built on 2024/11/29 at 09:20:50 ***due:*** *tue apr 02 @ 11:59pm*

This assignment contains both programming and written questions. For the written portion, you will typeset your answer, produce a PDF file called `hw8.pdf`, and upload it to Canvas. No other format will be accepted.

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Task 1: Breadth-First Search Running Time (2 points)

Our breadth-first (BFS) implementation from class uses `HashSet`s and `HashMap`s to keep track of progress. In this task, you'll reanalyze the running time of BFS if we use `TreeSet`s and `TreeMap`s instead. Show your work carefully. To keep things simple, we'll work with the BFS code shown above. It only reports vertices reachable from a source s. It has been rewritten from the version presented in class to make the operations more explicit.

```
Set<Integer> nbrsExcluding(
    UndirectedGraph<Integer> G,
    Set<Integer> vtxes,
    Set<Integer> excl
  ) {
    Set<Integer> union = new TreeSet<>(); // not HashMap
    for (Integer src : vtxes) {
        for (Integer dst : G.adj(src))
          if (!excl.contains(dst)) { union.add(dst); }
    }
    return union;
}
Set<Integer> bfs(UndirectedGraph<Integer> G, int s) {
    Set<Integer> frontier = new TreeSet<>(Arrays.asList(s));
    Set<Integer> visited = new TreeSet<>(Arrays.asList(s));

    while (!frontier.isEmpty()) {
        frontier = nbrsExcluding(G, frontier, visited);
        visited.addAll(frontier); // the i-th position is what's reached at i hops
    }

    return visited;
}
```

You should know that the `TreeSet` implementation uses a balanced binary search tree (BST), so `add`, `contains`, and `remove`, unlike in a `HashSet`, take $O(\log S)$ per operation, where S is the size of the collection. Though `addAll` may perform other optimizations, for the purpose of this task, assume that `addAll(X)` simply repeatedly calls `add` on each element of X.

You should also assume that `UndirectedGraph G` is implemented as an adjacency table using `HashMaps`, so `G.adj` takes $O(1)$.

Task 2: Random Permutations (2 points)

Remember that $[n]$ is the set $\{1,2,\ldots,n\}$. Let $p:[n]\to[n]$ be a random permutation on $[n]$ chosen uniformly among the $n!$ permutations. Consider the following code:

```
int minSoFar = Integer.MAX_VALUE;
int numUpdate = 0;
for (int i=1;i<=n;i++) {
    if (p(i) < minSoFar) {
        minSoFar = p(i);
        numUpdate++;
    }
}
```

We'll see that `numUpdate` is a random variable that depends on the permutation randomly chosen. Prove that at the end of the **for**-loop,

$$\ln(n+1) \le \mathbf{E}[\texttt{numUpdate}] \le 1+\ln n.$$

Task 3: HackerRank Problems (14 points)

Write your HackerRank ID in your PDF writeup.

There are *seven* problems in this set. You **must** write your solutions in Java (1.8). You will hand them in electronically on the HackerRank website. If you're planning on using a late token, please finish these HackerRank problems before the due date. The HackerRank "contest" ends at 11:59pm on the day the assignment is due.

Read Me! Many of the problems here are best solved using BFS. Once you have a good working implementation of BFS, it is just a matter of crafting the right graph to run it on. Pro tips: very often, you don't need to explicitly construct a graph—it is enough to be able to return the neighbors of a vertex on the fly.

You can find your problems at

`https://www.hackerrank.com/muic-dsoop-t-124-assignment-8`