15-451 Algorithms, Fall 2004

Homework # 4 **due: Tue-Wed, October 26-27, 2004**

Groundrules:

- This is an oral presentation assignment. As stated in the course information handout, you should work in groups of three. At some point before **Sunday October 24 at midnight** you should sign up for a 1-hour time slot on the signup sheet on the course web page.
- You are not required to hand anything in at your presentation, but you may if you choose. If you do hand something in, it will be taken into consideration (in a non-negative way) in the grading.

Problems:

1. **BSTs and dynamic programming.**

   Consider a binary search tree storing a set of keys $x_1 < x_2 < x_3 < \ldots < x_n$. Let's define the *cost* of handling a request for some key to be the number of comparisions made in searching for it (one plus the distance of the node from the root of the tree). For example, if the root is requested, the cost is 1.

   Given a particular sequence of requests, one can calculate the cost that would be incurred on that sequence by different possible binary search trees. The tree that attains the minimum cost is called the *optimal binary search tree* for that sequence.

   (a) For a fixed tree, the cost of a given sequence of requests clearly only depends on the number of times each key is requested, not on their order. Suppose that $n = 4$ and that x_1 is accessed once, x_2 is accessed 9 times, x_3 is accessed 5 times, and x_4 is accessed 6 times. Find an optimal binary tree for this set of requests. (There is more than one possible answer.)

   (b) In general, suppose the optimal binary search tree has x_i at the root, with L as its left subtree and R as its right subtree. Prove that L must be an optimal binary search tree for elements $x_1, \ldots, x_{i-1}$ and R must be an optimal binary search tree for elements $x_{i+1}, \ldots, x_n$.

   (c) Give a general algorithm for constructing the optimal binary tree given a sequence of counts $c_1, c_2, \ldots, c_n$ (c_i is the number of times x_i is accessed). The running time of your algorithm should be $O(n^3)$. Hint: use dynamic programming.

   Note #1: the notion of an optimal binary search tree is a lot like the notion of a Huffman tree, except that we also require the keys to be in search-tree order. This requirement is the reason that the greedy Huffman-tree algorithm doesn't work for finding optimal BSTs.

Note #2: it's actually possible to improve the running time to $O(n^2)$ by a simple modification to this dynamic-programming solution. But proving correctness for this faster version is a real pain.

2. Boruvka's MST Algorithm.

   Boruvka's MST algorithm (from 1926) is a bit like a distributed version of Kruskal. We begin by having each vertex mark the shortest edge incident to it. (For instance, if the graph were a 4-cycle with edges of lengths 1, 3, 2, and 4 around the cycle, then two vertices would mark the "1" edge and the other two vertices will mark the "2" edge.) For the sake of simplicity, assume that all edge lengths are distinct so we don't have to worry about how to resolve ties. This creates a forest F of marked edges. *(Convince yourself why there won't be any cycles!)* In the next step, each tree in F marks the shortest edge incident to it (the shortest edge having one endpoint in the tree and one endpoint not in the tree), creating a new forest F'. This process repeats until we have only one tree.

   (a) Show correctness of this algorithm by arguing that the set of edges in the current forest is always contained in the MST.
   (b) Show how you can run each iteration of the algorithm in $O(|E|)$ time with just a couple runs of Depth-First-Search and no fancy data structures. (Remember, this algorithm was from 1926!)
   (c) Prove an upper bound of $O(|E| \log |V|)$ on the running time of this algorithm.

3. Graph Searching

   Let G be a directed graph represented using an adjacency list. So, each node $G[i]$ has a list of all nodes reachable in 1 step from i (all out-neighbors of i). Suppose each node of G also has a value: e.g., node 1 might have value \$100, node 2 might have value \$50, etc.

   - Give an $O(|E| + |V| \log |V|)$ time algorithm that computes, for every node, the highest value reachable from that node (i.e., that you can get to by some path from that node). For instance, if it is possible to get to any node from any other node (G is "strongly-connected"), then for every node this will be the maximum value in the entire graph.

   Hint: one worthwhile preprocessing step is to sort nodes by value.