Problem Set 4

This problem set is due **Wednesday, April 4** at **11:59PM**.

Solutions should be turned in through the course website. **You must enter your solutions by modifying the solution template (in Python) which is also available on the course website.** The grading for this problem set will be largely automated, so it is important that you follow the specific directions for answering each question.

For multiple-choice and true/false questions, no explanations are necessary: your grade will be based only on the correctness of your answer. For all other non-programming questions, full credit will be given only to correct solutions which are described clearly and concisely.

Programming questions will be graded on a collection of test cases. Your grade will be based on the number of test cases for which your algorithm outputs a correct answer within time and space bounds which we will impose for the case. Please do not attempt to trick the grading software or otherwise circumvent the assigned task.

---

1. **An assortment of sorts (10 points)**

   (a) Merge sort on n integers in the range $\{1, \ldots, n^3\}$ requires time $\Theta(n^c \log n)$.
   What is c?

   (b) Counting sort on n integers in the range $\{1, \ldots, n^3\}$ requires time $\Theta(n^c)$.
   What is c?

   (c) Radix sort on n integers in the range $\{1, \ldots, n^3\}$ (with optimal choice of parameters) requires time $\Theta(n^c)$.
   What is c?

2. **Median of two arrays (20 points)**

   Let X and Y be two arrays, each containing n ordered values already in sorted order. Give the most efficient algorithm you can to find the median of all $2n$ elements in arrays X and Y. Prove correctness of your algorithm and analyze its running time.

3. **Cycle testing (20 points)**

   Design and analyze an algorithm for detecting if an undirected graph has an odd cycle. (A cycle of length k is a sequence of k distinct vertices $v_1, \ldots, v_k$ such that there are edges between v_1 and v_2, between v_2 and v_3, etc, and also an edge between v_k and v_1.)

4. **BFS or DFS? (10 points)**

   For each of the following problems, answer `'B'` if the most appropriate search algorithm is BFS, or `'D'` if the most appropriate search algorithm is DFS

   (a) You are a mouse who is trapped in a maze with no cycles. You have no memory, but you know left from right. Your escape strategy is closest to which search algorithm?

   (b) You are a pirate looking for hidden treasure on an island. You are at the location marked X on the map, but the maps is slightly inaccurate, so you believe the treasure to be at a nearby location. How do you determine the order in which to search the locations on the island?

   (c) You are Google Maps. Which search algorithm do you use to get driving directions?

   (d) Which search algorithm explores a graph in a manner reminiscent to a BST in-order traversal?

   (e) Which search algorithm is good at keeping track of shortest distances from the start node?

5. **True/False (30 points)**

   (a) Let G be an undirected graph. If we have a back edge when we run DFS on G, then the graph has a cycle.

   (b) Let G be a directed graph. If we have a cross edge when we run DFS on G, then the graph has a directed cycle.

   (c) The running time of insertion sort can be reduced to $O(n \cdot \log(n))$ if we use binary search when inserting each element into its appropriate position of the array instead of traversing the array backwards.

   (d) The running time of BFS is $O(V + E)$ irrespective of the graph representation.

   (e) Let G be a connected undirected graph, let v be a vertex in G, and let D be a directed graph obtained by orienting the edges of G arbitrarily. Then it is always the case that a DFS in D starting from v will explore the entire graph.[1]

   (f) If an undirected graph has vertices v_1, v_2, and v_3 in a triangle, then when performing BFS, AT LEAST two of v_1, v_2, and v_3 must be at the same level.

   (g) If an undirected graph has vertices v_1, v_2, and v_3 in a triangle, then when performing BFS, EXACTLY two of v_1, v_2, and v_3 must be at the same level.

   (h) If an undirected graph has vertices v_1, v_2, and v_3 in a triangle, then when performing DFS, no two of them can be on the same level. (We define "level" as the length of the path taken from the source in the DFS tree.)

   (i) Suppose that in the "Awkward Sort of Party" problem from Problem Set 2, each of the n people are assigned a vertex in a directed graph G. DFS is run on the graph, and a person arrives at the party when his vertex is first explored by the search, and leaves the party when his vertex is finished processing ("colored black" in the terminology of CLRS). True or False: At the conclusion of the party, no one will become a Twitter follower of anyone else.

   (j) A strongly connected component in a directed graph G is a maximal subset of vertices such that there is a directed path from any vertex in the set to any other vertex. True or False: Let C and D be two (distinct) strongly connected components of a directed graph G, and suppose that there is a directed edge from some vertex in C to some vertex in D. Then any depth-first search will either explore no vertices in D or will finish processing all vertices in D before it finishes processing all vertices in C. (By "finish processing," we mean, in the notation of CLRS, that the node has been "colored black.")

[1] The DFS does not restart from other vertices when the first search finishes

6. **Breadth-First Search (30 points)**

   One way of representing a graph in Python is as a dictionary `edges` mapping node numbers to lists of adjacent node numbers. The vertex set of the graph is the set of keys of the dictionary, that is, `edges.keys()`. A key `k` has a directed edge outwards to each key in the list `edges[k]`. This representation is basically an implemention of the adjacency lists discussed in class.

   Write a function `find_distances` that takes two arguments: an dictionary, `edges`, and a list of vertices, `sources`. It should return a dictionary `dist` which records the minimum distance from ANY source to each vertex of the graph, or `None` if it is unreachable.

   Your function should pass the following test cases:

```
graph = {0: [1,3], 1: [2], 2 : [0, 3],  3: [1]}
sources = [0]
dist = find_distances(graph, sources)
dist[0] == 0
dist[1] == 1
dist[2] == 2
dist[3] == 1

graph = {1: [2],
         2: ['skip a few' ],
         'skip a few' : [99, 199],
         98: [99]
         99: [100],
         100: ['skip a few']
         198: [199]
         199: [200]
         200: [] }
sources = [1, 100]
dist = find_distances(graph, sources)
dist[1] == 0
dist[2] == 1
dist['skip a few'] == 1
dist[98] == None
dist[99] == 2
dist[100] == 0
dist[198] == None
dist[199] == 2
dist[200] == 3
```