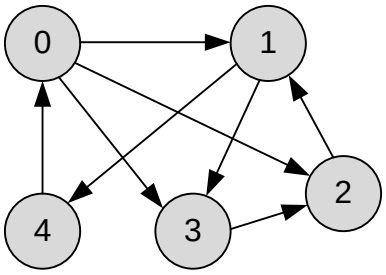
# Strongly connected components

**Definition.** A directed graph is called *strongly connected* if for every pair of vertices $u$ and $v$ vertices $u$ and $v$ are reachable from each other.

An arbitrary directed graph can be divided into strongly connected components, that are defined as the equivalence classes "$u$ is reachable from $v$ and $v$ is reachable from $u$".




**Theorem.** A directed graph is *strongly connected* (consists of only one strongly connected component) if and only if for any vertex $v$ the following two conditions are satisfied:

- all other vertices of the graph are reachable from the vertex $v$;
- vertex $v$ is reachable from any vertex of the graph;

Moreover, if for any one vertex $v$ these two conditions are satisfied, graph is strongly connected.

**Definition.** A *strongly connected component* of directed graph G(V, E) is such a maximum set of vertices $C \subseteq V$, that for every pair of vertices $u$ and $v$ vertices $u$ and $v$ are reachable from each other.

**Definition.** The transposed graph G(V, E) is a graph $G^T(V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$. That is, $E^T$ consists of the edges of G with their directions revrsed.

Graphs G and $G^T$ have the same strongly connected components, since $u$ and $v$ are reachable from each other in G if and only if they are reachable from each other in $G^T$.

The following ***Kosaraju*** algorithm finds strongly connected components in directed graph G (V, E) in linear time O (V + E).

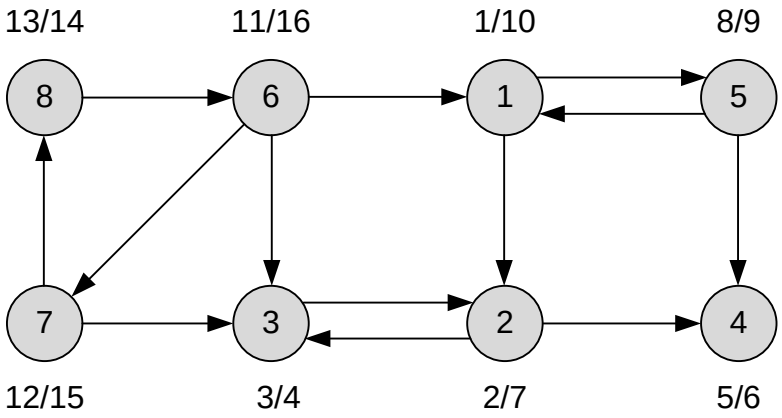
**Strongly Connected Components(G)**

1. Call DFS(G) to compute finishing times f[$v$] for each vertex $v \in V$.
2. Compute $G^T$.
3. Call DFS($G^T$), but in the main loop of DFS consider the vertices in order of decreasing f[$v$].
4. Output the vertices of each tree in the depth first forest formed in line 3 as a separate strongly connected component.

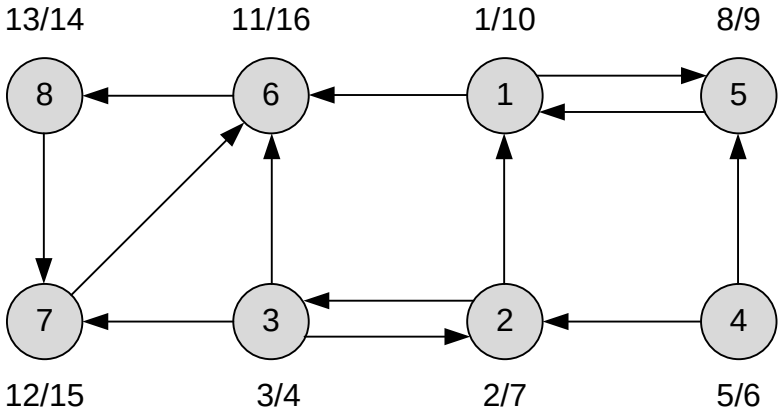
Based on this algorithm, a ***component graph*** $G^{SCC}(V^{SCC}, E^{SCC})$ is determined as follows. Suppose that G has strongly connected components $C_1, C_2, \ldots, C_k$. The vertex set $V^{SCC} = \{v_1, v_2, \ldots, v_k\}$ consists of vertices $v_i$ for each strongly connected component $C_i$ of G.

**Example.** Consider the next graph.

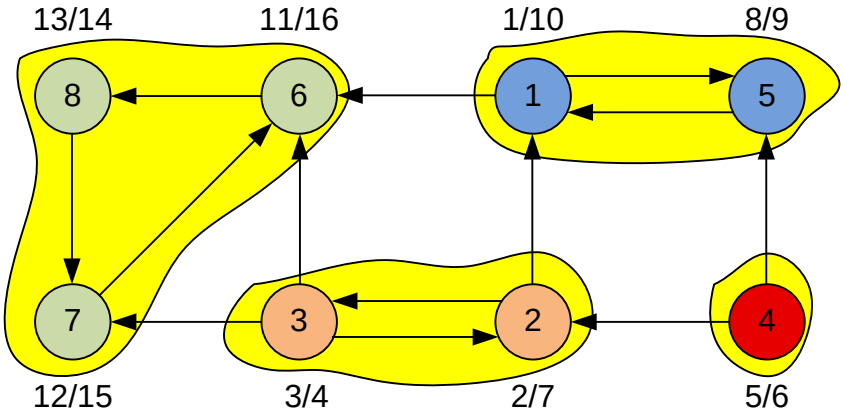
1. Call DFS(G), near each vertex *v* place the labels d[*v*] / f[*v*]:



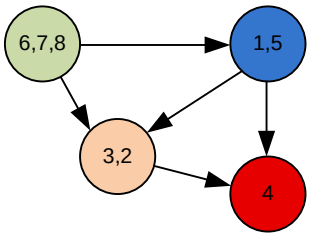

2. Compute $G^T$:




3. Call DFS($G^T$), considering the vertices in decreasing order of f[*v*].

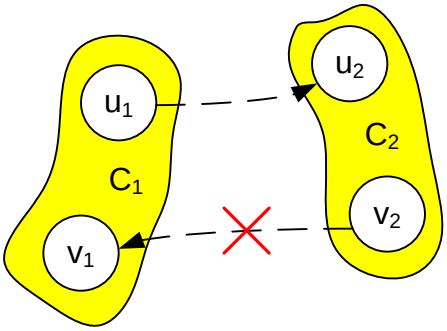



We get a graph of components (**condensation graph**):




4. The strongly connected components are: {6, 7, 8}, {3, 2}, {1, 5} and {4}.

**Lemma.** Let $C_1$ and $C_2$ be distinct strongly connected components in directed graph G(V, E). Let $u_1$, $v_1 \in C_1$, $u_2$, $v_2 \in C_2$, and suppose that there is a path $u_1 \Rightarrow u_2$ in G. Then there cannot also be a path $v_2 \Rightarrow v_1$.

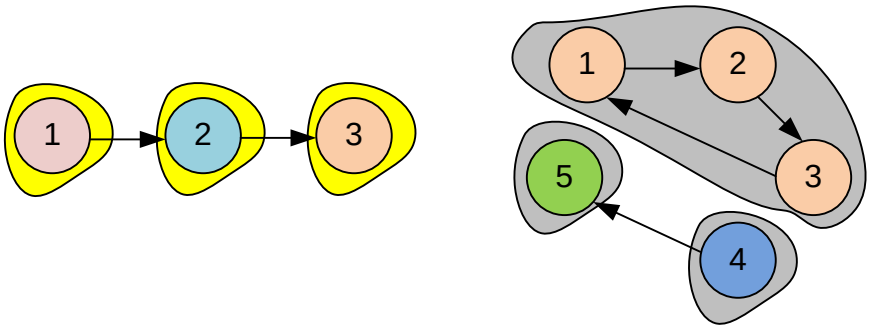



**Proof.** If we assume the opposite, then the vertices $u_1$ and $u_2$ will be reachable from one another ($u_2 \Rightarrow v_2 \Rightarrow v_1 \Rightarrow u_1$). That is, these vertices must belong to one strongly connected component.

**E-OLYMP** **2403. Strong connectivity** A strongly connected component in a directed graph is an arbitrary set of vertices such that from any vertex of this set there is a path to any other vertex of this set, and there is no other set with a similar property containing this set.

Given a directed graph. Find the number of strongly connected components in it.

► Find the number of strongly connected components in a directed graph. Graphs, given in samples, have the form:




Each of these graphs has 3 strongly connected components.

The input graph is stored in the adjacency list *g*. The inverse graph is stored in the adjacency list *gr*.

```
vector<vector<int> > g, gr;
vector<int> used, top;
```

Funtion ***dfs1*** implements the depth first search on the input graph. Put into the array *top* the sequence of vertices in the order in which the depth first search finishes their processing.

```
void dfs1(int v)
{
  used[v] = 1;
  for(int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (!used[to]) dfs1(to);
  }
  top.push_back(v);
}
```

Funtion ***dfs2*** implements the depth first search on the reversed graph. All the vertices that will be traversed as a result of a recursive call of ***dfs2*** function, belong to one strong connected component.

```
void dfs2(int v)
{
  used[v] = 1;
  for(int i = 0; i < gr[v].size(); i++)
  {
    int to = gr[v][i];
    if (!used[to]) dfs2(to);
  }
}
```

The main part of the program. Read the input data. Construct the reversed graph.

```
scanf("%d %d",&n,&m);
g.assign(n+1,vector<int>());
gr.assign(n+1,vector<int>());
for(i = 1; i <= m; i++)
{
  scanf("%d %d",&a,&b);
  g[a].push_back(b);
  gr[b].push_back(a);
}
```

Run the depth first search on the input graph. The sequence in which the processing of graph vertices is completed is stored in the *top* array.

```
used.assign(n+1,0);
for(i = 1; i <= n; i++)
  if (!used[i]) dfs1(i);
```

Run the depth first search on the reversed graph. Iterate the vertices of the reversed graph in the order of traversing the array *top* from right to left (from the end to the beginning). In the variable *c* count the number of strongly connected components.

```
used.assign(n+1,0);
c = 0;
for(i = n - 1; i >= 0; i--)
```

```
{
  v = top[i];
  if (!used[v])
  {
    dfs2(v);
    c++;
  }
}
```

Print the number of strongly connected components in the graph.

```
printf("%d\n",c);
```

**E-OLYMP** **1947. Condensation of the graph** Find the number of edges in the condensation of a given directed graph.

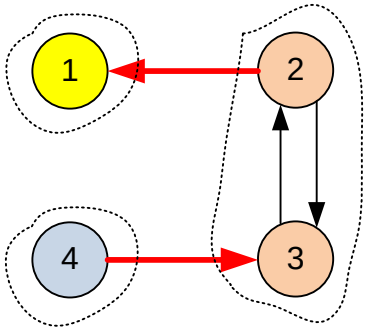
The condensation of a directed graph G is a directed graph G*, whose vertices are strongly connected components of G, and the edge in G* is present only if there exists at least one edge between the vertices of corresponding connected components.

The graph condensation does not contain multiple edges.

► Find the strongly connected components in the graph. Color all vertices of each strong connected component with one unique color. Let color[$i$] be the color of the $i$-th vertex. The number of used colors equals to the number of strong connected components.

Iterate over all the edges of the original graph. If the edge connects vertices of different colors, then it belongs to the graph condensation. Put all the edges ($a$, $b$) for which color[$a$] ≠ color[$b$] into the set $s$. Since we are using a set, not a multiset, multiple edges will not be taken into account. The number of elements in the set $s$ will be equal to the number of edges in the graph condensation.

The graph shown in the example has the form:




Graph condensation contains three vertices and two edges.

Store the input graph in the adjacency list $g$. Store the inverse graph (the graph with reversed edges) in the adjacency list $gr$. The edges of the condensed graph will be stored in the set of pairs $s$.

```
vector<vector<int> > g, gr;
vector<int> used, top, color;
set<pair<int,int> > s;
```

Funtion ***dfs1*** implements the depth first search on the input graph. Put into the array *top* the sequence of vertices in the order in which the depth first search finishes their processing.

```
void dfs1(int v)
{
  used[v] = 1;
  for(int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (!used[to]) dfs1(to);
  }
  top.push_back(v);
}
```

Funtion ***dfs2*** implements the depth first search on the reversed graph. All the vertices that will be traversed as a result of a recursive call of the ***dfs2*** function, belong to one strong connected component. Color all visited vertices with color *c*.

```
void dfs2(int v, int c)
{
  color[v] = c;
  for(int i = 0; i < gr[v].size(); i++)
  {
    int to = gr[v][i];
    if (color[to] == -1) dfs2(to,c);
  }
}
```

The main part of the program. Read the input data. Construct the reversed graph.

```
scanf("%d %d",&n,&m);
g.resize(n+1);
gr.resize(n+1);
for(i = 0; i < m; i++)
{
  scanf("%d %d",&a,&b);
  g[a].push_back(b);
  gr[b].push_back(a);
}
```

Run the depth first search on the input graph. The sequence in which the processing of graph vertices is completed is stored in the *top* array.

```
used.resize(n+1);
for(i = 1; i <= n; i++)
  if (!used[i]) dfs1(i);
```

Run the depth first search on the reversed graph. Iterate the vertices of the reversed graph in the order of traversing the array *top* from right to left (from the end to the beginning). The vertices included in the same strong connected component are colored with the same color. The current paint color is in the variable *c*.

```
color.assign(n+1,-1);
c = 0;
```

```
for(i = 1; i <= n; i++)
{
  v = top[n-i];
  if (color[v] == -1) dfs2(v,c++);
}
```

Variable *c* contains the number of connected components.

```
for(i = 1; i < g.size(); i++)
for(j = 0; j < g[i].size(); j++)
{
  to = g[i][j];
```

Iterate over all the edges of the graph (*i*, *to*). Check if the vertices *i* and *to* lie in different strongly connected components. If this is true, they are painted with different colors. Then the edge (*i*, *to*) belongs to the condensation of the graph, so insert into set *s* the pair (color[*i*], color[*to*]). Due to the fact that we use a set, not a multiset, multiple pairs will not be taken into account.

```
  if (color[i] != color[to])
    s.insert(make_pair(color[i],color[to]));
}
```

Print the number of edges in the graph condensation.

```
printf("%d\n",s.size());
```

**E-OLYMP 1104. Dominoes** Dominos are lots of fun. Children like to stand the tiles on their side in long lines. When one domino falls, it knocks down the next one, which knocks down the one after that, all the way down the line. However, sometimes a domino fails to knock the next one down. In that case, we have to knock it down by hand to get the dominos falling again.

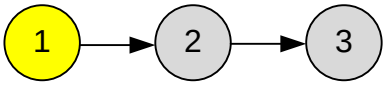
Your task is to determine, given the layout of some domino tiles, the minimum number of dominos that must be knocked down by hand in order for all of the dominos to fall.

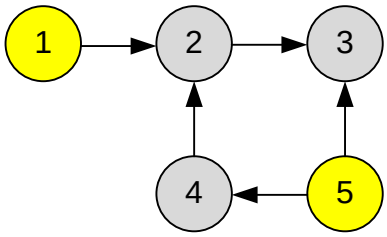
► Find the strongly connected components of the graph. Color all vertices of each strong connected component with one unique color. Let color[*i*] be the color of the *i*-th vertex. The number of used colors is the number of strongly connected components.

Obviously, if you push one domino tile with your hand, then all dominoes from the same connected component will necessarily fall. Let *cnt* be the number of connected components.

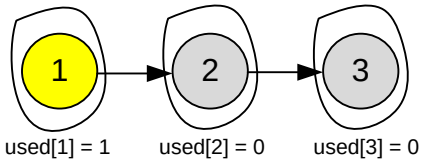
Create an array *used* of length *cnt*, which *i*-th element is 1, if it is necessary to push dominoes from the *i*-th component. Now find out which of the values used[*i*] should be set equal to 0. Iterate over all the edges of the graph. We are interested in those edges that connect different connected components. If, for example, the edge $i \to j$ is such (for it color[*i*] ≠ color[*j*]), then it is necessary to set used[color[*j*]] = 0. In this case, there is no need to knock dominoes from the component of color color[*j*]. Since by colliding dominoes from the component of color color[*i*], we'll surely knock all dominoes from the component of color color[*j*].

Graphs, given in samples, have the form:




In the first sample, it is enough to push domino number 1. In the second example, it is necessary to push dominoes numbered 1 and 5.




Graph from the first sample has 3 strongly connected components.

- because of an edge (1, 2) we do not need to knock vertex 2;
- because of an edge (2, 3) we do not need to knock vertex 3;

Store the input graph in the adjacency list *g*. The reverse graph (in which all edge directions are reversed) store in the adjacency list *gr*.

```
vector<vector<int> > g, gr;
vector<int> used, top, color;
```

The function ***dfs1*** implements depth first search on the input graph. In the array *top* store the vertices in the order in which the depth first search ends their processing.

```
void dfs1(int v)
{
  used[v] = 1;
  for(int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];
    if (!used[to]) dfs1(to);
  }
  top.push_back(v);
}
```

The function ***dfs2*** implements depth first search on the reversed graph. All vertices that will be traversed as a result of a recursive call of ***dfs2*** function belong to one strongly connected component. Color all the visited vertices with color *c*.

```
void dfs2(int v, int c)
{
  color[v] = c;
  for(int i = 0; i < gr[v].size(); i++)
  {
    int to = gr[v][i];
    if (color[to] == -1) dfs2(to,c);
  }
}
```

The main part of the program. Read the input data. Build the reversed graph.

```
scanf("%d %d",&n,&m);
g.resize(n+1);
gr.resize(n+1);
cnt = 0;
for(i = 0; i < m; i++)
{
  scanf("%d %d",&a,&b);
  g[a].push_back(b);
  gr[b].push_back(a);
}
```

Start the depth first search on the input graph. The sequence in which the graph vertices processing is completed is stored in the array *top*.

```
used.resize(n+1);
for(i = 1; i <= n; i++)
  if (!used[i]) dfs1(i);
```

Start the depth first search on the reversed graph. Iterate the vertices of the reversed graph in the order of traversing the array *top* from right to left (from end to start). The vertices included in one strongly connected component are colored with the same color. The current color is in the variable *c*.

```
color.assign(n+1,-1);
c = 0;
for(i = 1; i <= n; i++)
{
  v = top[n-i];
  if (color[v] == -1) dfs2(v,c++);
}
```

The variable *c* contains the number of strongly connected components.

```
used.assign(c,1);
for(i = 1; i < g.size(); i++)
for(j = 0; j < g[i].size(); j++)
{
  to = g[i][j];
```

Iterate over the edges of graph (*i*, *to*). Check if the vertices *i* and *to* lie in different strongly connected components. This is the case if they are colored in different colors. In this case, if we knock any domino from the strongly connected component, where domino (vertex) *i* is located, then the domino *to* will surely fall. This means that there is no sence to knock dominoes of color color[*to*], so set used[color[*to*]] = 0.

```
  if (color[i] != color[to]) used[color[to]] = 0;
}
```

In the variable *c* count the number of dominoes to be pushed. Equality used[*i*] = 1 means that no domino of color *i* will fall, no matter which domino of a different color is knocked. In this case, we'll definitely have to knock at least one domino of color *i*.

```
c = 0;
```

```
for(i = 0; i < used.size(); i++)
  if (used[i]) c++;
```

Print the answer.

```
printf("%d\n",c);
```

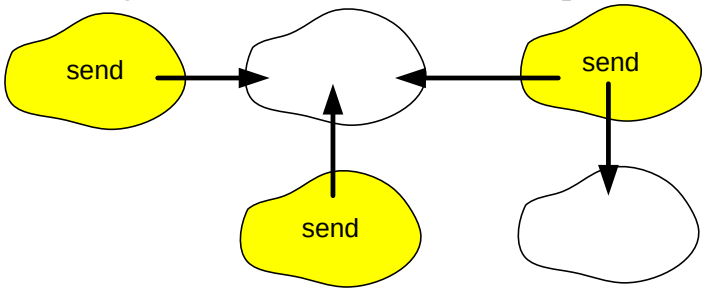
**E-OLYMP** **8553. Computer network** A computer network comprises $n$ computers, numbered from 0 to $n - 1$. Each one, after receiving a message, passes it to some other computers. If a message from computer $x$ can reach a computer $y$, this does not necessarily mean that a message from computer $y$ reaches the computer $x$. The system administrators want to determine the minimum number of computers from which a message has to be sent in order to reach all the computers in the network.

For better transmission of messages, they believe that the network needs to be extended by adding new connections between some computers, so when sending a message from an arbitrary computer it will be distributed to all others. For this purpose, it is necessary to determine the minimum number of new connections to be added, so that each of the computers can be used as initial for distribution of messages.

Write a program cnet that finds the minimal number of computers from which a message needs to be sent in order to be distributed to all computers in the network and finds the minimum number of new connections that need to be added, in order to allow a message, sent from each of the computers, to reach every other computer in the network.

► Find the strongly connected components in the graph. If there is one component (the graph is strongly connected), then a message for the entire network can be sent from any one computer, and the number of additional connections required is 0.

Let the graph be not strongly connected. Consider its condensation. For each connected component, find out whether there are outgoing and incoming edges. If some component does not have incoming edges, then in order to send a message to entire network, the message must be sent from a vertex belonging to this component. For the condensation below, the message should be sent from 3 computers.

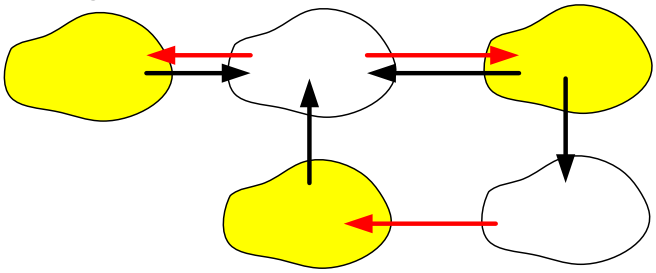



The message can be passed from any computer to any other in the network only if the graph is strongly connected. The minimum number of edges should be added to the original graph in such way as to make it strongly connected. For each connected component there must be both an incoming and outgoing edge. Let:

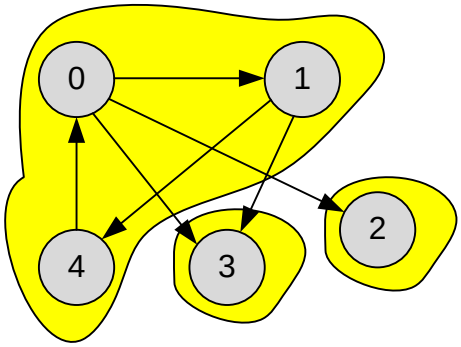
$c_1$ is the number of components without incoming edges (on the picture $c_1 = 3$);

$c_2$ is the number of components without outgoing edges (on the picture $c_2 = 2$);

Then its always possible to add max($c_1$, $c_2$) edges to make the graph strongly connected. For our example max($c_1$, $c_2$) = max(3, 2) = 3. To get a strongly connected graph, it is enough to add 3 edges.



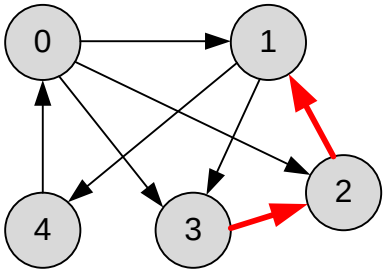
In the first test case the graph contains three strongly connected components.




There is one component that has no incoming edges: {0, 1, 4}, $c_1$ = 1;
There are two components that has no outgoing edges: {2}, {3}, $c_2$ = 1;

The message that should be distributed throughout the network, must be sent from components that have no incoming edges. We have one such component. The mailing can be done from one vertex (0, 1 or 4).

In order for the whole graph to become strongly connected, it is necessary to create edges outgoing from vertices 2 and 3, and at least one of the edges must be incoming into the component consisting of vertices {0, 1, 4}. For example, creating the following edges will make the graph strongly connected:




**E-OLYMP** **1936. Flights** The Chief Engineer Peter was asked to develop a new model of aircraft for the company "Air Bubundiya". It was found that the hardest part is choosing the optimal size of the fuel tank.

The Chief Cartographer of "Air Bubundiya" John has made a detailed map of Bubundiya. On this map he marked the fuel consumption for the flight between each pair of cities.

Peter wants to make the size of fuel tank as small as possible, so that the aircraft can fly from any city to any other (possibly with refueling in cities on the way).

► Let the tank size be $x$. Construct a graph in which there is a directed edge $(i, j)$ if and only if the amount of fuel required for a direct flight from the $i$-th city to the $j$-th city is no more than $x$ (that is, with the available tank, you can make a direct flight).

You can fly from any city to any one only if the graph is strongly connected. Instead of calculating the number of strongly connected components, we will use the following property.

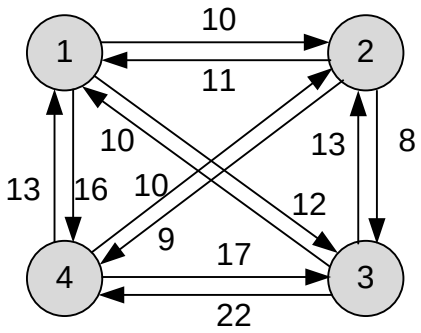
Graph is strongly connected if the following statements hold for any vertex $v$:

- from the vertex $v$ there is a path to all the other vertices;
- from any vertex there is a path to $v$;

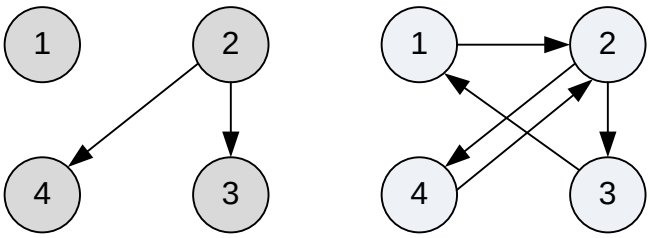
Moreover, if these statements are true for at least one vertex $v$, then they are true for all other vertices. These properties can be checked by running a depth first search on a graph, for example, from a zero vertex (for definiteness, the vertices are numbered from 0 to $n - 1$).

Thus, for a tank of size $x$, we have learned to determine whether an airplane can get from any city to any other. It remains to find the required minimum possible volume of the aircraft tank using the *binary search* method.

Graph, given in a sample, has the form:




On the left side, presented a graph of possible flights with a tank volume of 9, and on the right with a volume of 10.




With a tank of size 10, an airplane can fly from any city to any other.

Store the matrix of fuel consumption between cities in the array *graph*. The array *used* is used to mark the already visited vertices.

```
#define MAX 1010
int graph[MAX][MAX], g[MAX][MAX];
int used[MAX];
```

Start the function ***dfs*** of depth first search from the vertex *v*. If *type* = 0, then go along the edges according to their direction. When *type* = 1, depth first search is performed along the edges in the opposite direction.

```
void dfs(int v, int type)
{
  used[v] = 1;
  for(int i = 0; i < n; i++)
    if ((type ? g[i][v] : g[v][i]) && !used[i]) dfs(i,type);
}
```

The function ***AllVisited*** checks if all the edges of the graph are visited during the depth first search. The answer will be positive and the function will return 1 if all cells of the array *used* contain one.

```
int AllVisited(void)
{
  for(int i = 0; i < n; i++)
    if (!used[i]) return 0;
  return 1;
}
```

The main part of the program. Read the input fuel consumption matrix into the array *graph*.

```
scanf("%d",&n);
for(i = 0; i < n; i++)
for(j = 0; j < n; j++)
  scanf("%d",&graph[i][j]);
```

Find the minimum tank size using binary search on a segment [*L*; *R*]. Initially set [*L*; *R*] = [0; 2000000000].

```
L = 0; R = 2000000000;
while(L < R)
{
  Mid = (L + R) / 2;
```

At each iteration of the binary search, construct the adjacency matrix *g* of the directed graph: g[*i*][ *j*] = 1, if a direct flight with tank volume *Mid* can be made from city *i* to city *j*.

```
  for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
    g[i][j] = (graph[i][j] <= Mid);
```

Start depth first search from the zero vertex. If all the other vertices can be reached from zero vertex, and zero vertex can be reached from all the other vertices, then graph is strongly connected. In this case, the value of the variable *flag* remains 0.

```
  memset(used,0,sizeof(used));
  dfs(0,0); flag = 0;
  if (AllVisited())
```

```
    {
      memset(used,0,sizeof(used));
      dfs(0,1);
      if (!AllVisited()) flag = 1;
    } else flag = 1;
```

Recompute the boundaries of the binary search interval depending on the value of *flag*.

```
    if (!flag) R = Mid; else L = Mid + 1;
  }
```

Print the required minimum tank size *L*.

```
  printf("%d\n",L);
```

**E-OLYMP 1910. Empire** The Empire consists of $n$ planets. Lets label these planets with numbers from 1 to $n$. The planet with the number 1 is a capital of Empire, where the Emperor residence is located and the troops are prepared. On different planets of the empire the uprisings are often, which must be suppressed by force and immediately.

In order for troops to move quickly, the one-way teleports are installed on some planets. There are m teleports in total. Using the $i$-th teleport you can get instantly from planet $a_i$ to planet $b_i$ (but not vice versa). Thus, it is possible to crush the rebellion in time on some planet $x$, if there is a sequence of planets $p_1$, ..., $p_k$ ($k \geq 2$) such that $p_1 = 1$, $p_k = x$, and for each $1 \leq i < k$ there is a teleport from planet with number $p_i$ to the planet with number $p_{i+1}$. After crushing the uprising, the troops remain on the planet to maintain the order, so we do not need to worry about their return to the capital.

Check is there an opportunity using the existing system of teleports to crush the rebellion on any planet of the Empire. If so, print 0. Otherwise find the minimum number of teleports that must be built more so that the rebellion on any planet can be suppressed instantly. Each new teleport can be constructed between any two planets.

► In the graph representing the empire, add the least number of edges so that any vertex is reachable from vertex 1 (the capital).

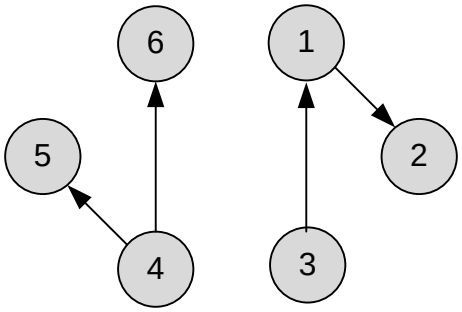
Note that new edges can only be constructed from the first vertex. Indeed, if the new arc is ($a$, $b$), then replacing it with (1, $b$), we will not change the reachability of the vertices from the capital.

Consider a graph condensation and choose a vertex that does not have incoming edges (the condensation graph is acyclic, such vertices always exist). Some strongly connected component corresponds to this vertex. From vertex 1 it is necessary to construct an edge to one of the vertices of this strongly connected component. There is one exception here – if vertex 1 itself is included in this strongly connected component, then no edge need to be constructed.

If in the condensation graph there are edges incoming to the vertex $v$, then there is no need to construct new edges in the original graph from vertex 1 to the vertices of the strongly connected component corresponding to $v$.

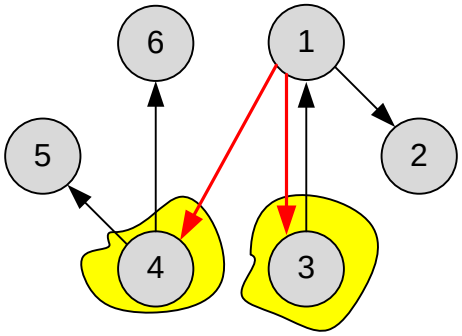
Thus, in this problem it is necessary to find the number of strongly connected components into which there is no incoming edges in condensation graph. The component that contains the capital should be processed separately.

Graph given in a sample, has the form:




To solve the problem, it is enough to build two additional teleports. You can, for example, build a teleport from planet 2 to planet 4 and from planet 5 to planet 3.

The given graph has 6 strongly connected components: each vertex forms one. Find the components without incoming edges. There will be two of them: those that contain vertices 3 and 4.




Thus, it is enough to build two new teleports from the capital (vertex 1), running respectively to vertices 3 and 4.

**E-OLYMP 1937. Fire safety** There are *n* houses in the city of Zelenograd. Some of them are connected by one way roads.

In recent times in Zelenograd the incidents of fires have increased. In this regard, the residents decided to build several fire stations in the city. But there was a problem – the fire engine traveling on the call, of course, can ignore the direction of the current road, however, the car returning from the job is obliged to follow the traffic rules (the people of Zelenograd piously respect these rules!).

It is clear that wherever the fire truck will be, it should be possible to return to the fire station where from it started its way. But the construction of stations costs a lot of money, so at the city council it was decided to build a minimum number of stations in such a way that this condition was hold. In addition for saving money, it was decided to build stations in the form of extensions to existing houses.

Your task is to write a program that calculates the optimal position of the stations.

► In this problem you need to find the minimum set of vertices reachable from all other vertices in the graph. Find the condensation of the graph. Consider a strongly connected component without any outgoing edge. Having arrived from other

components, you can put out the fire, but you will no longer be able to get home by following the road rules. Therefore, if there are no outgoing edges from some connected component, then it is necessary to build a fire station in it. There is no need to build other stations, since from any vertex it is always possible to get along the edges of the graph into a component without outgoing edges.

The input graph contains three strongly connected components. No edge comes out from components consisting of one vertex (4 and 5). Therefore, it is enough to build fire stations in them.