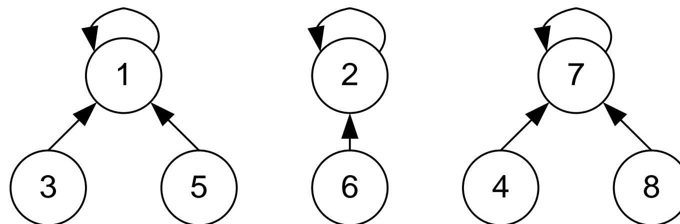
Disjoint Sets data structure

Let we have some elements, each one is located in some separate set. The data structure ***disjoint sets*** supports the following operations:

1. union of any two setsl
2. defining the set containing the given elementl
3. creation of a new element that is placed in a separate set

The elements of the sets will be stored in the form of trees: one tree corresponds to one set. The root of the tree is the ***representative*** (leader) of the set. Each node of the tree contains a pointer to the ***parent*** (father). The pointer of the node that itself is the representative of the set points to it itself.

Example. onsider a system of three sets {1, 3, 5}, {2, 6}, {4, 7, 8}.



The representatives of the first, second, and third sets, respectively, are vertices 1, 2, and 7.

It is convenient to store the sets not in the form of trees, but in one array that we shall call ***parent array***. Each of its elements contains a link to its ancestor in the tree. For the roots of trees, the ancestor will be themselves. For example, for the above sets, the parent array will look like this:

1	2	3	4	5	6	7	8
1	2	1	7	1	2	7	7

Vertices 1, 2, and 7, corresponding to the roots of the trees, are colored yellow.

The **interface** of disjoint sets system consists of three operations:
make_set(v) adds a new element v, placing it in a new set.

```
void make_set(int v)
{
  parent[v] = v;
}
```

Repr(v) returns the ***representative*** of the set containing the element v. It is selected in each set by the data structure itself and can be changed over the time. To find a representative of a set, one should move from the node along the pointers until the next node pointer points to itself.

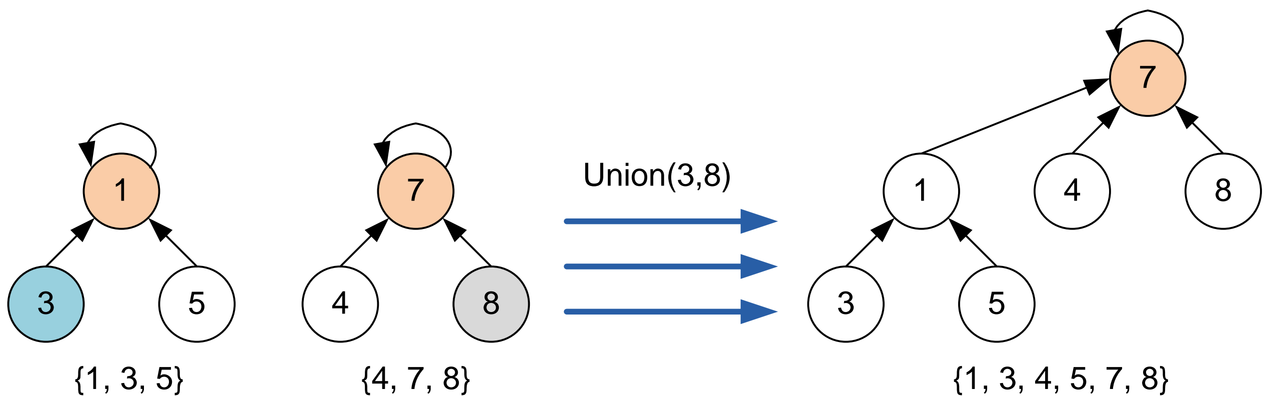
```
int Repr(int v)
{
```

```
  while (v != parent[v]) v = parent[v];
  return v;
}
```

Union(x, y) unites two sets containing elements x and y. If the elements belong to the same set (they have the same representative), then we do nothing. Otherwise, we attach one tree to another, for example declaring the ancestor of vertex x_1 (representative of x) the vertex y_1 (representative of y).

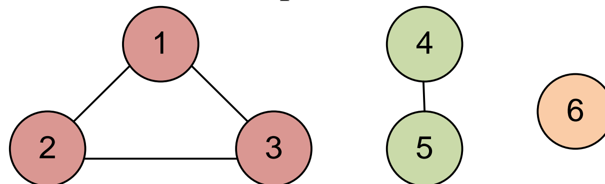
```
void Union(int x, int y)
{
  int x1 = Repr(x), y1 = Repr(y);
  if (x1 == y1) return;
  parent[x1] = y1;
}
```

Example. Lets unite the sets {1, 3, 5} and {4, 7, 8} from the above example by executing the function Union (3, 8). The representative of 3 is 1. The representative of 8 is 7. Since the representatives are different, let us declare vertex 7 as the ancestor of the vertex 1. After union we obtain the set {1, 3, 4, 5, 7, 8}.

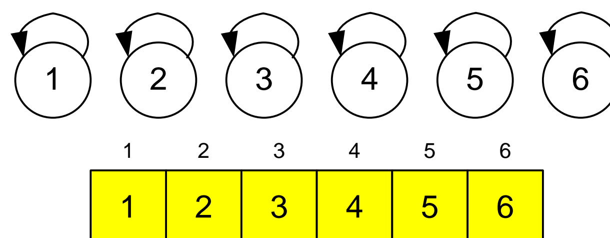


Finding the connected components in the graph

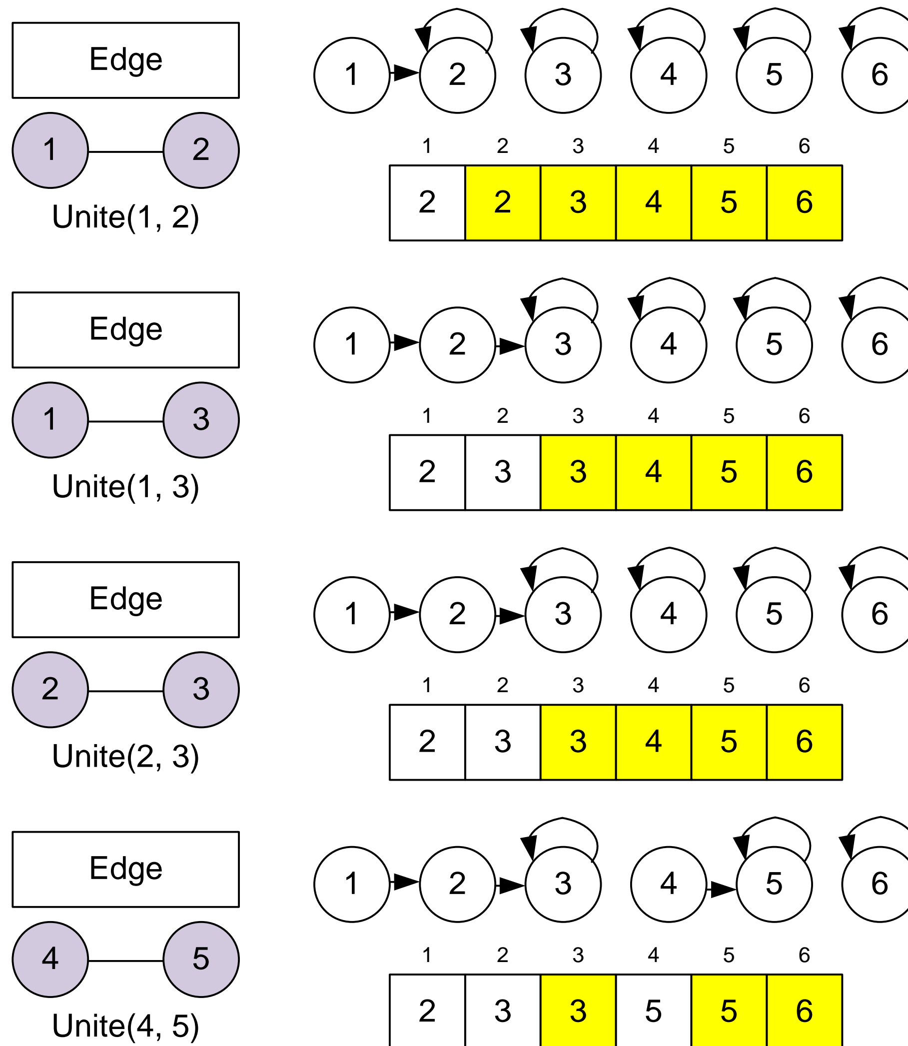
E-OLYMP **2269. Connected components** The undirected unweighted graph is given. Find the number of its connected components.



► Initially place each vertex v in its own set. Each vertex is a representative of itself.



Then, for each edge (u, v), unite the sets containing u and v. After all the edges are processed, two vertices are in the same connected component if and only if the corresponding objects are in the same set.



Number of connected components in the graph equals to the number of sets in the ***disjoint sets*** data structure. The number of sets equals to the number of representatives, namely to the number of such v for which parent[v] = v. In the example we have three representatives: 3, 5 and 6. So we have three connected components in the graph.

```
#include <stdio.h>
#define MAX 101

int mas[MAX];

int Repr(int n)
{
  while (n != mas[n]) n = mas[n];
  return n;
}

void Union(int x, int y)
{
  int x1 = Repr(x), y1 = Repr(y);
```

```
  if (x1 == y1) return;
  mas[x1] = y1;
}

int i, j, v, n, count = 0;

int main(void)
{
  scanf("%d", &n);

  // put each vertex to a separate set
  for (i = 1; i <= n; i++) mas[i] = i;

  // read adjacency matrix
  for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
  {
    scanf("%d", &v);
    // for each edge (i, j) call Union(i,j)
    if (i < j && v == 1) Union(i, j);
  }

  // count the number of representatives
  for (i = 1; i <= n; i++)
    if (mas[i] == i) count++;
  printf("%d\n", count);
  return 0;
}
```

E-OLYMP **982. Connectivity** Check whether the given undirected graph is connected.

► Use ***disjoint sets*** data structure. Graph is connected if it contains only one connected component.

E-OLYMP **776. Roads** Undirected graph is given. How many edges (minimum) we must add to the graph so that it becomes connected?

► Use ***disjoint sets*** data structure.

The above implementation of a system of disjoint sets is **ineffective**. Its possible to give an ecample when, after several unions of sets, we get a tree that is degenerated into a long chain. In this case, function ***Repr*** for finding the representative in a set runs in O(n), where n is the number of vertices in the tree.

Path compression heuristic

The path compression heuristic is designed to speed up the performance of the ***Repr*** function. After calling Repr(v), we find a representative p of the set containing v. Moving along the pointers from v to p, we can note that vertex v and all vertices traversed along the path, have their representative vertex p. Lets redirect their pointers directly to p.

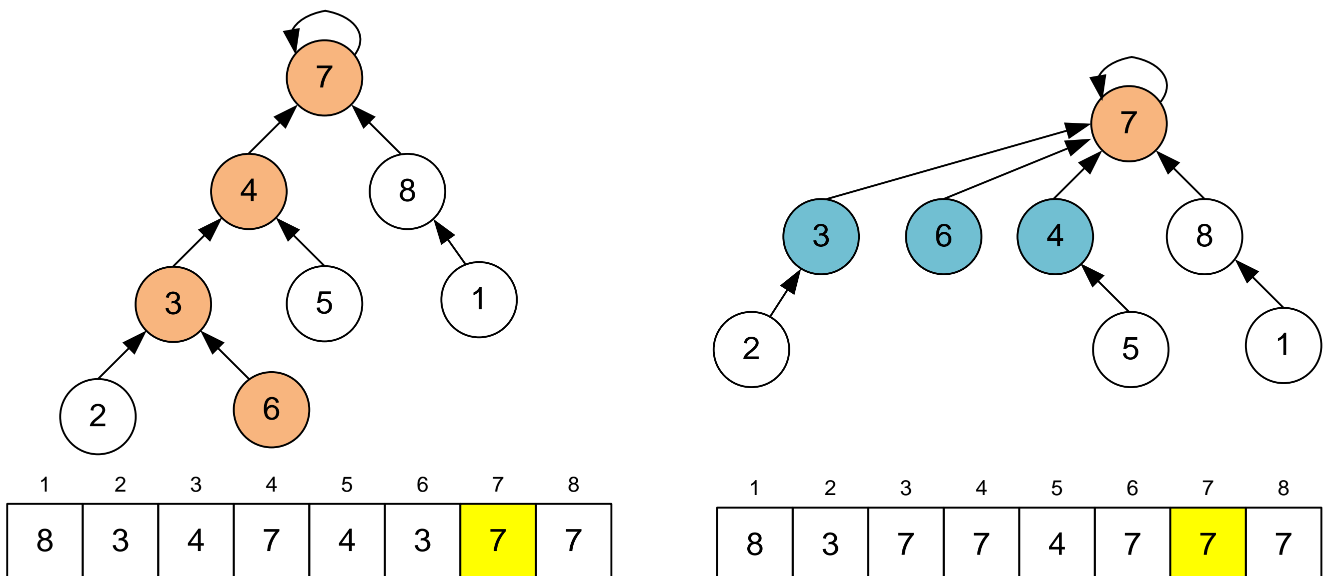
```
int Repr(int v)
{
```

```
  if (v == parent[v]) return v;
  return parent[v] = Repr(parent[v]);
}
```

As a result of recursive calls, a representative of the set is found, and then, in the process of reverse recursion, this representative is assigned to pointers for all visited elements on the path.

Now **parent** is a compressed array of ancestors, since for each vertex it stores not the immediate ancestor, but the ancestor of the ancestor, the ancestor of the ancestor of the ancestor, and so on. The parent array should be approached exactly as an ancestor array, possibly partially compressed.

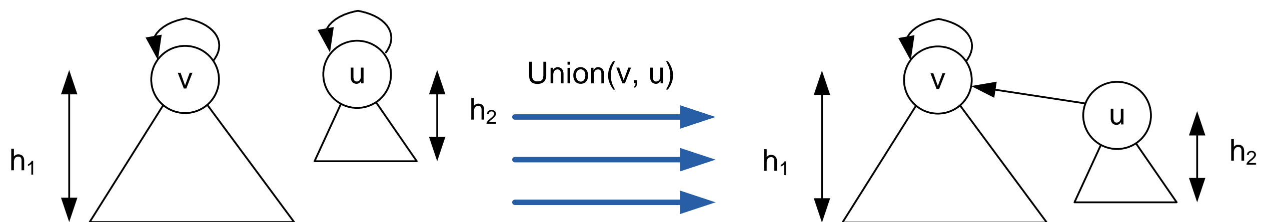
Example. A set is given on the left picture. On the right picture shows the same set after calling the ***Repr***(6) function.



Rank heuristic

The **rank heuristic** can speed up the running time of the algorithm. It changes a bit ***Union*** function: the decision which tree to join to another one is based on **ranks**.

The **rank** of a tree can be either *the number of vertices* in it, or its *depth*. In both cases, the heuristic is to join the lower ranked tree to the higher ranked tree when performing the ***union*** function.



Consider the implementation of a **rank heuristic** based on **tree sizes**. Let size[*v*] contains the size of the subtree rooted in *v*.

```
void make_set (int v)
{
  parent[v] = v;
  size[v] = 1;
}

void Union(int x, int y)
{
  x = Repr(x), y = Repr(y);
  if (x == y) return;
  if (size[x] < size[y]) swap(x,y);
```

The tree rooted at *y* is joined to the tree rooted at *x*. The size of the *x*-rooted tree is increased by the size of the *y*-rooted tree.

```
  parent[y] = x;
  size[x] += size[y];
}
```

Consider the implementation of a **rank heuristic** based on ***tree depth***. Value of depth[*v*] contains the depth of the subtree rooted at *v*. The depth of a tree with one vertex is considered to be zero.

```
void make_set (int v)
{
  parent[v] = v;
  depth[v] = 0;
}

void Union(int x, int y)
{
  x = Repr(x), y = Repr(y);
  if (x == y) return;
  if (depth[x] < depth[y]) swap(x,y);
```

The tree rooted at *y* is joined to the tree rooted at *x*.

```
  parent[y] = x;
  if (depth[x] == depth[y]) depth[x]++;
}
```

Both variants of the rank heuristic are equivalent in terms of asymptotics, therefore, in practice, one can apply either of them.

Theorem. When using the *path compression* and *rank* **heuristics** together, the running time per query is $O(a(n))$ on average, where $a(n)$ is the inverse Ackerman function, which grows very slowly. So slow that for all reasonable constraints of n it does not exceed 4 (approximately for $n \leq 10^{600}$).

Therefore, it is appropriate to say that the running time of a system of disjoint sets is practically constant.

E-OLYMP 37. Mail of the sponsor Undirected graph is given. How many edges (minimum) we must add to the graph so that it becomes connected?

► Since $n \le 10^5$, to solve the problem we must use a system of ***disjoint sets*** with **heuristics**. If you implement disjoint set data structure without heuristics, you'll get **Time Limit**.

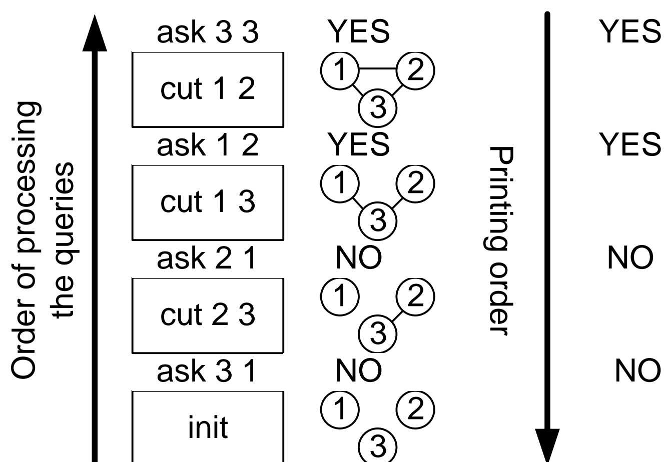
E-OLYMP 3728. Cut a graph The undirected graph is given. Two types of operations are performed on graph in the given order:

- **cut** – cut a graph, or delete an edge from a graph;
- **ask** – check, whether two given vertices lie in the same connected component.

It is known that after performing all types of cut operations there is no edges left in the graph. Give the result for each ask operation.

► Save all requests and start processing them in the reverse order. Start with a graph that does not contain edges (after all ***cut*** operations are performed, the set of edges is empty). To process the request "*cut u v*", we will add an edge between the vertices u and v. When the request "*ask u v*" arrives, we will check whether the vertices u and v lie in the same connected component. Store the answers to ***ask*** requests in the reverse order. Then print them in the required order.

Consider the order of processing the requests for the sample given and the order of printing the responses to the queries.



E-OLYMP 325. Dangerous route In one country there are n cities, some of which are connected by two-way routes. Cities are numbered by integers from 1 to n. In times of financial crisis the level of the crime in a state rose and the crime groups are organized. The most dangerous of these was "Timur and his gang" led by a notorious criminal circles by Timur, that commit robberies on most of the roads. As a result, some roads became dangerous to drive.

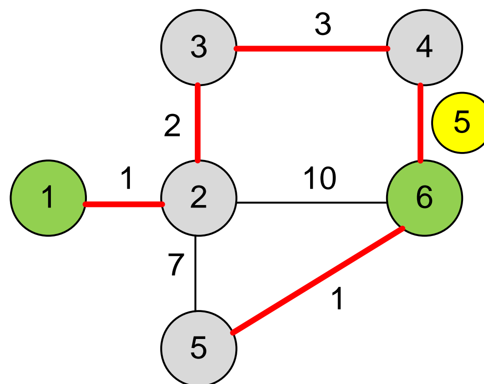
Baha must get out of city 1 and go to city n. Since he appreciate his life too much (and his wallet also), he decided to cheat Timur and go to destination by the least dangerous route, even if it is not the shortest. Baha defined the level of the danger for

each road as an integer from 0 (safe) to 10^6 (very dangerous). The danger of the route is the maximum value among the dangers of the roads that make up this route.

Help him to choose the safest route (one which has the minimum possible risk).

► Sort the roads (edges of the graph) in the order of increasing their danger. To solve the problem, we will use a system of **disjoint sets**. Initially, we create n sets, each contains one vertex. Iterate through the edges in ascending order of their danger. For each edge (a, b), unite the sets containing the vertices a and b. As soon as vertices 1 and n fall into one set, the algorithm ends. In this case, the path from the first vertex to the n-th one will already exist, and the danger of the roads through which this path passes will be no greater than the danger of the last considered road. Thus, if after considering the edge (x, y) the representatives of vertices 1 and n coincide, then the danger of the safest route will be equal to the danger of the road (x, y).

Consider the second test case.



The edges are processed in ascending order of danger. As soon as an edge with danger 5 is processed, vertices 1 and 6 become connected and the algorithm stops.