Introduction to Algorithms / Algorithms I  Fall 2016
Homework #3  Due: September 27, 2016, 1:30pm

Remember: you may work in groups of up to three people, but must write up your solution entirely on your own. Collaboration is limited to discussing the problems – you may not look at, compare, reuse, etc. any text from anyone else in the class. Please include your list of collaborators on the first page of your submission. You may use the internet to look up formulas, definitions, etc., but may not simply look up the answers online.

Please include proofs with all of your answers, unless stated otherwise.

1 Lower Bounds (34 points)

(a) Recall the jug matching problem from the last homework. You designed a randomized algorithm which took $O(n \log n)$ comparisons. Prove a matching lower bound for deterministic algorithms: prove that any deterministic algorithm which solves the problem must make at least $\Omega(n \log n)$ comparisons.

(b) Recall the median or sorted arrays problem from the last homework. You designed a deterministic algorithm which took $O(\log n)$ comparisons. Prove a matching lower bound: prove that any deterministic algorithm which solves the problem must make at least $\Omega(\log n)$ comparisons.

2 Search Trees (33 points)

We saw in class how to use binary search trees as dictionaries, and in particular how to use them to do insert and lookup operations. Suppose we want to modify the basic binary search tree to also let us perform the following operation (assuming that all keys are distinct): Given a key $x$, do a version of lookup($x$) that tells us how many keys are less than $x$ (we call this the rank of $x$)

Note that if our data was fixed, this would be easy. We could simply use a sorted array $A$. Then to find the rank of $x$ we simply do a binary search to find $x$ and then return its position minus 1.

But if we want to handle inserts as well, then one way to do this is through search trees. The goal of this problem is to modify normal binary search trees (nothing fancy like B-trees, red-black trees, AVL trees, etc.) so that the above operation can be done in $O(depth)$ time, and inserts can also be done in $O(depth)$ time. In particular, you should do the following:

(a) Describe an extra piece of information that you will store at each node of the tree, and describe how to use this extra information to do the above rank operation in $O(depth)$ time.

(b) Describe how to maintain this information in $O(depth)$ time when a new node is inserted (note that there are no rotations on an insert – it’s just the regular binary search tree insert, but you need to update information appropriately).

For example, a bad way to do this would be for every node to store the rank of its key. This information would let us do the rank operation quickly, but maintaining it on an insert might require updating all of the other nodes (which might be much larger than the depth).
3 Non-comparison sorting (33 pts)

(a) Suppose we are given an array of integers, but instead of all integers having the same length they can each have a different number of bits. So e.g. the number 0 or 1 takes one bit, the numbers 2, 3 take 2 bits, the numbers 4, 5, 6, 7 take three bits, etc. However, the total number of bits over all of the integers in the array is equal to $\ell$. Show how to sort the array in $O(\ell)$ time.

(b) Suppose now that we are given an array of strings (over some finite alphabet, say the letters a–z). Each string can have a different number of characters, but the total number of characters in all the strings (i.e. the sum over the strings of the length of the string) is equal to $\ell$. Show to sort the strings lexicographically in $O(\ell)$ time. Here lexicographic order is the standard alphabetic order, so for example $a < ab < b$. 
