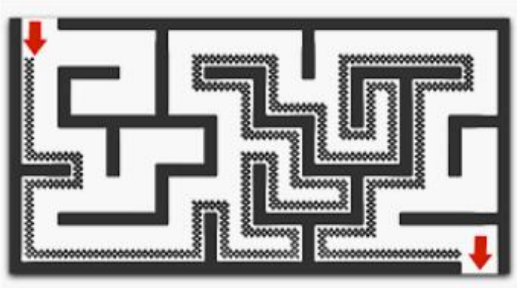
Depth-First Search

Article describes one of the classic methods of searching in the graph - dfs. The implementation of depth-first search on a disconnected (oriented) graph is presented. The technique of coloring the vertices and timestamps is described. The classification of the edges is given. The basic properties of paths and edges are formulated. The problems associated with DFS are considered.

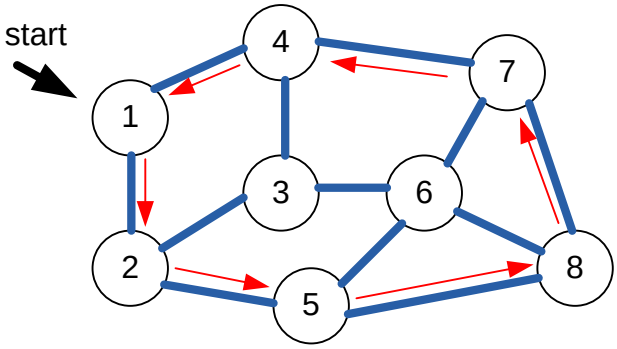
Imagine that you stay at the entrance to labyrinth. You know that somewhere there is a way out and it must be found. Labyrinth is a set of tunnels, located deep underground. You, unfortunately, do not have a flashlight or a torch, but you have a hint and a lack of fear in the dark.

Described conditions are sufficient to find a way out. It's enough to use the rule of **"right hand":** at the entrance of the labyrinth you should take the right hand on the wall and keep moving so that the right hand slides continuously along the wall. Doing so, you are sure to find an exit (unless of course there is a path to it from the point of entry; otherwise you will be taken back to the entrance).



Let's try to complicate the task. Let labyrinth be a collection of linked underground rooms and tunnels. There is only one entrance to the labyrinth (it is also the exit), and in each room there is a piece of gold. Your task is to collect all the pieces (exactly all!) and return to the same place where you entered the labyrinth.

If you do not impose additional conditions or your abilities (skills), the problem in this formulation is not solvable. In any case, a deterministic algorithm that 100% guarantees success in collecting all the golden pieces does not exist.



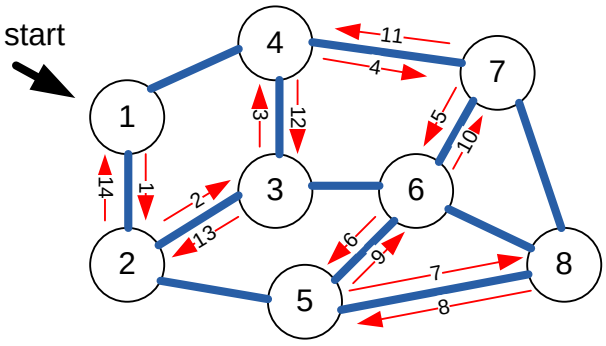
Suppose that in the latest version of the problem in each room there is a lamp that can be turned on only if you enter this room. Gold Miner (i.e. you) is already armed with a torch, as well as chalk, that can make any mark on the labyrinth walls. Once you enter a dark room, you can easily find the switch that turns the lamp on here. We

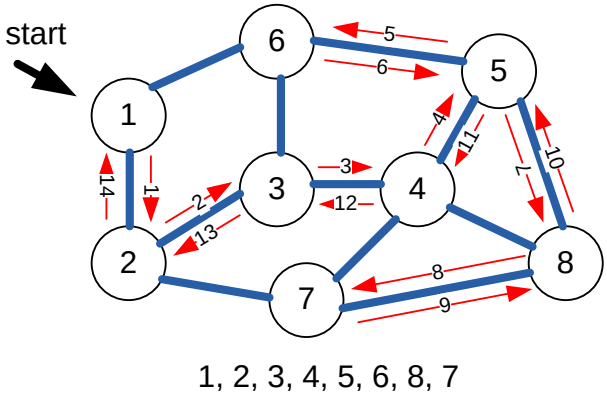
assume that any two adjacent rooms are connected directly with the short tunnel. So, being in the one side of the tunnel and looking into it, we can surely determine whether the light is burning on the other side or not (each tunnel always connects two rooms).

In this formulation, there exists a solution to the problem. It can be described by two rules:

1. If in the current room there is a tunnel leading to the room that is not visited yet (where the light is off), then we should go there. At the same time entering into a dark room, we usually turn on the light and mark with chalk tunnel through which we came here.

2. If from the current room we can ***not*** go to unvisited room (conditions in 1 are not satisfied), we return back through the tunnel along which we first came here (this tunnel is marked with chalk).





This method of traversing the labyrinth is called "**depth-first search**".

Depth-first search (DFS) is one of the methods to traverse a graph G = (V, E), the essence of which is to go "deep" while it is possible. While depth searching, the vertices of the graph are numbered, and the edges are marked. The traverse of vertices takes place according to the principle: if the current vertex has an edge leading to unvisited vertex, then go there; otherwise return back.

DFS starts with choosing the initial vertex *v* of the graph G, which is immediately marked as visited. Then for each unlabeled vertex that is adjacent to *v*, we recursively call dfs. When all the vertices that are reachable from *v* will be marked, the search ends. If in some (not initial) step of traversing the search stops, but some vertices marked as unvisited (this possible in the case of oriented or disconnected graph) then we randomly select one of unvisited vertices and start the search from it. The search process continues until all the vertices in G will be marked (visited).

Depth First Search can be run on a graph represented by:

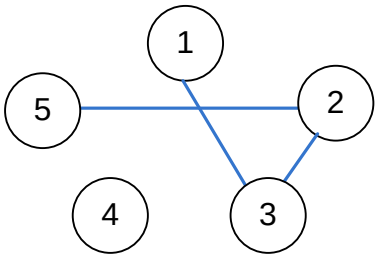
- adjacency matrix;
- adjacency list;

E-OLYMP 8760. Depth first search Undirected graph is given. Run depth first search from the given vertex v and print the numbers of vertices in the order of their first visit.

Input. First line contains number of vertices n ($n \le 100$) and edges m of undirected graph. Each of the next m lines contains two vertices a and b – an undirected edge of the graph. The last line contains vertex v.

Output. Run dfs(v) and print in one line the numbers of vertices in the order of their first visit.

Sample input	Sample output
5 3 1 3 2 3 2 5 5	5 2 3 1



► Let m be the adjacency matrix of the graph (m[i][j] = 1 if there exists an edge between vertices i and j, and m[i][j] = 0 otherwise), *used* is an array where *used*[i] = 1 if vertex i is used (visited) and *used*[i] = 0 otherwise. The numeration of the vertices in the graph starts from 1 (zero's row and column are not used).

```
#define MAX 101
int g[MAX][MAX], used[MAX];

// depth first search from the vertex v
void dfs(int v)
{
  // we entered the vertex v, print it
  printf("%d ", v);

  // mark vertex v visited
  used[v] = 1;

  // find an edge along which we can get into unvisited vertex i
  for (int i = 1; i <= n; i++)
    if ((g[v][i] == 1) && (used[i] == 0)) dfs(i);
}

int main(void)
{
  // read number of vertices and number of edges
  scanf("%d %d", &n, &m);

  // initialize arrays with 0
  memset(g, 0, sizeof(g));
  memset(used, 0, sizeof(used));

  // read m edges, graph is undirected
  for (i = 0; i < m; i++)
  {
    scanf("%d %d", &a, &b);
    g[a][b] = g[b][a] = 1;
  }
```

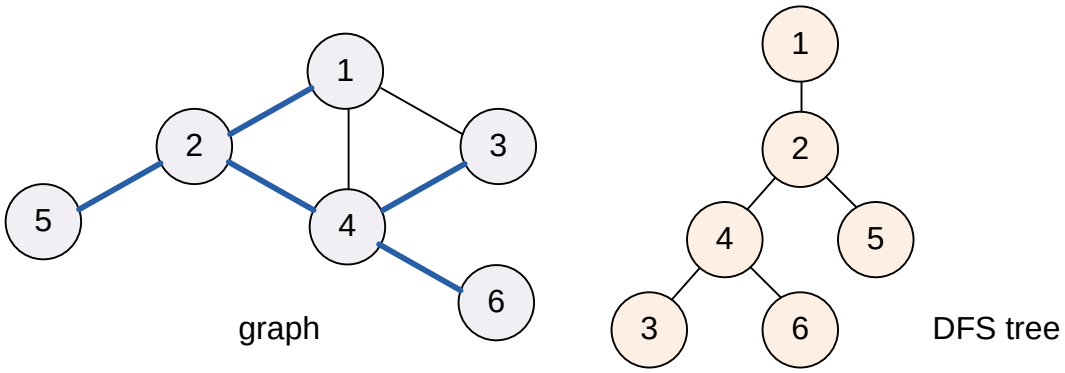
```
  // run depth first search from the vertex v
  scanf("%d", &v);
  dfs(v);
  printf("\n");
  return 0;
}
```

Consider the implementation of depth-first search using the adjacency list.

```
#include <cstdio>
#include <vector>
using namespace std;

vector<vector<int> > g;
vector<int> used;
int i, n, a, b;

void dfs(int v)
{
  // We switch on the lamp in the room (in the vertex v of the graph)
  used[v] = 1;

  // print the vertex v
  printf("%d ", v);

  // looking for a tunnel, through which you can get to the room that is not
  // visited
  // g[v] contains list of vertices adjacent to v:
  // g[v] = (to1, to2,..., tok) = (g[v][0], g[v][1],…, g[v][k]),k = g[v].size()
  for(int i = 0; i < g[v].size(); i++)
  {
    int to = g[v][i];

    // We have an edge (v, to). If to is not visited, go there
    if (used[to] == 0) dfs(to);
  }
}

int main(void)
{
  // vertices are numbered from 1 to n, initialize arrays
  scanf("%d %d", &n, &m);
  g.resize(n + 1);
  used.resize(n + 1);

  // read the graph
  for (i = 0; i < m; i++)
  {
    scanf("%d %d", &a, &b);
    g[a].push_back(b);
    g[b].push_back(a);
  }

  scanf("%d", &v);
  // run dfs from the vertex 1
  dfs(v);
  printf("\n");
  return 0;
}
```

E-OLYMP **978. Get a tree** Undirected connected graph is given. Remove some edges from it so that to get a tree. Print the edges remained in a tree.

► Run depth first search from the vertex 1. Print all the edges along which the depth first sesrch passes – this will be the resulting tree (depth first sesrch tree).

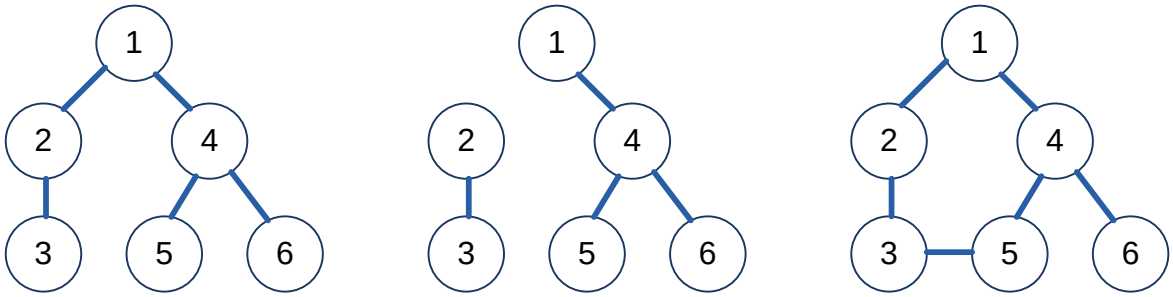


E-OLYMP 977. Is it a tree? Check if the given graph is a tree.

► If graph is a tree, then:

- Graph is connected;
- |V| = |E| + 1;
- Graph does not contain cycles;

From the first two conditions implies third condition. Graph is a tree is two first conditions are satisfied.



Connectivity of the graph can be checked in the next way: run depth first search from the first vertex. In the variable *c* count the number of visited vertices. If it equals to |V| = *n*, the graph is connected.

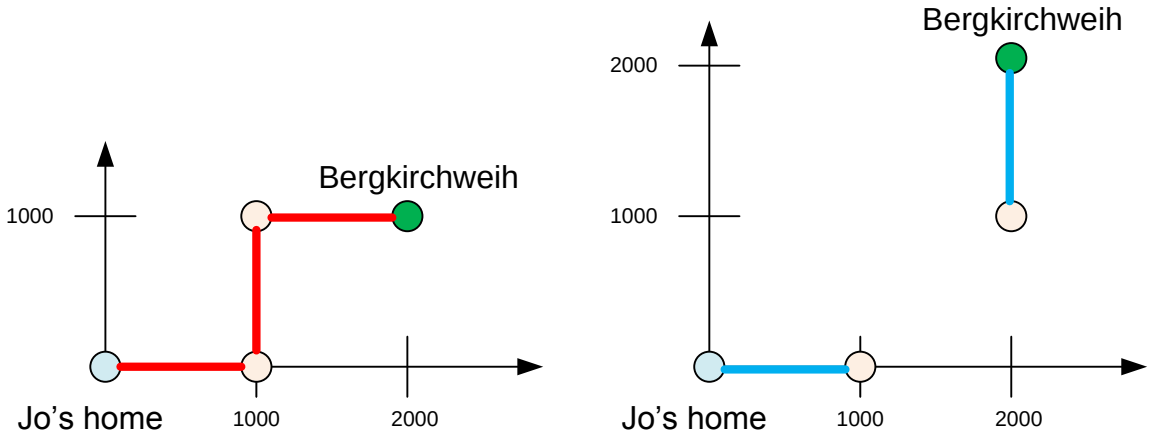
```
void dfs(int v)
{
  used[v] = 1;
  c++;
  for (int i = 1; i <= n; i++)
    if ((g[v][i] == 1) && (used[i] == 0)) dfs(i);
}
```

E-OLYMP 6033. Kastenlauf There are some points on a plane, given with its (*x*, *y*) coordinates:

- point 0 – Jo's home, start;
- points from 1 to *n* – stores selling beer;
- point (*n* + 1) – Bergkirchweih, finish

Jo and his friends want to run to the local fair, called Bergkirchweih. They can run from one point to another only if the distance betweeen them is no more than 1000

meters (they have only 20 bottles of beer and must drink one bottle every 50 meters). We must answer the question: can friends reach the Bergkirchweih?



► Let's build a graph with $(n + 2)$ vertices, numbered from 0 to $n + 2$. The edge exists between two vertices only if the Manhattan distance between them is no more than 1000. Manhattan distance between two points (x_1, y_1) and (x_2, y_2) equals to

$$|x_2 - x_1| + |y_2 - y_1|$$

For each test case we read first the number n of stores with beer.

```
scanf("%d", &n);
// 0 = Jo's house, start
// 1 .. n - stores selling beer
// n+1 - Bergkirchweih, finish
```

Read the coordinates of Jo's house, stores selling beer and Bergkirchweih.

```
for (i = 0; i < n + 2; i++)
  scanf("%d %d", &x[i], &y[i]);
```

Construct an adjacency matrix. If the Manhattan distance between points (x_i, y_i) and (x_j, y_j) is no more than 1000, then exists an edge between vertices i and j (g[i][j] = g[j][i] = 1).

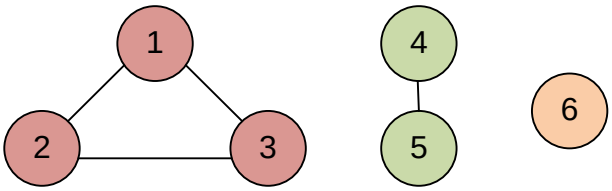
```
for (i = 0; i < n + 2; i++)
for (j = i + 1; j < n + 2; j++)
  if (abs(x[i] - x[j]) + abs(y[i] - y[j]) <= 1000)
    g[i][j] = g[j][i] = 1;
```

Connected components in undirected graphs

The ***connected component*** of the graph G = (V, E) is such subgraph G = (V', E') that for any $u, v \in$ V' there exists a path from u to v along the edges from E'.

Searching for all connected components in a graph means dividing its vertices into several groups such that inside one group you can go from any vertex to any other, and

there is no path between different groups. For example, the following graph has 3 connected components:



Graph is called ***connected***, if it contains only one connected component.

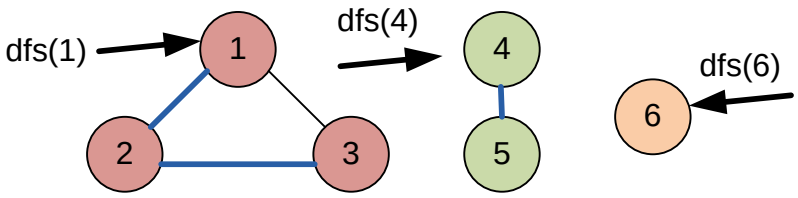
To find the connected components, we'll make a series of depth first search rounds. Run *dfs* from the first vertex. All the vertices that will be visited, are assigned to the first connected component. Look for the next unvisited vertex and run *dfs* from it. All the vertices visited from it form the second connected component. We continue the process of calling *dfs* until unvisited vertices exists.

The ***number of connected components*** *res* can be found in the next way:

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

To find the number of connected components for the graph given above, the following calls will be made:

- dfs(1): visit vertices 1, 2, 3. First connected component, *res* = 1;
- dfs(4): visit vertices 4, 5. Second connected component, *res* = 2;
- dfs(6): visit vertices 6. Third connected component, *res* = 3;



E-OLYMP **2269. Connected components** Find the number of connected components in undirected graph.

► Read adjacency matrix. Count the number of connected components.

E-OLYMP **982. Connectivity** Graph is given. Check is it connected.

► Read list of edges. Construct adjacency matrix. Count the number of connected components. Graph is connected if it contains only one connected component.

E-OLYMP **4000. Depth search** Given undirected graph and vertex *s*. Find the number of vertices in connected component where vertex *s* is located.

► Read adjacency matrix. Run DFS from the vertex *s*. Count the number of visited vertices during DFS.

E-OLYMP 776. Roads Given non-connected graph. Find minimum edges to add to get a connected graph.

► Read list of edges. Construct adjacency list because $n \leq 10.000$. Use DFS to count the number of connected components.

Timestamps

In depth first search algorithm you can use the **vertex color.** Initially all the vertices are colored *white*. When the vertex is passed for the first time, it turns *gray*. The vertex is ***processed***, if it is fully reviewed all of its adjacent vertices. When the vertex is processed, it becomes *black*.

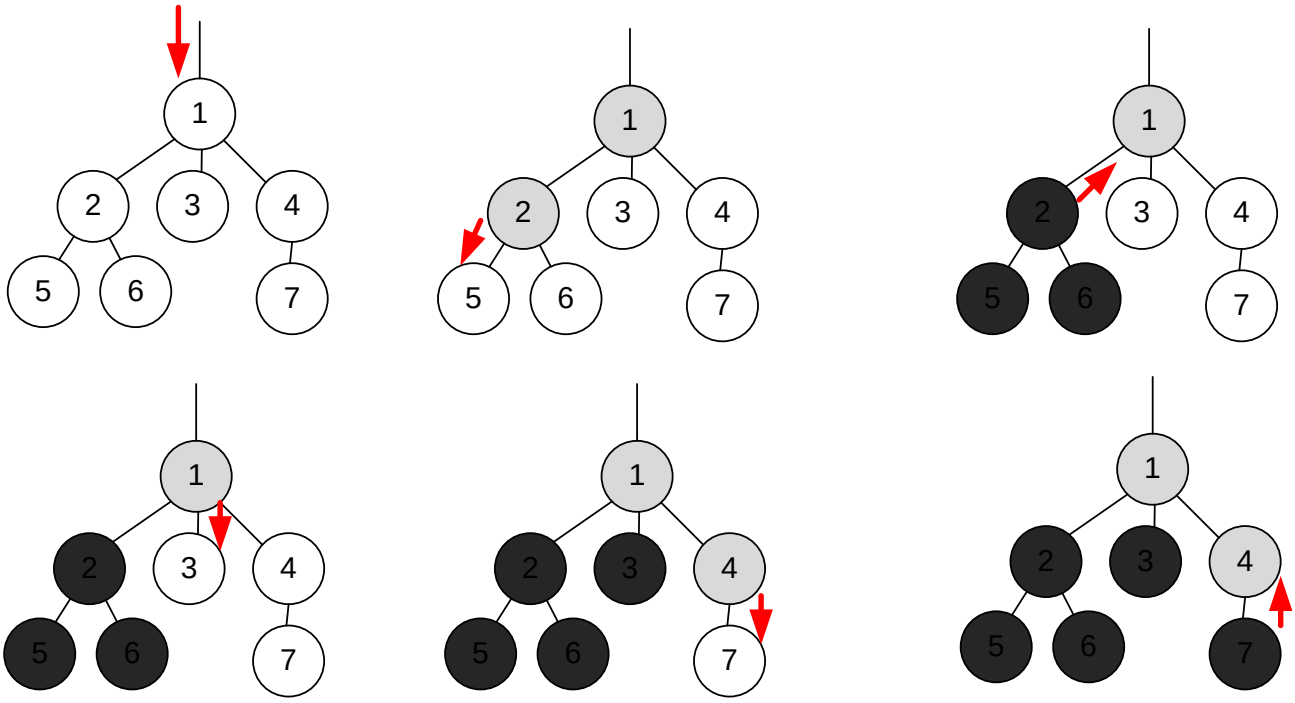
- used[*v*] = 0, vertex *v* is not visited (not colored);
- used[*v*] = 1, vertex *v* is visited first time, it is gray;
- used[*v*] = 2, vertex *v* is processed, it is black;

```
void dfs(int v)
{
  used[v] = 1;
  for (int i = 1; i <= n; i++)
    if ((g[v][i] == 1) && (used[i] == 0)) dfs(i);
  used[v] = 2;
}
```

Each vertex can be attributed with two **timestamps**:

- d[*v*] is the time when the vertex is detected and turned *gray*;
- f[*v*] is the time when the vertex is processed and turned *black*.

These labels are used in many graph algorithms and are useful for analyzing the properties of depth-first search.

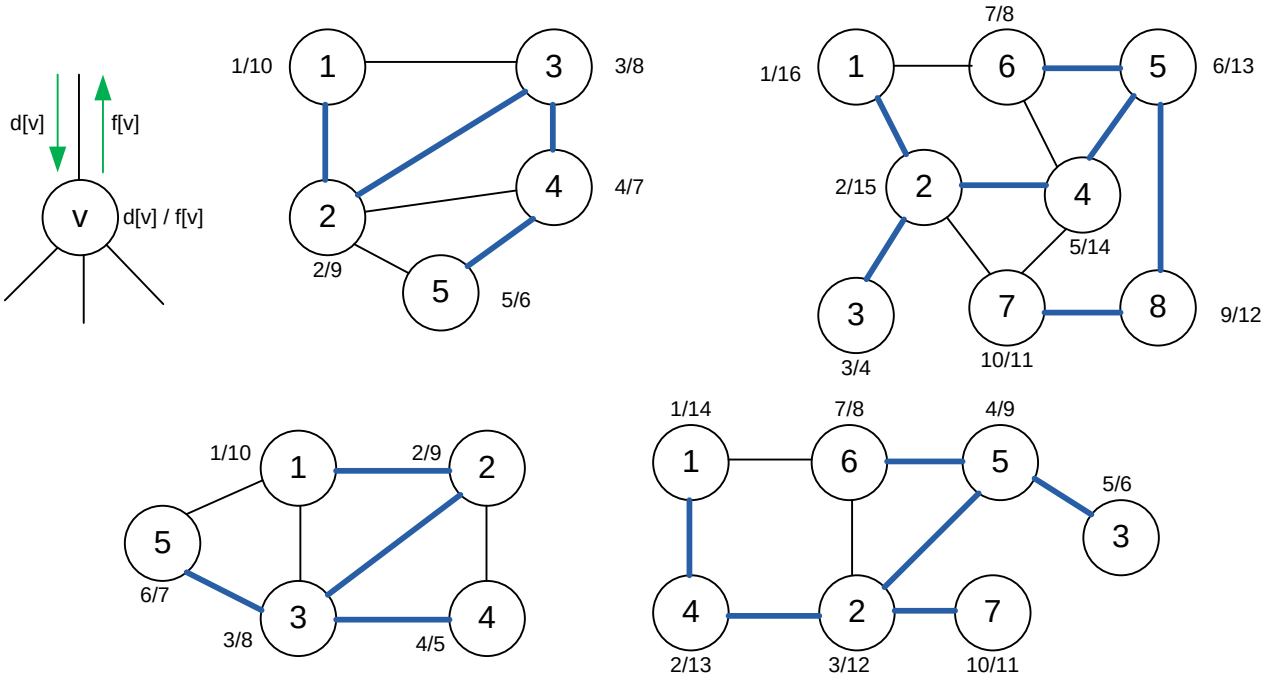


Timestamps $d[v]$ and $f[v]$ are integers from 1 to 2|V|. For each vertex v there is inequality: $d[v] < f[v]$. Vertex v is white till time $d[v]$, gray from $d[v]$ to $f[v]$ and black after $f[v]$.

Depth first search on a connected graph can be described as follows:

DFS (u)

1. Denote vertex u as passed and color it ***gray*** (enter to vertex u);
2. For each vertex v, adjacent to u, and colored to white, call dfs(v);
3. Color vertex u ***black*** (exit from vertex u);



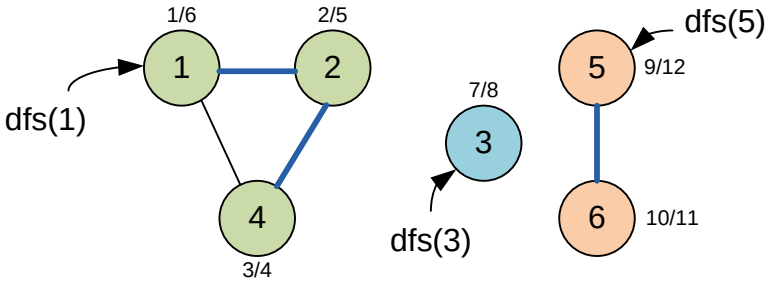
E-OLYMP 8761. Depth First Search - Timestamps Given undirected graph. Run DFS from vertex *v*. Print the values of timestamps d[*v*] / f[*v*] for each vertex *v*.

► Read list of edges. Construct adjacency matrix. Add timestamps to dfs function.

```
void dfs(int v)
{
  d[v] = t++;
  used[v] = 1;
  for (int i = 1; i <= n; i++)
    if ((g[v][i] == 1) && (used[i] == 0)) dfs(i);
  f[v] = t++;
}
```

Let *t* be the global variable, initialized to 0. In the *main* part run DFS from the vertex *v*.

E-OLYMP 9654. Depth First Search on a disconnected graph Given undirected disconnected graph. Run DFS on it. Print the values of timestamps when the vertices become **gray** / **black** for each vertex in the order of their first visit.



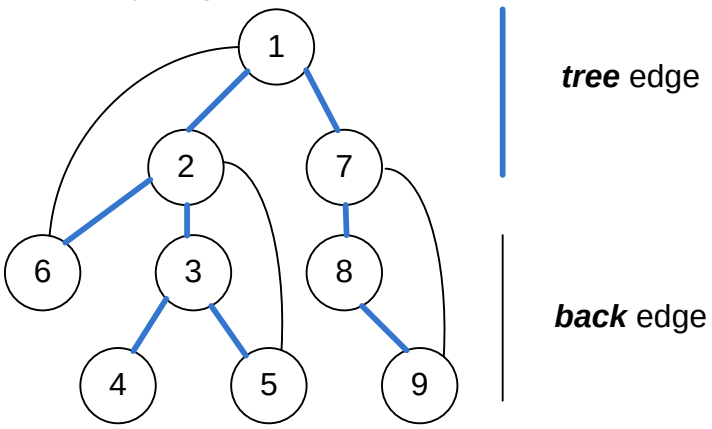
Vertex: 1, Gray 1, Black 6
Vertex: 2, Gray 2, Black 5
Vertex: 4, Gray 3, Black 4
Vertex: 3, Gray 7, Black 8
Vertex: 5, Gray 9, Black 12
Vertex: 6, Gray 10, Black 11

► Read list of edges. Construct adjacency matrix. Add timestamps to *dfs* function. Run *dfs* on disconnected graph.

Cycle detection in undirected graph

Edges of undirected graph can be divided into two groups:

- ***Tree edges*** – edges which are present in the **tree** obtained after applying DFS on the graph.
- ***Back edges*** – edges which are not part of DFS **tree**. (*u*, *v*) is a back edge if during dfs we try to go from *u* to *v*, but vertex *v* is already used.



If you draw the dfs tree on the plane, ***back edges*** usually run from *bottom* to *up*: 6 → 1, 5 → 2, 9 → 7.

Each ***back edge*** gives a cycle:

- 6 → 1: cycle 1 – 2 – 6 – 1
- 5 → 2: cycle 2 – 3 – 5 – 2
- 9 → 7: cycle 7 – 8 – 9 – 7

If undirected graph contains a back edge, then it contains a cycle.

Consider the picture above. Let we go from vertex 1 to vertex 2. Now we are in the vetex 2. If we try to move from 2 to 1, we see that vertex 1 is already visited, but the edge 2 – 1 does not give us a cycle. We can't move from the vertex to its parent (we can only return back to the parent if no other way found). So during dfs(*v*) for each vertex *v* we need to hold its parent vertex *prev*.

E-OLYMP 1390. Car race Given undirected disconnected graph. Check whether it contains a cycle.

► Read list of edges. Construct adjacency matrix. Set global variable *flag* = 0 meaning there is no cycle. When cycle will be found, set *flag* = 1.

```
void dfs(int v, int prev = -1)
{
  // mark vertex v as used
  used[v] = 1;

  // we try to go from v to i, i = 1,2,…,n
  for (int i = 1; i <= n; i++)

  // we can't go to parent vertex prev (vertex i can't be prev)
  // and there must be an edge from v to i (g[v][i] = 1)
    if ((i != prev) && g[v][i] == 1)

      // if vertex i is already used and i ≠ prev,
      // v – i is a back edge, set flag to 1.
      // otherwise run dfs from i with its parent v
      if (used[i] == 1) flag = 1; else dfs(i, v);
}
```

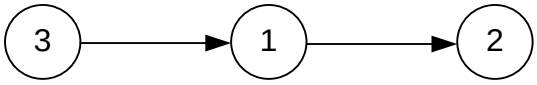
Run depth first search on disconnected graph (try to call DFS from each vertex).

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

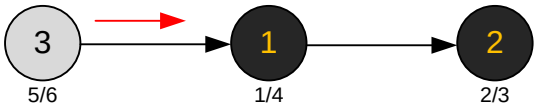
If after these dfs calls *flag* = 1, cycle exists. Otherwise there is no cycle.

Cycle detection in directed graph

Previous algorithm for cycle detection does not work for oriented graphs. Consider the next graph:

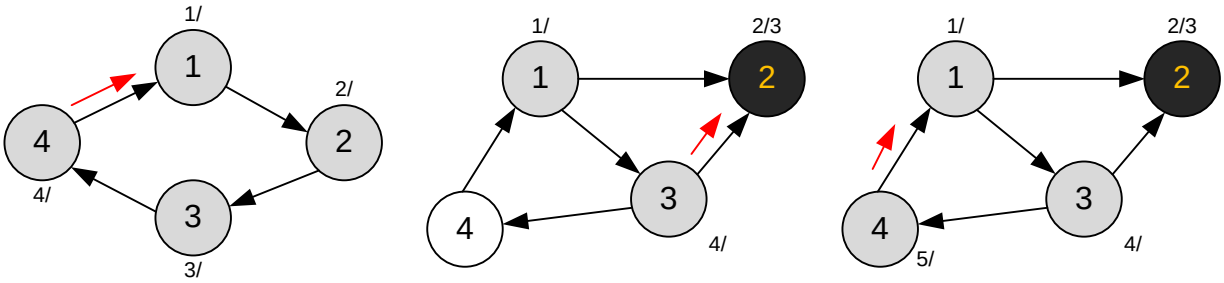


We start dfs from vertex 1, mark it as used. Then go to vertex 2, mark it as used. Return back from 2 to 1. Return back from 1. Now we run dfs from 3. Vertices 1 and 2 are already black. Vertex 3 is gray. An edge 3 – 1 runs into the vertex 1 which is already used.

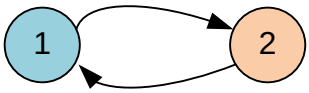


But an edge 3 – 1 does not create a cycle. It is because this edge runs into *black* vertex.

If directed graph contains an edge that runs into GRAY vertex, then it contains a cycle.



E-OLYMP **4004. Is there a cycle?** Given directed graph. Check whether it contains a cycle. For example, next graph contains a cycle:



► Declare adjacency matrix g and used array.

```
#define MAX 51
int g[MAX][MAX], used[MAX];
```

Depth first search from vertex *v*. *flag* is a global variable, initialised to 0 (there is no cycle initially). If cycle will be found, set *flag* to 1.

```
void dfs(int v)
{
  // mark vertex v as GRAY, make an entrance to v
  used[v] = 1;

  // we try to go from v to i, i = 1,2,…,n
  for (int i = 1; i <= n; i++)

    // if there exists an edge from v to i
    if (g[v][i])
    {

      // if vertex i is GRAY, we meet cycle
      if (used[i] == 1) flag = 1;

      // if vertex i is not visited, run dfs from there
      else if (used[i] == 0) dfs(i);
```

```
      // if vertex i is black (used[i] = 2), do nothing
    }
  // mark vertex v as BLACK, make an exit from v
  used[v] = 2;
}
```

Read adjacency matrix.

```
scanf("%d", &n);
for (i = 1; i <= n; i++)
for (j = 1; j <= n; j++)
  scanf("%d", &g[i][j]);
```

Run DFS like on disconnected graph.

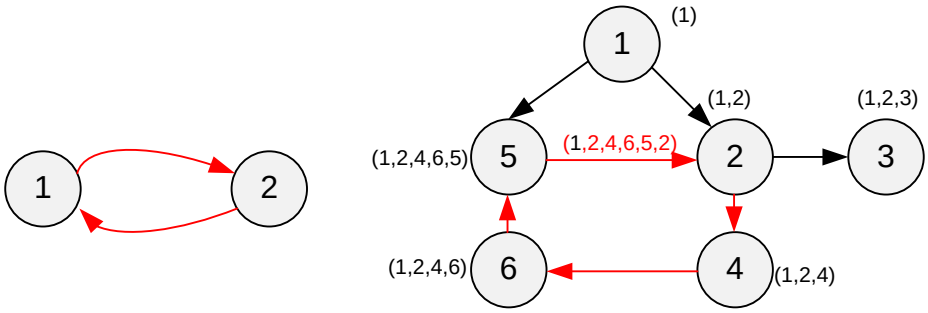
```
flag = 0;
for (i = 1; i <= n; i++)
  if (used[i] == 0) dfs(i);
```

Depending on the value of *flag*, print the answer.

E-OLYMP **2270. Find a cycle** Given directed graph. Check whether it contains a cycle. If yes, print any cycle.

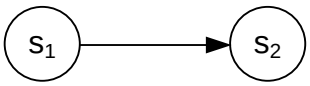
► Use adjacency list to run dfs to detect cycles in directed graph. In the stack (vector) keep the sequence of vertices you passed during dfs to restore found cycle.

These graphs are given at input. Near each vertex of the second graph the state of the stack is given. It contains a cycle 2 – 4 – 6 – 5.

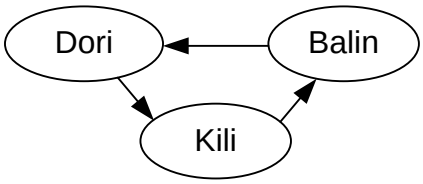


E-OLYMP **7945. Dwarves** Given directed graph. Check whether it contains a cycle.

► Create a graph which vertices are dwarves. Lets s_1 and s_2 be two dwarves. If $s_1 < s_2$, create a directed edge:



If graph contains a cycle, the statements are not consistent, we should output "**impossible**". For example, if for three dwarves we have the next relation among their heights: *Balin* < *Dori*, *Dori* < *Kili*, *Kili* < *Balin* then this is impossible.

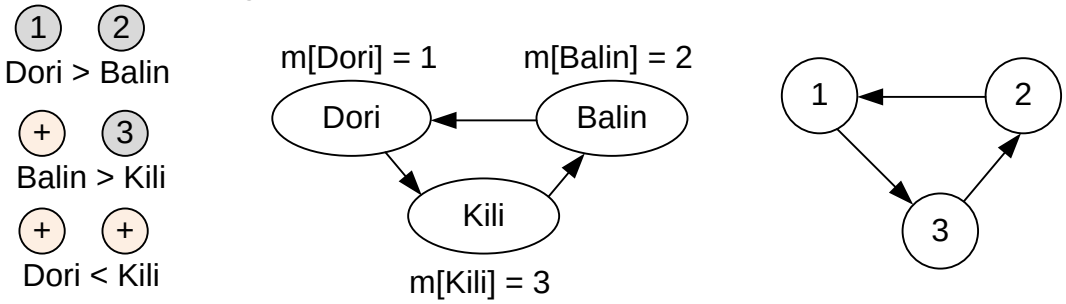


Input data contains names of the dwarves. We must assign to each dwarf the vertex (number). Let's declare

```
map<string, int> m;
```

and keep the number of the vertex for dwarf with the name *s* in m[*s*]. Data structure m is like a database that contains numbers of the vertices for each dwarf. We read input data and number the dwarves in the same order like they appear first time in the input.

For example, first appears Dori. We set m[Dori] = 1. Next appears Balin. We set m[Balin] = 2. In the second condition first appears Balin. m[Balin] is not 0 (it is already assigned the value). Then goes Kili. We set m[Kili] = 3.



Declare adjacency list for the graph:

```
vector<vector<int> > g;
```

Read the number of relations between the dwarves:

```
scanf("%d", &stat);
```

Number of dwarves does not exceed 10^4.

```
g.resize(10001);
```

Read the input relations and construct a graph.

```
// number of dwarves will be counted in n
n = 0;
for (i = 0; i < stat; i++)
{
  // read relation s1 < s2 or s1 > s2
  scanf("%s %c %s", s1, &ch, s2);
  // m[s] contains the vertex number for dwarf s
  // if m[s1] yet is not set up, assign vertex number ++n to dwarf s1
  if (m[s1] == 0) m[s1] = ++n;
  if (m[s2] == 0) m[s2] = ++n;

  // s1 < s2 for dwarves means m[s1] -> m[s2]
  // there is a directed edge from m[s1] to m[s2]
  int from = m[s1], to = m[s2];
  if (ch == '<')
    g[from].push_back(to); // from -> to
```

```
  else
  g[to].push_back(from); // to ->from
}
```

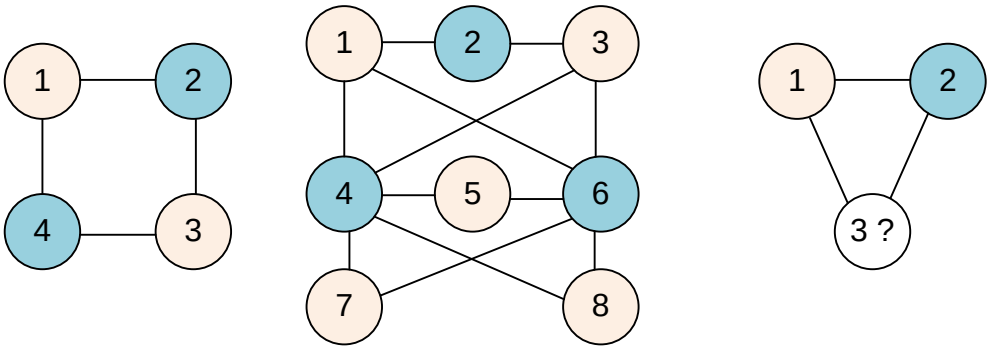
Run dfs on a directed graph like disconnected.

```
flag = 0; // no cycle initially
for (i = 1; i <= n; i++)
{
  if (used[i] == 0) dfs(i);
  if (flag == 1) break;
}
```

Depending on the value of *flag*, print the answer.

Bicolorable graph

Graph is called ***bicolorable*** if its vertices can be colored with two colores so that each edge connects vertices of different colors.



First two graphs are ***bicolorable***. Third one is not bicolorable, because the color of the vertex 3 can't coinside neither with color of vertex 1, nor with color of vertex 2.

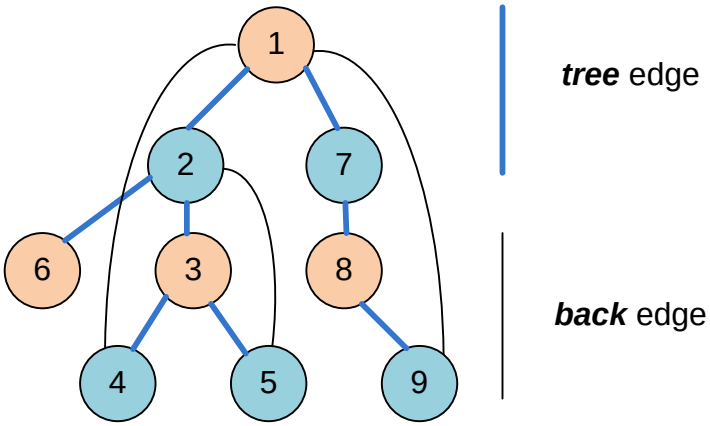
You can notice that **cycle** of even size is bicolorable, but cycle of odd size is not bicolorable. For example second graph is bicolorable and contains cycles of only even size.

To check is the graph *bicolorable*, run dfs from vertex 1. Lets color vertex *v* with one of the next ways:

- used[*v*] = 0, vertex *v* is not visited, it is not colored;
- used[*v*] = 1, vertex *v* is colored with color 1 (for example red);
- used[*v*] = 2, vertex *v* is colored with color 2 (for example blue);

Let's color starting vertex 1 with color 1. Then all its neighbours must be colored with color 2. All neighbours of vertex with color 2 must be colored with color 1 and so on. So if vertex *v* has color *col*, its neighbours should be colored with color 3 – *col*.

The problem can appear when we meet the back edge – the edge that runs into already visited vertex. If *v* – *u* is a back edge and their colors are the same, then graph is *not bicolorable*.



Everything was fine till we arrive to the vertex 5. Back edge 5 – 2 runs from blue vertex 5 into blue vertex 2. Graph is not bicolorable. We have a cycle of odd length: 2 – 3 – 5 – 2.

E-OLYMP **3165. Bicoloring** Given connected undirected graph. Check is it bicolorable.

► Declare adjacency matrix g and used array.

```
#define MAX 1001
int g[MAX][MAX], used[MAX];
```

Depth first search starts from vertex *v*. Vertex *v* is colored with *color*. *Error* is a global variable, initialised to 0 (initially let graph be bicolorable). If back edge will be found that connects vertices of the same color, set *Error* to 1.

```
void dfs(int v, int color)
{
  // if graph is not bicolorable, no sence to continue dfs
  if (Error) return;

  // color the vertex v
  used[v] = color;

  // we try to go from v to i, i = 1,2,…,n
  for (int i = 1; i <= n; i++)
    // if there is an edge from v to i
    if (g[v][i] == 1)
    {
      // if vertex i is not colored (not visited),
      // continue dfs from it. Color vertex i with 3 - color
      if (used[i] == 0) dfs(i, 3 - color);else
      // vertex i is already colored (visited)
      // if color of v and i are the same, graph is NOT bicolorable
      if (used[v] == used[i]) Error = 1;
    }
}
```

Main part of the program. Read the number of graph nodes *n*.

```
while (scanf("%d", &n), n)
{
```

Read the number of edges *l*.

```
  scanf("%d", &l);
  memset(g, 0, sizeof(g));
  memset(used, 0, sizeof(used));
```

Read the list of edges. Construct adjacency matrix.

```
  for (i = 0; i < l; i++)
  {
    scanf("%d %d", &a, &b);
    g[a][b] = g[b][a] = 1;
  }
```

Set *Error* to 0. Run dfs from vertex 1. Color starting vertex 1 with color 1.

```
  Error = 0; dfs(1, 1);
```

Depending on the value of *Error*, print the answer.

```
  if (Error) printf("NOT BICOLOURABLE.\n");
  else printf("BICOLOURABLE.\n");
}
```

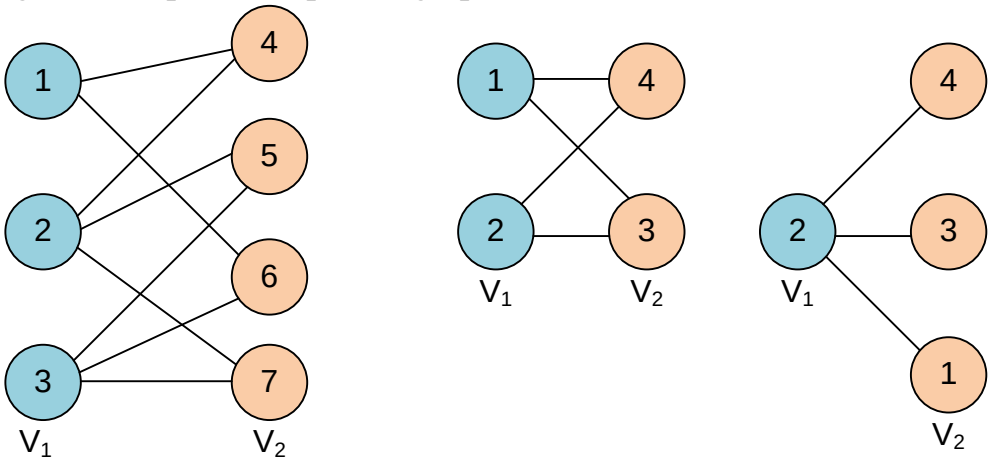
E-OLYMP **4002. Cheating away!** Given disconnected undirected graph. Check is it bicolorable.

► Use dfs for bicoloring detection. Run dfs from each vertex because graph is not connected.

Bipartite graph

Graph G(V, E) is called **bipartite** (or **bigraph**) if its vertices V can be divided into two *disjoint sets* ($V = V_1 \cup V_2$) so that every edge connects vertices from different sets.

Below given samples of bipartite graphs:



Graph is bipartite if and only if it is bicolorable.
Bipartite graphs can have cycles of even length only.

Run time of DFS

For each vertex v the call of dfs(v) is performed only once, since the call occurs only for *white* vertices, and after call it immediately colors to gray. In the body of dfs(v) loop is executed |V| times. Therefore, run time of the DFS will be $O(V^2)$, if the graph represents the adjacency matrix. If the graph represent the adjacency list, then in the body of dfs(v) loop is executed |adj(v)| times (through adj(v) we'll denote the set of edges emanating from v). $\sum_{v \in V} |adj(v)| = O(E)$, the running time of the depth-first search algorithm equals to $O(V + E)$.