COP 4516 Week 3 Individual: Trees (Solution Sketches)

Which Base is it Anyway?
To interpret a number in octal, multiply the digit in the kth place from the right by 8^k. For hexadecimal, you multiply this digit by 16^k. To avoid precision errors, use integers, don't use a built in power function. If you read the input in as an integer, it's easier to go through the digits backwards, in which case, you can keep separate variables that update the current power of 8 and 16. If you read the input as a string, you can go through the digits forwards and apply Horner's rule to update the value after processing each digit. (If the current octal value is val, and we just add in the digit c, then the new octal value will be 8*val + c.) Last, if any of the individual digits are 8 or 9, overwrite the octal value calculated to be 0.

Monkey Vines
Just like the postorder example shown in the class, the key is to write a recursive function that takes in (or has access to) the input string as well as a start index and end index:

```
int solve(String s, int sI, int eI) {
…
}
```

The goal of the function would be to solve the subproblem for the string s[SI..eI]. The base case cases correspond to leaf nodes. In the recursive case, we must scan through the substring to find two subtrees (we are guaranteed to have two). Any substring that represents a subtree must have an equal number of [s and]s. So, just start a counter of open brackets and close brackets. The first time you hit a close that makes the number equal to zero is your left subtree. So let's say we call this index endLeft. Then we want to make the following two recursive calls to the left and right subtrees, respectively:

```
solve(s, sI+1, endLeft)
solve(s, endLeft+1, eI-1)
```

Note that the first and last characters of the original recursive call must be the enclosing brackets for both of those subtrees.

Then, between these trees, the larger tree is the one that must be balanced out, so if one call returns 8 and the other returns 4, that means you have to make the other tree "weigh" 8 so that the whole thing weighs 16.

Spreading News

Consider solving this problem for the root of some tree of employees. At the root, you get to choose which order to tell each of your direct subordinates. Naturally, you want them in order of which subtree will take the longest to spread the message to the shortest. For example, say that there are four subtrees, and it would take 3, 9, 8 and 8 seconds respectively for the message to completely spread through each of those subtrees. We want to tell the subtree that takes 9 seconds first, followed by one of the subtrees that takes 8 seconds, followed by the next one that takes 8 seconds, followed by the one that takes 3 seconds. This is because waiting to tell the "slowest" subtree could potentially delay the total amount of time the message takes to spread throughout the whole tree. Given the times above and the ordering given, the message actually gets through the second subtree in 1 + 9 = 10 seconds, since it took one second to tell that subordinate, 2 + 8 = 10 seconds for the third subtree since that subordinate waited 2 seconds to get the message, 3 + 8 = 11 seconds for the fourth subtree, since that subordinate waited 3 seconds to get the message, and 4 + 3 = 7, since the last subordinate waited 4 seconds to get the message. Thus, for this case, it would take a minimum of 11 seconds for the message to spread.

If we reordered these calls as say, 9, 3, 8 and 8, then the corresponding finish times would 1+9 = 10 seconds, 2 + 3 = 5 seconds, 3 + 8 = 11 seconds and 4 + 8 = 12 seconds. Thus, this ordering of telling the subordinates would not lead to the minimum answer.

Thus, in the general case do the following:

1. Recursively solve the problem for each child, storing the answer from each recursive call in an array list.

2. Sort that list in reverse order. (So, for our example, it would be 9, 8, 8, and 3.)

3. Add i to the i^{th} of the list, starting with i = 1. (So, for our example it would be 10, 10, 11 and 7.)

4. Return the maximum of the values listed in step 3.

The base case is a single node, which takes time 0.

<u>Tree Sales</u>

We first note that a single query could be very expensive if we just stored each person's sales in their node in the tree structure. Then, to answer a single query, we would have to visit each node in the subtree of the queried node, which could be up to 100,000 steps and this could happen 100,000 times. (In reality, it wouldn't be this big, but we could still get close to 50000^2 operations total over 100,000 operations.) Thus, we need an idea to speed up a query. What if we store the ***<u>sum</u>*** of ALL nodes in a subtree at the root of that sub tree? So, the old storage method might look like what's drawn on the left and we can store tree sums instead as shown on the right, to speed up our ability to answer queries:

```
          10                                          100
      /         \                                  /        \
    5            18                              35          55
   / \          /                               /  \        /
13 17          7                              13  17      37
              / \                                         /  \
           11  19                                       11  19
```

Now, we have O(1) answers to queries (as long as we can access a node in O(1) time). But…how long does it take to update all the values for the tree on the right? Basically, a single sale affects each node on the ancestral path of the node making the sale. So say the node that currently has 11 makes a sale of $20, here are the edits to both tree structures:

```
          10                                          120
      /         \                                  /        \
    5            18                              35          75
   / \          /                               /  \        /
13 17          7                              13  17      57
              / \                                         /  \
           31  19                                       31  19
```

Basically, we just had to add 20 to the tree on the right to the nodes storing 11, 37, 55 and 100, respectively. Thus, the new run time of updating the tree for a sale is O(h), where h is the height of the tree. Luckily, the height of the tree is limited to 100 in the problem specification, so each update can be performed in 100 steps and each query in a single step. This gives us a total run time no worse than 100 x 100000 (max operations) = 10^7 steps, which is easily viable for a programming contest.