.second).size(); q++)
    {
```

```
        if ( directed==true ) B[((A.first)[2*q])][((A.first)[2*q+1])]=
(A.second)[q];

        if (directed==false)
        {
            if ((A.first)[2*q]<=(A.first)[2*q+1])
B[(A.first)[2*q]][(A.first)[2*q+1]]= (A.second)[q];
            if ((A.first)[2*q]>(A.first)[2*q+1])
B[(A.first)[2*q+1]][(A.first)[2*q]]= (A.second)[q];
        }

    }

    return 0;
}



int AdjMatrix2AdjVector (std::vector <int> & A, const std::vector <std::vector
<int>> &B, const bool weighted, const bool directed)
// Converts "Adjacency matrix" to "Adjacency vector".
// bool "weighted" sets if the graph is weighted or no. bool "directed" sets if
the graph is directed or no.
// For a weighted graph here are no multiple edges.
// Loops for an undirected unweighted graph counts as 2 edges
// For an undirected graph the data that is lower than the Main diagonal of the
matrix B is ignored
// In this function zero-value of any item of Adjacency matrix means "no such
edge" both for unweighted and weighted graph


{

    A.clear();
    if (B.size()==0) return -1;

    for (int y=0; y<B.size(); y++)  // lets test: if the matrix B is a "square"
        if (B.size()!=B[y].size()) return -1; // тест на квадратность


    int t;
    int c = 0;

    for (int i=0; i<B.size(); i++)
    {
        if (directed==false) c = i;
        for (int j=c; j<B.size(); j++) // for undirected graph lets see only not
lower than Main diagonal
        {
            t = B[i][j];
            if (t==0) continue;

            if ( (weighted==false)&&(directed==true)  )
            {
                if (t<0) {A.clear(); return -1;}

                for (int x = 0; x<t; x++)
                {
                    A.push_back(i);
                    A.push_back(j);
                }
            }

            if ( (weighted==true)&&(directed==true)  )
```

```
            {
                    A.push_back(i);
                    A.push_back(j);
                    A.push_back(t);
            }


            if ( (weighted==false)&&(directed==false)  )
            {
                if (t<0) {A.clear(); return -1;}

                if ((i==j) && ((t%2)!=0) )  {A.clear(); return -1;} // Loops for
undirected unweighted graph counts as 2 edges
                if ((i==j)) t = t/2;



                for (int x = 0; x<(t); x++)
                {
                    A.push_back(i);
                    A.push_back(j);
                }

            }

            if ( (weighted==true)&&(directed==false)  )
            {
                A.push_back(i);
                A.push_back(j);
                A.push_back(t);
            }


        }

    }

    return 0;
}



int AdjMatrix2AdjVector (std::pair < std::vector<int>, std::vector<double>> & A,
const std::vector <std::vector <double>> &B, const bool directed)
// Modification of the function AdjMatrix2AdjVector (see it above) for not-integer
(double) weights of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an
"Adjacency vector" without weights. But weights are set in the second one.
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the
first vector has its weight set as i-th element in the second one.
// For an undirected graph the data that is lower than the Main diagonal of the
matrix B is ignored

{

    (A.first).clear();
    (A.second).clear();

    if (B.size()==0) return -1;

    for (unsigned int y=0; y<B.size(); y++)  // lets test: if the matrix B is a
"square"
        if (B.size()!=B[y].size()) return -1; // тест на квадратность
```

```
    double t;
    int c = 0;

    for (int i=0; i<B.size(); i++)
    {
        if (directed==false) c = i;
        for (int j=c; j<B.size(); j++) // for undirected graph lets see only not
lower than Main diagonal
        {
            t = B[i][j];
            if (t==0.0) continue;


            if (directed==true)
            {
                    (A.first).push_back(i);
                    (A.first).push_back(j);
                    (A.second).push_back(t);
            }


            if (directed==false)
            {

                (A.first).push_back(i);
                (A.first).push_back(j);
                (A.second).push_back(t);
            }
        }
    }

    return 0;
}



int AdjVectorToAdjMap (const std::vector <int> &A, std::map <std::pair < int, int>
, int> &G2, const bool weighted)
// Converts Adjacency vector A to Adjacency map G2. Multiple edges will be joined
together.
// Parameter "weighted" sets if the graph A is weighted or no. Weights may be only
integers. If A is unweighted we consider that every edge have its weight = 1.
// Returns -1 if input data is not correct. Otherwise returns 0.
// Конвертирует Вектор смежности A в ассоциативный массив смежности G2.
Множественные ребра будут объединены с суммарным весом. Для невзвешенного графа
считаем вес всех ребер = 1.
// Возвращает -1 в случае некорректности исходных данных.
// Параметр weighted задает, является ли граф взвешенным (Истина) или нет.


{
    G2.clear();
    if (A.size()==0) return -1;
    if ( (A.size())%(2+weighted)!=0 ) return -1; // checking for input data
correctness

    std::pair < int, int> C;
    std::pair < std::pair < int, int>, int> D;

    if (weighted)
    {
    for (int i=0; i<A.size(); i=i+3)
    {
```

```
        C = std::make_pair(A[i], A[i+1]);
        D = std::make_pair(C, A[i+2]);

        if (G2.find(C)!=G2.end())
        {G2[C] = G2[C]+A[i+2]; continue;}

        if (G2.find(C)==G2.end())
        {G2.insert(D); continue;}


    }

    }

    if (!weighted)
    {
    for (int i=0; i<A.size(); i=i+2)
    {
        C = std::make_pair(A[i], A[i+1]);
        D = std::make_pair(C, 1);

        if (G2.find(C)!=G2.end())
        {G2[C] = G2[C]+1; continue;}

        if (G2.find(C)==G2.end())
        {G2.insert(D); continue;}

    }

    }

    return 0;

}



int AdjVectorToAdjMap (const std::pair < std::vector<int>, std::vector<double>> &
A, std::map <std::pair < int, int> , double> &G2)
// Converts Adjacency vector A to Adjacency map G2. Multiple edges will be joined
together.
// Returns -1 if input data is not correct. Otherwise returns 0.
// Конвертирует Вектор смежности A в ассоциативный массив смежности G2.
Множественные ребра будут объединены с суммарным весом.
// Возвращает -1 в случае некорректности исходных данных.

{
    G2.clear();

    if ((A.first).size()==0) return -1;
    if ((A.second).size()==0) return -1;
    if (  (A.first).size()!=((A.second).size())*2 ) return -1;


    std::pair < int, int> C;
    std::pair < std::pair < int, int>, double> D;


    for (int i=0; i<(A.second).size(); i++)
    {
        C = std::make_pair((A.first)[i*2], (A.first)[2*i+1]);
        D = std::make_pair(C, (A.second)[i]);
```

```
        if (G2.find(C)!=G2.end())
        {G2[C] = G2[C]+(A.second)[i]; continue;}

        if (G2.find(C)==G2.end())
        {G2.insert(D); continue;}

    }



    return 0;

}



int AdjMapToAdjVector (std::vector <int> &A, const std::map <std::pair < int, int>
, int> &G1)
// Converts Adjacency map G1 to Adjacency vector A. A is considered as weighted,
all weights are integers.
// Returns -1 if input data is not correct. Otherwise returns 0.
// Конвертирует Ассоциативный массив смежности G1 в вектор смежности A (только во
взвешенный, веса целочисленны).
// Возвращает -1 в случае некорректности исходных данных.



{
    A.clear();
    if (G1.size()==0) return -1;
    for (auto it =G1.begin(); it!=G1.end(); it++)
    {
        A.push_back(  (it->first).first);
        A.push_back(  (it->first).second);
        A.push_back  (it->second);

    }

    return 0;
}



int AdjMapToAdjVector (std::pair < std::vector<int>, std::vector<double>> & A,
const std::map <std::pair < int, int> , double> &G1)
// Converts Adjacency map G1 to Adjacency vector A. A is considered as weighted,
all weights are double.
// Returns -1 if input data is not correct. Otherwise returns 0.
// Конвертирует Ассоциативный массив смежности G1 в вектор смежности A (только во
взвешенный, веса имеют тип double).
// Возвращает -1 в случае некорректности исходных данных.


{
    (A.first).clear();
    (A.second).clear();
    if (G1.size()==0) return -1;
    for (auto it =G1.begin(); it!=G1.end(); it++)
    {
        (A.first).push_back(  (it->first).first);
        (A.first).push_back(  (it->first).second);
        (A.second).push_back  (it->second);

    }
```

```
    return 0;
}



int CheckUnvisit (std::vector <int> & Visited) // Вспомогательная функция для поиска первой непомеченной вершины в графе
// An auxiliary function that finds the first unmarked vertex in the graph (0 means unmarked)

{
    int b = -1;

    for (unsigned int w = 0; w<Visited.size(); w++)
        if (Visited[w] ==0)   // ищем номер первой необойденной вершины
        {b=w; break;}

    return b;
}



void EcycleDGraph (int t, std::vector <int> & R, const int V, std::vector <std::vector<int>> &B)
// Вспомогательная функция для поиска Эйлерова цикла в ОРИЕНТИРОВАННОМ графе, где он заведомо существует, нет изолированных вершин и нумерация вершин идет с 1.
// B - матрица смежности, содержащая кол-во ребер между вершинами, V - максимальный номер вершины

// An auxiliary function that finds Eulerian cycle in the DIRECTED graph without without checking of input data correctness
//(i.e. (1) the graph includes Eulerian cycle, (2) its vertices numbers start from "1", (3) the graph doesn't contain any isolated vertices).
// B is the Adjacency matrix, containing the number of edges between the vertices. V is the max number assigned to vertices.



{

    int f = 1;

    while (f!=0)  // Строим начиная с вершины за номером t путь и вычеркиваем пройденные ребра // Building up the path from vertex № t
    {

        for (int i = 1; i<(V+1); i++)
        {
            if (B[t][i]>=1)
            {

            R.push_back(t);
            B [t][i]--; // ребро вычеркнули // deleting edge
            t = i; // дальше ищем со следующей вершины // tet's continue from the end vertex of the deleted edge
            goto l1;
            }

        }
        f = 0;
```

```
        l1: ;



    }

    std::vector <int> T;
    T.clear();

    if (R.size()!=0)  // а теперь ищем замкнутые циклы ("пузыри"), относящиеся к какой-либо вершине из уже построенного в R пути. И так рукуррентно ("пузырь" может содержать еще "пузырь" и т.д.)
        // Now we should search for cycles related to every vertex of the path in the vector R recurrently.

    {

    for (unsigned int j = 0; j<R.size(); j++)
    {
        T.clear();
        EcycleDGraph (R[j], T, V, B);

        if (T.size()!=0)  // И если такой "пузырь" есть // And if we have found such cycle
        {
        T.push_back(R[j]); // lets finish the cycle by pushing back R[j] - it is both its begin and its end.
        for (unsigned int w = (j+1); w<R.size(); w++) // And then lets add to it the rest of the path. So we have in T the Path with cycle related to R[j]-vertex added.
            T.push_back(R[w]); // Дополним его значением R[j] (это его и начало и конец) и затем присовокупим остаток пути из R. Это и будет изначальный путь, в который вставили путь по "пузырю" из вершины R[j]


        R.resize(j); // Теперь обновим сам R. Обрежем все, что дальше вершины R[j] и добавим вместо этого тоткорректированный путь из T
        for (unsigned int e = 0; e<T.size(); e++) // Now it is time to update R. Lets cut all that is further than R[j] and add updated path from T (it contains cycle related to R[j]-vertex now)
            R.push_back(T[e]);
        T.clear();

        }


    }

    }

}




int EPathDGraph (std::vector <int> & A, std::vector <int> & R, const bool weighted, std::vector <int> & Isolated)
// Поиск Эйлерова пути либо Эйлерова цикла в ОРИЕНТИРОВАННОМ графе. Принимает на вход вектор смежности графа с указанием, взвешенный ли граф, а также заготовку R для найденного пути (цикла) и Isolated для изолированных вершин.
// При этом не считается изолированной вершина, имеющая лишь петли.
```

```
// Возвращает заполненные R и Isolated (если есть путь либо цикл, при этом возвращаемые значения соответственно 2 и 1) и пустые вектора и -1, если их не найдено.
// Эйлеров путь/ цикл ищется на всем графе, либо на единственной компоненте связности, при условии что прочие вершины - изолированные.
// Может работать с ориентированными графами с дублирующими ребрами и с множественными петлями. Нумерация вершин может осуществляться любыми целыми числами, в т.ч. отрицательными. При этом считается, что граф содержит все вершины, соответствующие всем числам от min (1, минимальный заданный номер вершины) по максимальный заданный номер вершины включительно.
// В процессе работы граф приводится к виду, чтобы вершины нумеровались начиная с 1. По окончанию работы исходная нумерация восстанавливается.

// Finding Eulerian Cycle or Path in directed graph (weighted or non-weighted) that may contain multiple edges and multiple loops.
// Returns Path/ Cycle as R, isolated vertices as Isolated. Returns value "1" if Eulerian cycle has been found or value "2" if Eulerian path has been found or "-1" together with empty R and Isolates if no cycle/ path found.
// If any vertex has loops only, such a vertex is not considered as an isolated one.
// Vertices may be numbered in different ways (they may be marked by both negative and non-negative integers). In doing so, we set that the graph contains vertices marked by all the integers from min (1, minimal number assigned to vertices) to maximal number assigned to vertices inclusive.
// In order to implement the function vertices may be renumbered to get started from "1"; after search is completed, the vertices will be assigned their original numbers.


{
    R.clear(); // здесь будет искомый цикл/ путь // vector for Eulerian cycle/ path
    Isolated.clear(); // Здесь будут храниться найденные изолированные вершины // vector for isolated vertices.


    if (A.size()==0) return -1;
    if ( (A.size())%(2+weighted)!=0 ) return -1; // checking for input data correctness

    int V; // здесь будет максимальный номер вершины // the max number of assigned to vertices
    int E = A.size()/(2+weighted); // the total number of edges in the graph


    int mn;
    RangeVGraph (A, V, mn, weighted);

    if (mn<1)  // Приведение вектора к нумерованию вершин с 1 // renumbering vertices to start from 1.
    {
        RenumVGraph (A, (1-mn), weighted);
        V = V+(1-mn);
    }

    std::vector <int> Visited (V+1, 0); // нулевой элемент использовать не будем: нумеруются вершины с 1
    Visited [0] = 1;

    std::vector <int> Vin(V+1, 0); //для подсчета входящих и исходящих в вершину
    std::vector <int> Vout (V+1, 0); // for counting in-edges and out-edges

    for (int q=0; q<A.size()-1-weighted; q= q+2+weighted)
    {
```

```
        Vin[(A[q+1])]++;
        Vout[(A[q])]++;
    }

    int t1, t2;
    int c=0;
    int c0=0;
    int c2=0;
    int c1=0;

    for (int a = 1; a<=V; a++)
    {
        if ((Vin [a]==0) && (Vout [a]==0) ) {Visited
[a]=2;Isolated.push_back(a);c0++;continue;} // такая вершина - изолированная //
such a vertex is isolated
        if (Vin [a]==Vout [a]) {c++; continue;} // у такой входы = выходам,
считаем их //such a vertex has the number of in-edges = the number of out-edges.
Lets count these vertices.
        if ((Vin [a]-Vout [a])==1) {c2++; t2 = a; continue;} // у такой на 1
больше входов, считаем их и запоминаем последнюю/ // such a vertex has the number
of in-edges - the number of out-edges = 1. Lets count them and remember the last
one.
        if ((Vin [a]-Vout [a])==-1) {c1++; t1 = a; continue;} // а у такой -
выходов, считаем их и запоминаем последнюю// such a vertex has the number of in-
edges - the number of out-edges = -1. Lets count them and remember the last one.


    }


    std::vector <std::vector <int>> B;
    MatrixSet(B, V+1, V+1, 0);


    for (unsigned int x = 0; x<A.size()-1-weighted; x = x+2+weighted)  //
Формируем матрицу смежности B, содержащую ко-во ребер даже для взвешенных графов
        // Forming Adjacency matrix and filling it with the number of edges
between vertices.
    {
        B[(A[x])][(A[x+1])]++;

    }

    int t = CheckUnvisit(Visited); // начнем искать путь/цикл с любой
неизолированной вершины // Lets start search from any non-isolated vertex
    sort (Isolated.begin(), Isolated.end());

    if ( ((c+c0)==V) && (c>=1) )  //In this case we may have a Cycle
    {
        EcycleDGraph(t, R, V, B); // Формируем цикл // find the cycle
        if (mn<1) // и если надо перенумеровываем вершины назад если необходимо //
Renumbering all the vertices back if needed
        {
            RenumVGraph (A, (-1+mn), weighted);
            RenumVGraph (R, (-1+mn), false, true);
            RenumVGraph (Isolated, (-1+mn), false, true);
        }
        R.push_back(R[0]); // в цикле начало = концу

        if ((R.size()-1)!=E) {Isolated.clear(); R.clear(); return -1;} // all the
edges must be in R, the total number of edges = E

        return 1;  //This means "Found Eulerian cycle"
    }
```

```
    if (((c+c0+c1+c2)==V) && (c1==1) && (c2==1)) // необходимое условие наличия
пути // in this case we may have Path
    {

        B[t2][t1]++; // добавим недостающее до цикла ребро // adding edge in order
to complete path to cycle
        EcycleDGraph(t1, R, V, B);  // find the cycle

        R.push_back(R[0]);

        for (int i = 0; i<(R.size()-1); i++)  // rearranging cycle to path
            if ((R[i] == t2) && (R[i+1] == t1))
            {
                R.pop_back();
                for (int q = (i+1);q<R.size();q++)
                {    R.insert(R.begin(), R[R.size()-1]); R.pop_back();}

            }


        if (mn<1) // Renumbering all the vertices back if needed
        {

            RenumVGraph (A, (-1+mn), weighted);
            RenumVGraph (R, (-1+mn), false, true);
            RenumVGraph (Isolated, (-1+mn), false, true);

        }

        if ((R.size()-1)!=E) {Isolated.clear(); R.clear(); return -1;} // all the
edges must be in R, the total number of edges = E

        return 2; // This means "Found Eulerian path"

    }


    Isolated.clear();
    R.clear();
    if (mn<1) // Renumbering all the vertices back if needed
    {

        RenumVGraph (A, (-1+mn), weighted);

    }

    return -1;
}


int EPathDGraph (std::pair < std::vector<int>, std::vector<double>> & A,
std::vector <int> & R, std::vector <int> & Isolated)
// Модификация функции EPathDGraph (см. выше) для случая нецелочисленных весов
ребер (double).
// Modification of the function EPathDGraph (see it above) for not-integer
(double) weights of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an
"Adjacency vector" without weights. But weights are set in the second one.
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the
first vector has its weight set as i-th element in the second one.
```

```
{
    R.clear(); // здесь будет искомый цикл/ путь // vector for Eulerian cycle/
path
    Isolated.clear(); // Здесь будут храниться найденные изолированные вершины //
vector for isolated vertices.

    if ((A.first).size()==0) return -1;
    if ((A.second).size()==0) return -1;
    if (  (A.first).size()!=((A.second).size())*2 ) return -1;

    return EPathDGraph (A.first, R, false, Isolated);

}



int DFSTS (const std::vector <int> & A, const int b, std::vector <int> & Visited,
std::vector <int> & order, const bool weighted)
// Вспомогательная функция для функции TSortHP. Проверки исходных данных не
проводится, вершины в ориентированном орграфе, заданном вектором смежности A, д.б.
нумерованы с 1.
// Граф может содержать петли (игнорируются).
// В процессе обхода раскрашиваем вершины в массиве Visited: 0 - непосещенная
(белая), 1 - посещена, но не отработана (серая), 2 - отработана (черная).
// weighted - истина, если взвешенный граф, иначе - ложь.
// Если найден цикл - возвращает 1 и пустой order.

// An auxiliary function for the function TSortHP. Works without any checking of
input data correctness. Vertices in the input directed graph (it is set by the
Adjacency vector A) are to be numbered starting from 1.
// The graph may contain loops (they will be ignored).
// During building topological sorting we shall colour vertices (using vector
Visited): 0 = unvisited (white), 1 = visited, but not still finished yet (grey), 2
= finished (black).
// Bool "weighted" should be set as "true" for weighted graph, "false" for
unweighted.
// If the graph contains cycle - returns 1 and empty "order".



{

    Visited [b] = 1;  // b - the vertex to start with. Vertex is to become grey
when starting working with it
    int f;


    for (unsigned int r = 0; r+1+weighted<=A.size(); r=r+2+weighted)
    {


        if ( (A[r]==b) && (A[r+1]==b) ) continue; // Если петля - идем дальше //if
a loop found - let's continue, loop will be ignored

        if ( (A[r]==b) && (Visited [ (A[r+1]) ] == 1) )  // in this case a cycle
is found
        {
            order.clear();
            return 1;
        }


        if ( (A[r]==b) && (Visited [ (A[r+1]) ] == 0) )  // нашли непосещенную?
непосещенная = 0  //found a non-visited vertex
```

```
        {
           Visited[(A[r+1])] = 1;
           f=DFSTS (A, (A[r+1]), Visited, order, weighted);
            if (f==1)
            {
                order.clear();
                return 1;
            }

        }


    }

    order.push_back(b);
    Visited [b] = 2;  // отработали и перекрасили в черный (=2) // now the vertex
is to become black (=2)

return 0;
}


int TSortHP (std::vector <int> & A, std::vector <int> & R, std::vector <int> &
order, std::vector <int> & Isolated, const bool weighted, const bool OnlyTS =
false)
// Функция для топологической сортировки в орграфе. Также в случае наличия
топологической сортировки (и при условии OnlyTS = false) ищет Гамильтонов путь и
перечень изолированных вершин. При этом не считается изолированной вершина,
имеющая лишь петли.
// Функция НЕ является функцией поиска именно Гамильтонова пути, он ищется ТОЛЬКО
в случае наличия топологической сортировки.
// Принимает на вход вектор смежности графа A с указанием, взвешенный ли граф
(параметр weighted), а также заготовку R для Гамильтонова пути, order для
топологической сортировки, Isolated для перечня изолированных вершин.
// Может работать с ориентированными графами с дублирующими ребрами и с
множественными петлями (петли будут игнорироваться).
// Нумерация вершин может осуществляться любыми целыми числами, в т.ч.
отрицательными. При этом считается, что граф содержит все вершины, соответствующие
всем числам от min (1, минимальный заданный номер вершины) по максимальный
заданный номер вершины включительно.
// В процессе работы граф приводится к виду, чтобы вершины нумеровались начиная с
1. По окончанию работы исходная нумерация восстанавливается.
// Если OnlyTS == false (нормальная работа функции):
// Возвращает 0, если найдены и топологическая сортировка, и Гамильтонов путь.
// Возвращает -1 и пустые R, order, Isolated, если в графе найден цикл.
// Возвращает 1 и пустой R, если есть топологическая сортировка, а Гамильтонова
пути нет.
// Если параметр OnlyTS == true, то ищется только топологическая сортировка
(данный режим предусмотрен для ускорения работы). Возвращает 0, если она найдена и
-1 если нет. Гамильтонов путь и изолированные вершины не возвращаются (R и
Isolated будут пусты в любом случае).

// The function finds topological sorting of directed graph (returned as vector
"order").
// ONLY IF topological sorting exists AND OnlyTS == false the function also checks
for Hamiltonian path (returned as vector R) and list of Isolated vertices
(returned as vector Isolated).
// The graph is set by Adjacency vector A, may be weighted or no (bool weighted).
// The graph may contain loops (they will be ignored).
// If any vertex has loops only, such a vertex is not considered as an isolated
one.
// The graph may contain multiple edges.
// Vertices may be numbered in different ways (they may be marked by both negative
and non-negative integers). In doing so, we set that the graph contains vertices
```

```
marked by all the integers from min (1, minimal number assigned to vertices) to
maximal number assigned to vertices inclusive.
// In order to implement the function vertices may be renumbered to get started
from "1"; after search is completed, the vertices will be assigned their original
numbers.
// So if OnlyTS == false:
// the function returns 0 if both topological sorting and Hamiltonian path found.
// the function returns -1 and empty Isolated, order, R if the graph contains
cycle.
// the function returns 1 if topological sorting found and, upon that, Hamiltonian
path doesn't exist.
// If OnlyTS == true, both R and Isolated will be returned empty (to make this
function faster). The function returns 0 if topological sorting is found and -1
otherwise.



{
    R.clear(); order.clear(); Isolated.clear();



    if (A.size()==0) return -1;
    if ( (A.size())%(2+weighted)!=0 ) return -1; // checking for input data
correctness

    int Vg; // здесь будет максимальный номер вершины // the max number of
assigned to vertices
    int E = A.size()/(2+weighted); // the total number of edges in the graph


    int mn;
    RangeVGraph (A, Vg, mn, weighted);

    if (mn<1)  // Приведение вектора к нумерованию вершин с 1 // renumbering
vertices to start from 1.
    {
        RenumVGraph (A, (1-mn), weighted);
        Vg = Vg+(1-mn);
    }

    if (!OnlyTS)
    {
    std::vector <int> Vin(Vg+1, 0); //для подсчета входящих и исходящих в вершину
    std::vector <int> Vout (Vg+1, 0); // for counting in-edges and out-edges


    // Поиск изолированных
    // Let's find isolated vertices

    for (int q=0; q<A.size()-1-weighted; q= q+2+weighted)
    {
        Vin[(A[q+1])]++;
        Vout[(A[q])]++;
    }


    for (int a = 1; a<=Vg; a++)
        if ((Vin [a]==0) && (Vout [a]==0) ) Isolated.push_back(a);

    if (mn<1) // Renumbering all the vertices back if needed
    {

        RenumVGraph (Isolated, (-1+mn), false, true);
```

```
    }

    // Конец поиска изолированных // end of finding isolated vertices
    }

    std::vector <int> Visited (Vg+1, 0); // нулевой элемент использовать не будем:
нумеруются вершины с 1  // all vertices are to be numbered starting from 1
    Visited [0] = 1;
    int b=CheckUnvisit(Visited);  // вершина откуда идет поиск  // let's start
from here

    int f;

    while (CheckUnvisit(Visited)!=-1)
    {
        b=CheckUnvisit(Visited);  // берем первую же непосещенную вершину  //
vertex not visited yet



         f=DFSTS (A, b, Visited, order, weighted);
         if (f==1)
         {
             R.clear();
             Isolated.clear();

             if (mn<1) // Renumbering all the vertices back if needed
             {

                 RenumVGraph (A, (-1+mn), weighted);

             }

             return -1; // cycle is found
         }

    }

    reverse (order.begin(),order.end());  // в массиве order - результат
топологической сортировки  // order contains topological sorting


    if (!OnlyTS)
    {

    std::vector < std::vector <int> > B;
    MatrixSet(B, Vg+1, Vg+1, 0);


    for (unsigned int x = 0; x<A.size()-1-weighted; x = x+2+weighted)  //
Формируем матрицу смежности B, содержащую ко-во ребер даже для взвешенных графов
        // Forming Adjacency matrix and filling it with the number of edges
between vertices.
    {
        if (A[x]!=A[x+1]) B[(A[x])][(A[x+1])]++;

    }



    int y = 1;

    if (order.size()==1) y=-1;
```

```
    for (unsigned int z = 0; z<(order.size()-1); z++)
    {
        if (B[(order[z])][(order[z+1])] < 1)
        {y=-1;break;}

    }




        if (y==-1)
        {
            R.clear();
                if (mn<1) // Renumbering all the vertices back if needed
                {

                    RenumVGraph (A, (-1+mn), weighted);

                    RenumVGraph (order, (-1+mn), false, true);

                }
                return 1; // means no Hamiltonian Path found, only Topological
Sorting returned

        }
        else
        {

            for (unsigned int i=0; i<order.size(); i++)
            {R.push_back(order [i]);

            }
        }

    }


            if (mn<1) // Renumbering all the vertices back if needed
            {

                RenumVGraph (A, (-1+mn), weighted);
                RenumVGraph (R, (-1+mn), false, true);
                RenumVGraph (order, (-1+mn), false, true);

            }

        return 0; // means both Hamiltonian Path and Topological Sorting returned
}




int TSortHP (std::pair < std::vector<int>, std::vector<double>> & A, std::vector
<int> & R, std::vector <int> & order, std::vector <int> & Isolated, const bool
OnlyTS = false)
// Модификация функции TSortHP (см. выше) для случая нецелочисленных весов ребер
(double).
// Modification of the function TSortHP (see it above) for not-integer (double)
weights of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an
"Adjacency vector" without weights. But weights are set in the second one.
```

```
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the
first vector has its weight set as i-th element in the second one.


{
    R.clear(); order.clear(); Isolated.clear();
    if ((A.first).size()==0) return -1;
    if ((A.second).size()==0) return -1;
    if (  (A.first).size()!=((A.second).size())*2 ) return -1;

    return TSortHP (A.first, R, order, Isolated, false, OnlyTS);

}




int DistanceBFA (std::vector <int> &A, std::vector <long long int> & D, const int
b, std::vector <int> & Prev, const bool weighted, int V = INT_MIN)
{

    // Рассчитывает расстояния от заданной вершины b до всех прочих в орграфе
(используется метод поиска в ширину).
    // Возвращается 1 в случае успеха (вектор D содержит кратчайшие расстояния от
вершины b до вершины i, а вектор Prev - индекс вершин-предков в таком пути).
    // По умолчанию вектор D содержит значения LLONG_MAX, а вектор Prev - "-1".
    // Если в ходе работы обнаружен цикл негативного веса, то функция возвращает -
1 и пустые вектора D и Prev.
    // На входе д.б. граф, заданный вектором смежности A (считается, что вершины
нумеруются с 0), номер исходной вершины b и флаг, является ли граф взвешенным
(const bool weighted). Для невзвешенных считается, что каждое ребро имеет вес = 1.
    // Также на вход подается номер наибольшей вершины V (если не передан,
рассчитывается самостоятельно как номер наибольшей вершины в ребрах)
    // Функция работает со взвешенными и с невзвешенными графами, причем они могут
содержать петли и множественные ребра. Ребра могут иметь как неотрицательный (в
т.ч. и нулевой), так и отрицательный вес.

    // The function counts the shortest distances from the vertex b to all
vertices in the graph (these distances are to be contained in vector D, i.e. D[i]
means the shortest distance from b to I).
    // By default vector D is filled with LLONG_MAX.
    // Vector Prev is intended to contain the number of the previous vertex for
every vertex in such shortest paths ("-1" value is set by default and means "this
vertex doesn't included in any such path").
    // The Breadth-first search method is used here.
    // The input graph should be directed, both weighted or unweighted (in case of
unweighted graph we consider every edge's weight as "1".) The graph may have loops
and multiple edges.
    // Input data: Adjacency vector A (it is supposed that vertices are numbered
starting from 0) and the maximum vertex number V (V may be not set, in this case
it will be the maximum vertex number of Adjacency vector A)
    // The edges may have weight of 0, >0, <0.
    // In case we found a negative weight cycle as well as input data is incorrect
the function returns "-1" and empty D and Prev.


    D.clear();
    Prev.clear();

    if (A.size()==0) return -1;

    if ((V<0)&&(V != INT_MIN)) return -1;
    if ( (A.size())%(2+weighted)!=0 ) return -1; // checking for input data
correctness
```

```
    int E = A.size()/(2+weighted); // the total number of edges in the graph


    int mn, mx;
    RangeVGraph (A, mx, mn, weighted);


    if (mn<0)  // // checking for input data correctness.
    {
        return -1;
    }


    if (mx>V) V = mx; // здесь будет максимальный номер вершины // the max number
of assigned to vertices

    if ((b<0) || (b>V))  // // checking for input data correctness: number of the
vertex b must be in range [0, V].
    {
        return -1;
    }

    D.resize(V+1, LLONG_MAX); // По умолчанию расстояния равны + бесконечность //
The default distance values is LLONG_MAX for every vertex


    Prev.resize(V+1, -1); // по умолчанию "предка" нет. // The previous vartexes
are not set by default (i.e. =-1 for every vertex)


    std::queue <int> Q; //вектор очереди


   D[b] = 0; //дистанция от первой вершины до себя = 0 // the distance from
starting vertex to itself = 0

   Q.push (b);

    unsigned int j;
    int i;
    long long int count = 0;
    long long int c = (long long int)(V*E);




    while (!(Q.empty()))
    {
        j = 0;// индекс пробега по вектору смежности A // indexes of beginning-
vertices of edges in A
        i = Q.front(); // номер очередной вершины из очереди к рассмотрению
(добавляем в конец, достаем с начала) // number of vertex to continue from

        while (j<=A.size()-2-weighted) // lets look through A
        {

            if (weighted)
            {
                // for a weighted graph:
```

```
                if (  (A[j]==i) &&  (A[j+1]==i) &&  (A[j+2]<0) )  {D.clear();
Prev.clear(); return -1;} // i.e. we have found a negative weight loop

                if (  (A[j]==i) && ((D[ (A[j+1] )] == LLONG_MAX) || (D[ (A[j+1]) ]
> (D[ i ]+ (long long int)(A[j+2]) ) ))  )
                // we should recount distance if we have found non-visited vertex
or we may reduce its distance from vertex b.

                {
                    Q.push(A[j+1]);  //эту j+1 вершину - в очередь // in this casr
we should push such vertex to quenue Q
                    count++;
                    D[(A[j+1])] = D[(i)]+(long long int)(A[j+2]); // по ней
посчитаем дистанцию // and recount its distance from b

                    Prev [(A[j+1])] = i;// и предка // and reset its previous
vertex too
                }

            }

            if (!weighted)
            {
            // для невзвешенного все аналогично, но считаем что вес каждого ребра
=1
            // for an unweighted graph lets put every edge has distance =1.


                if (  (A[j]==i) && ((D[ (A[j+1] )] == LLONG_MAX) || (D[ (A[j+1]) ]
> (D[ i ]+1) ))  )
                //если нашли вершину i и следующая за ней (за номером j+1 в
векторе A): (1) не исходная (от которой считаем, равная b),
                // и (2) дистанция до нее равна LLONG_MAX (там еще не были) либо
больше дистанции до i, увеличенной на A[j+1] (путь через i + последнее ребро из i)
                // we should recount distance if we have found non-visited vertex
or we may reduce its distance from vertex b.

                {
                    Q.push(A[j+1]);  //эту j+1 вершину - в очередь // in this casr
we should push such vertex to quenue Q
                    count++;
                    D[(A[j+1])] = D[(i)]+1; // по ней посчитаем дистанцию // and
recount its distance from b

                    Prev [(A[j+1])] = i;// и предка // and reset its previous
vertex too
                }

            }

            j = j+2+weighted;
        }

        Q.pop();

        if (count > (c)) //Значит, мы наткнулись на цикл отрицательного веса //
This means we found a negative weight cycle
        {
            D.clear();
            Prev.clear();

            return -1;
        }
```

```
    }



    return 0;
}


int DistanceBFA (std::pair < std::vector<int>, std::vector<double>> & A,
std::vector <long double> & D, const int b, std::vector <int> & Prev, int V =
INT_MIN)
// Модификация функции DistanceBFA (см. выше) для случая нецелочисленных весов
ребер (double).
// Modification of the function DistanceBFA (see it above) for not-integer
(double) weights of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an
"Adjacency vector" without weights. But weights are set in the second one.
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the
first vector has its weight set as i-th element in the second one.

{
    D.clear();
    Prev.clear();
    if ((A.first).size()==0) return -1;
    if ((A.second).size()==0) return -1;
    if (  (A.first).size()!=((A.second).size())*2 ) return -1;



    if ((V<0)&&(V != INT_MIN)) return -1;




    int E = (A.first).size()/(2); // the total number of edges in the graph


    int mn, mx;
    RangeVGraph (A.first, mx, mn, false);


    if (mn<0)  // // checking for input data correctness.
    {
        return -1;
    }


    if (mx>V) V = mx; // здесь будет максимальный номер вершины // the max number
of assigned to vertices

    if ((b<0) || (b>V))  // // checking for input data correctness: number of the
vertex b must be in range [0, V].
    {
        return -1;
    }

    D.resize(V+1, INFINITY); // По умолчанию расстояния равны + бесконечность //
The default distance values is INFINITY for every vertex
```

```
    Prev.resize(V+1, -1); // по умолчанию "предка" нет. // The previous vartexes
are not set by default (i.e. =-1 for every vertex)


    std::queue <int> Q; //вектор очереди


   D[b] = 0.0; //дистанция от первой вершины до себя = 0 // the distance from
starting vertex to itself = 0

   Q.push (b);

    unsigned int j;
    int i;
    long long int count = 0;
    long long int c = (long long int)(V*E);




    while (!(Q.empty()))
    {
        j = 0;// индекс пробега по вектору смежности A // indexes of beginning-
vertices of edges in A
        i = Q.front(); // номер очередной вершины из очереди к рассмотрению
(добавляем в конец, достаем с начала) // number of vertex to continue from

        while (j<A.second.size()) // lets look through A
        {


                if (  ( (A.first)[2*j]==i) &&  ( (A.first)[2*j+1]==i) &&  (
(A.second)[j]<0) )  {D.clear(); Prev.clear(); return -1;} // i.e. we have found a
negative weight loop

                if (  ( (A.first)[2*j]==i) && ( (D[ (A.first)[2*j+1] ] ==
INFINITY) || (D[ (A.first)[2*j+1] ] > (D[ i ]+ (long double)((A.second)[j]) ) )  )
)
                // we should recount distance if we have found non-visited vertex
or we may reduce its distance from vertex b.

                {
                    Q.push((A.first)[2*j+1]);  //эту вершину - в очередь // in
this casr we should push such vertex to quenue Q
                    count++;
                    D[ (A.first)[2*j+1] ] = D[ i ]+ (long double)((A.second)[j]);
// по ней посчитаем дистанцию // and recount its distance from b

                    Prev [ (A.first)[2*j+1] ] = i;// и предка // and reset its
previous vertex too
                }

            j = j+1;
        }

        Q.pop();

        if (count > (c)) //Значит, мы наткнулись на цикл отрицательного веса //
This means we found a negative weight cycle
        {
            D.clear();
            Prev.clear();

            return -1;
```

```
        }

    }

    return 0;
}
```

```
long long int MaxFlowGraph (std::vector <int> A, const bool weighted, int b, int e, std::vector <std::vector<int> >&Paths, std::vector <int> &Flows, std::vector < std::pair < int, int> > &MinCut)
// Функция для поиска максимального потока в графе из вершины b в вершину e; граф задается вектором смежности A;
// параметр weighted определяет, является ли граф взвешенным (если взвешенный - "Истина", при этом веса ребер здесь могут быть лишь целыми положительными).
// Может работать с множественными ребрами (рассматриваются как одно "объединенное" ребро с суммарным весом) и с множественными петлями (петли игнорируются).
// Вершины графа должны быть неотрицательны, веса ребер - только положительны
// Возвращает: (1) величину максимального потока, (2) заполненный вектор путей Paths, слагающих максимальный поток (один из возможных вариантов построения Paths, если их может быть несколько),
// (3) соответствующие этим путям значения потоков в векторе Flows, (4) перечень ребер минимального разреза графа в векторе MinCut (каждое ребро задано парой вершин).
//  В случае некорректных исходных данных или отсутствия пути из b в e возвращает -1 и пустые Paths, Flows, MinCut.


// Finds maximal flow from vertex b to vertex e in the graph A (set by Adjacency vector A).
// Parameter "weighted" sets if A is a weighted graph or no. All vertices of A should have only non-negative marks.
// For a weighted graph edges should have only positive values.
// Graph A may have multiple edges (multiple edges will be considered as one joined edge that have its weight = sum of the weights of all joined edges) and multiple loops (loops will be ignored).
// Returnes: maximal flow value and 3 vectors: vector Paths (contains all the paths of the maximal flow network (one of the possible solutions if >1 solutions exist))
// vector Flows (contains values of a flow for a index-relevant Path (i.e. Flows[0] for Paths [0], etc)),
// vector MinCut (contains the max-flow min-cut as an array of edges: every edge is set as a pair of its vertices).
// If input data is incorrect or there are no path from b to e returns -1 and empty Paths and Flows.



{

    long long int Result=0;// здесь будет результат // Here a result will be
    Paths.clear(); // здесь будут пути, слагающие макспоток
    Flows.clear(); // а здесь  - соответствующие им величины подпотоков, слагающие общий поток из b в e
    MinCut.clear(); // А здесь будет минимальный разрез

    if (A.size()==0) return -1;

    if ((b<0) || (e<0))  return -1;
    if ( (A.size())%(2+weighted)!=0 ) return -1; // checking for input data correctness
```

```
    int mn, mx;
    RangeVGraph (A, mx, mn, false, true);
    if (mn<0) return -1; // все вершины и веса д.б. неотрицательны // all vertices
should have non-negative marks. All weights shouldn't be negative too.



    if (FindIn(A, b, (2+weighted), 0) == -1) return -1; // Проверка что вершины b
(как исток) и e (как сток) есть в графе
    if (FindIn(A, e, (2+weighted), 1) == -1) return -1; // Checking for vertex b
(as source) and vertex e (as sink) both are in the graph

    if (weighted)
        if (FindIn(A, 0, (2+weighted), 2) != -1) return -1; // weighted graph
should not have zero-valued edges



    std::map <std::pair < int, int> , int> G2;
    G2.clear();


    if (AdjVectorToAdjMap (A, G2, weighted)==-1) return -1;


    int l=INT_MAX;
    int x;
    std::pair < int, int> C;


    std::vector <long long int> D; D.clear();
    std::vector <int> Prev; Prev.clear();
    std::vector <int> Path; Path.clear();

    int d = DistanceBFA (A, D, b, Prev, weighted);
    if (D[e]==LLONG_MAX) return -1; // в этом случае e недостижима из b // In this
case we have no path from b to e.


    while (d!=-1)
    {
        if (D[e]==LLONG_MAX) break; // в этом случае e недостижима из b  // In
this case we have no path from b to e.

        // строим путь из b в e
        // Let's build path from b to e.

        Path.clear();
        x = e;


        while (x!=-1)
        {
            Path.push_back(x);
            if (x==b)
            {

                break;
            }

            x = Prev[x];

        }
```

```
        std::reverse(std::begin(Path), std::end(Path));

        // построили путь из b в e
        // We have a path from b to e.

        l=INT_MAX;
        for (int q=0; q<Path.size()-1; q++)  // ищем ребро с самой низкой
пропускной способностью и записываем пропускную способноость в l
        {                                    // looking for an edge with the
minimal non-zero weight in the Path
            C = std::make_pair(Path[q], Path[q+1]);
            if ((l>G2[C])&&(G2[C]>0))
            {l = G2[C]; }

        }




        for (int q=0; q<Path.size()-1; q++)  // Вычитаем l по всему найденному
пути // recalculating weights of all edges of the Path
        {
           C = std::make_pair(Path[q], Path[q+1]);
            G2[C] = G2[C]-l;
            if (G2[C]==0)
            {
                G2.erase(G2.find(C));
                MinCut.push_back(C);
            }

        }


        Result=Result+l;
        Flows.push_back(l);
        Paths.push_back(Path);

        AdjMapToAdjVector (A, G2);


        d = DistanceBFA (A, D, b, Prev, weighted);
    }



for (int y=0; y<MinCut.size(); y++)
{

    if (D[(MinCut[y].first)]==LLONG_MAX) {MinCut.erase(MinCut.begin()+y); y--; }

}


    return Result;
}


long double MaxFlowGraph (std::pair < std::vector<int>, std::vector<double>> A,
int b, int e, std::vector <std::vector<int> >&Paths, std::vector <double> &Flows,
std::vector < std::pair < int, int> > &MinCut)
// Модификация функции MaxFlowGraph (см. выше) для случая нецелочисленных весов
ребер (double).
```

```
// Modification of the function MaxFlowGraph (see it above) for not-integer
(double) weights of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an
"Adjacency vector" without weights. But weights are set in the second one.
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the
first vector has its weight set as i-th element in the second one.



{

    long double Result=0.0;// здесь будет результат  // Here a result will be
    Paths.clear(); // здесь будут пути, слагающие макспоток
    Flows.clear(); // а здесь  - соответствующие им величины подпотоков, слагающие
общий поток из b в e
    MinCut.clear(); // А здесь будет минимальный разрез
    if ((b<0) || (e<0))  return -1;


    if ((A.first).size()==0) return -1;
    if ((A.second).size()==0) return -1;
    if (  (A.first).size()!=((A.second).size())*2 ) return -1;


    int mn, mx;
    RangeVGraph (A.first, mx, mn, false, true);
    if (mn<0) return -1; // все вершины д.б. неотрицательны  // all vertices
should have non-negative marks.



    if (FindIn(A.first, b, 2, 0) == -1) return -1; // Проверка что вершины b (как
исток) и e (как сток) есть в графе
    if (FindIn(A.first, e, 2, 1) == -1) return -1; // Checking for vertex b (as
source) and vertex e (as sink) both are in the graph


    for (unsigned int q=0; q<A.second.size(); q++)
        if (A.second[q]<=0.0) return -1; // weighted graph should not have
subzero-valued edges




    std::map <std::pair < int, int> , double> G2;
    G2.clear();


    if (AdjVectorToAdjMap (A, G2)==-1) return -1;


    double l=INFINITY;
    int x;
    std::pair < int, int> C;


    std::vector <long double> D; D.clear();
    std::vector <int> Prev; Prev.clear();
    std::vector <int> Path; Path.clear();

    int d = DistanceBFA (A, D, b, Prev);
    if (D[e]==INFINITY) return -1; // в этом случае e недостижима из b  // In this
case we have no path from b to e.
```

```
    while (d!=-1)
    {
        if (D[e]==INFINITY) break; // в этом случае e недостижима из b  // In this
case we have no path from b to e.

        // строим путь из b в e
        // Let's build path from b to e.

        Path.clear();
        x = e;


        while (x!=-1)
        {
            Path.push_back(x);
            if (x==b)
            {

                break;
            }

            x = Prev[x];

        }
        // построили путь из b в e
         // We have a path from b to e.

        std::reverse(std::begin(Path), std::end(Path));




        l=INFINITY;
        for (int q=0; q<Path.size()-1; q++)  // ищем ребро с самой низкой
пропускной способностью и записываем пропускную способноость в l
        {                                    // looking for an edge with the
minimal non-zero weight in the Path
            C = std::make_pair(Path[q], Path[q+1]);
            if ((l>G2[C])&&(G2[C]>0.0))
            {l = G2[C]; }

        }




        for (int q=0; q<Path.size()-1; q++)  // Вычитаем l по всему найденному
пути  // recalculating weights of all edges of the Path
        {
           C = std::make_pair(Path[q], Path[q+1]);
            G2[C] = G2[C]-l;
            if (G2[C]==0.0)

            {
                G2.erase(G2.find(C));
                MinCut.push_back(C);
            }

        }
```

```
        Result=Result+l;
        Flows.push_back(l);
        Paths.push_back(Path);

        AdjMapToAdjVector (A, G2);


        d = DistanceBFA (A, D, b, Prev);
    }



    for (int y=0; y<MinCut.size(); y++)
    {

        if (D[(MinCut[y].first)]==INFINITY) {MinCut.erase(MinCut.begin()+y); y--;
}

    }

    return Result;
}


int DistanceTS (std::vector <int> &A, std::vector <long long int> & D, const int
b, std::vector <int> & Prev, const bool weighted, int V = INT_MIN)
{

// Рассчитывает расстояния от заданной вершины b до всех прочих в орграфе. Метод
работает быстрее, чем DistanceBFA за счет предварительной топологической
сортировки орграфа.
// Однако метод неприменим для орграфов, содержащих любой цикл кроме петель,  в
т.ч. - множественных (петли будут игнорироваться).
// Возвращается 1 в случае успеха (вектор D содержит кратчайшие расстояния от
вершины b до вершины i, а вектор Prev - индекс вершин-предков в таком пути).
// По умолчанию вектор D содержит значения LLONG_MAX, а вектор Prev - "-1".
// Если был обнаружен цикл - возвращается -1 и пустые вектора D и Prev.
// На входе д.б. граф, заданный вектором смежности A (считается, что вершины
нумеруются с 0), номер исходной вершины и флаг, является ли граф взвешенным.
// Для невзвешенных графов считается, что каждое ребро имеет вес = 1. Для
взвешенных - длины ребер должны быть строго меньше INT_MAX.
// Также на вход подается номер наибольшей вершины V (если не передан,
рассчитывается самостоятельно как номер наибольшей вершины в ребрах)
// Функция работает со взвешенными и с невзвешенными графами, причем они могут
содержать петли и множественные ребра.
// Ребра могут иметь как неотрицательный (в т.ч. и нулевой), так и отрицательный
вес.

// The function counts the shortest distances from the vertex b to all vertices in
the graph (these distances are to be contained in vector D, i.e. D[i] means the
shortest distance from b to i).
// By default vector D is filled with LLONG_MAX.
// In doing so, vector Prev is intended to contain the number of the previvous
vertex for every vertex in such shortest paths ("-1" value is set by default and
means "this vertex doesn't included in any such path").
// This function seems to be faster than DistanceBFA, but DistanceTS works only
with graphs containing no cycles (except loops, multiply loops).
// The input graph should be directed, both weighted or unweighted (in this case
we consider every edge's weight as "1".) The graph may have loops and multiple
edges.
// Input data: Adjacency vector A (it is supposed that vertices are numbered
starting from 0) and the maximum vertex number V (V may be not set, in this case
it will be the maximum vertex number of Adjacency vector A)
```

```
// The edges of a weighted graph may have weight of 0, >0, <0, but only less than
INT_MAX (<INT_MAX).
// In case we found a cycle as well as input data is incorrect the function
returns "-1" and empty D and Prev.



    D.clear();
    Prev.clear();

    if (A.size()==0) return -1;

    if ((V<0)&&(V != INT_MIN)) return -1;
    if ( (A.size())%(2+weighted)!=0 ) return -1; // checking for input data
correctness



    int mn, mx;
    RangeVGraph (A, mx, mn, weighted);


    if (mn<0)  // // checking for input data correctness.
    {
        return -1;
    }


    if (mx>V) V = mx; // здесь будет максимальный номер вершины // the max number
of assigned to vertices

    if ((b<0) || (b>V))  // // checking for input data correctness: number of the
vertex b must be in range [0, V].
    {
        return -1;
    }

    D.resize(V+1, LLONG_MAX); // По умолчанию расстояния равны + бесконечность //
The default distance values is LLONG_MAX for every vertex


    Prev.resize(V+1, -1); // по умолчанию "предка" нет. // The previous vertices
are not set by default (i.e. =-1 for every vertex)



    std::vector <int> order; // здесь будет храниться топологическая сортировка
вершин
    order.clear();

    std::vector <int> R; // An auxiliary vector for the function TSortHP as such
parameter is required, the vector should stay empty
    R.clear();

    std::vector <int> I; //  An auxiliary vector for the function TSortHP as such
parameter is required, the vector should stay empty
    I.clear();

    int t=TSortHP (A, R, order, I, weighted, true);



    if (t==-1)  // if there are no topological sorting (i.e. the graph contains
cycle except any loops) this function can't work.
```

```
    {
        D.clear();
        Prev.clear();
        return -1;
    }


    if (mn!=0) order.insert(order.begin(), 0); // if vertices are numbrered
starting not from 0 - let's add 0

    if (V>mx)   //adding to "order" vertices that have number more than maximal in
A and <= V
        for (int ff = (mx+1); ff<=V; ff++)
             order.insert(order.begin(), ff);


    int bb;

    for (int i = 0; i< order.size(); i++)   // поиск нового номера вершины начала
поиска
    {

        if (order[i]==b) {bb = i; break;}

    }


    D[order[bb]] = 0; //дистанция от первой вершины до себя = 0

    std::vector <std::vector <int>> B (V+1);

    MatrixSet(B, V+1, V+1, INT_MAX);    // Создание заготовки матирицы B в виде 2-
хмерного массива (из векторов)


    for (unsigned int x = 0; x<=(A.size()-(2+weighted)); x = x+2+weighted)
    {
        if (weighted)
        {
            if ( (A[x]!=A[x+1]) && ( B[(A[x])][(A[x+1])] > A[x+2] ) )
B[(A[x])][(A[x+1])] = A[x+2];  // в матрицу смежности вставляем длину кратчайшего
ребра
        }

        if (!weighted)
        {
            if (A[x]!=A[x+1]) B[(A[x])][(A[x+1])] = 1;  // в матрицу смежности
вставляем длину ребра
        }
    }

    // Конец заполнения матрицы смежности



    for (int b = bb; b<V; b++)              // бежим по матрице смежности от
искомой вершины к большим номерам и пересчитываем дистанцию
        for (int j = b+1; j<=V; j++)
        {


            if (D[order[b]]==LLONG_MAX) continue; //если вершина b недостижима из
исходной, нечего от нее расстояния пересчитывать // We should recalculate
distances only from vertices that are accessible from vertex b as initial one.
```

```
            if ( (B[order[b]][order[j]]!=INT_MAX) && ((D[order[j]]==LLONG_MAX) ||
(D[order[j]]>(D[order[b]]+(long long int)(B[order[b]][order[j]]) ) ) ) )
{D[order[j]]=(D[order[b]]+ (long long int)(B[order[b]][order[j]]) );
Prev[order[j]]= order[b];}

        }



    return 0;
}


int DistanceTS (std::pair < std::vector<int>, std::vector<double>> & A,
std::vector <long double> & D, const int b, std::vector <int> & Prev, int V =
INT_MIN)
// Модификация функции DistanceTS (см. выше) для случая нецелочисленных весов
ребер (double).
// Modification of the function DistanceTS (see it above) for not-integer (double)
weights of edges of a graph.
// Graph is represented here as a pair of 2 vectors. The first one is an
"Adjacency vector" without weights. But weights are set in the second one.
// So an edge that is set by the pair of vertices indexed as 2*i, 2*i+1 in the
first vector has its weight set as i-th element in the second one.



{
    D.clear();
    Prev.clear();
    if ((A.first).size()==0) return -1;
    if ((A.second).size()==0) return -1;
    if (  (A.first).size()!=((A.second).size())*2 ) return -1;

    if ((V<0)&&(V != INT_MIN)) return -1;



    int mn, mx;
   RangeVGraph (A.first, mx, mn, false);


    if (mn<0)  // // checking for input data correctness.
    {
        return -1;
    }


    if (mx>V) V = mx; // здесь будет максимальный номер вершины // the max number
of assigned to vertices

    if ((b<0) || (b>V))  // // checking for input data correctness: number of the
vertex b must be in range [0, V].
    {
        return -1;
    }

    D.resize(V+1, INFINITY); // По умолчанию расстояния равны + бесконечность //
The default distance values is INFINITY for every vertex


    Prev.resize(V+1, -1); // по умолчанию "предка" нет. // The previous vertices
are not set by default (i.e. =-1 for every vertex)
```

```
    std::vector <int> order; // здесь будет храниться топологическая сортировка
вершин
    order.clear();

    std::vector <int> R; // An auxiliary vector for the function TSortHP as such
parameter is required, the vector should stay empty
    R.clear();

    std::vector <int> I; //  An auxiliary vector for the function TSortHP as such
parameter is required, the vector should stay empty
    I.clear();

    int t=TSortHP (A.first, R, order, I, false, true);



    if (t==-1)  // if there are no topological sorting (i.e. the graph contains
cycle except any loops) this function can't work.
    {
        D.clear();
        Prev.clear();
        return -1;
    }


    if (mn!=0) order.insert(order.begin(), 0); // if vertices are numbrered
starting not from 0 - let's add 0

    if (V>mx)   //adding to "order" vertices that have number more than maximal in
A and <= V
        for (int ff = (mx+1); ff<=V; ff++)
             order.insert(order.begin(), ff);


    int bb;

    for (int i = 0; i< order.size(); i++)   // поиск нового номера вершины начала
поиска
    {

        if (order[i]==b) {bb = i; break;}

    }


    D[order[bb]] = 0.0; //дистанция от первой вершины до себя = 0

    std::vector <std::vector <double>> B (V+1);

    MatrixSet(B, V+1, V+1, INFINITY);    // Создание заготовки матирицы B в виде
2-хмерного массива (из векторов)


    for (unsigned int x = 0; x<=((A.first).size()-(2)); x = x+2)
    {

            if ( ((A.first)[x]!=(A.first)[x+1]) && (
B[((A.first)[x])][((A.first)[x+1])] > (A.second)[x/2] ) )
B[(A.first)[x]][(A.first)[x+1]] = (A.second)[x/2];  // в матрицу смежности
вставляем длину кратчайшего ребра
   }
```

```
    // Конец заполнения матрицы смежности


    for (int b = bb; b<V; b++)              // бежим по матрице смежности от
искомой вершины к большим номерам и пересчитываем дистанцию
        for (int j = b+1; j<=V; j++)
        {


            if (D[order[b]]==INFINITY) continue; //We should recalculate distances
only from vertices that are accessible from vertex b as initial one.

            if ( (B[order[b]][order[j]]!=INFINITY) && ((D[order[j]]==INFINITY) ||
(D[order[j]]>(D[order[b]]+(long double)(B[order[b]][order[j]]) ) ) ) )
{D[order[j]]=(D[order[b]]+ (long double)(B[order[b]][order[j]]) ); Prev[order[j]]=
order[b];}

        }



    return 0;
}



#endif
```