01 - Component labeling

Write a program to solve the component labeling problem. A binary image is stored as an n * n (n ≤ 65536) array of 0s ad 1s. The 1s represent objects, while the 0s represent empty space between objects. The component labeling problem is to associate a unique positive integer with every object. When the program completes, every 1-pixel will have a positive integer label. A pair of 1-pixels have the same label if and only if they are the same component (object). There are at most 2^{32}-1 components. The 1-pixels are in the same component if they are linked by a path of 1-pixels. Two 1-pixels are contiguous if they are adjacent to each other, either horizontally or vertically. For example, given this image:

```
1 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0
0 1 1 1 0 0 0 0
0 1 1 0 1 1 1 1
0 0 0 0 1 0 0 1
1 1 1 0 1 1 0 1
1 1 1 1 0 1 1 1
0 0 0 0 0 0 1 1
```

one (but certainly not the only) valid output would be:

```
 1  0  0  0  0  0  0  0
 0 10  0 10  0  0  0  0
 0 10 10 10  0  0  0  0
 0 10 10  0 29 29 29 29
 0  0  0  0 29  0  0 29
41 41 41  0 29 29  0 29
41 41 41 41  0 29 29 29
 0  0  0  0  0  0 29 29
```

Note that a 0 in a particular position of the input image results in a 0 in the same position in the output image. If two positions in the output image have the same integer value, it means there is a path of 1s between the two positions in the input image.

component_labeling.hh

```
/**
 * @param input the input n*n bool matrix
 * @param n the number of matrix rows and columns
 * @param output the output n*n long unsigned int matrix
 */
void component_labeling(long unsigned int** output,const long unsigned int n,const bool** input);
```

02 - 0-1 knapsack

We have a knapsack of volume V and n types of items, (i=0,1,...,n-1). Items of type i have weight w_i and volume v_i. The objective is to determine how many items of each type should be placed in the knapsack so as to maximize the total weight of the knpsack without exceeding its volume (V). At most one item of each type can be selected.

For instance, if we have a knapsack of volume V=1000 and n=5 types of items, i_0 (weight 40 and volume 200), i_1 (weight 50 and volume 314), i_2 (weight 100 and volume 198), i_3 (weight 95 and volume 500), and i_4 (weight 30 and volume 300). The most valuable knapsack not exceeding the volume limit contains only the items i_0, i_2, and i_3.

knapsack.hh

```
typedef struct {
        long unsigned int weight;
        long unsigned int volume;
} item;
/**
 * @param output The output array of item quantities
 * @param capacity The knapsack capacity
 * @param n The number of available items
 * @param items The array of n available items
 */
void knapsack(long unsigned int* output,const long unsigned int capacity,const long unsigned int n,const item* items);
```

03 - Harmonic progression sum

The simplest harmonic progression is

$$1/1, 1/2, 1/3, ...$$

Let $S_n = \Sigma_{i=1}^{n} (1/i)$;

Compute these sums to arbitrary precision after the decimal point. For example, $S_7 = 2.592857142857$, to 12 digits of precision after the decimal point.

```
/**
 * @param argc the argc
 * @param argv the argv
 * @param output the value of the sum
 * @param n the number of terms that will be considered in the summation
 * @param d the number of digits after the decinal point
 */
void sum(int argc, char** argv, char* output,const long unsigned int d,const long unsigned int n);
```

04 - Game of life

The Game of Life is not your typical computer game. It is a 'cellular automaton', and was invented by Cambridge mathematician John Conway.

This game became widely known when it was mentioned in an article published by Scientific American in 1970. It consists of a collection of cells which, based on a few mathematical rules, can live, die or multiply. Depending on the initial conditions, the cells form various patterns throughout the course of the game.

The Rules

- For a space that is 'populated':
  - Each cell with one or no neighbors dies, as if by loneliness.
  - Each cell with four or more neighbors dies, as if by overpopulation.
  - Each cell with two or three neighbors survives.
- For a space that is 'empty' or 'unpopulated'
  - Each cell with three neighbors becomes populated.

life.hh

```
/**
 * @param argc the argc
 * @param argv the argv
 * @param output the final state of the n * n matrix at the g^th generation
 * @param g the number of generations to calculate
 * @param n the order of the matrix
 * @param input the initial state of the n * n matrix at the 0^th generation
 */
void life(int argc,char* argv[],bool** output,const long unsigned int g,const long unsigned int n,const bool** input);
```

05 - Binary search tree

A binary search tree is a way of organizing n keys from a linearly ordered set to ensure their retrieval in Θ(log n) time. Given the probability of each key being accessed, this task is to create an optimal binary search tree that minimizes the average search time. For instance, suppose we have the following probabilities:

brown	0.16
dog	0.13
fox	0.06
jumped	0.08
lazy	0.07
over	0.17
quick	0.05
the	0.28

Then, the corresponding binary tree is:

arvore

06 - Convex hull

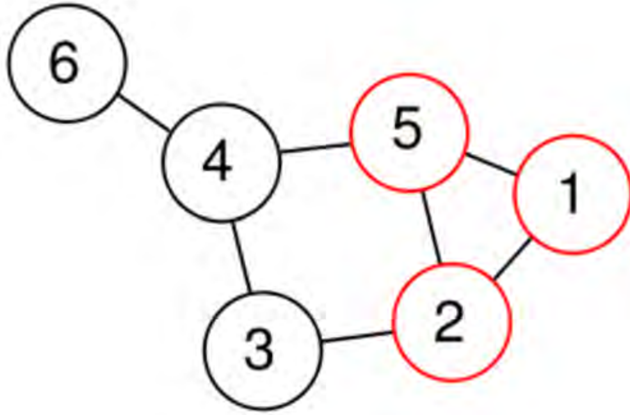
A convex hull is the smallest convex set containing a given collection of points in a real linear space. Computing a convex hull (or just "hull") is one of the first sophisticated geometry algorithms, and there are many variations of it. The most common form of this algorithm involves determining the smallest convex set (called the "convex hull") containing a discrete set of points. This algorithm also applies to a polygon, or just a set of segments, whose hull is the same as the hull of its vertex point set. There are numerous applications for convex hulls: collision avoidance, hidden object determination, and shape analysis to name a few.

The input for this task consists of a set of points, with the corresponding x and y coordinates. The output consists of the total number of hulls, that is, after the first hull is found, all the points within this hull are excluded, so that another hull can be found. This process continues until all hulls are found, and the total number of hulls is found.

```
typedef struct {
        long unsigned int x;
        long unsigned int y;
} point;
/**
 * @param argc the argc
 * @param argv the argv
 * @param h the size of the set of points on the convex hull
 * @param output the set of points on the convex hull
 * @param n the size of the set of input points
 * @param input the set of input points
 */
void convex_hull(int argc,char* argv[],long unsigned int* h,point* output,const long unsigned int n,const point* input);
```

07 - Maximum Clique

A clique in a graph is a set of pairwise adjacent vertices, or in other words, an induced subgraph which is a complete graph. The maximum clique problem, is to find the largest clique in a graph. In the graph in the figure below, vertices 1, 2 and 5 form a clique (which is maximum), because each has an edge to all the others. This task is to find all maximum cliques in a graph.



The input for this task consists of a graph. The output for this task consists of all maximum cliques in the input graph.

```
/**
 * @param argc the argc
 * @param argv the argv
 * @param clique2 the input graph (edges)
 * @param n the number of nodes
 * @param output the output set of maximum cliques
 */
void clique(int argc, char** argv, set >& output, const set >& clique2, const unsigned int n);
```

08 - Matrix multiplication

The product C of two matrices A and B is defined as

$c_{ik}==a_{ij}b_{jk}$, (1)

where j is summed over for all possible values of i and k and the notation above uses the Einstein summation convention. The implied summation over repeated indices without the presence of an explicit sum sign is called Einstein summation, and is commonly used in both matrix and tensor analysis. Therefore, in order for matrix multiplication to be defined, the dimensions of the matrices must satisfy

$(n \times m)(m \times p)==(n \times p)$, (2)

where $(a \times b)$ denotes a matrix with a rows and b columns. Writing out the product explicitly,

$[c_{11}\ c_{12}\ ...\ c_{1p};\ c_{21}\ c_{22}\ ...\ c_{2p};\ ...;\ c_{n1}\ c_{n2}\ ...\ c_{np}]==[a_{11}\ a_{12}\ ...\ a_{1m};\ a_{21}\ a_{22}\ ...\ a_{2m};\ ...;\ a_{n1}\ a_{n2}\ ...\ a_{nm}][b_{11}\ b_{12}\ ...\ b_{1p};\ b_{21}\ b_{22}\ ...\ b_{2p};\ ...;\ b_{m1}\ b_{m2}\ ...\ b_{mp}]$, (3)

where

$c_{11} = a_{11}b_{11}+a_{12}b_{21}+...+a_{1m}b_{m1}$ (4)

$c_{12} = a_{11}b_{12}+a_{12}b_{22}+...+a_{1m}b_{m2}$ (5)

$c_{1p} = a_{11}b_{1p}+a_{12}b_{2p}+...+a_{1m}b_{mp}$ (6)

$c_{21} = a_{21}b_{11}+a_{22}b_{21}+...+a_{2m}b_{m1}$ (7)

$c_{22} = a_{21}b_{12}+a_{22}b_{22}+...+a_{2m}b_{m2}$ (8)

$c_{2p} = a_{21}b_{1p}+a_{22}b_{2p}+...+a_{2m}b_{mp}$ (9)

$c_{n1} = a_{n1}b_{11}+a_{n2}b_{21}+...+a_{nm}b_{m1}$ (10)

$c_{n2} = a_{n1}b_{12}+a_{n2}b_{22}+...+a_{nm}b_{m2}$ (11)

$c_{np} = a_{n1}b_{1p}+a_{n2}b_{2p}+...+a_{nm}b_{mp}$. (12)