1 Multiple MSTs

Recall a graph can have multiple MSTs if there are multiple spanning trees of minimum weight.

(a) For each subpart below, select the correct option and justify your answer. If you select "never" or "always," provide a short explanation. If you select "sometimes", provide two graphs that fulfill the given properties — one with multiple MSTs and one without. Assume G is an undirected, connected graph with at least 3 vertices.

1. If **some** of the edge weights are **identical**, there will

   ◯ never be multiple MSTs in G.

   ◯ sometimes be multiple MSTs in G.

   ◯ always be multiple MSTs in G.

   Justification:

2. If **all** of the edge weights are **identical**, there will

   ◯ never be multiple MSTs in G.

   ◯ sometimes be multiple MSTs in G.

   ◯ always be multiple MSTs in G.

   Justification:

(b) Suppose we have a connected, undirected graph G with N vertices and N edges, where all the **edge weights are identical**. Find the maximum and minimum number of MSTs in G and explain your reasoning.

Minimum: ________
Maximum: ________

Justification:

(c) It is possible that Prim's and Kruskal's find **different** MSTs on the same graph `G` (as an added exercise, construct a graph where this is the case!). Given any graph `G` with integer edge weights, modify the edge weights of G to **ensure** that (1) Prim's and Kruskal's will output the same results, and (2) the output edges still form a MST correctly in the original graph. You may not modify Prim's or Kruskal's, and you may not add or remove any nodes/edges.
Hint: Look at subpart 1 of part a.

2 Topological Sorting for Cats

The big brain cat, Duncan, is currently studying topological sorts! However, he has a variety of **curiosities** that he wishes to satisfy.

(a) Describe at a high level in plain English how to perform a topological sort using an algorithm we already know (hint: it involves DFS), and provide the time complexity.

(b) Duncan came up with another way to possibly do topological sorts, and he wants you to check him on its correctness and tell him if it is more efficient than our current way! Let's derive the algorithm.

1. First, provide a logical reasoning for the following claim (or a proof!): Every DAG has at least one source node, and at least one sink node.

2. Duncan wishes to extend from the Graph class to create a DAG class. He wants to eventually add a method that enables topological sorting, but needs to write some helper methods first! Complete the following instance methods `computeInDegrees` and `findAllSourceNodes()`.

```
public class Graph {
    public Graph(int V) // Create empty graph with v vertices, numbered 0 to V - 1
    public void addEdge(int v, int w) // Adds edge from v to w
    Iterable<Integer> adj(int v) // Gets vertices adjacent to v
    int V() // Number of vertices
    int E() // Number of edges
}

public class DAG extends Graph {
    // Computes the number of incoming edges to a vertex
    public int[] computeInDegrees() {
        int[] indegree = ______________________;

        for (_________________________________) {

            for (__________________________) {

                ______________________________

            }
        }

        return indegree;
    }

    // Finds all source nodes in the graph
    public List<Integer> findAllSourceNodes(int[] indegree) {
        List<Integer> sources = new ArrayList<>();

        for (_________________________________) {

            if (__________________________) {

                __________________________

            }
        }

        return ___________________;
    }
}
```

Runtime of `computeInDegrees` is:

Runtime of `findAllSourceNodes` is:

3. Now, make the following observation: If we remove all of the source nodes from a DAG, we are guaranteed to have at least one new source node. Inspired by this fact, and using the previous parts, complete the `topologicalSort()` method. What is its runtime?

```
public class DAG extends Graph {
    public int[] computeInDegrees() { ... }
    public List<Integer> findAllSourceNodes(int[] indegree) { ... }

    public List<Integer> topologicalSort() {
        List<Integer> sorted = new ArrayList<>();

        ____________________________________________

        // Hint: add elements from another iterable here
        Queue<Integer> sources = new ArrayDeque<>(_______________________);

        while (________________________________) {

            int source = sources.poll();

            ______________________________________

            for (______________________________________) {

                ______________________________________

                if (________________________) {

                    ________________________
                }
            }
        }
        return sorted;
    }
}
```

Runtime of `topologicalSort` is:

4. Venti, the bard from Mondstadt is allergic to cats. He wanted to trick Duncan and created a DAG object, but it actually represents a graph with a cycle! How can you modify the method `topologicalSort()` above to detect whether the graph has a cycle?

3 A Wordsearch

Given an N by N wordsearch and N words, devise an algorithm (using pseudocode or describe it in plain English) to solve the wordsearch in $O(N^3)$. For simplicity, assume no word is contained within another, i.e. if the word "bear" is given, "be" wouldn't also be given.

If you are unfamiliar with wordsearches or want to gain some wordsearch solving intuition, see below for an example wordsearch. Note that the below wordsearch doesn't follow the precise specification of an N by N wordsearch with N words, but your algorithm should work on this wordsearch regardless.

Example Wordsearch:

C	M	U	H	O	S	A	E	D
T	R	A	T	H	A	N	K	A
O	C	Y	E	S	R	T	U	T
N	I	R	S	A	I	O	L	S
Y	R	R	M	T	N	N	H	R
Y	E	A	E	V	A	R	U	E
A	A	A	I	M	E	L	C	R
N	H	D	J	Y	U	A	C	I
T	Y	S	A	A	R	S	U	C
A	R	S	I	G	Y	E	S	A

ajay	anton
crystal	eric
grace	isha
luke	naama
rica	sarina
sherry	shreyas
sohum	sumer
tony	vidya

Hint: Add the words to a Trie, and you may find the `longestPrefixOf` operation helpful. Recall that `longestPrefixOf` accepts a `String key` and returns the longest prefix of `key` that exists in the `Trie`, or **`null`** if no prefix exists.