Articulaton points

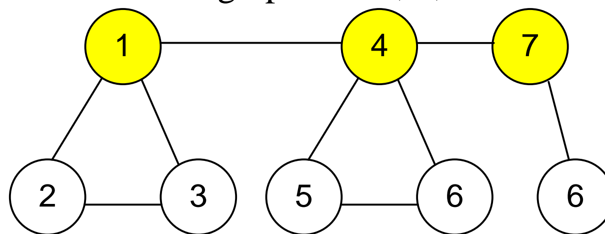
The articulation point of an undirected graph is a vertex which removal makes the graph disconnected. Removing a vertex also means removing of all the edges outgoing from it.

Naive algorithm in O(V * (V + E))

For each vertex *v* of original graph:

а) delete *v* from the graph;
б) check if the graph is connected;
в) return *v* back to graph;

The articulation points for the next graph are: 1, 4, 7.

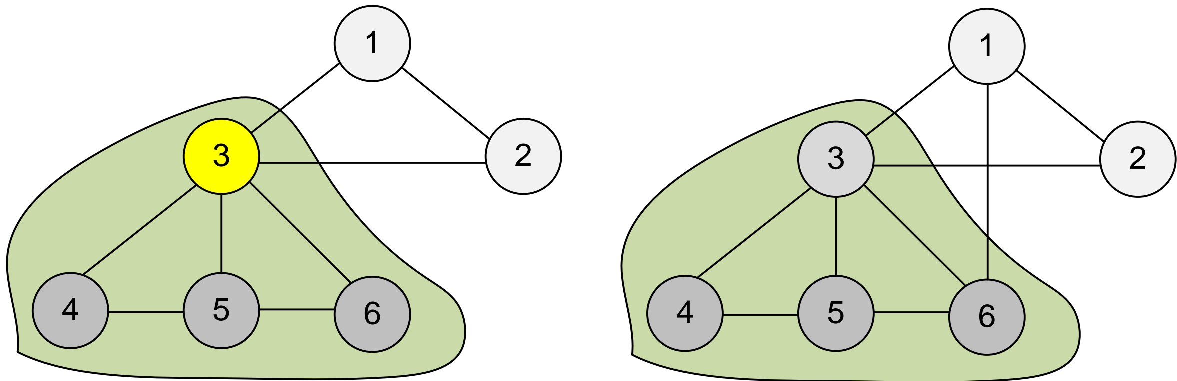


Optimal algorithm in O(V + E)

The following algorithm finds all articulation points and is based on depth first search. The running time of the algorithm is $O(n + m)$, where n is the number of vertices and m is the number of edges in the graph.

Start the depth first search traversal from some vertex, that we'll further call the *root*.

1. Suppose when traversing in depth, we iterate over all edges from the vertex $v \neq root$. If the current edge (*v*, *to*) is such that from the vertex *to* and from any of its descendants in the DFS tree there is no backward edge to any ancestor of the vertex *v*, then the vertex *v* is an articulation point. If the depth first search has looked through all the edges from the vertex *v*, and found the edges satisfying the above conditions, then the vertex *v* is not an articulation point.
2. Consider now the remaining case $v = root$. This vertex is an articulation point if and only if it has more than one son in the depth first search traversal. (if, having left *root* along an arbitrary edge, we could not traverse the entire graph, then *root* is the articulation point).



In the *left* picture vertex 3 is an articulation point. From subtree with root at 3 you can't go out (up) the subtree, because there is no back edges.

In the *right* picture vertex 3 is ***not*** an articulation point. There is a back edge from 6 to 1, from subtree with a root in 3 you can reach vertex 1 that is an ancestor of the vertex 3.

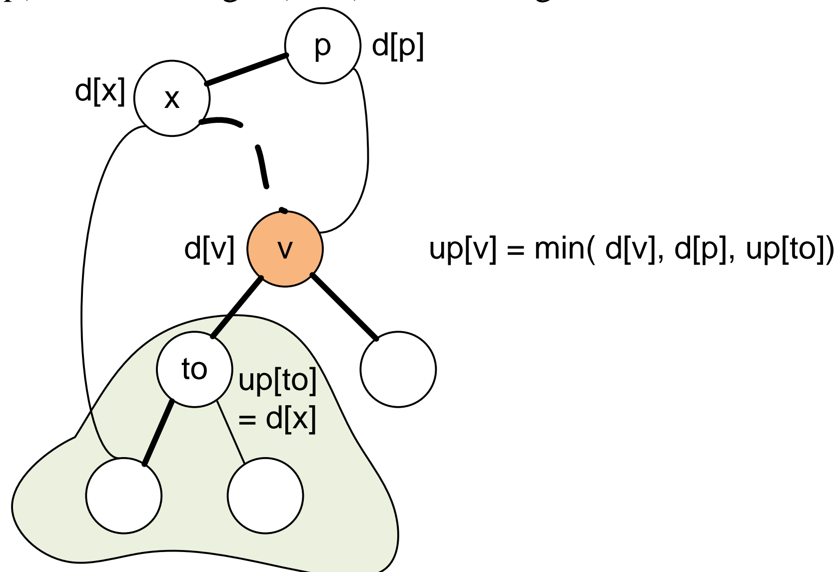
Let d[v] be the time of entering the depth first search into the vertex v. Let's introduce an array up[v], that will allow us to respond to the above requests. Define up[v] as minimum among:

- time d[v] of entrance to the vertex v;
- time d[p] of entrance to each vertex p, that is the endpoint of some back edge (v, p);
- all values of up[to] for eah vertex to, that is the immediate son of v in dfs tree (for the tree edge (v, to)).

Informally, one can assert that up[v] is equal to d[p] of the topmost vertex p of the dfs tree, that can be reached from the subtree v using back edges that are present only in this subtree.

$$\text{up}[v] = \min(\text{d}[v], \text{d}[p], \text{up}[to]),$$

where (v, p) is a back edge, (v, to) is a tree edge.



There is a back edge from the vertex v or from one of its descendants to its ancestor if and only if there is such a such son to that up[to] < d[v]. That is, if the inequality up[to] ≥ d[v] holds for some tree edge (v, to), then the vertex v is ***an articulation point***. If up[to] = d[v], then in the dfs subtree with vertex v there is a back edge that arrives exactly at v.

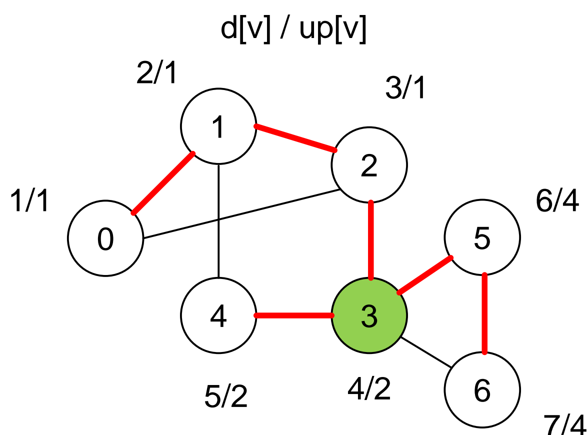
For the initial vertex *root*, the specified criterion does not apply; for it, the number of direct sons in DFS tree should be counted.

Root is an articulaton point if it has more that one son is DFS tree.

Leaf can never be an articulaton point.

If there exists a tree edge (v, to) such that up[to] ≥ d[v], then v is an articulation point.

Consider the graph shown below. Start depth first search from the vertex 0. The edges of the dfs tree are in read bold lines. Each vertex has labels d[v] / up[v]. Graph has three back edges: (2, 0), (4, 1), and (6, 3). The labels up[v] are placed in the reverse order of dfs.



up[6] = min(d[6], d[3]) = min(7, 4) = 4. One back edge (6, 3), no outgoing tree edges.

up[5] = min(d[5], up[6]) = min(6, 4) = 4. No outgoing back edges, one tree edge (5, 6).

up[4] = min(d[4], d[1]) = min(5, 2) = 2. One back edge (4, 1), no tree edges.

up[3] = min(d[3], up[4], up[5]) = min(4, 2, 4) = 2. No outgoing back edges, two tree edges (3, 4) and (3, 5).

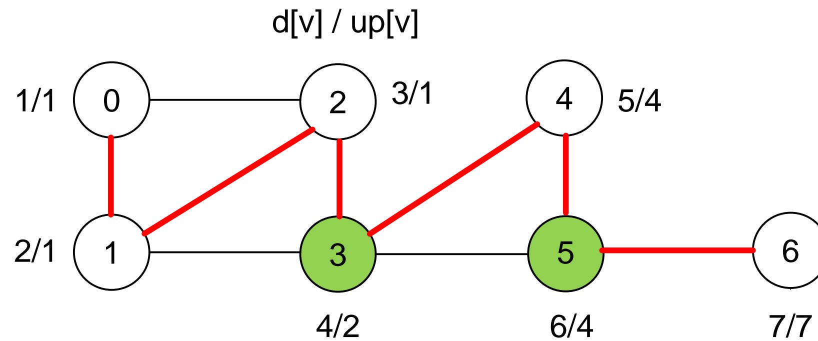
up[2] = min(d[2], d[0], up[3]) = min(3, 1, 2) = 1. One back edge (2, 0), one tree edge (2, 3).

up[1] = min(d[1], up[2]) = min(2, 1) = 1. No outgoing back edges, one tree edge (1, 2).

up[0] = min(d[0], up[1]) = min(1, 1) = 1. No outgoing back edges, one tree edge (0, 1).

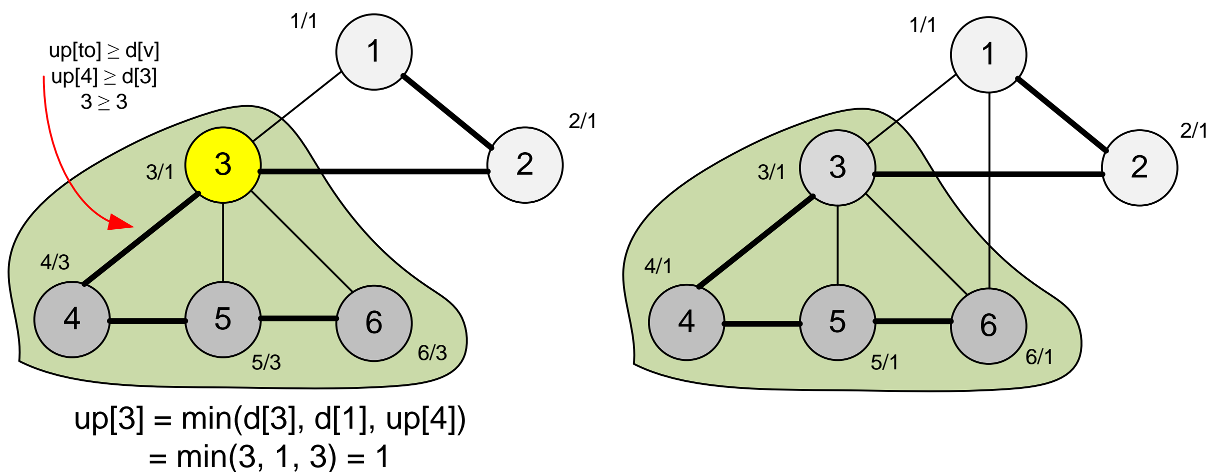
Vertex 3 will be an articulation point, since there is a tree edge (3, 5) for which 4 = up[5] ≥ d[3] = 4. Vertex 0 will not be a articulation point, since only one tree edge leaves it.

Consider the next graph:



Vertex 3 will be the articulation point, since there is an edge (*v*, *to*) = (3, 4), for which 4 = up[4] ≥ d[3] = 4.

Vertex 5 will be the articulation point, since there is an edge (*v*, *to*) = (5, 6) for which 7 = up[6] ≥ d[5] = 6.



Declare the adjacency matrix of the graph *g*, the array *used* where we'll mark the visited vertices, as well as additional arrays *d* and *up*.

```
#define MAX 100
int g[MAX][MAX], used[MAX], d[MAX], up[MAX];
```

Run the depth first search from the vertex *v*. The ancestor of *v* is *p*. If *v* is the root of the dfs tree, set *p* = -1. In the variable *children* count the number of children at the root node.

```
void dfs (int v, int p = -1)
{
  int to, children;
```

When entering the vertex v, mark it visited. Set the label d[v] equal to the current timestamp *time*. Initially set up[v] to be equal to d[v].

```
used[v] = 1;
d[v] = up[v] = time++;
children = 0;
```

Iterate over the vertices *to* that can be reached from v. For this, the element g[v][*to*] of the adjacency matrix must contain one. It is necessary to consider three cases:

1. (v, *to*) is a tree edge, that we traverse in the opposite direction (in this case $to = p$)
2. (v, *to*) is a back edge (in this case used[*to*] = 1 and $to \neq p$)
3. (v, *to*) is a tree edge (in this case used[*to*] = 0)

```
for (to = 0; to < n; to++)
{
  if(!g[v][to]) continue;
  if (to == p)  continue;
```

If vertex *to* is visited, then (v, *to*) is a back edge. Recompute the value of up[v].

```
  if (used[to])
    up[v] = min (up[v], d[to]);
  else
  {
```

Otherwise start the depth first search from the vertex *to*. (v, *to*) is a tree edge. Recompute up[v].

```
    dfs (to, v);
    up[v] = min (up[v], up[to]);
```

If up[*to*] $\geq$ d[v] and v is not a root ($p \neq -1$), then the vertex v is an articulation point.

```
    if ((up[to] >= d[v]) && (p != -1))
      printf("%d ",v);
```

Count the number of vertices *to*, into which the depth first search is run from the vertex v.

```
    children++;
  }
}
```

If v is a root ($p = -1$) and the number of its sons in dfs tree is greater than 1, then the vertex v is an articulation point.

```
if ((p == -1) && (children > 1))
 printf("%d ",v);
}
```

The main part of the program. Read the input graph. The first line contains the number of vertices *n*. It is followed by pairs of vertices that describe the edges of the graph.

```
scanf("%d",&n);
memset(g,0,sizeof(g));
while(scanf("%d %d",&a,&b) == 2)
  g[a][b] = g[b][a] = 1;
```

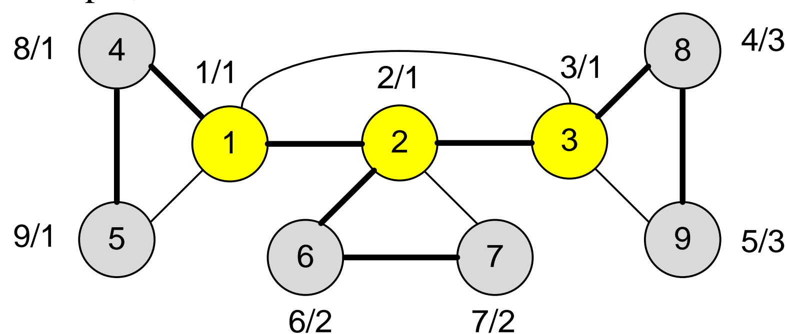
Set *time* to one and start the depth first search.

```
time = 1;
for(i = 0; i < n; i++)
  if (!used[i]) dfs(i);
```

E-OLYMP 1945. Articulation points The undirected graph is given. Find all its articulation points.

► Search for the articulation points using the depth first search. For each vertex *v*, compute the labels d[*v*] / up[*v*]. A vertex *v* is an articulation point if there exists an edge (*v*, *to*) of the depth first search tree such that the inequality up[*to*] ≥ d[*v*] holds. This inequality means that from the vertex *to*, which is a son of *v*, along the back edges from the subtree with the vertex *to*, one can go no higher than the vertex *v*.

Graph gven in a sample, has the form:



The edges of the depth first search tree are marked with bold lines. Near each vertex *v* there are labels d[*v*] / up[*v*]. The articulation points are highlighted in color.

Vertex 1 is an articulation point because for edge (1, 4) up[4] ≥ d[1] (1 ≥ 1).

Vertex 2 is an articulation point because for edge (2, 6) up[6] ≥ d[2] (2 ≥ 2).

Vertex 3 is an articulation point because for edge (3, 8) up[8] ≥ d[3] (3 ≥ 3).

Since the number of vertices in the graph is large, store the graph in an adjacency list. The array *used* is used to mark the already visited vertices. To solve the problem, we'll use two additional arrays *d* and *up*. The list of vertices, that are articulation points, will be saved in the set *ArtPoints*.

```
vector<vector<int> > graph;
vector<int> used, d, up;
set<int> ArtPoints;
```

The function ***dfs*** starts the depth first search from the vertex *v* and searches for articulation points. If *v* is the root of the ***dfs*** tree, set *p* = -1. In the variable *children* count the number of children at the root node. Found articulation points are saved in set *ArtPoints*.

```
void dfs (int v, int p = -1)
{
  int i, to, children;
```

When entering the vertex *v*, mark it visited. Set the label d[*v*] equal to the current timestamp *time*. Initially set up[*v*] to be equal to d[*v*].

```
  used[v] = 1;
  d[v] = up[v] = time++;
  children = 0;
```

Iterate over the vertices *to* that can be reached from *v*. It is necessary to consider three cases:

1. (*v*, *to*) is a tree edge, that we traverse in the opposite direction (in this case *to* = *p*)
2. (*v*, *to*) is a back edge (in this case used[*to*] = 1 and $to \neq p$)
3. (*v*, *to*) is a tree edge (in this case used[*to*] = 0)

```
  for (i = 0; i < graph[v].size(); i++)
  {
    to = graph[v][i];
    if (to == p)  continue;
```

If vertex *to* is visited, then (*v*, *to*) is a back edge. Recompute the value of up[*v*].

```
    if (used[to])
      up[v] = min (up[v], d[to]);
    else
    {
```

Otherwise start the depth first search from the vertex *to*. (*v*, *to*) is a tree edge. Recompute up[*v*].

```
      dfs (to, v);
      up[v] = min (up[v], up[to]);
```

If up[*to*] ≥ d[*v*] and *v* is not a root ($p \neq -1$), then the vertex *v* is an articulation point.

```
      if ((up[to] >= d[v]) && (p != -1)) ArtPoints.insert(v);
```

Count the number of vertices *to*, into which the depth first search is run from the vertex *v*.

```
      children++;
    }
  }
```

If v is a root (p = -1) and the number of its sons in dfs tree is greater than 1, then the vertex v is an articulation point.

```
  if ((p == -1) && (children > 1)) ArtPoints.insert(v);
}
```

The main part of the program. Read the undirected graph into adjacency list *graph*.

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

Run the depth first search. Graph can be disconnected.

```
time = 1;
for(i = 1; i <= n; i++)
  if (!used[i]) dfs(i);
```

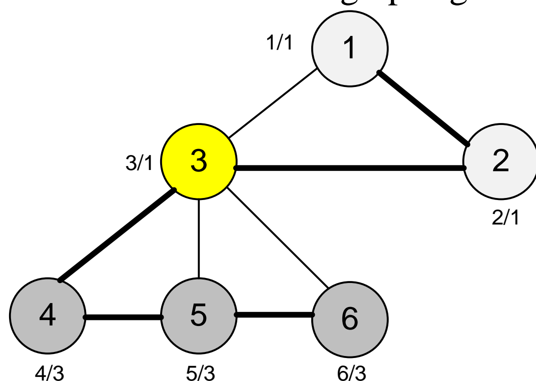
Print the number of articulation points, as well as them in ascending order.

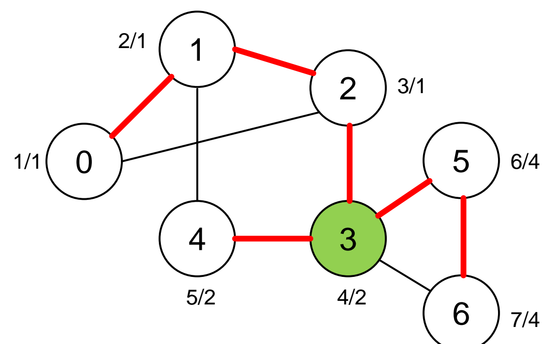
```
printf("%d\n",ArtPoints.size());
for(iter = ArtPoints.begin(); iter != ArtPoints.end(); iter++)
  printf("%d\n",*iter);
```

E-OLYMP **10224. Articulation points - Timestamps** Undirected graph is given. Run depth first search from the given vertex v. Print the timestamps d[v] and up[v] for each vertex v in the increasing order of vertices.

► Labels d[v] / up[v] are used to find articulation points. Perform a depth first search and set up the indicated labels.

Place the labels in the graphs given in the first and second test cases.





E-OLYMP **2964. Magnetic Cushions** The City of the Future is built up with skyscrapers. To move between them and transport parking some of skyscrapers triples are connected with triangular cushions made from unipolar magnets. Each cushion connects exactly 3 skyscrapers and a top view on it is a triangle with vertices in

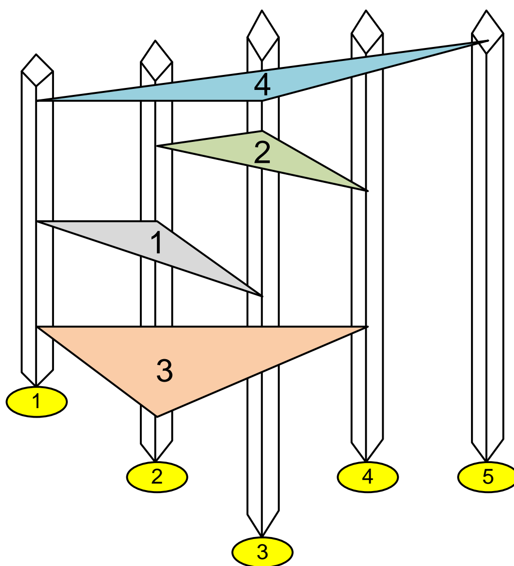
skyscrapers. This allows moving freely between the skyscrapers. The pillows can be constructed at different levels, so one skyscraper can be connected with different couples using different pillows, so two skyscrapers can be connected with multiple pillows (either with different third skyscraper or with the same). For example, there may be two cushions at different levels between skyscrapers 1, 2 and 3, and moreover, a magnetic cushion between 1, 2 and 5.

The system of magnetic pillows is organized so that you can use them to get from one skyscraper to any other in this city (from one pillow to another you can move inside a skyscraper), but maintaining each of them requires a lot of energy.

Write a program that finds which magnetic pillows can not be removed from the city structure, because removal of even just one of them leads to the fact that there exist skyscrapers from which now you can not get to some other skyscrapers, and people will become very sad.

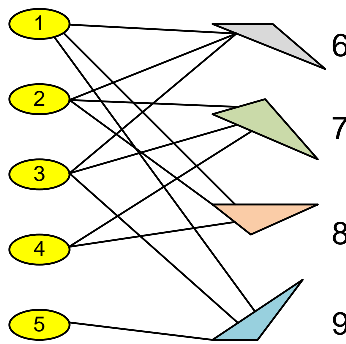
► For the magnetic cushion, we introduce an additional vertex, connecting it to each vertex that is connected by this cushion. Let's number the new vertices from $n + 1$ to $n + m$. The constructed graph will contain $n + m$ vertices. Next, find all articulation points with numbers greater than n. If vertex i ($i > n$) is an articulation point, then this means that switching off magnetic cushion number $i - n$ is impossible without reporting a violation in the city.

In the first sample there are 5 skyscrapers and 4 magnetic cushions, placed as shown below.



If pillow number 4 is removed, skyscraper number 5 will be cut off from the rest of the buildings.

Let's build a graph of 5 + 4 = 9 vertices as described in the analysis of the algorithm. The resulting graph has one articulation point 9, the number of which is greater than 5. Therefore, the pillow number 9 – 5 = 4 cannot be removed without breaking the connectivity in the city.



Bridges

The **bridge** in the graph is an edge, which removing disconnects the graph (splits into two or more connected components).

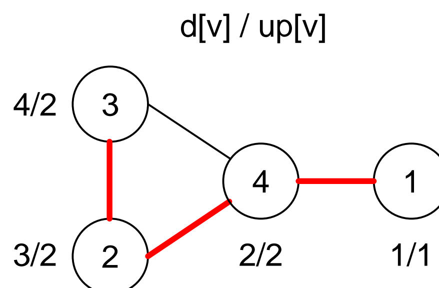
Theorem. An edge of a graph is a ***bridge*** if and only if it is not contained in any simple cycle.

Proof. If an edge belongs to a simple cycle, then its ends are reachable from each other even after removing this edge, therefore, removing such an edge cannot lead to decomposition into several connected components.

Conversely, if the graph remains connected after removing the edge (a, b), then there is a simple path from a to b that does not contain (a, b). By adding an edge (a, b) to this path, we get a simple cycle.

Consequence. Back edge in dfs can't be a bridge. This follows from the fact that any back edge is contained in some simple cycle.

Start the depth first search, place the d[v] and up[v] labels. There is a back edge from a vertex v or its descendant to its ancestor if and only if there is a son *to* such that up[*to*] < d[v]. If for some tree edge (v, *to*) the equality up[*to*] = d[v] holds, then in the dfs subtree with the vertex v there is a back edge that comes exactly at v. If up[*to*] > d[v], then the edge (v, *to*) is a bridge. Any back edge can't be a bridge.



The edge (1, 4) is a bridge since 2 = up[4] > d[1] = 1.

If there exists a tree edge (v, to) such that up[to] > d[v], then (v, to) is a bridge.

Function ***dfs*** runs depth first search from the vertex *v*.

```
void dfs (int v, int p = -1)
{
  int i, to;
  used[v] = 1;
  d[v] = up[v] = time++;
  for (i = 0; i < graph[v].size(); i++)
  {
    to = graph[v][i];
    if (to == p)  continue;
    if (used[to])
      up[v] = min (up[v], d[to]);
    else
    {
      dfs (to, v);
      up[v] = min (up[v], up[to]);
      if (up[to] > d[v]) printf("%d %d\n",v,to);
    }
  }
}
```

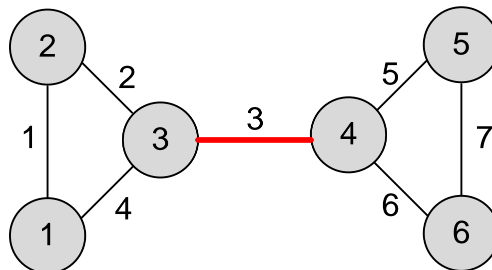
Function ***FindBridges*** finds the bridges.

```
void FindBridges(void)
{
  time = 1;
  for(int i = 1; i <= n; i++)
    if (!used[i]) dfs(i);
}
```

E-OLYMP 1943. Bridges The undirected graph is given. Find all its bridges.

► Start the depth first search, place the d[*v*] and up[*v*] labels. There is a back edge from a vertex *v* or its descendant to its ancestor if and only if there is a son *to* such that up[*to*] < d[*v*]. If for some tree edge (*v*, *to*) the equality up[*to*] = d[*v*] holds, then in the dfs subtree with the vertex *v* there is a back edge that comes exactly at *v*. If up[*to*] > d[*v*], then the edge (*v*, *to*) is a bridge. Any back edge can't be a bridge.

Graph, given in a sample, has the form:



Since the number of vertices in the graph is large, we'll store the graph as an adjacency list *graph*. The array *used* is used to mark the already visited vertices. To solve the problem, we'll use two additional arrays *d* and *up*. In the set of *Bridges*, we'll

collect the numbers of the edges that are bridges. For each input edge (a, b), remember its number in the mapping *mp*.

```
vector<vector<int> > graph;
vector<int> used, d, up;
set<int> Bridges;
map<pair<int,int>, int> mp;
```

Function ***Edge*** returns the edge, the pair of vertices (a, b), where $a < b$.

```
pair<int,int> Edge(int a, int b)
{
  if (a > b) swap(a,b);
  return make_pair(a,b);
}
```

Function ***dfs*** runs depth first search from the vertex v. Place the labels d[v] and up[v]. The vertex p is the ancestor of v in the search tree.

```
void dfs (int v, int p = -1)
{
  used[v] = 1;
  d[v] = up[v] = time++;
  for (int i = 0; i < graph[v].size(); i++)
  {
    int to = graph[v][i];
    if (to == p)  continue;
    if (used[to])
      up[v] = min (up[v], d[to]);
    else
    {
      dfs (to, v);
      up[v] = min (up[v], up[to]);
      if (up[to] > d[v]) Bridges.insert(mp[Edge(v,to)]);
    }
  }
}
```

Function ***FindBridges*** finds the bridges.

```
void FindBridges(void)
{
  time = 1;
  for(int i = 1; i <= n; i++)
    if (!used[i]) dfs(i);
}
```

The main part of the program. Read the input graph. For each edge (a, b) store its number in the mapping *mp*. We need this so that we can print the bridges not as pairs of vertices they connect, but as the numbers of the input edges.

```
scanf("%d %d",&n,&m);
graph.resize(n+1); used.resize(n+1);
d.resize(n+1); up.resize(n+1);
for(i = 1; i <= m; i++)
```

```
{
  scanf("%d %d",&a,&b);
  graph[a].push_back(b); graph[b].push_back(a);
  mp[Edge(a,b)] = i;
}
```

Find the bridges. The numbers of the edges that are bridges are stored in the set *Bridges*.

```
FindBridges();
```

Print the number of bridges. On the next line print the numbers of the edges that are bridges in ascending order.

```
printf("%d\n",Bridges.size());
for(iter = Bridges.begin(); iter != Bridges.end(); iter++)
  printf("%d ",*iter);
printf("\n");
```

Biconnected components

A graph without bridges is called biconnected. The maximum biconnected subgraph of a graph is called a *biconnected component* or *block*. Note that any two different biconnected components either do not have common vertices, or have one common vertex, which is the articulation point.

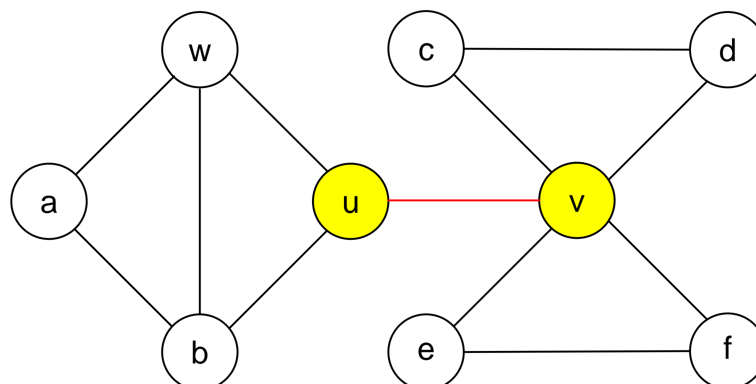
Finding the articulation points and biconnected components of a given graph is important when studying the reliability of communication and transport networks.

There are:

- ***vertex biconnectivity*** (without articulation points)
- ***edge biconnectivity*** (without bridges) biconnectivity.

Graph below contains:

- ***Articulation points***: u, v.
- ***Bridges***: (u, v).
- ***Blocks***: $\{a, b, w\}$, $\{b, u, w\}$, $\{a, b, u, w\}$, $\{c, d, v\}$.



Theorem. If the vertex is incident to the bridge and is not a hanging, then it will be the *articulation point*.

Theorem. Let each vertex of the graph v be labeled with d[v] and up[v] using depth first search. Then graph will contain exactly as many maximal edge biconnected components as there are vertices v for which d[v] = up[v].

The following program prints edge biconnected components. When traversing the graph using dfs, we'll store the traversed vertices in the stack. Then, upon completion of the computation the label up[v], starting from the top of the stack and up to v, there will be the vertices of the graph that belong to one edge biconnected component.

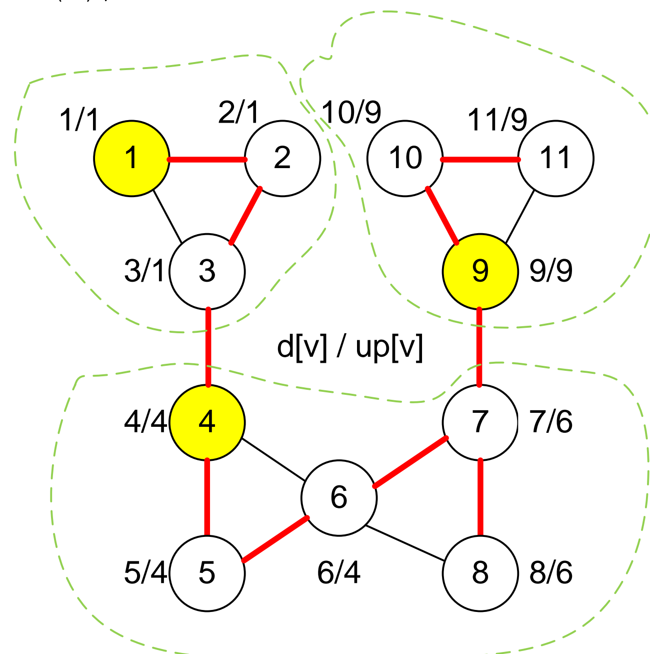
```
void dfs (int v, int p = -1)
{
  int i, to, y;
  used[v] = 1;
  d[v] = up[v] = time++;
  _Stack.push_back(v);
  for (i = 0; i < graph[v].size(); i++)
  {
    to = graph[v][i];
    if (to == p)  continue;
    if (used[to])
      up[v] = min (up[v], d[to]);
    else
    {
      dfs (to, v);
      up[v] = min (up[v], up[to]);
    }
  }
```

The equality d[v] = up[v] holds for exactly as many vertices v as there are edge biconnected components in the graph. Moreover, for each biconnected component, there is exactly one vertex with the specified property. Print the contents of the stack starting from the top of the stack and up to v.

```
  if (d[v] == up[v])
  {
    printf("Biconnected Component:");
    while(1)
    {
      y = _Stack.back(); _Stack.pop_back();
      printf(" %d",y);
      if (y == v) break;
    }
    printf("\n");
  }
}
```

Run the depth first search on the graph. Vertices in the graph are numbered from 1 to n.

```
time = 1; _Stack.clear();
for(i = 1; i <= n; i++)
  if (!used[i]) dfs(i);
```



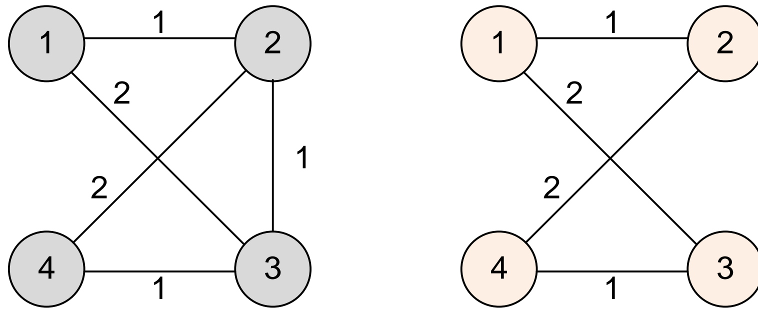
The graph contains three edge biconnected components. Exactly three vertices *v* have the equality d[*v*] = up[*v*] (for vertices 1, 4, 9).

E-OLYMP **2622. Reliable nets** T You're in charge of designing a campus network between buildings and are very worried about its reliability and its cost. So, you've decided to build some redundancy into your network while keeping it as inexpensive as possible. Specifically, you want to build the cheapest network so that if any one line is broken, all buildings can still communicate. We'll call this a minimal reliable net.

► A network is reliable if the graph that represents it is doubly edge connected. Biconnectivity is checked by depth first search – to ensure it, the absence of bridges in the graph is necessary. If the input graph is not biconnected, then the required network does not exist. In the case of biconnectivity, graph allows the presence of articulation points.

Using dynamic programming and masks, iterate over all the edges and try to remove them from the input graph. That is, iterate over all possible subgraphs. As soon as the next subgraph is no longer biconnected (it contains a bridge), stop the search. Among all biconnected subgraphs, look for the one that has the lowest cost.

Consider the first graph from the sample. The minimum reliable network is shown on the right. The graph on the right does not contain bridges, its cost is 6.



The input graph is stored in the adjacency matrix *g*. The arrays *used*, *d* and *up* are used to check if the graph is biconnected. The cell best[*mask*] stores the minimum network cost that can be formed from the edges specified by the mask *mask*.

```
#define INF 0x3F3F3F3F
#define MaxV 30
int g[MaxV][MaxV];
int used[MaxV], d[MaxV], up[MaxV];
int best[1<<20];
```

Store the list of edges of the input graph in the array of structures E.

```
struct Edge
{
  int u, v, dist;
} E[21];
```

Function ***dfs*** that runs depth first search, checks for bridges in the graph. If a bridge is present, the variable *IsBridge* is set to 1.

```
void dfs (int v, int p = -1)
{
  if (IsBridge) return;

  used[v] = 1;
  d[v] = up[v] = time++;

  for (int to = 1; to <= n; to++)
  {
    if ((to == p) || !g[v][to]) continue;
    if (used[to])
      up[v] = min (up[v], d[to]);
    else
    {
      dfs (to, v);
      up[v] = min (up[v], up[to]);
      if (up[to] > d[v]) IsBridge = 1;
    }
  }
}
```

Function ***IsBiconnected*** returns 1, if graph is biconnected. For this, there must be no bridges in the graph.

```
int IsBiconnected(void)
```

```
{
  time = 1; IsBridge = 0;
  memset(used,0,sizeof(used));
  memset(d,0,sizeof(d));
  memset(up,0,sizeof(used));

  for(int i = 1; i <= n; i++)
  {
    if (!used[i]) dfs(i);
    if (IsBridge) break;
  }
  return !IsBridge;
}
```

Compute the minimum network cost that can be formed from the edges specified by the mask *mask*. The length of all edges of subgraph specified by *mask* equals to *CurLen*.

```
int go(int mask, int CurLen)
{
  int i, opt;
```

If the value of best[*mask*] is already calculated (it is not equal to INF), then we return it. If the current subgraph is not biconnected, then stop the search process, the value of best[*mask*] is set to INF – 1, which is considered already calculated.

```
  if(best[mask] != INF) return best[mask];
  if (!IsBiconnected()) return best[mask] = INF - 1;
  best[mask] = CurLen;
```

Iterate over the edges included in the subgraph. Remove only one *i*-th edge from the graph and recursively solve the problem for the subgraph specified by the mask *mask* XOR 2^i. The length of the edges of this graph will be equal to *CurLen* – E[*i*].*dist*.

```
  for(i = 0; i < m; i++)
  {
    if (mask & (1 << i))
    {
      g[E[i].u][E[i].v] = g[E[i].v][E[i].u] = 0;
      opt = go(mask ^ (1 << i),CurLen - E[i].dist);
      if (opt < best[mask]) best[mask] = opt;
      g[E[i].u][E[i].v] = g[E[i].v][E[i].u] = 1;
    }
  }
  return best[mask];
}
```

The main part of the program. Read the edges of the graph and store them in the array E. Build the adjacency matrix *g*. The edge lengths are stored only in array E. Compute the lengths of all edges in the variable *TotLen*.

```
  while(scanf("%d %d",&n,&m), n + m)
  {
    memset(best,0x3F,sizeof(best));
```

```
    memset(g,0,sizeof(g));
    res = INF;
    TotLen = 0;

    for(i = 0; i < m; i++)
    {
      scanf("%d %d %d",&E[i].u ,&E[i].v,&E[i].dist);
      g[E[i].u][E[i].v] = g[E[i].v][E[i].u] = E[i].dist;
      TotLen += E[i].dist;
    }
```

Find the answer and print it depending on whether the required reliable network exists. For the graph containing all edges corresponds a mask $2^m - 1$, consisting of m units.

```
    res = go((1 << m) - 1, TotLen);
    if (res >= INF - 1)
      printf("There is no reliable net possible for test case %d.\n",cs++);
    else printf("The minimal cost for test case %d is %d.\n",cs++,res);
  }
```