**CS 180** Homework 6

**Problem 1**

given two permutations of the same range of numbers 1, 2 … n, array A and array B

sort array A in O(n log n) time but also move the corresponding indices in B and store the original indices in a hash map

*e.g.* A = [3, 1, 4, 2], B = [2, 3, 4, 1] → A = [1, 2, 3, 4], B = [3, 1, 2, 4], I = indices of A = [1, 3, 0, 2]

for every element i of array B

if B[i] > B[i + 1]

return (I[i], I[i+1]) as a tuple // this is the pair of inverted indices i, i+1

// else if we didn’t find a pair, we reverse A and try again

reverse A // this can be done in O(n) time and O(1) space using XOR swap

reverse B

reverse indices array

for every element i of array B

if B[i] > B[i + 1]

return (I[i], I[i+1]) as a tuple

The complexity of this algorithm would be O(n log n) because we’ll be using an efficient algorithm, like merge sort, to sort A. All these commands are in parallel so it would be a constant x n log n which is just O(n log n)

**Problem 2**

1. By the definition given, a lexicographically minimum tree has the smallest tuple (w1, … , wn-1) where the sum of all wi in the tuple is the smallest. The minimum spanning tree definition then is already proven just by the information already given in the prompt.

data structure

// variables

mst

buildMST(graph G)

start by using Kruskal’s or Prim’s algorithm to build an MST from G

findPath(graph G, u, v)

mst = buildMST(G)

run DFS or BFS from u to v to find lowest weight path between u and v

This data structure starts by building an MST from the input graph. The definition of an MST is a connected and acyclic graph with minimal total weight so we would just a BFS or DFS search from our source node to our target node to find the least weight path between u to v. There are other paths we could’ve taken from u to v but this resulting one would be the smallest in total weight.

Alternatively, another solution involves constructing an adjacency matrix for the graph G with the cells being the weight from node x to node y. We would then start from the least possible weight and find all edges of that weight. If the graph is connected from u to v, we would return, otherwise we would continue checking the heavier weighted edges.

**Problem 3**

1. To prove that a graph with distinct edge weights can only have one MST, we can use another theorem about graphs that may or may not have distinct edge weights. The theorem goes as:

Given a weighted graph G, another graph T which is an MST for G, and e = ab which is an edge of G that is not in T then:

1. weight(e) >= weight(f) for any edge in T that connects vertex a to b. This is because the MST minimizes all the edge weights; if weight(e) was less than weight(f), then T would not be an MST.
2. If weight(e) == weight(f) then we can obtain another MST for graph G by replacing e with f

We then suppose there are two MST’s for graph G, namely T and T”, and let e be the cheapest edge of G in either MST. If we apply the theorem this graph that has distinct edge weights, then the first proof would be that weight(e) > weight(f) instead of >= weight(f) because no two edges have the same weight.

If e is in T, then T” union with e would contain a cycle. We then call e” to be the edge in T” that’s not in T. So then T” union edge e and without edge e” is the same graph as T. Since weight(e) < weight(e”) then T < T” and that’s a contradiction because we assumed both T and T” to be MSTs. Therefore there can only be one MST for a graph with unique edge weights.

1. To obtain all minimum spanning trees of a graph that may or may not have distinct edge weights:

allSpanningTrees(graph G)

start by running Kruskal’s or Prim’s algorithm to find one MST of the graph

compute the total weight of the MST

generate all spanning trees of the graph

compute the total weight of all spanning trees generated and only output the ones with weight equal to MST

Kruskal’s algorithm runs in O(E log V) and Prim’s runs in O(E + V log V). Filtering out the resulting spanning trees should run in linear O(n) time, n being the number of spanning trees. The most computationally heavy operation here is computing all spanning trees which can be done in O(E log(V) + V + n) with n being the number of trees generated.

**Problem 4**

We can prove this O(n log n) lower bound on computing a convex hull by using the graph of y = x2 as an example. This graph is a representation of the points xi, .. ,xn being embedded onto a graph through the operation xi2. Since a parabola is a convex hull, this is a valid example. We then note that the points appearing along the x-axis of the parabola correspond to the sorted order of the input numbers xi, …, xn. To iterate through this list of sorted numbers and construct the parabola would require at least O(n) time. However in other cases when the input is not given in sorted order, we would have to first sort them. Therefore, there is a lower bound on computing a convex hull depends on the complexity of sorting the points which is O(n log n) at best. Therefore every algorithm for computing the convex hull of a set of n points needs at least Ω(n log n) time.