April 30, 2019

Problem 1: Lower Bounds [10 points] Prove that any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \log n)$ time in the worst case. (Hint: Consider how to reduce the sorting problem to performing a set of operations on a binary search tree. In other words, show that if a faster algorithm existed for constructing a binary search tree then you would violate the $\Omega(n \log n)$ comparison-based sorting lower bound.)

All we need to do (and this is explicit in CLRS chapter 12) is perform an inorder traversal of the binary search tree. Because we know that an inorder traversal takes $O(n)$ time, all we would need to do is to gather the keys of each node as we traverse and do whatever we like with them (store them in an array/data structure, print them out, etc.) The proof works by contradiction. We assume the number of elements n is greater than 1, because a list of 1 element can be sorted in constant time or is by definition sorted.

Proof:

1) Assume that we can contruct a binary search tree with a comparison-based algorithm from an arbitrary list of n elements in better than $\Omega\, n \log n$ time.

2) A binary search tree by definition is a data structure which has at least one node (else it is an empty tree), and for each node x there are up to two possible immediate children of that node x_{left} and x_{right} such that the key of the left child of x is less than or equal to x and the key of the right child of x is greater than or equal to x.

3) By (2), the leftmost node of any BST is the node with the lowest-valued key, and the rightmost node of any BST is the node with the highest-valued key. If there is only one node it is both the leftmost and rightmost and hence both the lowest- and the highest-valued.

4) For each left leaf or child l of a binary search tree, the parent of l has a key whose value is greater than or equal to the the value of l's key, and each right leaf or child r has a parent node whose key's value is less than or equal to r's key. Therefore each parent node p has a key value p_{val} such that $l_{val} \leq p_{val} \leq r_{val}$.

5) From (3) and (4), the keys in a BST are ordered such that the leftmost leaf is has the lowest value, followed by its parent, followed by its right sibling (if it has one). This parent is itself either a left or right child whose parent's key is either greater than or less than it's own key, all the way up from the leaves to the root, in a binary search tree that has height lg$n$. The reverse is true of the rightmost node and its relatives. This logic also applies to every subtree of any BST.
6) By definition, an inorder traversal of a binary tree traverses the BST by first visiting the leftmost leaf, then its parent, then its right sibling. It continues in this fashion by visiting the left, parent(or root), and right nodes/subtrees of the BST until it has visited the rightmost leaf.

7) By (5) and (6), an inorder BST traversal visits the elements of the tree in increasing order, and would therefore have access to a sorted permutation of the n elements in the tree.

8) It takes $O(n)$ to traverse a tree, because we are just accessing the elements, and are not per-

forming any comparisons.

9) There is a proven $\Omega\, n \log n$ lower bound on comparison-based sorting algorithms.

10) By (1), (8), and (9) we have a contradiction, because we have discovered a comparison-based sorting algorithm (the construction of the BST coupled with the inorder traversal) that can sort in better than $\Omega\, n \log n$ time.

11) Therefore, it must be the case that we cannot construct a binary search tree using a comparison-based algorithm in better than $\Omega\, n \log n$ time. Because of this, we can think of a BST as being a kind of pre-sorted data structure.

Problem 2: Median of Medians The 'Median-of-medians' selection algorithm presented in class divides the input into groups of 5. Using a group of odd size helps keep things a little simpler (because otherwise the group medians are messier to define), but why the choice of 5?

(a) [10 points] Show that the same argument for linear worst-case time complexity works if we use groups of size 7 instead.

The argument from the text (for groups of 5) states the following: After the initial round of median-finding (after we've determined the median of each group of 5 elements), we know we will have at least half of these medians that are greater than or equal to the 'median-of-medians.' This means that at least 3 elements in half of the $\lceil \frac{n}{5} \rceil$ groups are also greater than or equal to the 'mom.' From this, they derive the known quantity for elements greater than the 'mom':

$$3\left(\lceil\frac{1}{2}\rceil \cdot \lceil\frac{n}{5}\rceil - 2\right) \geq \frac{3n}{10} - 6$$

They subtract 2 groups from the total to account for the group with the 'mom' in it and the group with less than 5 elements. From there they add up the costs and determine that to split the initial array, call insertion sort n times on a constant number of items, and recursively call the algorithm costs, in the worst case, $T(n) = T(\lceil \frac{n}{5} \rceil) + T(\lceil \frac{7n}{10} \rceil + 6) + an$

Using the same reasoning for groups of 7, we would have a similar situation; after splitting the input into $\lceil \frac{n}{7} \rceil$groups of 7, we would still have at least half of the medians greater than the 'mom,' but with a couple of differences.

Instead of each of these groups thereby having 3 greater elements, each group whose median was greater than the 'mom' would actually have 4 elements greater than or equal to the 'mom', because the median would have 3 elements greater than itself (rather than 2 in the case of groups of 5).

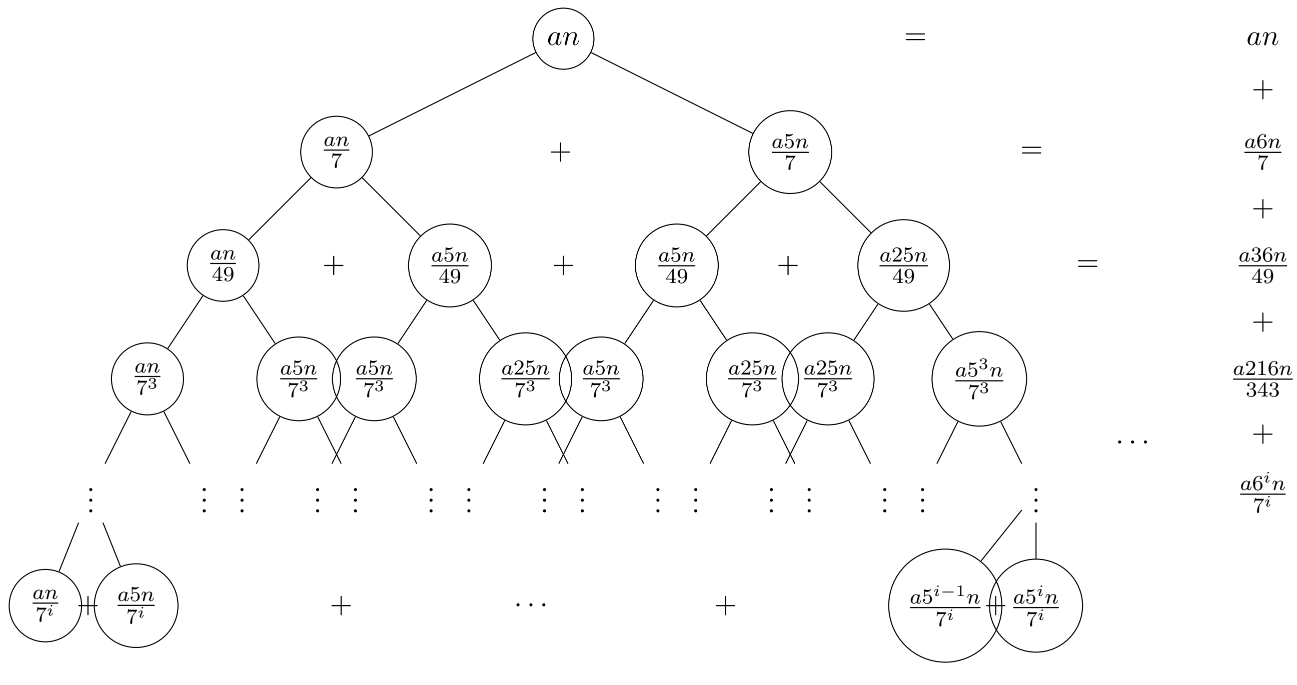
We could still account for the two groups that do not fit this criteria by subtracting them out. Therefore, the known quantity of elements greater than the 'mom' after grouping and median finding in each group of 7, is:

$$4\left(\lceil\frac{1}{2}\rceil \cdot \lceil\frac{n}{7}\rceil - 2\right) \geq \frac{2n}{7} - 8$$

Furthermore, we'd have a similar complexity for breaking this problem up, with slight changes. It would still take constant time to call insertion sort on all of the groups of 7 elements, but we'd have to call it $O(n)$ times, and there is the other constant factor of splitting into groups of 7; all of this would cost $\lceil\frac{n}{7}\rceil + an$. This gives us a recurrence that looks similar but obviously has different fractions of the problem input size; Looking at the situation where we always have bad luck and we don't find the 'mom' right away, but have to recur on the larger group of elements left over, we'd get this:

$$T(n) = T(\frac{n}{7}) + T(\frac{5n}{7}) + 8 + an$$

Without worrying about adding or subtracting 8, let's now take a look at a recursion tree to discover if, in fact, we have the same situation as we did with groups of 5 (costs are on the right [forgive the formatting, all attempts to correct spawned hydra-heads]):



The point of this messy picture is to show that at increasing levels of the recursion tree, we have decreasing costs. Waving our hands a bit, we can see that this is happening as the denominator of each level's costs is multiplied by 7, while the numerator, as levels increase, is outpaced by the denominator. Intuitively, and from our knowledge of the Master Theorem, if the costs are dominated at the root, then we will have case 3. Obviously the MT can't be applied to this situation formally, but we can observe that an is the largest cost, and we have decreasing costs all the way down until we've run out of inputs to the problem. We know from our proof of the 5-groups version of this problem that the fractions of the recurrences to this problem break down to an infinite geometric series that, in closed form, evaluates to a constant multiplied by the costs incurred by dividing and conquering, which we know are $O(n)$. In other words, dividing the problem into groups of 7 also $\in O(n)$.

Looking at this a bit more formally, we can see that the right node of the rightmost branch of each level of the recursion tree is always the node that contains the largest subproblem (this is the only branch where the 5 in the numerator is raised to the same power as the 7 in the denominator) and therefore incurs the largest cost for each level. Therefore, if there are i levels on this unlucky side, then this subtree will be greater than or equal to all subtrees to the left of it (in height), because the problem size will approach zero more quickly for smaller subproblems (subtrees from leftmost to right). We can set $T(1) = 1$ as our base case for this rightmost subtree, and solving:

$$\begin{aligned} T(1) &= T(n\frac{5^i}{7^i}) \\ 1 &= n\frac{5^i}{7^i} \\ \frac{1}{n} &= \frac{5^i}{7^i} \\ n &= \frac{7^i}{5^i} \\ n &= (\frac{7}{5})^i \\ i &= \log_{\frac{7}{5}} n \end{aligned}$$

This tells us that we will need to recursively call this algorithm at most $\log_{\frac{7}{5}} n$ times. So, we could consider the just costs at the rightmost node for each level from the root to the rightmost leaf of a tree of height $\log_{\frac{7}{5}} n$:

$$
\begin{aligned}
& an \sum_{i=0}^{\log_{\frac{7}{5}} n} \frac{5^i}{7^i} \\
&= an \sum_{i=0}^{\log_{\frac{7}{5}} n} (\frac{5}{7})^i \\
&= an \cdot \frac{1 - \frac{5}{7}^{\log_{\frac{7}{5}} n+1}}{1 - \frac{5}{7}} \\
&= an \cdot \frac{7}{2} \cdot (1 - \frac{5}{7}^{\log_{\frac{7}{5}} n+1}) \\
&= an \cdot \frac{7}{2} \cdot (1 - \frac{5}{7}^{\log_{\frac{7}{5}} n} \cdot (\frac{5}{7})) \\
&= an \cdot \frac{7}{2} \cdot (1 - \frac{7}{5}^{-1^{\log_{\frac{7}{5}} n}} \cdot \frac{5}{7}) \\
&= an \cdot \frac{7}{2} \cdot (1 - \frac{7}{5}^{\log_{\frac{7}{5}} n^{-1}} \cdot \frac{5}{7}) \\
&= an \cdot \frac{7}{2} \cdot (1 - \frac{7}{5}^{\log_{\frac{7}{5}} \frac{1}{n}} \cdot \frac{5}{7}) \\
&= an \cdot \frac{7}{2} \cdot (1 - \frac{1}{n} \cdot \frac{5}{7}) \\
&= an \cdot \frac{7}{2} \cdot (1 - \frac{5}{7n}) \\
&= an \cdot \left(\frac{7}{2} - \frac{7}{2} \cdot \frac{5}{7n}\right) \\
&= an \cdot \left(\frac{7}{2} - \frac{5}{2n}\right) \\
&= \frac{an7}{2} - \frac{an5}{2n} \\
&= \frac{an7}{2} - \frac{a5}{2} \\
&= \frac{a(n7-5)}{2} \\
&= 3.5an \\
&= O(n)
\end{aligned}
$$

We know from the recursion tree that this branch incurs the highest cost of this tree, and even after adding the other costs (small fractions of n) we will still have a result in linear time.

To illustrate more clearly the weight of the right we could recognize that the costs at each level add up to $an\frac{6^i}{7^i}$ and set $i = \log_{\frac{7}{6}} n$. We can derive a similar solution by summing all the levels to get a constant multiplied by an to discover the total costs:

$$
\begin{aligned}
& an\sum_{i=0}^{\log_{\frac{7}{6}} n} \frac{6^i}{7^i} \\
&= an\sum_{i=0}^{\log_{\frac{7}{6}} n} (\frac{6}{7})^i \\
&= an\cdot\frac{1-\frac{6}{7}^{\log_{\frac{7}{6}} n+1}}{1-\frac{6}{7}} \\
&= an\cdot 7\cdot(1-\frac{6}{7}^{\log_{\frac{7}{6}} n+1}) \\
&= an\cdot 7\cdot(1-\frac{6}{7}^{\log_{\frac{7}{6}} n}\cdot(\frac{6}{7})) \\
&= an\cdot 7\cdot(1-\frac{7}{6}^{-1^{\log_{\frac{7}{6}} n}}\cdot\frac{6}{7}) \\
&= an\cdot 7\cdot(1-\frac{7}{6}^{\log_{\frac{7}{6}} n^{-1}}\cdot\frac{6}{7}) \\
&= an\cdot 7\cdot(1-\frac{7}{6}^{\log_{\frac{7}{6}} \frac{1}{n}}\cdot\frac{6}{7}) \\
&= an\cdot 7\cdot(1-\frac{1}{n}\cdot\frac{6}{7}) \\
&= an\cdot 7\cdot(1-\frac{6}{7n}) \\
&= an\cdot\left(7-7\cdot\frac{6}{7n}\right) \\
&= an\cdot\left(7-\frac{6}{n}\right) \\
&= an7-\frac{an6}{n} \\
&= an7-a6 \\
&= a(n7-6) \\
&= 7an \\
&= O(n)
\end{aligned}
$$

It looks like the rightmost branch of the tree costs roughly half of the total cost at each level for sufficiently large n, which is an interesting result.

(b) [10 points] Show that groups of size 3 results in superlinear time complexity.

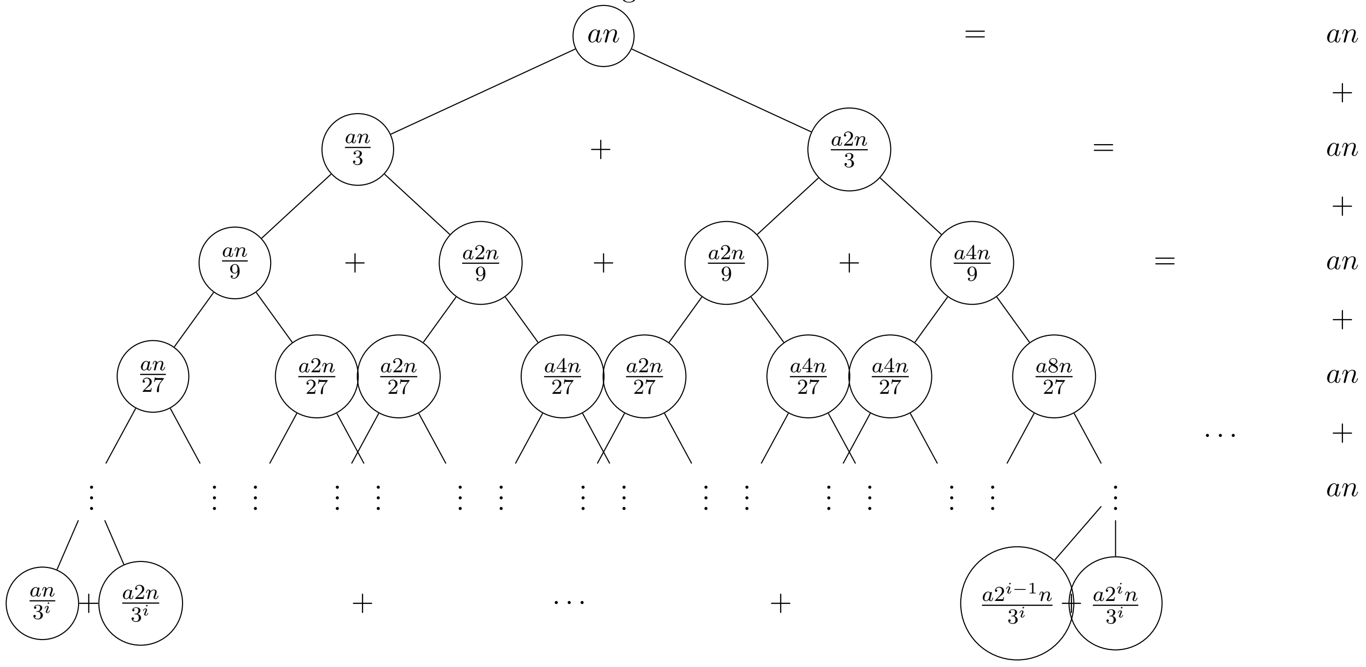
This instance of the problem is actually super interesting, because you would imagine that, like the groups of 5 or 7, it would also have linear time complexity. Let's jump right in and then discuss after, using the same logic as above to determine elements larger than x pre-'mom':

$$2\left(\lceil\frac{1}{2}\rceil \cdot \lceil\frac{n}{3}\rceil - 2\right) \geq \frac{n}{3} - 4$$

Again, the complexity is similar for the first stage of the algorithm. Copying and pasting, it takes constant time to call insertion sort on all of the groups of 3 elements, but we'd have to call it $O(n)$ times, plus the constant factor of splitting into groups of 3; all of this would cost $\lceil\frac{n}{3}\rceil + an$. The recurrence:

$$T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + 4 + an$$

Ditch the 4 and let's climb down the tree once again:



Unfortunately, we have a curious problem here. Namely, each level of recursion incurs the exact same cost of an. This is similar to situations that correspond to Master Theorem case 2. Again, we can't use the MT, but we can observe that at all levels of the tree the costs are evenly split and therefore we can't, like above, take the root's costs and ignore the rest of the tree. Instead, we must take the costs of all of the levels of recursion. How far does this tree go down? Using the same logic as above, the rightmost subtree's leaves are at $T(1) = 1$ and $i = \log_{\frac{3}{2}} n$. We don't really need to perform the summation

$$\sum_{i=0}^{\log_{\frac{3}{2}} n} an$$

this time because we can see that this will cost us around $an \log_{\frac{3}{2}} n$ so this is (as we would expect for MT case 2 as well) $\in O(n\, log\, n)$.