Problem 1. Let F be the set of all functions over the nonnegative integers. Which of the following properties does big-O satisfy?

1a (5 points). Reflexivity: for every $f(n)$ in F, $f(n)$ is in $O(f(n))$.
ANSWER: yes. To see this, note that if $c=1$ and $n_0=1$, then $f(n) \le cf(n)$ for every $n \ge n_0$.

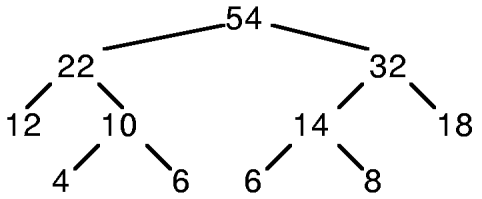
1b (5 points). Antisymmetry: if $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(f(n))$, then $f(n) = g(n)$.
ANSWER: no. One counterexample is the case where $f(n)=n$ and $g(n)=2n$.

1c (5 points). Transitivity: if $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
ANSWER: yes. To see this, suppose $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$. Then there are numbers $c_1 \ge 0$, $n_1 \ge 0$, $c_2 \ge 0$, $n_2 \ge 0$ such that $f(n) \le c_1 g(n)$ for every $n \ge n_1$ and $g(n) \le c_2 h(n)$ for every $n \ge n_2$. If we let $c_3 = c_1 c_2$ and $n_2 = \max(n_1, n_2)$, then $f(n) \le c_2 g(n)$ for every $n \ge c_2$.

1d (5 points). Totality: for all $f(n)$, $g(n)$ in F, either $f(n)$ is in $O(g(n))$ or $g(n)$ is in $O(f(n))$.
ANSWER: no. One counterexample is the case where $f(n) = 1$ if n is odd and 0 if n is even, and $g(n) = 0$ if n is odd and 1 if n is even.

Problem 2. Use Huffman's algorithm to create a binary tree of minimal weighted path length with the weights 4, 6, 6, 8, 12, and 18 at its leaf nodes. What is the tree's weighted path length?

ANSWER:



The weighted path length is
12*2+4*3+6*3+6*3+8*3+18*2
= 24+12+18+18+24+36
= 132

Problem 3a. Let T be an AVL tree of n nodes. Suppose we represent T using an array (like I did for binary trees in the first week of class). In the worst case, how big will the array be? A big-O answer is OK as long as it is a tight bound.

1. *ANSWER: On page 220 it says that T has height $h \le 1.44 \lg n$. Thus in the worst case, the number of nodes in the array will be $1+2+\ldots+2^h = 2^{h+1}-1 = O(2^{1.44 \lg n}) = O(n^{1.44})$.*

Problem 3b (5 points). Does it make any sense to represent an AVL tree in this way? Why or why not?

ANSWER: no. The rotation operations required for insertion and deletion will take a long time because we'll have to move large numbers of nodes from one location to another in the array. Furthermore, the insertion operations may overflow the array if they increase the tree's height.

Problem 4. Suppose you want to search a linked list of n keys $K_1, K_2, \ldots, K_n$, where the probability of K_i being sought is p_i. Suppose that each key K_i costs c_i to read, so that the cost of searching for K_i is $c_1+c_2+\ldots+c_i$. In order to get the smallest expected search cost, what order should the keys be stored in?

ANSWER: the answer to this is on page 209 of the book, in problem 10. The order that will give the smallest expected search cost is the one for which $p_1/c_1 \ge p_2/c_2 \ge \ldots \ge p_n/c_n$.

Problem 5. What is the Boyer-Moore matrix for the string "abcabcabc"?

ANSWER:

```
a b c a b c a b c
```

Problem 5. What is the Boyer-Moore matrix for the string "abcabcabc"?

ANSWER:

	a	b	c	a	b	c	a	b	c
a	0	3	3	0	6	6	0	9	2
b	3	0	3	6	0	6	9	0	1
c	3	3	0	6	6	0	9	9	0
Other	3	3	3	6	6	6	9	9	9

Problem 6a. Let *S* be an AVL tree. Explain how to write a procedure HEIGHT*(S)* that returns the height of *S* and runs in time *O*(lg *n)*, where *n* is the number of nodes in *S*. You don't need to write the actual code, as long as you make it very clear how the program works.

ANSWER: the idea is to find the length of a longest path from the root to a leaf node:

n := the root of the tree
counter := 0
loop
    if n is a leaf then return counter
    if n's balance is -1 then n := n's left child, else n := n's right child
    counter := counter+1
repeat

Problem 6b (5 points). How would you modify your procedure if *S* were a 2-3 tree?

ANSWER: If S is a 2-3 tree then no node has a "balance field", and it doesn't matter which path you follow from the root to a leaf node, because they all have the same length. Thus, replace the 5'th statement of the above procedure with "n := n's left child".

Problem 7a. Let *S* and *T* be AVL trees of the same height *h*. Suppose every key of *S* is smaller than every key of *T*. Explain how to write a function CONCATENATE*(S,T)* that runs in time *O(h)* and returns an AVL tree that contains all of the nodes of *S* and *T*. You don't need to write the actual code, as long as you make it very clear how the program works.

ANSWER: Find S's rightmost node K by starting at the root and following the right-child pointers all the way down. Remove K from S using the standard procedure for deleting nodes from AVL trees. This produces a modified tree S' whose height is either h or h-1. Instead of returning the node K to the freelist, make it the root of a new tree U whose left subtree is S' and whose right subtree is T.

Problem 7b (5 points). What problems would you encounter if you tried to adapt the CONCATENATE function for use on 2-3 trees?

ANSWER: If the height of S' is h-1 then the tree U will not be a 2-3 tree.

Problem 8 (5 points). Let *T* be the tree shown in Figure 7.24a on page 248 of the book. If you invoke SPLAY*(F,T)*, this will do a sequence of rotations on *T*. For each rotation, tell what node is rotated and the direction in which it is moved. You don't need to draw the trees produced by these rotations.

ANSWER: rotate E to the left, G to the right, H to the right, C to the left, and L to the right.