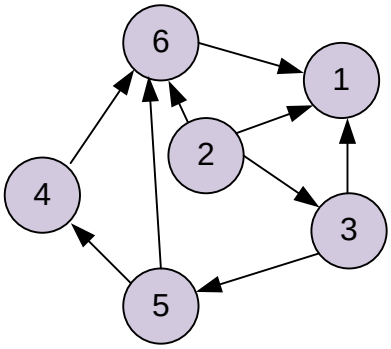
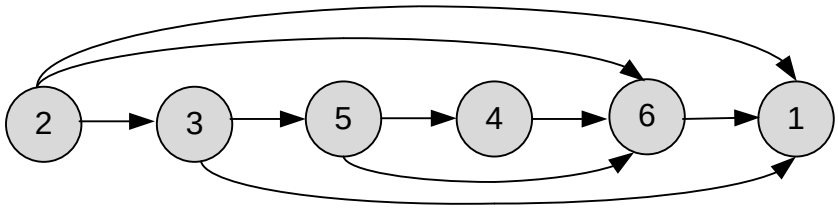
Topological sort

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], 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], 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.

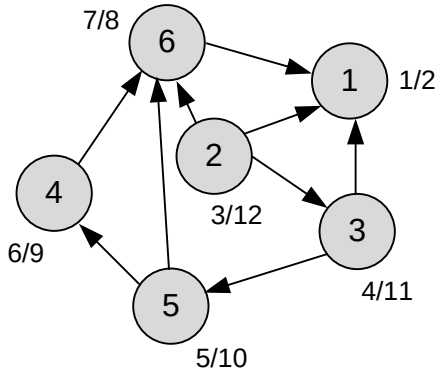
Theorem. The topological sort of vertices in a graph is possible if and only if it does not contain cycles.

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.

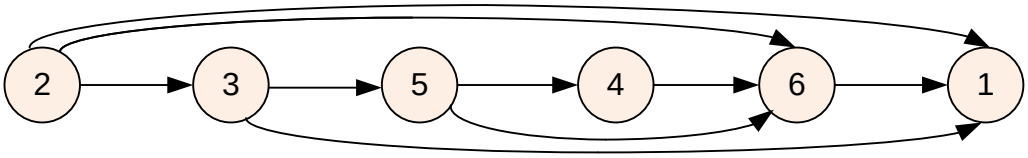
The complexity of topological sort algorithm equals to the time it takes to traverse all the vertices of the graph using ***dfs*** algorithm, that is O(n + m).

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*].

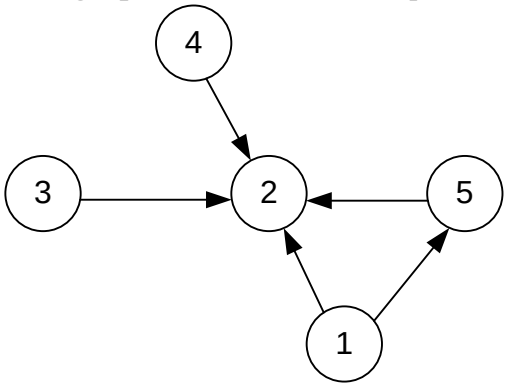


The first vertex to be colored black will be 1. It will be the last vertex in topological order. The second vertex colored black will be 6. The last will be vertex 2.

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



Exercise. Look at the next graph and answer the questions:



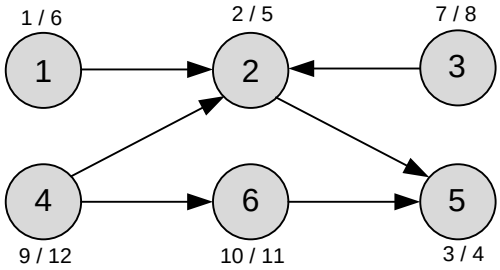
- Find any topological ordering for the graph vertices;
- Find lexicographically ***minimum*** topological sort;
- Find lexicographically ***maximum*** topological sort;
- Is the topological sort ***unique***? Give an example of the graph with unique topological sort.

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*.

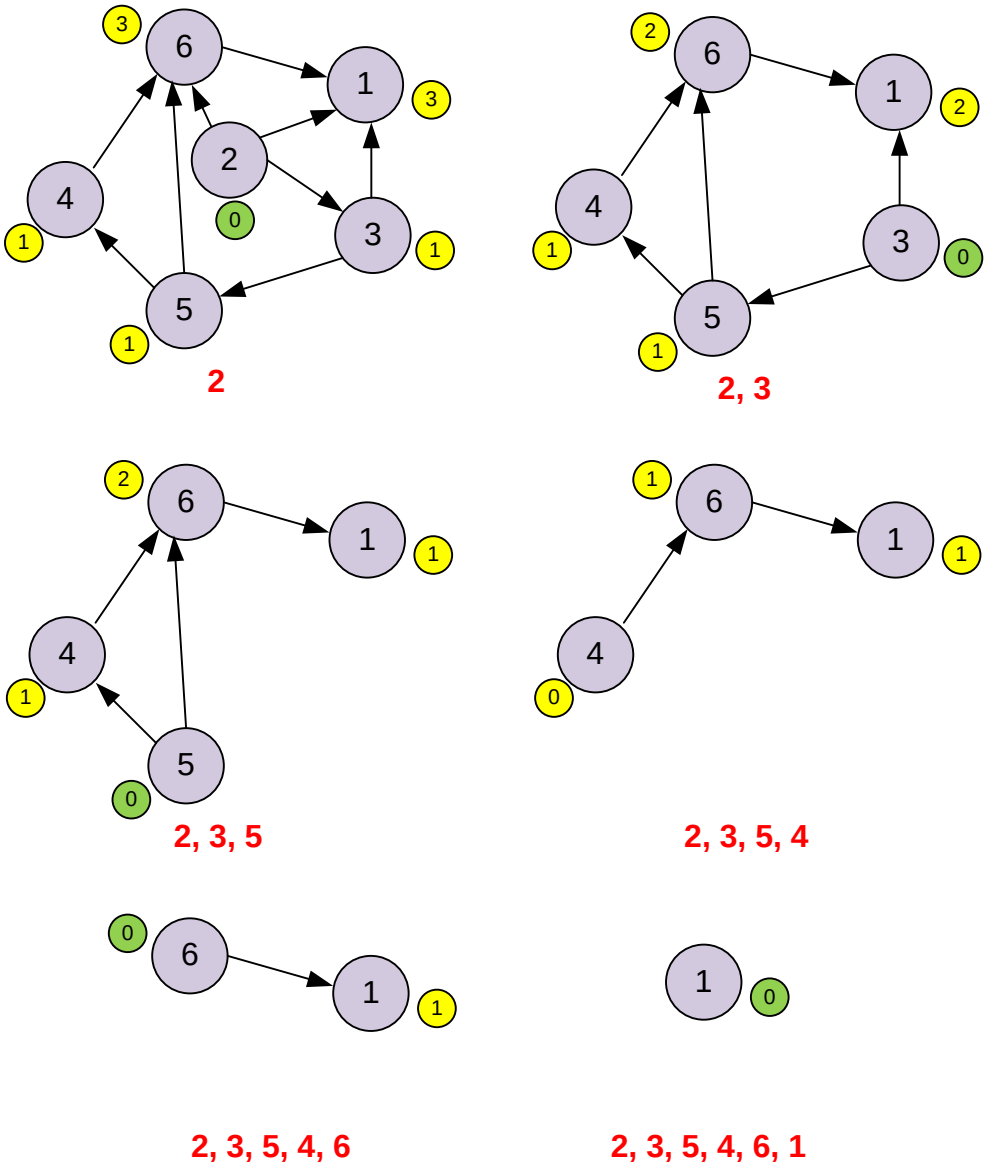
```
  for(i = n - 1; i >= 0; i--)
    printf("%d ",top[i]);

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 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.



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 data.

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

For each edge (*a*, *b*) increase InDegree[*b*] by 1.

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

Push all vertices with incoming degree 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
{
```

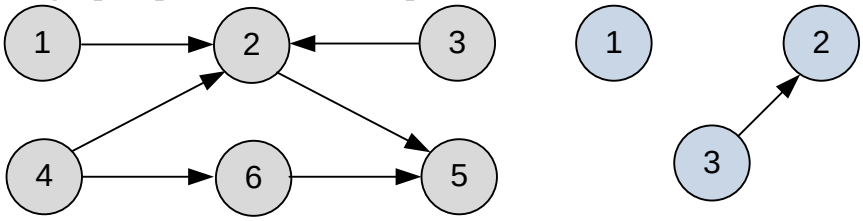
Print the vertices of the graph in topological order.

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

E-OLYMP 10235. Ordering tasks John has *n* tasks to do. Unfortunately, the tasks are not independent and the execution of one task is only possible if other tasks have already been executed.

► In the problem it is necessary to perform topological sorting of the vertices of the directed graph.

Consider the graphs presented in sample.



Possible topological sorts for the first graph are:

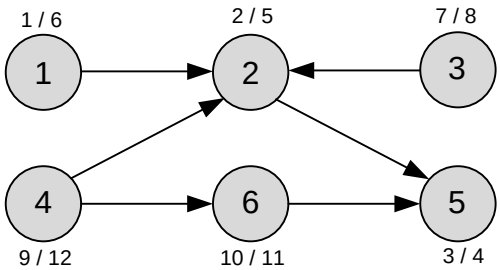
- 3, 1, 4, 2, 6, 5;
- 1, 3, 4, 6, 2, 5;
- 4, 1, 6, 3, 2, 5;

E-OLYMP 10651. The smallest topological sort The directed unweighted graph is given. Find the lexicographically smallest topological ordering of its vertices.

► In this problem one must find the lexicographically smallest topological sort. Let's use the Kahn algorithm. Instead of the classic queue, we'll use a priority queue or a set.

Initially insert to the set the vertices that have no incoming edges (from which the topological sort can start). At each iteration, we'll extract the smallest element from the set – this will ensure the construction of the lexicographically smallest topological sort.

The graph from example has the next form:



The lexicographically ***smallest*** topological sort is

1, 3, 4, 2, 6, 5

The lexicographically ***biggest*** topological sort is

4, 6, 3, 1, 2, 5

E-OLYMP 10652. The unique topological sort The directed unweighted graph is given. Find out if it has a unique topological ordering of its vertices.

►.

E-OLYMP 10648. Avengers NurlashKo, Nurbakyt, and Zhora are members of the last ninja clan fighting against even emperor Ren'swild reign. After devastating

defeat in an open battle, they decided split their army into three camps and wage a guerrilla war.

One Emperor Ren's ridiculous reforms allows to pass roads between cities only in one direction. Also, he chose the allowed directions of the roads in such way, so that it's impossible to start and return to the same city after passing several roads.

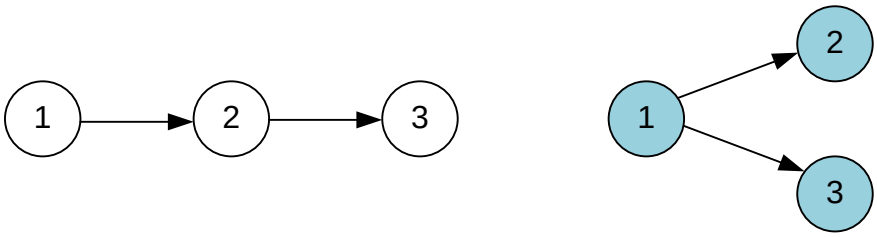
Right now, the clan is deciding where to place their camps. Emperor Ren's army makes regular raids inspecting some path. If Army crushes all three of the camps during their raid, clan wouldn't be able to regroup and would loose the war. Help the clan to choose three cities, so that there is no path that passes through all three of these cities.

► In the graph, you need to find any three vertices that do not lie on the same path.

Let's start Kahn's algorithm for topological sort. Find two vertices that will be in the queue at the same time. In this case, there is no path where these two vertices lie. Adding any third vertex to them, we get the answer.

If, when implementing the Kahn algorithm, the queue always contains at most one vertex, then all the vertices of the graph lie on the same path.

The graphs from the test cases have the form:



In the first example, all three vertices lie on the same path. In the second example, three vertices do not lie on the same path.

E-OLYMP 4861. Sections in Makhovniki Little Peter loves computers and wants to learn programming. In the small town of Makhovniki, where he lives, there is a network of programming sections on a wide variety of subjects. When Peter went to sign up, he saw a large list of *n* sections. Peter wants to be a comprehensively developed person, so he was going to learn in all these sections. But when he was going to sign up for all the classes at once, it turned out that everything was not so simple. First of all, at one time it is allowed to study only in one of these *n* sections. Secondly, some teachers put forward input requirements for the knowledge of students, consisting in the knowledge of the courses of some other sections!

Peter wants to become a great programmer, so such trifles do not stop him. Indeed, all he needs is just to make up the correct order for visiting sections, in order to meet all the input requirements – this is a very simple task, accessible even to a very inexperienced programmer.

Before sitting down to make up the order of visiting the sections, Peter carefully read the conditions of study and found another important point. It turns out that in order to attract schoolchildren, in all sections there is a system of encouraging pupils with candies. This means that at the end of the next round, the student is given several boxes of chocolates, more and more with each passing section. On the other hand, in each

section the number of candies in a box is different, depending on the complexity of the course. More specifically – for passing the i-th section, if this section goes in the general list with the number j, the student is given as much $n^{i-1} * j$ candies – such generous people are programmers. Peter decided to combine the useful with the pleasant – now he wants to choose such an order of visiting sections so that to get as many candies as possible, however this task is no longer within his power. Help the future great man find such an order.

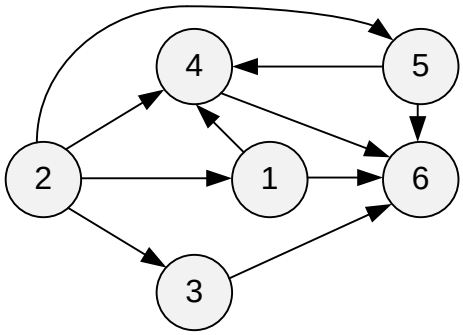
► Let $(p_0, p_1, p_2, \ldots, p_{n-1})$ be the order in which Petya will visit the sections. Then you should maximize the value

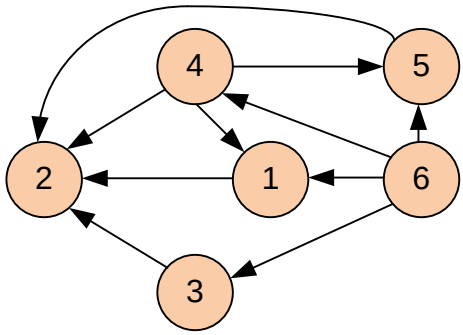
$$n^0 * p_0 + n^1 * p_1 + n^2 * p_2 + \ldots + n^{n-1} * p_{n-1}$$

Since n is fixed in the problem, the specified value will be maximum when the sequence $(p_{n-1}, \ldots p_2, p_1, p_0)$ is lexicographically the largest.

Construct a reverse graph. Find the lexicographically largest topological sort in it and print it in reverse order. The number of candies eaten for the constructed sequence will be the maximum.

The graph from the statement has the form





lexicographically biggest

6, 4, 5, 3, 1, 2

For the reverse graph, the lexicographically largest topological sort is (6, 4, 5, 3, 1, 2). The order of attending the sections will be in the reverse order: (2, 1, 3, 5, 4, 6).

E-OLYMP **9044. Lonely island** There are many islands that are connected by one-way bridges, that is, if a bridge connects islands a and b, then you can only use the bridge to go from a to b but you cannot travel back by using the same. If you are on island a, then you select (uniformly and randomly) one of the islands that are directly reachable from a through the one-way bridge and move to that island. You are stuck on an island if you cannot move any further. It is guaranteed that after leaving any island it is not possible to come back to that island.

Find the island that you are most likely to get stuck on. Two islands are considered equally likely if the absolute difference of the probabilities of ending up on them is $\leq 10^{-9}$.

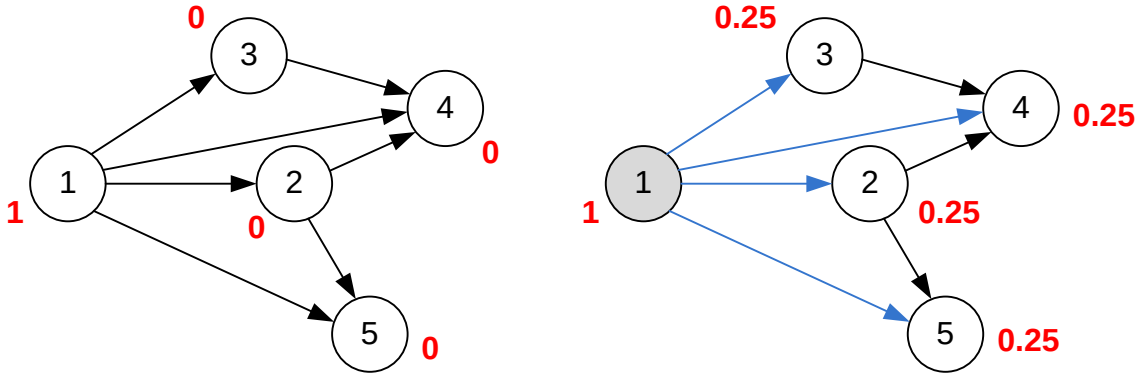
► Let's start the Kahn's algorithm for topological sort. For each vertex v, compute the number of incoming InDegree[v] and outgoing OutDegree[v] edges from it. Let us denote by prob[v] the probability to be at the vertex v. Initially, prob[r] = 1 for the starting vertex r, for other vertices prob[v] = 0.

Next, for each vertex *v*, in the order of topological sorting, iterate over the edges (*v*, *to*) and increase the value of prob[*to*] by prob[*v*] / OutDegree[*v*].

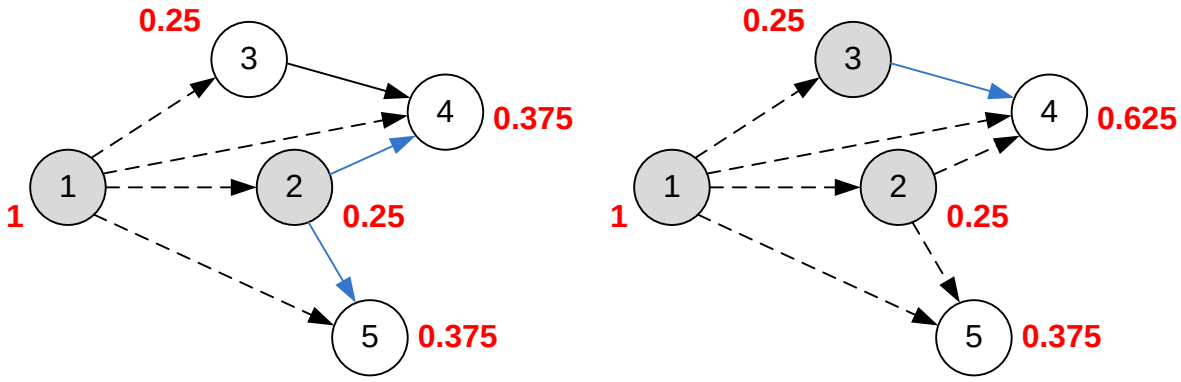
You can get stuck at the vertex *v*, for which OutDegree[*v*] = 0. Find the maximum value among prob[*v*], for which OutDegree[*v*] = 0.

Simulate Kahn's algorithm for the graph given in the problem. For each vertex of the graph we assign the probability to be there. Initially (figure on the left) the probability to be at all vertices is 0, except for the starting first, for which probability is 1.

The first vertex in topological order will be 1. Four edges comes out of it, therefore the probability prob[1] = 1 will be divided between 4 vertices: the value prob[1] / 4 = 0.25 should be added to prob[2], prob[3], prob[4] and prob[5].



Next, vertex 2 will be added to the topological order. Two edges come out of it. prob[2] / 2 = 0.125 should be added to prob[4] and prob[5] (left figure below). The next vertex will be 3. One edge comes out of it. Add prob[3] / 1 = 0.25 to prob[4] (figure on the right). Then vertices 4 and 5 will be processed, but they do not contain outgoing edges and the probabilities will not be recalculated.



A dead-end vertex (that has no outgoing edges) with the maximum probability to be there has number 4. There is only one such vertex in the graph.