Online (auto-graded) version of this precept: https://stepik.org/lesson/217879

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

EXERCISE 1: Cycle Detection Using BFS

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).

```java
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);
                  }
            }
      }
}
```

EXERCISE 2: Cycle Detection Using DFS

Consider the following Depth-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.

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

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

    for (int i = 0; i < G.V(); i++)
        edgeTo[i] = -1;

    return hasCycle(G, marked, edgeTo, 0);
}

private static boolean hasCycle(Graph G, boolean [] marked, int [] edgeTo, int v) {

    marked[v] = true;

    for (int w : G.adj(v)) {
        if (!marked[w]) {
            edgeTo[w] = v;
            hasCycle(G, marked, edgeTo, w));
        }
    }
}
```

EXERCISE 3: Running Time Analysis

A. What is the order of growth of the running time of the DFS and BFS algorithms for cycle detection (as a function of V and E) in the *best case*? What is the order of growth in the *worst case*?

B. Re-implement `hasCycle()` such that the running time is *constant* in V and E.

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



}
```