601.433/633 Introduction to Algorithms **Fall 2024**
Homework #3 **Due:** September 24, 2024, 9am

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 Group Sorting (67 points)

We say that an array A of size n is *k-group sorted* if it can be divided into k consecutive groups, each of size n/k, such that the elements in each group are larger than the elements in earlier groups, and smaller than elements in later groups. The elements within each group need not be sorted.

For example, the following array is 4-group sorted:

1	2	4	3	7	6	8	5	10	11	9	12	15	13	16	14

Note that every array is 1-group-sorted, and only sorted arrays are n-group sorted. For the rest of this problem we will only care about deterministic algorithms (and lower bounds against deterministic algorithms). You may assume that all elements are distinct, and if you want to you may assume that n and k are powers of 2.

(a) (17 points) Describe an algorithm that k-group-sorts an array in $O(n \log k)$ (i.e., in at most $O(n \log k)$ time it must turn an array which is not k-group sorted into one that is). Prove correctness and running time.

(b) (17 points) Prove that any comparison-based k-group-sorting algorithm requires $\Omega(n \log k)$ comparisons in the worst case.

(c) (16 points) Describe an algorithm that completely sorts an already k-group-sorted array in $O(n \log(n/k))$ time. Prove correctness and running time.

(d) (17 points) Prove that any comparison-based algorithm to completely sort an already k-group-sorted array requires $\Omega(n \log(n/k))$ comparisons in the worst case.

2 Range Queries (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. Some of you might naturally wonder why we bother to do this, when hash tables (which we will talk about later) already allow us to do this. While there are many good reasons to use search trees rather than hash tables, one informal reason is that search trees can in some cases be either used directly or easily extended to allow efficient queries that are difficult or impossible to do efficiently in a hash table.

An important example of this is a *range query*. Suppose that all keys are distinct. In addition to being able to insert and lookup (and possibly delete), we want to allow a new operation *range(x, y)* which is supposed to return the number of keys in the tree which are at least x and at most y.

In this problem we will only be concerned with normal binary search trees (nothing fancy like B-trees, red-black trees, AVL trees, etc.). Recall that in binary search trees of height h, inserts can be done in $O(h)$ time. For all of the parts, prove both correctness and running time.

(a) (11 points) Given a binary search tree of height h, show how to implement *range(x, y)* in $O(h + k)$ time, where k is the number of elements that are at least x and at most y.

Can we do this operation even faster? It turns out that we can! In particular, for a binary search tree of height h, we can do this in $O(h)$ time. We will prove this in the rest of the problem.

(b) (11 points) 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 range query in $O(h)$ time.

Hint: think of keeping track of a size.

(c) (11 points) Describe how to maintain this information in $O(h)$ 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).