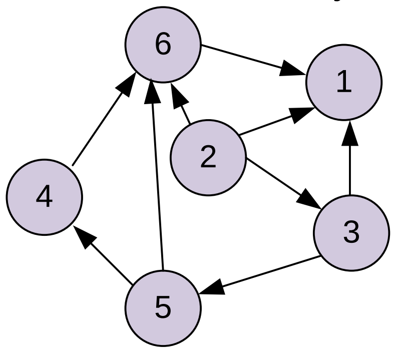
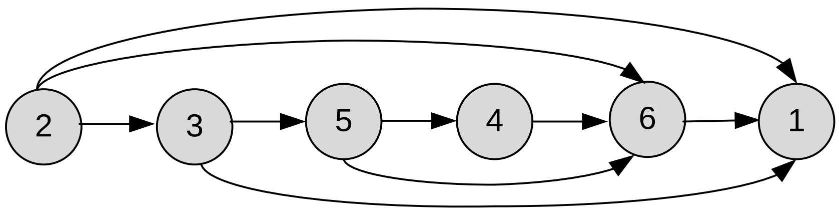
Topological sort

The problem of topological sorting of a graph is to indicate such a linear order on its vertices so that any edge leads from a vertex with a lower number to a vertex with a higher number. Obviously, if there are cycles in the graph, then there is no such order.

Example. Consider a graph that does not contain cycles:



Sort topologically its vertices: 2, 3, 5, 4, 6, 1.



Theorem. An acyclic graph always has a vertex without incoming edges.

Proof. Assume the opposite, let an edge enters each vertex. For an arbitrary vertex x, we denote one of these edges by (prev[x], x). The sequence x, prev[x], $\text{prev}^2[x]$, … is infinite, and the number of vertices in the graph is finite. Consequently, some vertex y will occur twice in this sequence. Consider the part of this sequence between repetitions: y, prev[y], $\text{prev}^2[y]$, …, y. Expanding this sequence in the opposite direction, we get a cycle in the graph. We came to a contradiction.

Perform a topological sort of the vertices. The very first vertex in this topological order hasn't incoming edges.

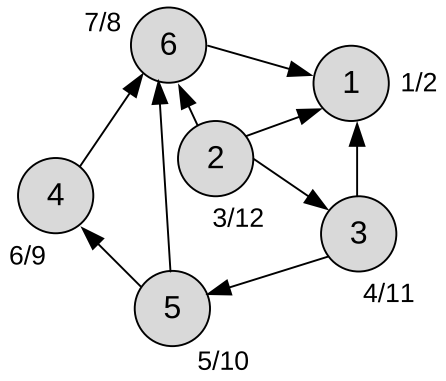
Theorem. The topological sort of vertices in a graph is possible if and only if it does not contain cycles.

Topological sort implementation using depth first search

The problem of topological sort can be solved using depth first search. Initially, all vertices are white. When the ***dfs*** enters the vertex, it becomes gray. When the vertex is processed, it turns black. The order of the vertices in a topological sort is the inverse order in which the vertices become black.

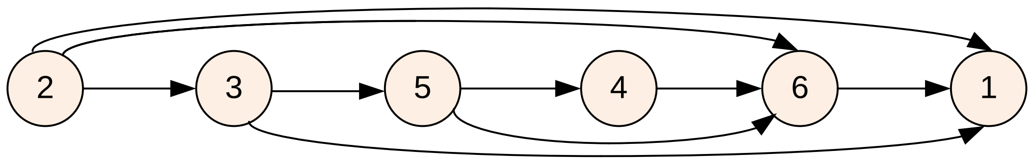
The complexity of topological sort algorithm equals to the time it takes to traverse all the vertices of the graph using ***dfs*** algorithm, that is O(n + m).

Example. Start a depth first search on the graph. Next to each vertex *v*, place the labels d[*v*] / f[*v*]. To determine the topological sort order, one should sort the graph vertices in descending order of labels f[*v*].



The first vertex to be colored black will be 1. It will be the last vertex in topological order. The second vertex colored black will be 6. The last will be vertex 2.

v	2	3	5	4	6	1
f[v]	12	11	10	9	8	2

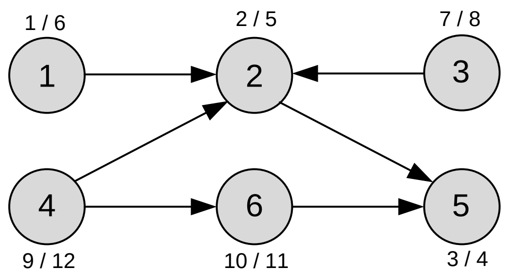


E-OLYMP 1948. Topological sort The directed unweighted graph is given. Sort topologically its vertices.

► Topological sorting is performed using the depth first search. Initially, all vertices are white. When the ***dfs*** enters the vertex, it becomes gray. When the vertex is processed, it turns black. The order of the vertices in a topological sort is the inverse of the order in which the vertices become black. That is, the first (last) fully processed vertex in ***dfs*** will be the last (first) in the topological sort.

The vertices of a graph cannot be topologically sorted if there is a cycle in the graph. Since the graph is directed, there should be no edges going to the gray vertices during ***dfs***.

The graph shown in the sample, has the form:



Place the labels d[*v*] / f[*v*] near each vertex *v*. Topologically sorted vertices are arranged in descending order of labels f[*v*].

v	4	6	3	1	2	5
f[v]	12	11	8	6	5	4

Since the number of vertices in the graph is large, store the graph as an adjacency list *g*. Store the vertex labels in the array *used*:

- used[*i*] = 0, if vertex *i* is not visited yet (vertex is white);
- used[*i*] = 1, if vertex *i* is visited already, but its processing is not finished yet (vertex is gray);
- used[*i*] = 2, if vertex *i* is processed already (vertex is black);

Store the vertices in array *top* in the order of completion of their processing during ***dfs***.

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

Function ***dfs*** implements the depth first search from the vertex *i*.

```
void dfs(int i)
{
```

We entered the vertex *i*. Make it gray.

```
  used[i] = 1;
```

Iterate over the vertices *to*, where we can go from *i*.

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

If the directed edge (*i*, *to*) goes to the gray vertex, then graph contains a cycle.

```
    if (used[to] == 1) Error = 1;
```

If the vertex *to* is not visited yet, run recursively ***dfs*** from it.

```
    if (used[to] == 0) dfs(to);
  }
```

Finish processing the vertex *i*. Make it black and add it to the array *top*.

```
  used[i] = 2;
  top.push_back(i);
}
```

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

```
scanf("%d %d",&n,&m);
g.resize(n+1); used.resize(n+1);

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

Run the depth first search on directed graph.

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

If graph contains a cycle (during ***dfs*** *Error* = 1 is set), then print -1.

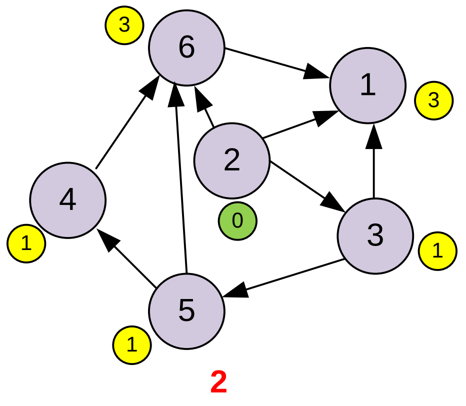
```
if (Error) printf("-1");
else
```

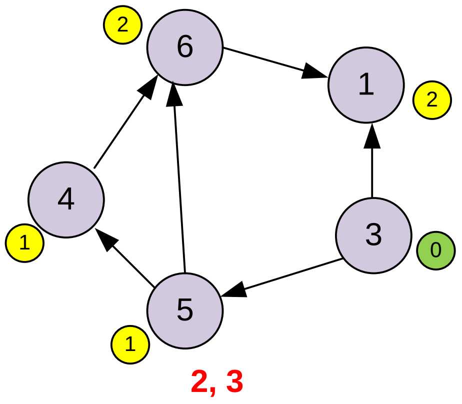
Print the vertices of the graph in the reverse order of the one in which they were pushed into the array *top*.

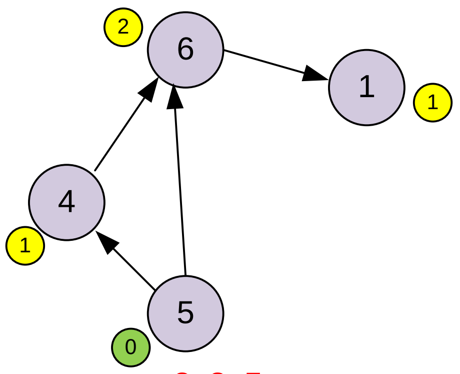
```
  for(i = n - 1; i >= 0; i--)
    printf("%d ",top[i]);

printf("\n");
```

Topological sort implementation using Kahn algorithm

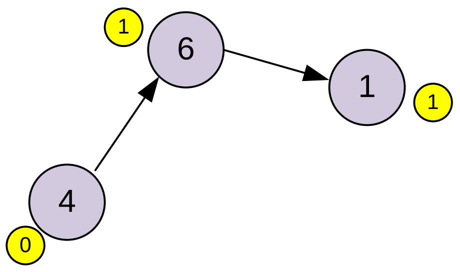
Compute the incoming degree for each vertex. Push the vertices with zero incoming degree into the queue. While the queue is not empty, pop the vertex out of the queue and add it to the end of the topological order. For each vertex *v* removed from the queue, simulate the removal of all edges (*v*, *u*) outgoing from it. That is, for each such edge, the incoming degree of the vertex *u* should be decreased by one. If after this reduction the incoming degree of the vertex *u* becomes zero, push *u* into the queue. The algorithm runs until the queue becomes empty. If all the vertices have been queued, then the topological order is constructed. Otherwise, after removing some vertices, we get a graph without vertices of degree zero. This is possible only if there is a cycle in the graph. In this case there is no topological ordering.



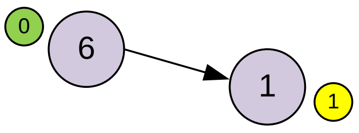




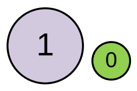
2, 3, 5



2, 3, 5, 4



2, 3, 5, 4, 6



2, 3, 5, 4, 6, 1

The input graph is stored in the adjacency list *graph*. Store the incoming degrees of vertices in the array *InDegree*. Push the topologically sorted vertices of the graph into the array *top*.

```
vector<vector<int> > graph;
vector<int> InDegree, top;
deque<int> q;
int i, j, a, b, n, m, v, to;
```

Read the input graph.

```
scanf("%d %d",&n,&m);
graph.assign(n+1,vector<int>());
InDegree.assign(n+1,0);
for(i = 0; i < m; i++)
  scanf("%d %d",&a,&b),graph[a].push_back(b);
```

Iterate over all the edges of the graph. Compute the incoming degrees of all vertices. For each edge (*i*, *to*) increase InDegree[*to*] by 1.

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

Push all vertices with incoming degrees zero into the queue *q*.

```
for(i = 1; i < InDegree.size(); i++)
  if (!InDegree[i]) q.push_back(i);
```

Continue the algorithm until the queue *q* is not empty.

```
while(!q.empty())
{
```

Pop the vertex *v* from the queue and push it to the end of the topological order.

```
  v = q.front(); q.pop_front();
  top.push_back(v);
```

Delete the edges (*v*, *to*) from the graph. For each such edge decrease the input degree of the vertex *to*. If the degree of the vertex *to* becomes zero, push it into the queue, from where it will be pushed into the topological order list.

```
  for(i = 0; i < graph[v].size(); i++)
  {
    to = graph[v][i];
    InDegree[to]--;
    if(!InDegree[to]) q.push_back(to);
  }
}
```

If not all *n* vertices are pushed into the array *top*, then graph contains a cycle and topological sort is impossible.

```
if (top.size() < n)
  printf("-1\n");
else
{
```

Print the vertices of the graph in topological order.

```
  for(i = 0; i < top.size(); i++) printf("%d ",top[i]);
  printf("\n");
}
```