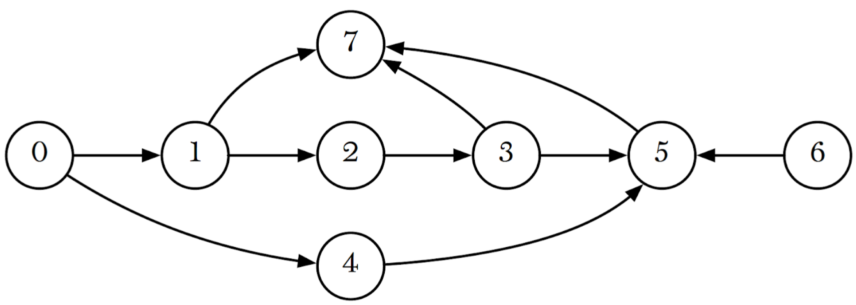
EXERCISE 1: Shortest Common Ancestor

In a directed graph, a vertex x is an ancestor of v if there exists a (directed) path from v to x. Given two vertices v and w in a rooted directed acyclic graph (DAG), a shortest common ancestor `sca(v, w)` is a vertex x which:

- is an ancestor to both v and w;
- minimizes the sum of the distances from v to x and w to x (this path, which goes from v to x to w, is the shortest ancestral path between v and w).

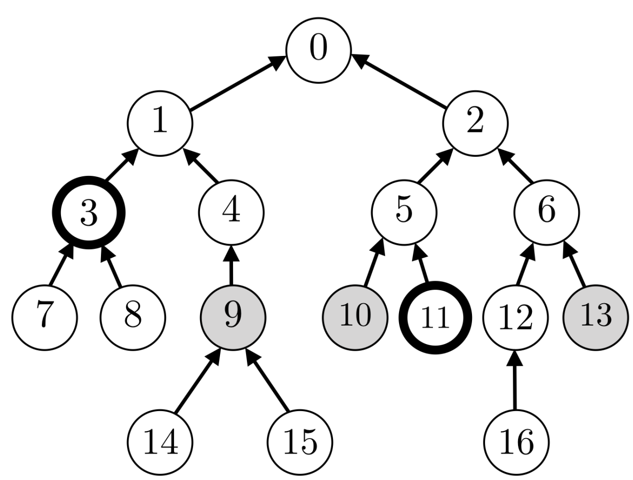
A. In the following digraph, find the shortest common ancestor of vertices `1` and `4`, and give the sum of the path lengths from these vertices to all common ancestors, and then circle the shortest.



B. Describe an algorithm for calculating the shortest common ancestor of two vertices v and w. Your algorithm should run in linear time (proportional to $V + E$).

C. How would your algorithm differ if we are interested in the shortest ancestral path between two ***sets*** of vertices A and B instead of two vertices? I.e. between any vertex v in A and any vertex w in B.

In the example, $A = 3, 11$ and $B = 9, 10, 13$. The shortest common ancestor is 5 (between 10 and 11).



EXERCISE 2: Cycle Detection Using BFS

Note. You can also use the online version of this exercise, which allows testing your code and receiving instant feedback:

https://stepik.org/lesson/217879

The online version also has an ***extra exercise*** for the bored!

Consider the following Breadth-First Search code. What modifications should be made in order for the `hasCycle()` method to return `true` if the graph has a simple cycle and `false` otherwise? Assume that the graph is *connected*, *undirected* and does not have parallel edges or self-loops.

Def. A *cycle* is a path with at least one edge whose first and last vertices are the same. A *simple cycle* is a cycle with no repeated edges or vertices (except the requisite repetition of the first and last vertices).

```
private static boolean hasCycle(Graph G) {

      boolean[] marked = new boolean[G.V()];
      int[] edgeTo = new int[G.V()];

      Queue<Integer> q = new Queue<Integer>();
      marked[0] = true;
      q.enqueue(0);

      while (!q.isEmpty()) {
            int v = q.dequeue();          // v is the current node
            for (int w : G.adj(v)) {      // for every neighbor w of v
                  if (!marked[w]) {
                        edgeTo[w] = v;
                        marked[w] = true;
                        q.enqueue(w);
                  }
            }
      }
}
```

B. What is the order of growth of the running time of this algorithm (as a function of V and E) in the *best case*? What is the order of growth in the *worst case*?

EXERCISE 3: Detecting Directed Cycles

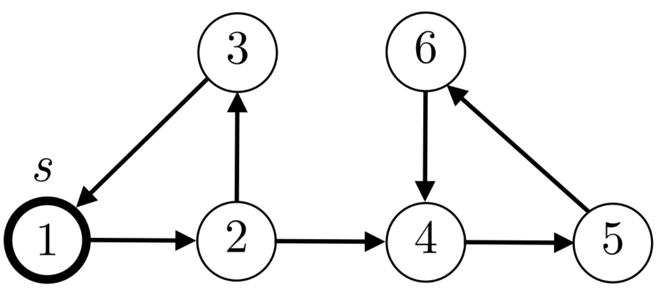
An online version of this exercise is available at: https://stepik.org/lesson/219467

A. Consider the graph G given below and the marked vertex s. Show in the given box what the output would be if `depthFirstSearch` is called on G and s.

```
private boolean[] marked;

public void depthFirstSearch(Digraph G, int s) {
      marked = new boolean[G.V()];
      dfs(G, s);
}

private void dfs(Digraph G, int v) {
      marked[v] = true;
      StdOut.println("Starting " v);
      for (int w : G.adj(v)) {
            if (!marked[w])
                  dfs(G, w);
      }
      StdOut.println("Finished " + v);
}
```



B. Consider the following modified version of the `dfs` method. Explain with the simplest counterexample why this code is not a correct cycle detection code.

```
private void dfs(Digraph G, int v) {
      marked[v] = true;

      for (int w : G.adj(v)) {
            if (!marked[w])
                  dfs(G, w);
            else StdOut.print("Cycle found!");
      }
}
```

C. Briefly describe how depth-first search could be modified to detect cycles in a digraph.

D. Fill the blank lines in the following DFS code so that it prints "`Cycle found!`" if and only if there is a cycle in the graph. Assume that the graph is connected.

```
private boolean[] marked;
private boolean[] onStack;

public void checkCycles(Digraph G, int s) {
      marked = new boolean[G.V()];
      __________________________________________
      dfs(G, s);
}

private void dfs(Graph G, int v) {
      marked[v] = true;
      __________________________________________
      for (int w : G.adj(v)) {
            if (!marked[w])
                  dfs(G, w);
            else if (_______________________)
                  StdOut.print("Cycle found!");
      }
      __________________________________________
}
```