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

Bella Lu
(Partners: Jecia Mao, Luna Liu)

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.

Description of algorithm

We can use a process similar to a deterministic quicksort algorithm.

BPFRT $\rightarrow O$(length of array)
Splitting $\rightarrow O$(length of array)
Recursive call $\rightarrow$?

We only need to do this algorithm until there are k groups left, and every time we pick a median, the left side will be smaller and the right side will be bigger.

For example, after the first median is picked:

$k/2$ groups	median	$k/2$ groups

For the sake of this algorithm, since n is a power of 2, we can always choose the smaller of the two numbers in the middle, then let the median go into the group on the left for splitting.

$k/4$ groups	median	$k/4$ groups	$k/4$ groups	median	$k/4$ groups

...

1 group	median	1 group	...	1 group	median	1 group

The question is how we will determine the ending condition. We stop when we have reached k groups.

We can think of the starting condition as having enough elements for k groups, then every time we split, the size decreases by a factor of 2. The number of groups that the recursion calls on follows the pattern below:

$k \rightarrow k/2 \rightarrow k/4 \rightarrow ... \rightarrow 1$

Since k decreases by a factor of 2 every time, there are a total of $\log(k)$ levels.

Strong induction proof for correctness

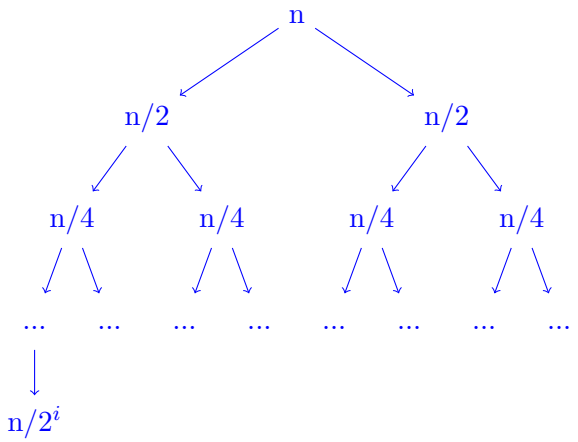
Base Case: In the problem statement, it is given that all arrays are 1-group-sorted. So, the base case, $k = 1$, holds.

For the sake of induction, let's assume that that for all $n \leq \frac{k}{2}$ groups (assuming n and k are powers of 2), the algorithm holds, meaning that it will be k-group-sorted.

Now we must prove that the algorithm holds for k groups. When we have k groups, we can use BPFRT to find the median. Since n is a power of 2, we can always choose the smaller of the two numbers in the middle, then let the median go into the group on the left for splitting. After splitting, we will have two arrays of size $\frac{k}{2}$ groups. By our inductive hypothesis, the algorithm will hold for all subsequent lengthed groups.

$\therefore$ The algorithm is correct. ■

Recursion tree proof for runtime



Let the first level be 0.

As said above, both BPFRT and splitting both take O(length of array). Each node does $O(n/2^i)$ work because 2^i is the length of the array after every time we split the array into halves.

As proved before in the description, there are a total of $\log(k)$ levels. As we see above, each level has 2^i nodes, so each level does $2^i \cdot n/2^i = O(n)$ work. Thus, the total work is $n \cdot \log k$.

$\therefore$ This algorithm is correct and has $O(n \log k)$ running time.

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

We need to find the total number of nodes in the decision tree. Then, we would take the log of that number because this gives us the lower bound (i.e. we can at most get rid of half of the nodes each time we compare).

For example, if we have 16 numbers, and we want to put them into 4 groups, then the number of ways to do so would be $\binom{16}{4,4,4,4}$, because each group would have 4 numbers.

In general, if we want to find the number of ways to put n numbers into k groups, each having the same number of elements (i.e. $\frac{k}{n}$), then there would be $\binom{n}{\frac{k}{n}, \frac{k}{n}, \ldots, \frac{k}{n}}$ number of ways. This equates to $\frac{n!}{(\frac{n}{k}!)^k}$ because there is a total of k terms on the bottom.

$$\begin{aligned}
&\log(\text{number of comparisons}) \\
&= \log[\frac{n!}{(\frac{n}{k}!)^k}] \\
&= \log(n!) - \log[(\frac{n}{k}!)^k] \\
&= \log(n!) - k\log(\frac{n}{k}!) \\
&\text{(by Stirling's approximation)} \\
&= n\log(n) - (\frac{n}{k})k\log(\frac{n}{k}) \\
&= n\log(n) - n\log(\frac{n}{k}) \\
&= n\log(n) - n\log(n) + n\log(k) \\
&= n\log(k) \\
&= \Omega(n \log k)
\end{aligned}$$

$\therefore$ We have proven 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.

Description of algorithm

Since the array is already k-group-sorted, we only need to sort within each group. By definition, each group will have $\frac{n}{k}$ elements. We can sort each group with mergesort.

Proof of runtime

To sort each group of $\frac{n}{k}$ elements, mergesort takes O[(number of elements)log(number of elements)], which is $O(\frac{n}{k} \log(\frac{n}{k}))$.

In total, there are k groups, so the total running time is $O(k\frac{n}{k} \log(\frac{n}{k})) = O(n \log(\frac{n}{k}))$.

Proof of correctness

This algorithm is correct because being k-group-sorted means that all the elements in each group is larger than the earlier groups, and smaller than the later groups. So, completely sorting within each groups gives the correct final ordering.

More rigorously, we can prove it by cases. Let $i \leq j$ be two arbitrary elements in the array before the algorithm (i.e. it is currently k-group-sorted):

Case 1: i and j are in different groups. After the algorithm, i and j are in the correct order because the group themselves are already sorted before the algorithm, which comes in the definition of being k-group-sorted.

Case 2: i and j are in the same group. After mergesort within that group, i and j are in the correct order.

$\therefore$ We have proved 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.

If the array is already k-group-sorted, then to find the total number of combinations, we can do it per group. Then, we would take the log of that number because this gives us the lower bound (i.e. we can at most get rid of half of the nodes each time we compare).

For example, we had 16 numbers, and they are already sorted into 4 groups. The order in each group are scrambled, but each element in each group are larger than the elements in earlier groups, and smaller than later groups. So, we only need to look at the ways to order within each group, then multiply each group. If there are 4 elements in each group, then the number of combinations is 4!. Since they're are 4 groups in total, so the total number of

combinations is $(4!)^4$.

In general, the total number of combinations to order an already k-group-sorted array is $(\frac{n}{k}!)^k$.

$$\begin{aligned}
&\log(\text{number of comparisons}) \\
&= \log(\tfrac{n}{k}!)^k \\
&= k\log(\tfrac{n}{k}!) \\
&\text{(by Stirling's approximation)} \\
&= (\tfrac{n}{k})k\log(\tfrac{n}{k}) \\
&= n\log(\tfrac{n}{k}) \\
&= \Omega(n\log(\tfrac{n}{k}))
\end{aligned}$$

$\therefore$ We have proven 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.

(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.

Description and proof of runtime

1. Find x
We can first use the binary search tree find function to find x. If x is not present, then we stop when we encounter a number smaller than x, then go back up once to get the smallest

number greater than x, which takes at most one more step, which is constant. This process takes $O(h)$ because the most nodes we can traverse to find x is the height, which is h.

2. Go to y
To find the number of elements between x and y, we can use in-order traversal from x, since we have already found the position in the previous step, to y. We terminate the process when we reach y, or the largest node smaller than y. This process takes $O(k)$ because we must traverse every element between x and y, by the nature of in-order traversal. By the definition, k is the number of elements that are at least x and at most y.

By adding the two operations, we get $O(h + k)$ time.

Proof of correctness

By definition of a binary tree, any node's left child is always smaller and the right child is always greater than the node. Thus, we can always find x by going right if x is greater than the current node. If we hit a node smaller than x, where x isn't in the tree, the binary tree let us find the next greater node by simply going to its parent.

By definition of in-order traversal, it will traverse the nodes in numerical order. If we start at x and end at y, we will count all the nodes that is between x and y.

Thus, no nodes are double counted, and we don't count any nodes that we don't want.

$\therefore$ The operation *range*(x, y) can be implemented in $O(h + k)$ time and is correct. ■

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.

Description

At each node, we can store the number of nodes under it as an extra piece of information. We can find the number of nodes to the left or right of the node by simply subtracting the number stored in the node on the other side.

1. Start at the root.

2. Do the same steps as if we are finding x. Every time we go right, meaning that the node is smaller than x, we add the number of nodes in the left branch to a running total. This means counting the number of nodes smaller than x.

3. We can stop when we find x, or the smallest number greater than x present (found by going to the parent of that smaller node).

4. Start from the root then do the same for y. Every time we go right, meaning that the node is smaller than y, we add the number of nodes in the left branch to a second running total. This means counting the number of nodes smaller than y.

5. We can stop when we find y, or the largest number smaller than y present.

Proof of strong induction for correctness

Let a be an arbitrary number. The algorithm should find the number of nodes smaller than a in the tree.

Base case: When we have one node with no children, the stored information will be zero. If the singular node is equal to a, we return zero (num of nodes in left branch). If the singular node is greater than a, we return zero (num of nodes in left branch). If the singular node is smaller than a, we return one (num of nodes in left branch + 1).

For the sake of induction, we assume that for all trees that have $n < k$ nodes, we return the correct number of nodes that is smaller than a.

For any tree that has $n = k$ nodes, the root will either be less than, greater than, or equal to a.

Case 1: Root is less than a
We would add the number of nodes in the left branch plus 1 (for the root). We will then look at the right child of the root, which will be a sub-tree that has $n < k$ nodes. By the inductive hypothesis, this will return the correct number of nodes that is smaller than a.

Case 2: Root is greater than a
We don't care about the number of nodes in the right branch. We will then look at the left child of the root, which will be a sub-tree that has $n < k$ nodes. By the inductive hypothesis, this will return the correct number of nodes that is smaller than a.

Case 3: Root is equal than a
We would return the the number of nodes in the left branch.

Final answer:
(number of nodes smaller than y) - (number of nodes smaller than x)
= number of nodes in between x and y

Since we can correctly find the number of nodes less than any given number, the algorithm is correct.

Proof of runtime

This process takes $O(h)$ twice, because we are essentially simply finding x then finding y, which each takes $O(h)$, with constant adding/subtracting in between. The operations are done at most $O(h)$ times because we traverse at most $O(h)$ nodes.

$\therefore$ This extra piece of information makes range query in $O(h)$ time. ■

(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).

Description

When we insert, we can add 1 to the extra piece of information in every node that is in the path of inserting the node.

Proof of correctness

When we insert a new value, its parent will have one more node under it, which causes the parent of the parent of have one more node node under it. Thus, this information is incremented by 1 for every node that is in the path of the root to the new element.

Proof of runtime

This process is done in $O(h)$ because the time to insert a new element in a binary search tree is at most its height, and we simply update the information as we go down the insertions path. The increment is done in constant time, and it will be done at most $O(h)$ times.

$\therefore$ We can maintain this information in $O(h)$ time when a new node is inserted. ■