Сведения о документе / About

Настоящий документ содержит общее описание CBioInfCpp.h и содержащихся в нем функций.

Документы CBioInfCpp.h, About_CBioInfCpp.rtf, About_CBioInfCpp.pdf (все указанные файлы размещены в настоящем каталоге) составляют собой одно произведение, которое распространяется на условиях лицензии Creative Commons Attribution 4.0 International Public License (сокращенно - CC BY, гиперссылка на текст лицензии: https://creativecommons.org/licenses/by/4.0/legalcode.ru).

Автор CBioInfCpp.h, About_CBioInfCpp.rtf, About_CBioInfCpp.pdf – Черноухов Сергей (chernouhov@rambler.ru)

This document contains general data on CBioInfCpp.h library.

CBioInfCpp.h, About_CBioInfCpp.rtf, About_CBioInfCpp.pdf are written by Sergey Chernouhov (chernouhov@rambler.ru).

Функции CBioInfCpp.h и их предназначение/ Functions included in CBioInfCpp.h and their possible use

CBioInfCpp.h содержит ряд функций, которые могут быть полезны как при решении различных задач в области биоинформатики, так и других задач, связанных с работой со строками и графами.

Все функции объявлены и полностью определены в файле CBioInfCpp.h. При этом в данном файле также объявляется включение библиотек iostream, fstream, string, vector, set, algorithm, queue, map, cmath, stack, limits.h, cstdlib, ctime, float.h, unordered_map, unordered_set, utility, functional, необходимых для корректной работы всех функций.
Да, наверное, такой подход и не является каноническим, но он позволяет быстро начать работу (достаточно лишь включить CBioInfCpp.h в свою программу с помощью include).

При работе с графами применены структуры данных «Вектор смежности» и «Ассоциативный массив смежности» - см. раздел "Функции работы с графами / Working with graphs".

PS Настоящая версия CBioInfCpp.h содержит не так много функций. Однако автор будет рад, если эта она получит дальнейшее развитие.

CBioInfCpp.h contains a number of functions that may be used as in bioinformatics as well as in other fields related to working with graphs and strings.

All the function are defined in the only file CBioInfCpp.h. Also the libraries iostream, fstream, string, vector, set, algorithm, queue, map, cmath, stack, limits.h, float.h are included. Yes, this way may be considered as "not the only best", but it allows to start immediately by writing "include CBioInfCpp.h" at the program beginning.

The data structures "Adjacency vector" and "Adjacency map" to represent a graph are used in the CBioInfCpp (see the section "Working with graphs").

PS The current version of CBioInfCpp.h contains not so many functions. But it would be nice if this library grows up.

Функции ввода-вывода/ Input-Output functions

```
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:
// Сюда будут записываться индексы (обозначения) строк
// Here indexes of strings will be contained

DataS:
// Сюда будут записываться сами строки
// Here data strings will be contained


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


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 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 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 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.


template < typename TMC>
int MatrixCout (std::vector <std::vector <TMC>> & B, char g = ' ')
// Обобщение функции: ввод матрицы (элементов стандартных типов TMF) 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
// "Fouts" Matrix of standard type elements 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.


template < typename TMF>
int MatrixFout (std::vector <std::vector <TMF>> & B, std::ofstream & fout, char g
= ' ')
// Обобщение функции: ввод матрицы (элементов стандартных типов TMF) 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 of standard type elements 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 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.


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.


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.


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.


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


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


template < typename TVC>
```

```
int VectorCout (const TVC &P)
// Обобщение функции: вывод вектора/ списка/ и др. итерируемых контейнеров,
содержащих элементы стандартных типов, на экран через пробелы
// "Couts" vector/ list/ etc container of standard type elements to screen.
Returns -1 if the container is empty


template < typename TVF>
int VectorFout (const TVF &P, std::ofstream &fout)
// Обобщение функции: вывод вектора/ списка/ и др. итерируемых контейнеров,
содержащих элементы стандартных типов, в файл через пробелы
// "Fouts" vector/ list/ etc container of standard type elements to screen.
Returns -1 if the container is empty


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.


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.


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 его 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.


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 его 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.


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.
```

```
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.


int GraphCout (const std::map <std::pair < int, int> , int> &P)
// Вывод графа, заданного ассоциативным массивом смежности, на экран: каждое ребро
выводится в новой строке. Веса ребер целочисленны.
// "Couts" a graph that is set by Adjacency map P to screen: one edge in one line.
// Returns -1 if input data is not correct. Otherwise returns 0.


int GraphFout (const std::map <std::pair < int, int> , int> &P, std::ofstream
&fout)
// Вывод графа, заданного ассоциативным массивом смежности, в файл: каждое ребро
выводится в новой строке. Веса ребер целочисленны.
// "Fouts" a graph that is set by Adjacency map P to file: one edge in one line.
// Returns -1 if input data is not correct. Otherwise returns 0.


int GraphCout (const std::map <std::pair < int, int> , double> &P, unsigned int
prec = 4, bool scientifique = false)
// Модификация функции GraphCout для заданного ассоциативным массивом смежности
графа (см. выше) для случая нецелочисленных весов ребер.
// Modification of the function GraphCout for Adjacency map (see it above) for
not-integer (double) weights of edges.


int GraphFout (const std::map <std::pair < int, int> , double> &P, std::ofstream
&fout, unsigned int prec = 4, bool scientifique = false)
// Модификация функции GraphFout для заданного ассоциативным массивом смежности
графа (см. выше) для случая нецелочисленных весов ребер.
// Modification of the function GraphFout for Adjacency map (see it above) for
not-integer (double) weights of edges.

int GraphCout (const std::map <std::pair < int, int> , std::vector<int> > &P, bool
in_line=false)
// Вывод графа, заданного "мега-мапом", на экран. Веса ребер целочисленны.
// Мега-мап представляет собой ассоциативный массив смежности, предназначенный для
хранения графов с множественными петлями и множественными ребрами.
// При этом ключом является пара чисел, задающих вершины ребер графа между ними, а
значением - вектор весов для всех ребер между этими вершинами.
// Параметр in_line задает, выводить ли все значения вектора-значения в 1 строку
после ключа, или же каждое значение вектора выводится после ключа в новой строке.

// "Couts" a graph that is set by Adjacency mega-map P to screen.
// Adjacency mega-map is an extended version of Adjacency map (and it is built
basing on std::map too) for containing graphs that have multiply loops and
multiply edges.
// In doing so, the key value of the map is a pair of integers that sets edge(s)
between them and the mapped value is a vector that contains weights of all edges
between these vertices.
// Parameter "in_line" sets how to print mapped value of the edges: all values of
the mapped vector in one line or key value + every value of the mapped vector as
separate line.
```

```
// Returns -1 if input data is not correct. Otherwise returns 0.


int GraphFout (const std::map <std::pair < int, int> , std::vector<int> > &P,
std::ofstream &fout, bool in_line=false)
// Вывод графа, заданного "мега-мапом", в файл. Веса ребер целочисленны.
// Мега-мап представляет собой ассоциативный массив смежности, предназначенный для
хранения графов с множественными петлями и множественными ребрами.
// При этом ключом является пара чисел, задающих вершины ребер графа между ними, а
значением - вектор весов для всех ребер между этими вершинами.
// Параметр in_line задает, выводить ли все значения вектора-значения в 1 строку
после ключа, или же каждое значение вектора выводится после ключа в новой строке.


// "Fouts" a graph that is set by Adjacency mega-map P to file: one edge in one
line.
// Adjacency mega-map is an extended version of Adjacency map (and it is built
basing on std::map too) for containing graphs that have multiply loops and
multiply edges.
// In doing so, the key value of the map is a pair of integers that sets edge(s)
between them and the mapped value is a vector that contains weights of all edges
between these vertices.
// Parameter "in_line" sets how to print mapped value of the edges: all values of
the mapped vector in one line or key value + every value of the mapped vector as
separate line.
// Returns -1 if input data is not correct. Otherwise returns 0.


int GraphCout (const std::map <std::pair < int, int> , std::vector<double> > &P,
bool in_line=false, unsigned int prec = 4, bool scientifique = false)
// Модификация функции GraphCout для заданного "мега-мапом" (см. выше) для случая
нецелочисленных весов ребер.

// Modification of the function GraphCout for mega-map (see it above) for not-
integer (double) weights of edges.


int GraphFout (const std::map <std::pair < int, int> , std::vector <double> > &P,
std::ofstream &fout, bool in_line=false, unsigned int prec = 4, bool scientifique
= false)
// Модификация функции GraphFout для заданного "мега-мапом" (см. выше) для случая
нецелочисленных весов ребер.
// Modification of the function GraphFout for mega-map (see it above) for not-
integer (double) weights of edges.

int CoutSuffixTree (const std::vector <int> & Tree, const std::string DataS, bool
tree=true, bool strings = true)
// "Couts" the suffix tree set by Tree (its format see at SuffixTreeMake' s
description). The string itself is set by DataS.
// No checking of input data correctness is here.
// If "tree" = true couts every 4 numbers  that set each edge of the tree in line.
// If "strings" = true couts every substring that relevant to each edge in line.

// Выводит на экран суффиксное дерево, заданное вектором числе в порядке,
указанном в описании функции SuffixTreeMake. Сама строка задается DataS. Проверка
корректности исходных данных не проводится.
// Если "tree" = true, выводит по каждому ребру квартет чисел, его задающих
(каждый квартет - вновой строке).
// Если "strings" = true, выводит по каждому ребру соотвествующую ему подстроку из
строки DataS


int FoutSuffixTree (const std::vector <int> & Tree, const std::string DataS,
std::ofstream &fout, bool tree=true, bool strings = true)
```

```
// "Fouts" the suffix tree set by Tree (its format see at SuffixTreeMake' s
description). The string itself is set by DataS.
// No checking of input data correctness is here.
// If "tree" = true couts every 4 numbers  that set each edge of the tree in line.
// If "strings" = true couts every substring that relevant to each edge in line.

// Выводит в файл суффиксное дерево, заданное вектором числе в порядке, указанном
в описании функции SuffixTreeMake. Сама строка задается DataS. Проверка
корректности исходных данных не проводится.
// Если "tree" = true, выводит по каждому ребру квартет чисел, его задающих
(каждый квартет - вновой строке).
// Если "strings" = true, выводит по каждому ребру соотвествующую ему подстроку из
строки DataS
```

Функции работы со строками / Working with strings

```
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.


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

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


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

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


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.


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


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


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


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

```
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 - величину 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).


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 длиной от 1 до MaxWordLength символов включительно 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 lengths of the repeat that have its starting position as previous element in the Result.
// Returns -1 and empty Result if input data is incorrect.


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.


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.


void GMapMonoisotopicMassTableLD (std::map <char, long double> & MassTable)
//Generates Monoisotopic mass table in the map (long double)


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.


int GPPM (const std::vector <std::string> &s, std::vector <std::vector <long double>> & B, const std::string &Alph, long double z = 0.0, long double d = 2.0)
{

// Генерирует позиционную матрицу вероятностей 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 набор пустой или же строки в нем 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 и заполненная B).
// z и d - параметры для сглаживания (pseudocount): используется формула (Ns+z)/(N+d*z)

// Generates a position probability matrix (PPM) 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 0 items or its 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. If success returns 0 and generated B.
// z and d are pseudocount parameters: (Ns+z)/(N+d*z) is used


double PDist (const std::string& s1, const std::string& s2)
// counts p-distance without checking of the input data correctness


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.


int ConsStringQ1 (std::vector <std::string> &DataS, std::vector<std::string> &QDataS, std::string &TempS, std::string &QTempS, const int method = 1, const std::string &Alph = "ACGT", const int Phred=33)
// Generates a consensus string upon std::vector <std::string> DataS as an input collection of strings and QDataS as their quality.
// The result will be as TempS as consensus string and QTempS as its quality string.
// If multiply consensus strings exist returns the one of them.
// The parameter "Alph" set an alphabet, by default Alph = "ACGT" (other symbols are not considered for forming the consensus string).
// The parameter "Phred" sets the quality scale (by default as Phred33).
```

```
// There are 4 methods to construct the consensus string (set by the parameter
"method"):
// 0 - default - the symbol that has the maximal average probability (quality) for
a given position in consensus string will be chosen. It will have the same
quality.
// 1 - the symbol that has the maximal sum of probability (quality) for a given
position in consensus string will be chosen. it will have the quality = sum of
probability (quality) of this char / number of times that the char occurs at given
position.
// 2 - the symbol that has the maximal sum of probability (quality) for a given
position in consensus string will be chosen. it will have the max quality
(probability) of this char at given position.
// 3 - the symbol that has the maximal probability (quality) for a given position
in consensus string will be chosen. It will have the same quality.
// One may set QDataS as empty: in this case quality will considered as "some
equal" for every symbol at every position.
// So in order to find a consensus string upon a given collection without quality
one may choose method №1 or №2 and empty QDataS.
// If no symbol of the Alph has been found at a given position - the ' ' will be
set there both in the consensus string TempS and in quality string QTempS.
// If any input data is incorrect returns -1 and empty TempS and QTempS, otherwise
returns 0.

// Формирует консенсусную строку (если возможно несколько вариантов такой строки -
то один из них) согласно набру строк DataS с соотвествующим качеством (строки,
задающие качество даны в QDataS на соответствующих позициях).
// Результат - консенсусная строка TempS и соотвествующая ей строка качества
QTempS. В случае успеха возвращается 0, в случае некорректных вводных данных - -1
и пустые указанные строки.
// Параметр "Alph" задает алфавит: только эти символы учитываются при формировании
консенсуса.
// Параметр "Phred" задает шкалу качества (по умолчанию - Phred33).
// Предусмотрены 4 метода формирования консенсусной строки:
// 0 - по умолчанию - символы на каждой данной позиции выбираются согласно
максимальной средней вероятности (качеству), качество присваивается то же.
// 1 - символы на каждой данной позиции выбираются согласно максимальной суммарной
вероятности (качеству), качество присваивается как среднее по данному символу.
// 2 - то же, но качество присваивается как максимальное по данному символу на
данной позиции.
// 4 - выбор символа (и присвоение ему качества) осуществляется исходя из
максимального качества (вероятности) символа на данной позиции.
// Если передать пустой QDataS, то предполагается, что качество везде "какое-то
одинаковое", строка QTempS возвращается пустой. Т.обр., для определения
консенсусной строки согласно лишь частоте
// появления символов достаточно выбрать метод = 1 или 2 и пустой QTempS.
// Если ни одного символа из алфавита на данной позиции не найдено - ставится ' '
и качество по данной позиции задается как ' '.


int ConsStringQ2 (std::vector <std::string> &DataS, std::vector<std::string>
&QDataS, std::string &TempS, std::string &QTempS, const int method = 1, const
std::string &Alph = "ACGT", const char tr = '^', const int Phred=33)
// Модификация функции ConsStringQ1 (см. выше) для целей учета всех вариантов
консенсусной строки.
// В консенсусной строке позиции разделяются символом tr (по умолчанию = '^'), при
этом на каждой позиции может быть несколько символов из алфавита, если с т.зр.
критерия выбора они равновероятны.
// Modification of ConsStringQ1 (see it above) for all the version of consensus
string. For that every position may have >= 1 symbols (if different symbols may be
chosen for this position).
// The positions are separated by the symbol tr (by default is set as '^').
```

```
int JoinOverlapStrings (std::multimap <long long int, std::string> & Locuses,
std::map <long long int, std::string> & JoinedLocuses, const bool Aggregate =
false, const bool NoQuality = false, const int method = 0, const std::string &Alph
= "ACGT", const int Phred=33)
// Функция, которая принимает на вход Locuses, содержащий набор подстрок из
некоторой неизвестной строки, в формате: стартовая позиция в неизвестной строке ->
сама (под)строка,
// и объединяет пересекающиеся строки (результат записывается в JoinedLocuses в
том же формате).
// Области пересечений строятся как консенсусные строки:
// - с помощью функции ConsStringQ1, если необходим какой-либо один из приемлемых
вариантов (bool Aggregate = false), или с помощью ConsStringQ2, если необходимы
все варианты (bool Aggregate = true) (подробнее см описания данных функций),
// - с выбранным методом построения консенсусной строки (0 - 3, подробнее о
методах также  см. описания ConsStringQ1 и ConsStringQ2),
// - с учетом качества (NoQuality=false) или нет (NoQuality=true); при учете
качества считается, что первая половина каждой строки в Locuses содержит
собственно саму строку, а вторая половина - строку, задающую ее качество.
// - Параметр качества задается по шкале Phred с помощью соответствующего
параметра.
// - Алфавит задается параметром Alph, так что при формировании консенсусных строк
будут учитываться только символы из данного алфавита.
// Таким образом, если необходимо просто соединить все пересекающиеся строки в
нектором наборе без учета качества, необходимо задать NoQuality = true, задать
алфавит и в Locuses разместить саму коллекцию объединяемых строк.
// В случае NoQuality = true будет всегда использоваться метод №1
// Например, коллекция вида  0->ACGT, 1->TGTA, 1->TT, 10->TT, 11->TCA в этом
случае объединится как 0->ATGTA, 10->TTC.

// The function joins overlapping strings from Locuses (contain substrings of the
unknown string as pairs: start position -> string) and writes the resulting
collection of the joined strings to JoinedLocuses (has the same format).
// The overlaps are to constructed as consensus strings:
// - using ConsStringQ1 (if any one version of result needed, set bool Aggregate =
false for it) or using ConsStringQ2 (if all the version needed, set bool Aggregate
= true for it), see info on ConsStringQ1 and ConsStringQ2 for details.
// - using the chosen method of consensus generating (set by parameter "method",
see info on ConsStringQ1 and ConsStringQ2 for details).
// - taking into account quality (NoQuality=false, scale is set by parameter
"Phred") or no (NoQuality=true).
//   NOTE that if we take quality into account (NoQuality=false) it is expected
that the first half of every string in Locuses means the string to be joined
itself, and the other half - its quality.
// - the Alphabet should be set by the string Alph. In doing so, only symbols of
the Alph will be taken into account for consensus string generation.
// If NoQuality = true method #1 will be used always.
// So, if we need to join collection 0->ACGT, 1->TGTA, 1->TT, 10->TT, 11->TCA in
any way without any additional info,
// we should set NoQuality = true, Aggregate = false, and have the result: 0-
>ATGTA, 10->TTC.


long double ProfileProbableMer (const std::string &s, int n, const std::vector
<std::vector <long double>> & B, std::vector <std::string> & DataS, long double d
= 0.0000001, std::string Alph = "ACGT")
// Finds all most probable n-mer of the given string s upon position probability
matrix (PPM) B and the Alphabet Alph.
// Returns their probability and all these n-mers contained in DataS.
// If any data incorrect returns empty DataS and -1.0.
// Parameter d sets precision for doubles comparison.

// Находит все наиболее вероятные n-меры в строке s исходя из позиционной матрицы
вероятностей B и возвращает их в векторе DataS. Возвращает значение
соответствующей вероятности.
```

```
// Если данные некорректны, возвращает пустой DataS и -1.0
// Параметр d задает точность при расчете вероятностей.


int MedianString (const int WordLenght, const std::vector <std::string> & DataS,
std::vector <std::string> & MedianS, std::string Alph = "ACGT")
// Finds all median strings (having length = WordLenght) upon given array of
strings in the vector DataS and given Alphabet set by the string Alph.
// Returns distance between found median string(s) and the pattern (i.e. DataS),
all found median strings will contained in the vector MedianS.
// If any data incorrect returns empty MedianS and -1.

// Находит все "медианные" строки заданной длины WordLenght исходя из набора
исходных строк в векторе DataS.
// Возвращает значение дистанции от найденных медианных строк и набора исходных
строк DataS. Сами медианные строки будут в MedianS.
// Если данные некорректны, возвращает пустой MedianS и -1.

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 EditDistA (const std::string &s1, const std::string &s2, std::string &sr1,
std::string &sr2)
// Расширенная версия функции EditDist (см. выше). Возвращает также Edit Distance
Alignment строк s1 и s2 как строки sr1 и sr2 (если несколько вариантов возможны -
один из возможных).
// Extended version of the function EditDist (see it above). Returns also Edit
Distance Alignment of s1 and s2 as sr1 and sr2 (one possible version if many
exists).


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 FindMutatedRepeatsED (const std::string &strShort, const 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 соответствующей 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 mismatch 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 for 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.


int SuffixTreeMake (const std::string &DataS, std::vector <int> & Tree, int b=1)
// Suffix Tree constructing upon a string DataS. If DataS is empty returns -1, if success returns 0.
// Suffix Tree will be contained in the vector Tree, every edge as quartet of integers:
// number of tne start-vertex of edge, number of end-vertex of edge, starting position of substring in DataS, the length of this substring.
// "b" sets the number to start numerating of vertices of suffix tree.
// Построение суффиксного дерева Tree по строке DataS. Возвращает -1, если длина строки <1, в случае успеха вернет 0.
// Суффиксное дерево возвращается в виде вектора Tree, представленного в виде квартетов чисел:
// номер вершины-начала ребра, номер вершины-конец ребра, стартовая позиция подстроки в строке DataS, ее (подстроки) длина.
// параметр b задает, с какого номера будут нумероваться вершины в суффиксном дереве.


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


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.


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: Here the 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".
// Trie: Здесь будет само дерево в виде набора триплетов чисел. Первые два задают ребро графа, а третье - соответствующий символ (букву). Вершины графа нумеруются с 1.


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


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


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


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


int Num (std::string & Numbers, int &a1,int &a2, double &a3)
// Перегон строки, содержащей 3 числа, разделенных пробелами (пары целых 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't contain any other
symbols) to int a1,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.


void Num (std::string & Numbers, std::vector <std::string> & A)

// Modification of "Num"-functions (see them above) for strings.
// Converts string that consists of some strings (separated by spaces) to a vector
of strings
// Модификация функций семейства Num (см. выше) для строк: Перегон строки со
строками, разделенными пробелами, в массив (вектор) строк A.
```

Функции работы с графами / Working with graphs

При работе с графами здесь используется такая структура данных как «**Вектор смежности**».

Невзвешенный граф

Назовем вектором смежности для взвешенного графа упорядоченный набор (массив) четного кол-ва чисел (a[2i], a[2i+1],..., где 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].

Взвешенный граф: целочисленные веса ребер

Назовем вектором смежности для взвешенного графа упорядоченный набор (массив) чисел (a[3i], a[3i+1], a[3i+2],..., где 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").
Данный 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].

Взвешенный граф: нецелочисленные веса ребер (double)

Виду того, что в одном массиве (векторе) нельзя хранить разнородные элементы, предложена следующая реализация. Граф хранится в паре векторов (std::pair < std::vector<int>, std::vector<double>>) где первый вектор является вектором смежности графа без указания весов, а второй вектор содержит соответствующие веса. Таким образом, для ребра, задаваемого парой вершин под индексами 2*i, 2*i+1 первого вектора, вес будет равен элементу под индексом i второго вектора.

Структура данных «Вектор смежности» достаточно компактна, занимает меньше памяти, чем матрица смежности (для разреженных графов), относительно просто реализуется и может быть удобна для решения ряда задач.

При этом:

- Вершины графа могут быть промаркированы и отрицательными числами. При этом в ряде функций происходит приведение номеров вершин к неотрицательным либо положительным при реализации функции (ответ же выдается в исходных, неприведенных номерах вершин).
- Графы могут содержать множественные ребра и множественные петли.

Дополнение. Для обеспечения более быстрого доступа к значению веса ребра в графе, с лета 2019 года в CBioInfCpp также используется модификация Вектора смежности – **Ассоциативный массив смежности**, реализованный на базе контейнера Map. При этом ключом является пара целых чисел int, задающих номера вершин, а значением – вес ребра между ними: int либо double.
При использовании «классического» **Ассоциативного массива смежности** минусом является невозможность хранения множественных ребер. Для решения этой задачи может использоваться «**расширенный ассоциативный массив смежности**» (т. наз. «**мега-мап**»: здесь ключом также является пара чисел, задающих номера соединяемых ребром(-ами) вершин, а значением - вектор весов для всех ребер между этими вершинами – int или double).

The "**Adjacency vector**" is a data structure to represent a graph is used in the library CBioInfCpp.

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).

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].

Adjacency vector of weighted graph: weights are integers
Let "Adjacency vector" of weighted graph (all weights are integers) be a data structure, that contains array of integers such as a[3i], a[3i+1], a[3i+2],... (0-basing indexing).
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].

Adjacency vector of weighted graph: weights are doubles
As we are not able to create an array (or a vector) that contains both integers and doubles let's do the following. Let a graph is represented here as a pair of 2 vectors (i.e. std::pair < std::vector<int>, std::vector<double>>). 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.

The **"Adjacency vector"** may be considered as another one data structure to represent a graph. It is compact, smaller than Adjacency Matrix (for sparse graphs), simple to implement and it may be a convenient one for some tasks and problems solving.

By the way, vertices of a graph may be marked by both positive and non-positive numbers here. In order to implement some function vertices may be renumbered to get started from "0" or "1"; in doing so, the vertices will be assigned their original numbers before the function is complete.

Also any graph may have multiple edges and multiple loops.

Since Summer of 2019 CBioInfCpp also have a modification of Adjacency vector - **Adjacency map**. Adjacency map allows to have quicker access to edge's weight (the key is a pair of vertices numbers (int), mapped value, i.e. weight, may be int or double).
As Adjacency map can't deal with multiple edges, its extended version i.e. **Adjacency mega-map** may be used in such cases. In doing so, the key value of the map is a pair of integers that sets edge(s) between them and the mapped value is a vector that contains weights (int or double) of all edges between these vertices.

```
int UWGraphRead (std::ifstream & fin, std::vector <int> & A)
// Чтение невзвешенного графа в вектор смежности A.
// 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 (i.e. to vector A).
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.


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, 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.


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 std::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 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.
```

```
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


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


int AdjVector2AdjMatrix (std::vector <int> & A, std::vector <std::vector <int>>
&B, const bool weighted, const bool directed)
// Converts "Adjacency vector" A to "Adjacency matrix" B.
// 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
// Returns 0 if success. Returns -1 and empty B if no.


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


int AdjMatrix2AdjVector (std::vector <int> & A, std::vector <std::vector <int>>
&B, const bool weighted, const bool directed)
// Converts "Adjacency matrix" B to "Adjacency vector" A.
// 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 (so if the Main
diagonal of the matrix B contain any odd number for such graph the function will
return -1)
// 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
// Returns 0 if success. Returns -1 and empty A if no.


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


int AdjVectorToAdjMap (const std::vector <int> &A, std::map <std::pair < int, int>
, int> &G2, const bool weighted, const bool directed = true)
// 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.
// Parameter "directed" sets if the graph A is directed or no. For undirected
graph numers of nodes of every edge will be written to G2 in increasing order.
// 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.
Параметр directed задает, является ли граф ориентированным (Истина) или нет.
// Для неориентированных графов номера вершин каждого ребра будут записаны в G2
порядке возрастания.


int AdjVectorToAdjMap (const std::pair < std::vector<int>, std::vector<double>> &
A, std::map <std::pair < int, int> , double> &G2, const bool directed = true)
// Converts Adjacency vector A to Adjacency map G2. Multiple edges will be joined
together.
// Parameter "directed" sets if the graph A is directed or no. For undirected
graph numbers of nodes of every edge will be written to G2 in increasing order.
// 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.
// Параметр directed задает, является ли граф ориентированным (Истина) или нет.
Для неориентированных графов номера вершин каждого ребра будут записаны в G2
порядке возрастания.
// Возвращает -1 в случае некорректности исходных данных.


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.


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.
```

```
int AdjVectorToAdjMegaMap (const std::vector <int> &A, std::map <std::pair < int,
int> , std::vector<int> > &G2, const bool weighted, const bool directed = true)
// Converts Adjacency vector A to Adjacency mega-map G2.
// Adjacency mega-map is an extended version of Adjacency map (and it is built
basing on std::map too) for containing graphs that have multiply loops and
multiply edges.
// In doing so, the key value of the map is a pair of integers that sets edge(s)
between them and the mapped value is a vector that contains weights of all edges
between these vertices.
// 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.
// Parameter "directed" sets if the graph A is directed or no. For undirected
graph numers of nodes of every edge will be written to G2 in increasing order.
// Returns -1 if input data is not correct. Otherwise returns 0.
// Конвертирует Вектор смежности A в ассоциативный "мега-мап" G2. Для
невзвешенного графа считаем вес всех ребер = 1.
// Мега-мап представляет собой ассоциативный массив смежности, предназначенный для
хранения графов с множественными петлями и множественными ребрами.
// При этом ключом является пара чисел, задающих вершины ребер графа между ними, а
значением - вектор весов для всех ребер между этими вершинами.
// Возвращает -1 в случае некорректности исходных данных.
// Параметр weighted задает, является ли граф взвешенным (Истина) или нет.
// Параметр directed задает, является ли граф ориентированным (Истина) или нет.
Для неориентированных графов номера вершин каждого ребра будут записаны в G2
порядке возрастания.


int AdjVectorToAdjMegaMap (const std::pair < std::vector<int>,
std::vector<double>> & A, std::map <std::pair < int, int> , std::vector<double> >
&G2, const bool directed = true)
// Модификация функции AdjVectorToAdjMegaMap (см. выше) для случая нецелочисленных
весов ребер графа.
// Modification of the function AdjVectorToAdjMegaMap (see it above) for not-
integer (double) weights of edges of a graph.


int AdjMegaMapToAdjVector (std::vector <int> &A, const std::map <std::pair < int,
int> , std::vector <int> > &G1)
// Converts Adjacency mega-map G1 to Adjacency vector A. A is considered as
weighted, all weights are integers.
// Adjacency mega-map is an extended version of Adjacency map (and it is built
basing on std::map too) for containing graphs that have multiply loops and
multiply edges.
// In doing so, the key value of the map is a pair of integers that sets edge(s)
between them and the mapped value is a vector that contains weights of all edges
between these vertices.
// Returns -1 if input data is not correct. Otherwise returns 0.

// Конвертирует "мега-мап" G1 в вектор смежности A (только во взвешенный, веса
целочисленны).
// Мега-мап представляет собой ассоциативный массив смежности, предназначенный для
хранения графов с множественными петлями и множественными ребрами.
// При этом ключом является пара чисел, задающих вершины ребер графа между ними, а
значением - вектор весов для всех ребер между этими вершинами.
// Возвращает -1 в случае некорректности исходных данных.


int AdjMegaMapToAdjVector (std::pair < std::vector<int>, std::vector<double>> & A,
const std::map <std::pair < int, int> , std::vector<double> > &G1)
// Модификация функции AdjMegaMapToAdjVector (см. выше) для случая нецелочисленных
весов ребер графа.
// Modification of the function AdjVectorToAdjMegaMap (see it above) for not-
integer (double) weights of edges of a graph.
```

```
int CheckUnvisit (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)


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 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.


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.


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.


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.


int DistanceBFA (std::vector <int> &A, std::vector <long long int> & D, const int
b, const bool weighted, int V = INT_MIN)
// Модификация функции DistanceBFA (см. выше): нет поиска массива предков и вместо
поиска в ширину применен алгоритм Беллмана — Форда.
// Рассчитывает расстояния от заданной вершины b до всех прочих в орграфе.
// Возвращается 1 в случае успеха (вектор D содержит кратчайшие расстояния от
вершины b до вершины i).
```

```
// 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 пустой вектор D.
// 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.

// Modification of the function DistanceBFA (used Bellman-Ford algorithm instead of Breadth-First Search and here is no search for previous vertices for every vertex in such shortest paths).
// 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.
// The Bellman-Ford algorithm 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.


int DistanceBFA (std::pair < std::vector<int>, std::vector<double>> & A,
std::vector <long double> & D, const int b, int V = INT_MIN)
// Модификация функции DistanceBFA (вместо поиска в ширину применен алгоритм Беллмана — Форда для нецелочисленных весов ребер (double)), массив предков не строится.
// Modification of the function DistanceBFA (used Bellman-Ford algorithm instead of Breadth-First Search and here is no search for previous vertices for every vertex in such shortest paths) 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.


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".


int CycleToPath (std::vector <int> Cycle, std::vector<int> &Path)
// Функция "вставляет" цикл Cycle в путь Path с первого значения из Path, которое
встречается в Cycle.
// В случае успеха вернет 0. В случае неуспеха или некорректных исходных данных
вернет -1.

// The function integrate cycle Cycle to the path Path (starting vertex will be
the first from the Cycle that may be found in the Path)
// Returns 0 if success. Returns -1 if it is impossible or input data are
incorrect


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.


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.


int DFS_for_NBPaths (const std::vector <int> & A, const bool w, const int b, const
int bconst, std::vector <int> & Visited, std::vector <int> & Path)
// An auxiliary function for NBPaths (see it below): DFS for finding isolated
cycles
// Вспомогательная функция для NBPaths (см. ниже): обход в глубину графа для
поиска изолированных циклов


void Circuit_for_NBPaths (const std::vector <int> & A, const bool w, int V,
std::vector <std::vector<int> >&Paths, std::vector <int> & Visited)
// An auxiliary function for NBPaths (see it below): finding isolated cycles
// Вспомогательная функция для NBPaths (см. ниже) для поиска изолированных циклов


int NBPaths (std::vector <int> & A, bool w, std::vector <std::vector<int> >&Paths,
bool directed = true)
// Finds all maximal non-branching Paths in a graph, that is set by Adjacency
vector A.
// Parameter "w" sets if A is a weighted graph or no. Parameter "directed" sets if
A is a weighted graph or no.
// The result will be in std::vector <std::vector<int> > Paths. If input data is
incorrect returns -1 and empty Paths. If input data is correct returns 0.
// Vertices may be numbered in different ways (they may be marked by both negative
and non-negative integers). 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.
// The input graph A may have multiple edges (multiple edges will be considered as
non branching paths) and multiple loops (any loop will considered as a non
branching path).

// Функция для поиска всех максимально длинных неразветвляющихся путей в графе,
заданном вектором смежности A. Параметр w задает, является ли граф взвешенным, или
нет.
// Параметр directed - является ли граф ориентированным.
// Результат возвращается в std::vector <std::vector<int> > Paths.
// Возвращает -1 и пустой Paths в случае некорректности исходных данных. В случае
успеха вернет 0.
// Может работать с графами, вершины которых заданы любыми целыми числами, в т.ч.
- отрицательными.
```

```
// Может работать с графами множественными ребрами и множественными петлями
(каждое множественное ребро и каждая петля рассматриваются как отдельный путь).


int NBPaths (std::pair < std::vector<int>, std::vector<double>> & P, std::vector
<std::vector<int> >&Paths, bool directed = true)
// Модификация функции NBPaths (см. выше) для случая нецелочисленного веса ребер.
// Modification of the function NBPaths (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 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 или же
отсутствия любой из этих вершин в графе 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 or there are no
such vertices (b or e) in the graph returns -1 and empty Paths, Flows, MinCut.


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.


int SubGraphsInscribed (std::vector <int> A, std::vector <int> B,
std::unordered_set<std::vector <int>, VectorIntHash> & Result, bool directed =
true, bool InscribedOnly = true, const bool PreThinning=true, const unsigned int
HowManySubgraphs = 0)
// The function initially was intended to find all inscribed subgraphs in
unweighted graph A that are isomorphic to unweighted graph B.
// "Inscribed" means here that
// (1) this subgraph is "glued" to other parts of A only by edges that connected
to those vertices of this subgraph that are begin/ end ones of any max-length non-
branching path of it,
// or (2) this subgraph is a separate connected component of the graph A.
// I.e. for graph B = {0->2, 10->2, 2->3, 3->4, 4->5, 4->6} we could find only A1
= {0->2, 1->2, 2->3, 3->4, 4->5, 4->6} as inscribed isomorphic subgraph of A = {0-
>2, 7->1, 1->2, 2->3, 3->4, 4->5, 4->6}.
// But if we add edge 3->8 to A (in this case A = {3->8, 0->2, 7->1, 1->2, 2->3,
3->4, 4->5, 4->6}), we couldn't find any inscribed isomorphic to B subgraph of A.

// Since 12/2020 the function can also be used to find all (not only inscribed)
subgraphs of unweighted graph A that are isomorphic to unweighted graph B.
// To do so the function was modified to work with all edges of these graphs as
well as with their max-length non-branching paths (that had been implemented
before).

// Graph A is set by an Adjacency vector.
// Graph B is set by an Adjacency vector.
// Both A and B are unweighted.
// Bool "directed" sets if A and B are directed graphs or no.

// If InscribedOnly == false the function finds all (not only inscribed) subgraphs
of unweighted graph A that are isomorphic to unweighted graph B.
// If InscribedOnly == true the function looks for "inscribed" ones only.
// PreThinning is an additional parameter that sets should the function skip stage
PreThinning of input data or no.
// As working time both of the function as a whole and its stages is rather
depends on input data sometimes one may try to play with this parameter.
// HowManySubgraphs sets the upper limit how many subgraphs should be found. If
HowManySubgraphs == 0 the function looks for all fitting subgraphs.

// Result: an unordered_set named "Result" that contains all subgraphs of A found
(set by Adjacency vectors).
// The function also returnes the number of different subgraphs of A that are
isomorphic to B (i.e. the size of "Result");
// if input data are incorrect returns -1 and empty Result.
// A and B may have >=1 connected components; also A and B may have multiply
edges.

// Функция изначально находила все "вписанные" подграфы графа A, изоморфные графу
B.
// «Вписанным» подграфом графа A здесь называется такой его подграф, который может
быть «приклеен» к другим частям графа A только за счет ребер,
// инцидентных лишь граничным вершинам его (подграфа) неразветвляющихся путей
максимальной длины (при этом граф A может содержать и иные компоненты связности).
// Т.е. для графа B = {0->2, 10->2, 2->3, 3->4, 4->5, 4->6} будет найден
изоморфный ему подграф A1, "вписанный" в граф A = {0->2, 7->1, 1->2, 2->3, 3->4,
4->5, 4->6}, и этот подграф A1 = {0->2, 1->2, 2->3, 3->4, 4->5, 4->6},
// однако если в граф A добавить ребро 3->8 (A = {3->8, 0->2, 7->1, 1->2, 2->3, 3-
>4, 4->5, 4->6}), то функция не найдет в A изоморфных B "вписанных" подграфов.

// Графы A и B д.б. невзвешеными, задаются векторами смежности, могут быть
ориентированными (bool directed = true) и нет.
```

```
// С декабря 2020 функция позволяет находить все подграфы графа A, изоморфные графу B, а не только "вписанные".
// Для этого нужно задать параметр InscribedOnly = 0. Если InscribedOnly == 1, то поиск идет быстрее, но только по "вписанным" подграфам.
// Адаптация осуществлена дополнением возможности рассмотрения всех ребер рассматриваемых графов (до этого рассматривались только non-branching paths).

// Функция может работать с графами, содержащими несколько несвязных компонет, а также с графами с множественными ребрами.
// При работе с множественными ребрами из A выбираются те, что имеют кратность не менее, чем соответствующие им в B

// Параметр PreThinning определяет, нужно ли производить предварительное "прореживание". Вспомогательный параметр для оптимизации времени работы,
// т.к. скорость прохождения различных этапов алгоритма, как и самого алгоритма, сильно зависит от исходных данных.
// Параметр HowManySubgraphs задает верхний предел кол-ва подграфов, которое нужно найти. Если HowManySubgraphs == 0, то производится поиск всех возможных подграфов.


int SubGraphsInscribed (std::vector <int> A, std::vector <int> B,
std::set<std::vector <int>> & Result, bool directed = true, bool InscribedOnly = true, const bool PreThinning=true, const unsigned int HowManySubgraphs = 0)
// The version of the function SubGraphsInscribed: results will be in set Result. It works slower.
// Версия функции SubGraphsInscribed: результаты будут в set Result. Работает несколько дольше.


template < typename Tf>
int SubGraphsInscribedM (std::vector <int> A, std::vector <int> B,
std::set<std::vector <int>> & Result, bool directed = true, bool InscribedOnly = true, const bool PreThinning=true, const unsigned int HowManySubgraphs = 0,
std::map<int, Tf> Af={}, std::map<int, Tf> Bf={})
// The experimental version of the function SubGraphsInscribed: vertices of graphs may have marks (set by std::map<int, Tf> Af for graph A and by std::map<int, Tf> Bf for graph B).
// As isomorphic may be considered vertices, that (1) have no marks or (2) both have equal marks.
// If Af or Bf have a mark for a vertex which number doesn't appears at graph A / graph B, it should be considered as incorrect input data (returns -1).

// Экспериментальная версия функции SubGraphsInscribed: Вершины графов A и B могут иметь метки (заданы в std::map<int, Tf> Af для A и в std::map<int, Tf> Bf для B).
// Вершины будут считаться изоморфными если они либо (1) обе не имеют меток, либо (2) обе имеют метки, причем равные.
// Случай наличия в Af либо Bf ключа, не совпадвющего с номером какой-либо вершины в A или B рассматривается как некорректные исходные данные (возвращает -1).


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, multiple 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.


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.


int DistanceTS (std::vector <int> &A, std::vector <long long int> & D, const int b, const bool weighted, int V = INT_MIN)
// Модификация функции DistanceTS (см. выше) - не рассчитывается массив "предков"
// Modification of the function DistanceTS (see it above): no Prev finding


int DistanceTS (std::vector <int> &A, std::vector <int> & D, const int b, std::vector <int> & Prev, const bool weighted, int V = INT_MIN)
// Модификация функции DistanceTS (см. выше) - расстояния в int.
// Modification of the function DistanceTS (see it above): distances will be int, not long long int. Be careful with data.


int DistanceTS (std::vector <int> &A, std::vector <int> & D, const int b, const bool weighted, int V = INT_MIN)
// Модификация функции DistanceTS (см. выше) - расстояния в int и не рассчитывается массив "предков".
 // Modification of the function DistanceTS (see it above): distances will be int, not long long int. Be careful with data. No Prev finding too.
```

```
int DistanceTS (std::pair < std::vector<int>, std::vector<double>> & A,
std::vector <long double> & D, const int b, int V = INT_MIN)
// Модификация функции DistanceTS (см. выше) для случая нецелочисленных весов
ребер (double) и без поиска массива "предков".
// Modification of the function DistanceTS (see it above) for not-integer (double)
weights of edges of a graph. No Prevs finding too.
// 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.


int MakeSubgraphSetOfVertices (const std::vector <int>&A, std::vector
<int>&Subgraph, const std::set <int> &Vertices, const bool weighted)
// Функция для выделения подграфа Subgraph из графа A, такого что все вершины
искомого подграфа задаются set <int> Vertices.
// Граф A задается вектором смежности A, веса - целочисленны или граф
невзвешенный.
// параметр weighted задает, является ли граф взвешенным.

// The function makes a subgraph Subgraph from the graph A as follows: we take
only vertices from the set <int> Vertices.
// Graph A is set by Adjacency vector A. It may be weighted or no (set by
"weighted"), weights may be integers only.


int MakeSubgraphSetOfVertices (const std::vector <int>&A, std::vector
<int>&Subgraph, const std::unordered_set <int> &Vertices, const bool weighted)
// Функция для выделения подграфа Subraph из графа A, такого что все вершины
искомого подграфа задаются unordered_set <int> Vertices.
// Граф A задается вектором смежности A, веса - целочисленны или граф
невзвешенный.
// параметр weighted задает, является ли граф взвешенным.

// The function makes a subgraph Subgraph from the graph A as follows: we take
only vertices from the unordered_set <int> Vertices.
// Graph A is set by Adjacency vector A. It may be weighted or no (set by
"weighted"), weights may be integers only.


int MakeSubgraphSetOfVertices (const std::pair < std::vector<int>,
std::vector<double>> & A, std::pair < std::vector<int>, std::vector<double>>
&Subgraph, const std::set <int> &Vertices)
// Модификация функции MakeSubgraphSetOfVertices (см.выше) для случая
нецелочисленных весов ребер.
// Modification of the function MakeSubgraphSetOfVertices (see it above) for not-
integer (double) weights of edges of a graph.


int MakeSubgraphSetOfVertices (const std::pair < std::vector<int>,
std::vector<double>> & A, std::pair < std::vector<int>, std::vector<double>>
&Subgraph, const std::unordered_set <int> &Vertices)
// Модификация функции MakeSubgraphSetOfVertices(см. выше) для случая
нецелочисленных весов ребер для unordered_set
// Modification of the function MakeSubgraphSetOfVertices (see it above) for not-
integer (double) weights of edges of a graph and unordered_set Vertices.


int AdjVectorEdgesMultiplicity (const std::vector <int> &A, std::map <std::pair <
int, int> , int> &G2, const bool weighted, bool directed = true)
// Counts multiplicity of edges of a graph that is set by Adjacency vector A.
```

```
// 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.
// Parameter "directed" sets if the graph A is directed or no.
// For a DIRECTED graph the result will be formed in the map G2 as follows:
// key is the pair of integers corresponding to the source and the sink vertices of an edge, and the value is its multiplicity.
// For example "G2[std::pair <int, int>(1, 2)] = 3" means that directed edge 1->2 has its multiplicity = 3.
// For UNDIRECTED graph G2 will contain edge multiplicity for both keys std::pair <int, int> (vertex1, vertex 2) and std::pair <int, int> (vertex2, vertex 1).
// For example for undirected graph will be so: "G2[std::pair <int, int>(1, 2)] = G2[std::pair <int, int>(2, 1)] = 3".
// Returns -1 and empty G2 if input data is not correct. Otherwise returns 0.

// Возвращает кратность ребер графа, заданного вектором смежности A, в ассоциативном массиве смежности G2.
// Параметр weighted задает, является ли граф взвешенным (Истина) или нет.
// Параметр directed задает, является ли граф ориентированным (Истина) или нет.
// К примеру, для ориентированного графа "G2[std::pair <int, int>(1, 2)] = 3" означает, что направленное ребро 1->2 имеет кратность = 3.
// Для неориентированного будут храниться одинаковые значения для пар вершин (vertex1, vertex 2) и (vertex2, vertex 1).
// Т.е. для неориентированного графа будет верной подобная запись: G2[std::pair <int, int>(1, 2)] = G2[std::pair <int, int>(2, 1)] = 3
// Возвращает -1 и пустой G2 в случае некорректности исходных данных. Если успех - вернет 0.


int AdjVectorEdgesMultiplicity (const std::vector <int> &A, std::unordered_map <std::pair < int, int> , int, PairIntHash> &G2, const bool weighted, bool directed = true)
// Modification to return result in unordered_map
// Модификация AdjVectorEdgesMultiplicity (см. выше) для возврата результата в unordered_map.


int AdjVectorEdgesMultiplicity (const std::pair < std::vector<int>, std::vector<double>> & A, std::map <std::pair < int, int> , int> &G2, bool directed = true)
// Modification of the function AdjVectorEdgesMultiplicity (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.
// Returns -1 and empty G2 if input data is not correct. Otherwise returns 0.

// Модификация AdjVectorEdgesMultiplicity (см. выше) для случая графа с нецелочисленными весами ребер.


int AdjVectorEdgesMultiplicity (const std::pair < std::vector<int>, std::vector<double>> & A, std::unordered_map <std::pair < int, int> , int, PairIntHash> &G2, bool directed = true)
// Modification to return result in unordered_map
// Модификация AdjVectorEdgesMultiplicity (см. выше) для возврата результата в unordered_map.


int NeighborJoiningUndirectedGraph (const std::vector <std::vector <int>> B, std::pair < std::vector<int>, std::vector<double>> &A)
// Конструирует дерево (неориентированный граф) методом присоединения ближайшего соседа на основе матрицы дистанций B, результат - дерево - возвращается в векторе смежности A. Нумерация вершин графа ведется с 1.
```

```
// Возвращает 0 в случае успеха; в случае некорректных данных (пустая или не
квадратная  или содержащая отрицательные элементы матрица B), вернет -1.

// Generates a tree (undirected graph) using Neighbor Joining method (as an
Adjacency vector A, 1-based indexing of vertices) upon a distance matrix B.
// If any data incorrect (B has zero lines (i.e. empty) or has negative values or
is not a square matrix) returns empty A and -1. If success returns 0.


int UPGMA_UndirectedGraph (const std::vector <std::vector <int>> B, std::pair <
std::vector<int>, std::vector<double>> &A)
// Конструирует дерево (неориентированный граф) методом UPGMA на основе матрицы
дистанций B, результат - дерево - возвращается в векторе смежности A. Нумерация
вершин графа ведется с 1.
// Возвращает 0 в случае успеха; в случае некорректных данных (пустая или не
квадратная  или содержащая отрицательные элементы матрица B), вернет -1.

// Generates a tree (undirected graph) using UPGMA method (as an Adjacency vector
A, 1-based indexing of vertices) upon a distance matrix B.
// If any data incorrect (B has zero lines (i.e. empty) or has negative values or
is not a square matrix) returns empty A and -1. If success returns 0.


int NeighborJoiningUndirectedGraph (const std::vector <std::vector <double>> B,
std::pair < std::vector<int>, std::vector<double>> &A)
// Модификация функции для нецелочисленной матрицы дистанций B.
// Modification for distance matrix B of doubles.


int UPGMA_UndirectedGraph (const std::vector <std::vector <double>> B, std::pair <
std::vector<int>, std::vector<double>> &A)
// Модификация функции для нецелочисленной матрицы дистанций B.
// Modification for distance matrix B of doubles.


void DFSAllPathsDGraph (std::vector <int>&A, std::vector <int> &Visited, const int
&b, const int &e, const bool weighted, std::set <std::vector <int>> & GPath,
std::vector <int> &Path)
// An auxiliary function for AllPathsDGraph (see it below)
// Вспомогательная функция для AllPathsDGraph (см. ниже)


int AllPathsDGraph (std::vector <int>&A, const int &b, const int &e, const bool
weighted, std::set <std::vector <int>> & GPath)
// An experimental function to find all paths from the vertex b to the vertex e of
directed graph that is set by Adjacency vector A. May be too slow or have some
mistakes.
// A may be weighted or no (set by weighted).
// Returns 0 and set of paths found in GPath, if input data are incorrect returns
-1 and empty GPath.
// Экспериментальная функция для поиска всех путей в ориентированном графе из
вершины b в вершину e. Может работать неточно и долго.
// Граф задается вектором смежности A. weighted задает, взвешенный ли он.
// Возвращает 0 и найденные пути в GPath, в случае некорректных исходных данных
вернет пустой GPath и -1.


int DFS_for_Cycles (const std::vector <int> & A, const bool w, const int b, const
int bconst, std::vector <int> & Visited, std::vector <int> & Path, std::set
<std::vector <int>> &GPath, int &mn, const bool &directed)
// An auxiliary function for Cycles_in_Graph (see it below): DFS for finding
cycles
// Вспомогательная функция для Cycles_in_Graph (см. ниже): обход в глубину графа
```

```
int Cycles_in_Graph (std::vector <int> & A, const bool w, std::set
<std::vector<int> >&Paths, std::set <int> &StartV, const bool directed = true)
// An experimental function to find simple cycles in graph that is set by
Adjacency vector A. May be too slow or have some mistakes.
// A may be weighted or no (set by w) and directed or no (set be directed).
// If StartV is not empty, the function searches only for cycles that contain any
vertex in StartV.
// Returns the number of cycles found and set of cycles themselves in Paths, if
input data are incorrect returns -1 and empty Paths.
// Экспериментальная функция для поиска простых циклов в графе. Может работать
неточно и долго.
// Если множество StartV непустое, ищет циклы только через эти вершины.
// Граф задается вектором смежности A. w задает, взвешенный ли он, a directed -
ориентированный ли он.
// Возвращает кол-во найденных циклов и сами найденные циклы в Paths, в случае
некорректных исходных данных вернет пустой Paths и -1.


int DFSSCC1 (const std::vector <int> & A, const int b, std::vector <int> &
Visited, std::vector <int> & order, const bool weighted)
//Вспомогательная функция для упорядочивания вершин орграфа для поиска
сильносвязных его компонент

// An auxiliary function to order vertices of a graph for the function
SCCGraph_Vertices


int DFSSCC2 (const std::vector <int> & A, const int b, std::vector <int> &
Visited, std::vector <int> & order,std::set <int> & R, const bool weighted)

//Вспомогательная функция для поиска вершин очередной сильно связной компоненты
для SCCGraph_Vertices

// An auxiliary function to find vertices of a graph for a strongly connected
component (for the function SCCGraph_Vertices).

int SCCGraph_Vertices (std::vector <int> & A, std::vector <std::set <int>> & G,
const bool weighted, int mn = 0, int V = INT_MIN)
// Функция для нахождения наборов вершин по всем компонентам сильной связности
(т.е. областей сильной связности) ориентированного графа, заданного вектором
смежности A. Параметр weighted задает, является ли граф взвешенным.
// Также на вход подается номер наибольшей вершины V (если не передан,
рассчитывается самостоятельно как номер наибольшей вершины в ребрах) и номер
минимальной вершины (по умолчанию = 0).
// В случае успеха возвращает число сильно связанных компонент. Возвращает и
заполненный вектор G, каждый элемент которого - набор вершин очередной компоненты
связности.
// В случае некорректных входных данных возвращает -1 и пустой G.

// The function finds collection of vertices for each strongly connected component
of the directed graph, that is set by an Adjacency vector A.
// These collections are to be contained in vector G, i.e. G[i] contains a
collection of vertices of the i-th strongly connected component.
// Input data: Adjacency vector A, 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 minimum vertex number mn (== 0 by default), bool weighted, that sets if the
graph is weighted.
// If input data is incorrect the function returns "-1" and empty G.
```

```
int CCGraph_Vertices (std::vector <int> & A, std::vector <std::set <int>> & R,
const bool weighted, int mn = 0, int V = INT_MIN)
// Функция для нахождения наборов вершин по всем компонентам связности (т.е.
областей связности) неориентированного графа, заданного вектором смежности A.
Параметр weighted задает, является ли граф взвешенным.
// Также на вход подается номер наибольшей вершины V (если не передан,
рассчитывается самостоятельно как номер наибольшей вершины в ребрах) и номер
минимальной вершины (по умолчанию = 0).
// В случае успеха возвращает число связанных компонент. Возвращает и заполненный
вектор R, каждый элемент которого - набор вершин очередной компоненты связности.
// В случае некорректных входных данных возвращает -1 и пустой R.

// The function finds collection of vertices for each connected component of the
undirected graph, that is set by an Adjacency vector A.
// These collections are to be contained in vector R, i.e. R[i] contains a
collection of vertices of the i-th connected component.
// Input data: Adjacency vector A, 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 minimum vertex number mn (== 0 by default), bool weighted, that sets if the
graph is weighted.
// If input data is incorrect the function returns "-1" and empty R.
```

Вспомогательные функции / Auxiliary functions

```
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


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


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


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


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 0.
// Returns index in vector (int) of the first element = a. Search starts from index "start" with step = "step". If step<1 step will be set as 1. If start<0 start will be set as 0. If no such element found the function returns -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 0.
// Returns index in vector (long long int) of the first element = a. Search starts from index "start" with step = "step". If step<1 step will be set as 1. If start<0 start will be set as 0. If no such element found the function returns -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 0.
// 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 step<1 step will be set as 1. If start<0
start will be set as 0. If no such element found the function returns -1.
// 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".


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 0.
// 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 step<1 step will be set
as 1. If start<0 start will be set as 0. If no such element found the function
returns -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 0.
// 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 step<1 step will be set as 1. If start<0
start will be set as 0. If no such element found the function returns -1.
// 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".


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 0.
// 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 -1. If step<1 step will be set as 1. If start<0 start will be
set as 0.


int FindIn (std::vector <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 0.
// Returns index in vector (string) of the first element = a. Search starts from
index "start" with step = "step". If step<1 step will be set as 1. If start<0
start will be set as 0. If no such element found the function returns 0.
```

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


int GraphVerticesNumbersCompress (std::vector <int> & P, const bool weighted)
// Renumbering of vertices of the graph P (set by Adjacency vector P) to make the
sequence of numbers of vertices as a row of not-negative integers with no blanks.
Weighted sets if the graph P is weighted.
// Перенумеровывает вершина графа, заданного вектором смежности P, таким образом,
чтобы все номера вершин составляли ряд неотрицательных целых чисел без пропусков.
Параметр weighted задает, является ли граф взвешенным.


std::string CIGAR1 (const std::string &S0, const std:: string & S2, int npos = 0)
// Формирует строку CIGAR по результату "прикладывания" строки S2 к строке S0 с
позиции npos;
// в случае некорректных данных (длина какой-л. строки равна 0, начальная позиция
отрицательна или приложенная S2 "выезжает" за границу S0 - возвращает пустую
строку
// Generates CIGAR-string as a result of "fitting" of the string S2 to s0
(starting position == npos). If any data is incorrect returns empty string.


int ScoreStringMatrix (const std::vector <std::string> &s)
// Returnes a score (i.e. total number of mismatches) upon vector of strings s. If
input data is incorrect returns 0.
// Возвращает суммарное количество всех несовпадений символов по каждой позиции по
набору строк s. Если данные некорректны вернет -1.


int GenRandomUWGraph (std::vector <int> &P, int v, int e, int b=0)
// Вспомогательная функция для генерации случайного невзвешенного графа в векторе
смежности P
// An auxiliary function that generates a random unweighted graph P (set by
Adjacency vector P)
// e means the number of edges, v means the number of vertices, b means the
minimal number to be assigned to vertex


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"


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 partitions (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"
```

```
template < typename TMEE>
int MatrixEraseElement (std::vector <std::vector <TMEE>> & B, const int &j)

// Удаление элемента j из матрицы B, матрица может содержать элементы произвольных типов.
// Нумерация элементов - с нуля. если матрица пустая или элемент в ней не содержится (задан слишком большой его номер) - выдаст -1, если успех - 0.
// Строки матрицы могут быть, в принципе, не равны друг другу по длине

// Erasing an element #j from matrix B (0-based indexing)
// If input data are incorrect return -1. If success returns 1
// NB Matrix B may have rows of different length.


int UWGraphFromWGraph (const std::vector <int> &P1, std::vector <int> &P2)
// Construct unweighted graph P2 upon weighted graph P1 (only edges without their weights are to be included to P2).
// P1 and P2 are set by adjacency vector.
// If success returns 0. If input data incorrect returns -1 and empty P2
// Конструирует невзвешенный граф P2 из взвешенного P2 (копируются ребра без весов). Вектора заданы векторами смежности.
// Если успех - возвращает 0, если исходные данные некорректны - вернет -1 и пустой P2.


int WGraphFromUWGraph (const std::vector <int> &P1, std::vector <int> &P2)
// Construct weighted graph P2 upon unweighted graph P1 (let the weight of every edge is = 1).
// P1 and P2 are set by adjacency vector.
// If success returns 0. If input data incorrect returns -1 and empty P2
// Конструирует взвешенный граф P2 из невзвешенного P2 (считается, что вес любого ребра =1). Вектора заданы векторами смежности.
// Если успех - возвращает 0, если исходные данные некорректны - вернет -1 и пустой P2.



int PathFromPrev (std::vector <int> &Path, const std::vector <int> &Prev, const int b, const int e, const bool ignoreb = false)
// Вспомогательная функция. Конструирует путь Path в графе от вершины b до e согласно массиву предков Prev. Prev [i] содержит номер вершины-предка для вершины i, либо -1, если предка нет.
// Путь собирается "от конца", т.е. от вершины e. Если bool ignoreb = false, то ищется путь строго от b до e. Иначе - началом пути также может быть некоторая вершина, не имеющая предка.
// Если входные данные некорректны, вернет -1 и пустой Path. В случае успеха вернет 0.

// An auxiliary function. Constructs path Path in some graph from vertex b to vertex e upon an array of Prev.
// Prev [i] contains the number of the previous vertex of the vertex i in the Path or -1 if vertex i has no previous vertex.
// Returns -1 and empty Path if input data are incorrect. If success returns 0.
// If bool ignoreb = false the function looks for path from b to e exactly.
// Otherwise it may also return a path starting from some vertex with no previous one, if only such path may be constructed to the vertex e (from-end-to begin construction is implemented).


std::string GenerateAlphabet (const std::vector <std::string> &DataS)
// Generates an alphabet upon the given vector of strings DataS. Symbols will be ordered under ASCII.
// Формирует алфавит из символов, входящих в набор строк DataS. Порядок символов в алфавите задается ASCII.
```

```
std::string GenerateAlphabet (const std::string &DataS)
// Generates an alphabet upon the given string DataS. Symbols will be ordered
under ASCII.
// Формирует алфавит из символов, входящих в строку DataS. Порядок символов в
алфавите задается ASCII.
```