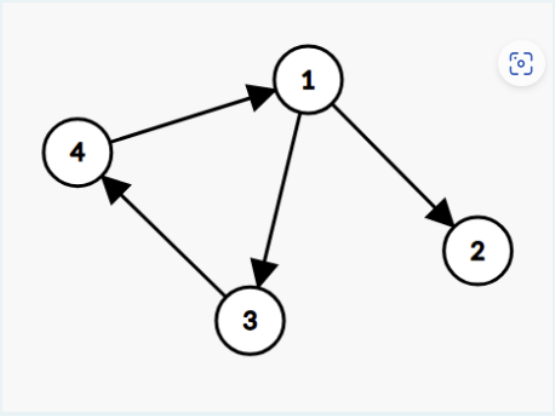
QUIZ-NACHBESPRECHUNG



Welche *Adjazenzmatrix* gehört zum gerichteten Graph oben?

a.

0	1	1	0
0	0	0	0
0	0	0	1
1	0	0	0

b.

0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0

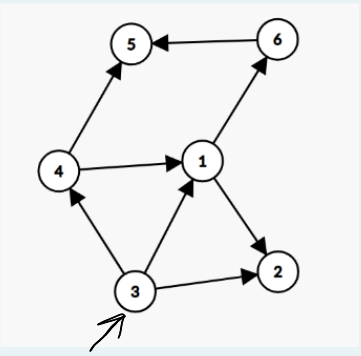
c.

1	1	1	0
0	1	0	0
0	0	1	1
1	0	0	1

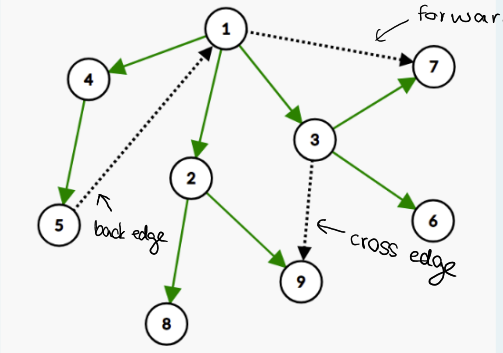
Wahr oder falsch: Ein längster (gerichteter) Pfad in einem gerichteten Graph ohne gerichtete Zyklen muss in einer Senke enden.

Bitte wählen Sie eine Antwort:

- Wahr
- Falsch



Welcher Knoten kommt in einer *topologischen Sortierung* des obigen Graphen zuerst?



Im obigen Graph zeigen die grünen, durchgezogenen Kanten einen *Tiefensuchbaum* an. Klassifizieren Sie die restlichen Kanten. (*Hinweis: Jede Option sollte **genau einmal** verwendet werden.*)

$(3, 9)$ Auswählen ...

$(1, 7)$ Auswählen ...

$(5, 1)$ Auswählen ...

Wir führen eine *Tiefensuche (DFS)* auf einem gerichteten Graph $G = (V, E)$ aus und bestimmen alle *Pre-/Post-Zahlen*.

Wir stellen fest, dass es eine Kante $(u, v) \in E$ gibt mit

$$\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v).$$

Was können wir über G schliessen?

(Erinnern Sie sich daran, dass die Pre-/Post-Zahl eines Knotens den Zeitpunkt angibt, wann er von der Tiefensuche zum ersten/letzten Mal besucht wird.)

Wählen Sie eine Antwort:

- a. G hat einen gerichteten Zyklus.
- b. G hat keinen gerichteten Zyklus.
- c. Keines der beiden.

SERIE 7 - COMMON MISTAKES

- DP[i][0] = 1 DP[0][j] = 0
  → was ist DP[0][0]?

- Rekursionen begründen

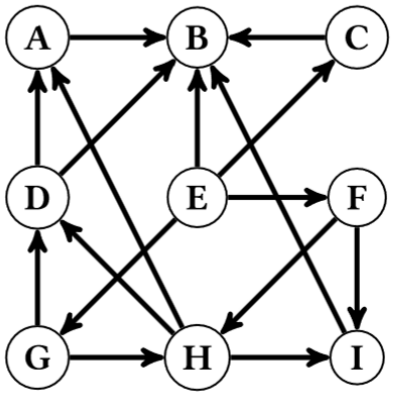
TOPOLOGICAL SORTING

Idee: Graph nach 'Abhängigkeiten' sortieren, z.B. bei Task-Graphen

Ansatz:
- Finde Senke v
- Setze v an letzte Stelle
- Entferne v und löse rekursiv

Aufgabe

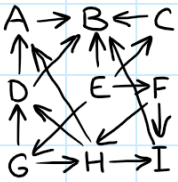
(c) The following graph has a topological sorting. If so, give a topological sorting; if not, prove why no topological sorting can exist.



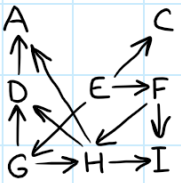
EGCFHIDAB

c)

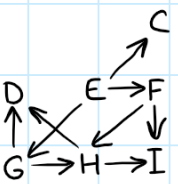
We compute one possible topological order as following:

1) 

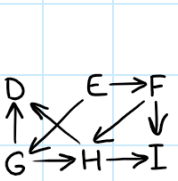
topological order:

2) 

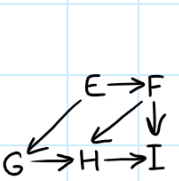
topological order:

3) 

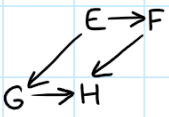
topological order:

4) 

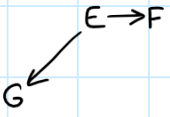
topological order:

5) 

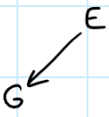
topological order:

6) 

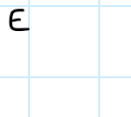
topological order:
IDCAB

7) 

topological order:
HIDCAB

8) 

topological order:
FHIDCAB

9) 

topological order:
GFHIDCAB

10) topological order:
EGFHIDCAB

B

AB

CAB

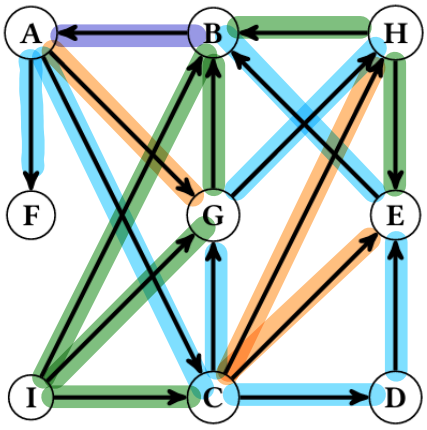
DCAB

∃ topologische Sortierung ⟺ ∄ gerichteter Zyklus

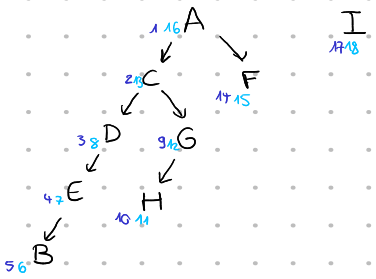
DFS

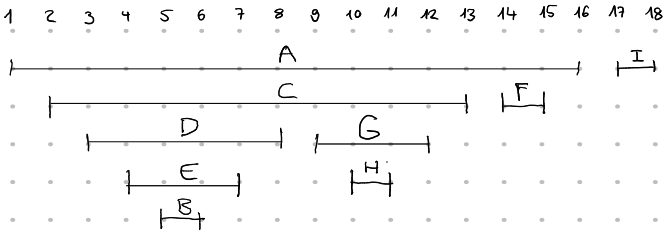
Idee: Laufe so lange einem Pfad entlang bis du an einem Knoten ankommst, den du schon gesehen hast, dann laufe zurück bis du weitere „Abzweigung" findest

```
private void DFS(int x) {
  visited[x] = true;
  for (int i = 0; i < degree[x]; i++) {
    if (!visited[edges[x][i]]) {
      DFS(edges[x][i]);
    }
  }
}
```



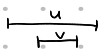
Aufgabe: Zeichne DFS-Baum startend bei A und nummeriere mit Pre- und Postnummern

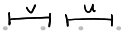




tree edges: Kanten im DFS-Baum

back edges: $e=(u,v)$  → ∃ Zyklus

forward edges: $e=(u,v)$  → ∃ zwei unterschiedliche Pfade von u nach v

cross edges: $e=(u,v)$ 

im ungerichteten Graphen: back edge = forward edge

keine cross edges → wieso?

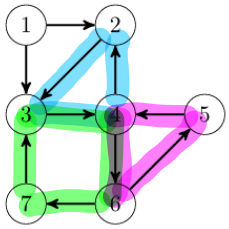
topologische Sortierung $\hat{=}$ umgekehrter Postorder (wenn es keine back edges gibt)

Anwendungsbeispiele:

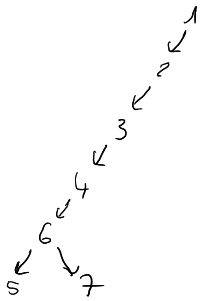
- Erreichbarkeitsprobleme / Zusammenhangskomponenten → u von v erreichbar? → DFS(v) visited[u] = true
- Zyklen prüfen
- topologische Sortierung

Prüfungsaufgabe HS23

/ 2 P d) *Depth-first search:* Consider the following directed graph:



i) Draw the depth-first tree resulting from a depth-first search starting from vertex 1. Process the neighbors of a vertex in increasing order.



ii) Write out two edges e_1, e_2 such that the directed graph above has a topological ordering after removing e_1 and e_2 (the vertex set does not change).

Remark: There could be multiple valid solutions. In this case, you only need to write down one of them.

Kanten wählen, die in zwei Zyklen sind → (4,2) und (4,6)

Aufgabe (Bonus HS23)

Exercise 9.4 *Number of paths in DAGs* **(1 point)**.

Let $G = (V, E)$ be a directed graph without directed cycles[1] (i.e., a directed acyclic graph or short DAG). Assume that $V = \{v_1, \ldots, v_n\}$ (for $n = |V| \in \mathbb{N}$). Further assume that v_1 is a source and v_n is a sink. The goal of this exercise is to find the number of paths from v_1 to v_n.

(a) Prove that there exists a topological sorting of G that has v_1 as first and v_n as last vertex.

Using part (a), we assume from now on that the sorting $v_1, v_2, \ldots, v_n$ of the vertices is a topological sorting. We can achieve this by renaming the vertices. Part (a) tells us then that we do not need to rename v_1 and v_n.

Annahme: für alle gerichteten v_1-v_n-Pfade P: $v_1 = v_{i_0}, v_{i_1}, \ldots, v_{i_\ell} = v_n$ gilt $i_0 < i_1 < \ldots < i_\ell$.

(c) Describe a bottom-up dynamic programming algorithm that, given a graph G with the property that $v_1, \ldots, v_n$ is a topological sorting, returns the number of v_1-v_n paths in G in $O(|V| + |E|)$ time. You can assume that the graph is provided to you as a pair (n, Adj) of the integer $n = |V|$ and the adjacency lists Adj. Your algorithm can access $Adj[u]$, which is a list of vertices to which u has a direct edge, in constant time. Formally, $Adj[u] := \{v \in V \mid (u, v) \in E\}$.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Hint: *Define the entry of the DP table as $DP[i]$ = number of paths in G from v_i to v_n.*

a) v_1, v_n entfernen → topologische Sortierung im restlichen Graphen finden
→ v_1 und v_n anzufügen

c) 1) DP[1...n]

2) s. Hint

3) Base Case: DP[n] = 1

Rekursion: $DP[i] = \sum_{\substack{(v_i, v_j) \in E \\ \text{mit } j \in \{i+1, \ldots, n\}}} DP[j]$ + Begründung

4) for i=n...1

5) DP[1]

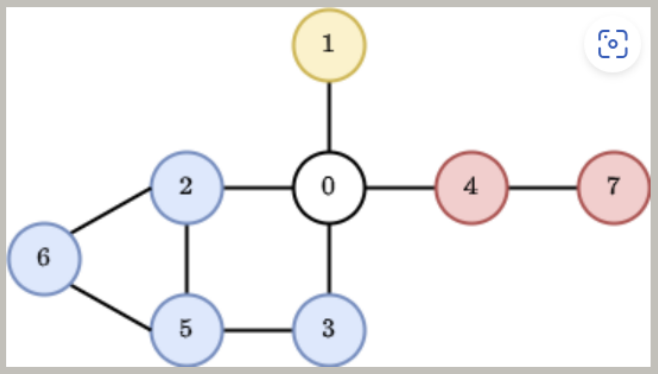
6) O(|V| + |E|)

Graph Sets

You are given an undirected connected graph with n nodes numbered from 0 to $n-1$ and m edges between them.

The nodes of the graph are partitioned into sets in the following way. Node 0 forms a set of its own. Suppose that node 0 is removed from the graph. Then, the nodes in each connected subgraph of the resulting graph forms another set.

An example is shown below, in which the nodes are colored according to the set they are in. Specifically, the sets in this example are $\{0\}$, $\{1\}$, $\{2,3,5,6\}$, and $\{4,7\}$.



Given such a graph, you have to answer queries of the following type:

1. **hasCycle()**: Return 1 if the graph has a cycle, and 0 otherwise.
2. **hasCycleWithoutNodeZero()**: Suppose that node 0 is removed from the graph. Return 1 if the resulting graph has a cycle, and 0 otherwise.
3. **isSameSet(x, y)**: Given two nodes x and y, return 1 if they are in the same set, and 0 otherwise.
4. **getShortestPath(x, y)**: Given two nodes x and y **that are in different sets**, return the shortest-path distance between them.

You have to implement the four methods described above. In addition, we provide an **intialize()** method which we guarantee to run before any of the queries. Feel free to use this method, for example, to initialize information that you want available in all of the queries (of course, the time consumed by **intialize()** counts toward the time limit of the problem).

For an easier implementation of the queries, **recall and make use of the fact** that the graph is connected.

Grading (24 points):

1. **hasCycle()** (4 points): This query will be run at most once. An $O(n+m)$-time implementation gets 4 points.
2. **hasCycleWithoutNodeZero()** (4 points): This query will be run at most once. An $O(n+m)$-time implementation gets 4 points.
3. **isSameSet(x, y)** (8 points): If q is the number of queries of this type, an $O(n+m+q)$-time implementation gets 8 points and an $O(q(n+m))$-time implementation gets 4 points.
4. **getShortestPath(x, y)** (8 points): If q is the number of queries of this type, an $O(n+m+q)$-time implementation gets 8 points and an $O(q(n+m))$-time implementation gets 4 points.

Meine Lösung (geht bestimmt auch effizienter),
Aufgabe 4 benötigt BFS (nächste Woche)

```java
import java.util.Arrays;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;


class Main {
  public static void main(String[] args) {

    int tests = In.readInt(); // number of tests
    for (int t = 0; t < tests; t++) {
      int n = In.readInt(); // number of nodes
      int m = In.readInt(); // number of edges
      int q = In.readInt(); // number of queries

      int[][] edgeArray = new int[m][2]; // array of edges
      for (int i = 0; i < m; i++) {
        edgeArray[i][0] = In.readInt();
        edgeArray[i][1] = In.readInt();
      }

      Graph G = new Graph(n, m, edgeArray); // graph
      G.initialize();

      // queries
      for (int i = 0; i < q; i++) {
        int type = In.readInt();
        if (type == 1) {
          // hasCycle
          Out.println(G.hasCycle());
        } else if (type == 2) {
          // hasCycleWithoutNodeZero
          Out.println(G.hasCycleWithoutNodeZero());
        } else if (type == 3) {
          // isSameSet
          int x = In.readInt();
          int y = In.readInt();
          Out.println(G.isSameSet(x, y));
        } else if (type == 4) {
          // getShortestPath
          int x = In.readInt();
          int y = In.readInt();
          Out.println(G.getShortestPath(x, y));
        }
      }
    }
  }
}
```

```java
class Graph {
  private int n;                // number of nodes
  private int m;                // number of edges
  private int[] degree;         // degrees[i]: the degree of vertex i
  private int[][] edges;        // edges[i][j]: the endpoint of the j-th edge of vertex i
  private boolean[] visited;    // visited[i]: whether node i has been visited
  int cycle_without_zero;
  int component;
  int[] components;
  int[] d;

  Graph(int n, int m, int[][] edgeArray){
    this.n = n;
    this.m = m;
    degree = new int[n];
    edges = new int[n][];
    visited = new boolean[n];

    for (int i = 0; i < m; i++) {
      degree[edgeArray[i][0]]++;
      degree[edgeArray[i][1]]++;
    }
    for (int i = 0; i < n; i++) {
      edges[i] = new int[degree[i]];
      degree[i] = 0;
    }
    for (int i = 0; i < m; i++) {
      edges[edgeArray[i][0]][degree[edgeArray[i][0]]++] = edgeArray[i][1];
      edges[edgeArray[i][1]][degree[edgeArray[i][1]]++] = edgeArray[i][0];
    }
  }
```

```java
  public void initialize() {

    cycle_without_zero = 0;
    components = new int[n];
    component = 0;
    for(int i=1; i<n; i++){
      if(components[i] == 0){
        component++;
        components[i] = component;
        DFS_without_0(i);
      }
    }
    Queue<Integer> q = new LinkedList<Integer>();
    q.add(0);
    d = new int[n];
    while(!q.isEmpty()){
      int u = q.poll();
      for(int i = 0; i<edges[u].length; i++){
        if(d[edges[u][i]] == 0){
          d[edges[u][i]]= d[u]+1;
          q.add(edges[u][i]);
        }
      }
    }
  }
```

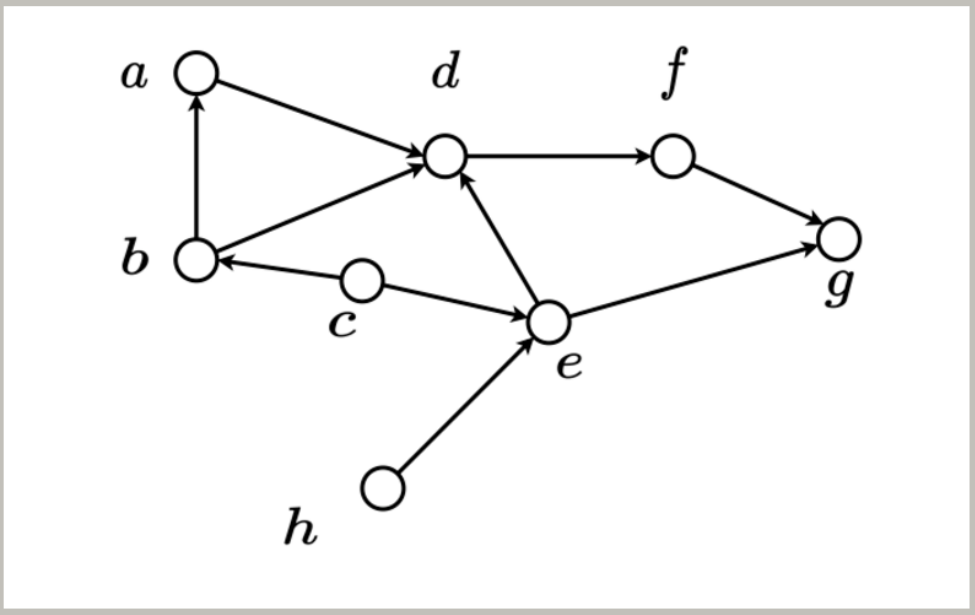
```java
  public int hasCycleWithoutNodeZero() {
    if(m -degree[0] > n-component-1) return 1;
    return 0;
  }

  public int isSameSet(int x, int y) {
    if (components[x] == components[y]) return 1;
    return 0;
  }

  public int getShortestPath(int x, int y) {
    return d[x]+d[y];
  }

  private void DFS_without_0(int x) {
    for (int i = 0; i < degree[x]; i++) {
      if(edges[x][i] != 0){
        if (components[edges[x][i]] == 0) {
          components[edges[x][i]] = component;
          DFS_without_0(edges[x][i]);
        } else {
          cycle_without_zero = 1;
        }
      }
    }
  }

}
```

Andere DFS Aufgabe (Bonus von HS22)

In Between

You are given a directed graph $G = (V, E)$ as an adjacency list and two vertices $x, y \in V$. Here we say that a vertex z is *between* x and y if there exist paths in G such that (1) you can reach z from x and (2) you can reach y from z. Your goal is to count the number of vertices between x and y.

For example, consider the following graph.



Between nodes h and g there are 5 vertices $\{h, e, d, f, g\}$. Note that the orders of the given vertices matters: between g and h there would be 0 vertices.

You need to implement your solution as a method **countInBetween(G, x, y)** in the file Main.java. You get one point for each passing test. To pass both test sets, your solution needs to run in time $O(|V| + |E|)$.

Attention: You are **NOT** allowed to use additional imports, other than the imports already included in the code template.

Lösungsidee:

1. DFS(x)
2. neue Adjazenzliste mit allen Kanten umgekehrt
3. DFS(y) mit neuer Adjazenzliste
4. alle Knoten die sowohl in DFS(x) als auch in DFS(y) erreicht wurden, befinden sich zwischen x und y.