Midterm Notes, 2022

March 24, 2022

1 Fruit Jumble

This problem was fairly standard (and was on the practice midterms). The basic idea is to look at the invariant for each sorting algorithm. BubbleSort has the largest elements sorted at the end of the list (and so has to be D or E), SelectionSort has the smallest items sorted at the beginning of the list (and so has to be A, C, or E). QuickSort is partitioned around the pivot, which is the first element (Elderberry) and so has to be C. Since E is clearly neither BubbleSort or SelectionSort (since both the beginning *and* end are sorted), that resolves those three. InsertionSort has the beginning of the list sorted and the end identical to the unsorted list, so has to be B (as it is unchanged from the unsorted list from Mango onwards). The trickiest to identify is probably MergeSort, but (having eliminated E as nonsense), there is only one left: F.

2 Asymptotically Approaching Answers

These were three questions on asymptotic analysis.

a. The smallest function bigger than $n^2 \log n$ is n^3. The function $O(n^2)$ is too small.

b. The base of the logarithm is just a constant multiplier, and big-O notation is not impacted by constants.

c. Notice that every time N doubles, the time is multiplied by 4. That indicates that the relationship is quadratic. (The function is somewhere close to $(N/20,000)^2$).

3 Recurring Recurrences

These were a few questions on recurrences.

a. This is a basic recurrence that is very similar to the recurrence for QuickSelect, except that we are dividing by 4 instead of 2. If you expand it out, you end up with $n + n/4 + n/16 + n/256 + \ldots$ which sums to something definitively less than $2n$.

b. This is just the recurrence for QuickSelect where the pivot is the median value. At each step, you spend n times partitioning around the pivot, and then recurse on a sub-problem of size $n/2$.

c. This function divides the the array into three pieces of size $n/3$ and recurses on one of them (i.e., the one from $n/3$ to $2n/3$). It also looks at $2n/3$ items in the array prior to recursing. Thus the recurrence is something like $T(n) = T(n/3) + 2n/3$.

d. The outer loop has n iterations, the inner loop has $\log n$ iterations.

4 How Fast Is It

This question asked for the asymptotic running time of a variety of algorithms.

a. Listing elements in a binary search tree is just an in-order traversal so $O(n)$.

b. The only way to find the median element in an AVL tree is to enumerate the items from smallest to largest. There is no other obvious way to figure out where the median will be. (Think of the situation at the root: just by looking at the heights of the two children, you cannot determine whether the median is in the left or right subtree. That's why the Rank-Select tree augments each node with the weight. But an AVL tree does not have the weight.) So this takes $\Theta(n)$ time.

c. QuickSort is no faster on a sorted list than an unsorted list. (In fact, if you just use the first item as the pivot, it would be worse!) So in this case, it is still $O(n \log n)$.

d. Similarly, QuickSelect does not run any faster or slower on a sorted list. (If the list is sorted, or likely to be sorted, you should just a different approach! I wonder if there is a good algorithm that adapts to how sorted the list is?)

e. In this case, the load on the hash table is $m/n = \log n$, so the expected length of a hash table chain (and hence the cost of a query) is $O(\log n)$.

f. A deletion can trigger a rotation at each step up the tree, i.e., $\Theta(\log n)$ rotations.

5 Where can I find the answer?

1. None of these solutions led to a valid binary search. All accessed the array outside the range $[0, n-1]$ leading to in invalid array access error.

   - For case I, consider the input array [123] and assume the search key is 4. In the first iteration, start is 0, end is 2, and mid is 1. It updates start to 2, recursing on the right half. In the second iteration, start is not larger than end, so it continues to compute mid = 2. It then recurses again on the right half, with start equal to 3. It now finds that start is larger than end and so tries to return. To do so, it access the array at $A[start]$, causing an out-of-bounds exception.
   - For case II, it is similar to the previous case except recursing on the right side instead of the left. Consider the input array $[1, 2]$ and assume the search key is 0. In the first iteration, start is 0, end is 1, and mid is 0. It updates end to -1, recursing on the left half. In the second iteration,it now finds that start is greater-than-or-equal-to end and so tries to return. To do so, it access the array at $A[end]$, causing an out-of-bounds exception.
   - For case III, the same type of problems occur. Consider the input array $[1, 2]$ and assume the search key is 0. In the first iteration, start is 0, end is 1, and mid is 0. It updates end to -1, recursing on the left half. In the second iteration, end is less than start, so it does not return. In this case, it computes mid to be equal to 0 again. And so it again recurses setting end = -1. You will notice that this results in an infinite recursion.

   Binary search is surprisingly hard!

2. Surprisingly, this condition is not strong enough to allow for efficient search. Consider the example:

   $$A = [8, 8, 8, 8, 8, 9, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8]$$

   If you search for 9, there is no way to find it in faster than linear time. Every other query you make to the array provides no information as to where to find the 9. And notice that this satisfies the requirements that no items later in an array is more than 1 bigger than an item earlier in the array. (Of course, this will not work if you do not allow duplicates. In that case, it may be possible—but probably tricky!)

3. This is another question on sorting almost sorted arrays. In this case, you can design a search algorithm that works (I think). But simple versions of binary search will not work. In particular, a good example array to keep in mind is: $A = [1, 101, 2, 102, 3, 103, 4, 104, 5, 105, 6]$. The requirement for the array ensures that the even indexed array slots are sorted and that the odd indexed array slots are sorted, so two binary searches is enough! But the algorithm given does not work.

   The key problem with the proof is the following: it is true that whenever the search key is in the first half of the array, then the algorithm recurses on the right half. But the converse is *not* true, i.e., it is *not* true that whenever it recurses on the first half, then the key is in the first half. (So statement 2 of the proof is not correct, as the final part of statement 2 asserts that.) Consider the example where we are searching for key 4 in the example above, where mid is 103. Because $4 < 103$, we recurse on the left half. But of course key 4 is in the right half.

6 Putting Everything in Order

These questions were mostly about sorting lists containing only two values. Unfortunately, most sorting algorithms do not get any improvement in this case. Thus, BubbleSort, SelectionSort, InsertionSort, and MergeSort see no change in their running time (i.e., $O(n^2)$ and $O(n \log n)$.) QuickSort with 3-way partitioning, however, does work better! After only one partition, the array is sorted and QuickSort will terminate after two iterations. The last part (F) simply required recognizing that the array was partitioned array 42.

7 Trees

1. In this case, I decided to accept either answer as correct, since there were several different ways to interpret the question. The goal of the question was to ask about the depth of the leaf that is the closest to the root, as every insert has to search all the way down to a leaf. Can you have an AVL tree with a leaf at depth $< \log n$? Yes! Draw a tree where the left sub-tree is a complete binary tree and the right sub-tree is maximall unbalanced. Even at height 4, it is possible. So in that sense, you can in the best case do an insert that looks at less than $\log n$ levels of the tree. Does every AVL tree have a leaf with depth $< \log n$? No! If you draw a perfectly balanced tree, then every leaf is at depth $>= \log n$ and so no insert can be faster than $\log n$. Finally, one additional confusion is that the question did not ask about the depth of the leaf but about the running time (and not the asymptotic running time), so it would be reasonable to argue that even if the insert only needed to traverse $\log n/2$ levels of the tree, then the time would be $> \log n$ since clearly it must perform at least two operations per level. Given that there were several different completely reasonable ways to interpret this question, I accepted either answer as correct.

2. We know that $n \leq 2^{h+1} - 1$ for every binary tree. So statement (IV) is true. The rest are not. For (I), consider a FibTree(4) (from the other question) which has height 3 and 7 nodes. The same example implies that (III) is also not true. For (II), consider a complete tree of height 2 containing 7 nodes.

3. If two nodes are siblings, we know their height differs by at most 1. If two nodes are siblings, we know their depth differs by 0. But for an arbitrary two nodes, their depth can differ significant, up to a $\Theta(\log n)$ factor. Imagine an AVL tree that is maximally unbalanced: for a node at height h, its right child is always height $h - 2$ and its left child is height $h-1$. Notice that the rightmost child will have a depth of approximately $h/2$, while the left child will have a depth of h. Since the total height of of this tree is $\Theta(\log n)$, we conclude that the difference in depth is also $\Theta(\log n)$. (You can also draw some simple examples to see this, i.e., how the difference in depth keeps increasing as n gets larger.)

4. A FibTree is height-balanced, by definition. Notably, you can show that FibTree(n) has height $n - 1$. You can prove this by induction, as a FibTree(1) has height 0, a Fibtree(2) has height 1, and a Fibtree(n) has two FibTrees as children, one with height $n - 2$ and one with height $n - 3$ (by induction).

5. This follows from the same analysis we did in class (thought it can also just be solved by logic). First, the mathematical approach:

$$\begin{aligned} n_h &\geq n_{h-1} + n_{h-3} + 1 >= 2n_{h-3} \\ n_h &\geq 2^{h/3-1} \\ log_2(n) + 1 &\geq h/3 \\ h &\leq (log_2(n) + 1) \end{aligned}$$

Note that the -1 term in the second term is simply to avoid floor/ceiling problems in case h is not divisible by 3. Now, let's think about this question logically. The key observation is that this tree is necessarily less well balanced than an AVL tree. (Every AVL tree satisfies the 3AVL requirement, while not every 3AVL tree is an AVL tree, and intuitively you can unbalance the tree more.) The answer cannot by (1), because that is not even true for a regular AVL tree. Similarly, answer (2) is actually also not true for a regular AVL tree, since $2/\log 3 \approx 1.25$. (This would require a bit of approximation, i.e., you know that $\sqrt{2}$ is about 1.4, i.e., close to 1.5, so $2\sqrt{2} \approx 3$, so $\log 3 \approx 1.5$. Hence $2/1.5 \approx 4/3$.) The maximum height of an AVL tree, by contrast, is about $1.44 \log n$. Answers (6), (7), and (8) are implausibly large, as a few examples should show. So the most tempting alternative answer is presumably (4). However to get a height that is quadratic (or even cubic) requires a lot more imbalance. Thinking back to the maximum difference in depth between two leaves, this means that a tree of height h can have a leaf of depth $h/3$, but no shallower. That would mean that the shallowest leaf had a depth of $\Theta(\log^2(n))$ which is impossible. (Count the minimum number of nodes in a tree where every leaf has depth $\Theta(\log^2(n))$.) So by the process of elimination, (3) seems likely to be the answer. (But it is best to do the math above to verify, of course.)

6. This problem is just about simulating the AVL tree insertion algorithm.

7. If x is out of balance after the insertion, it means that after the insertion the left and right children of x differ in height by at least 2. Thus before th insertion, they must differ in height by at least 1. Hence the children of x must have height 6 and 5. We know that C is the shallower tree, since otherwise increasing the height of y wouldn't cause x to be out of balance. Hence y has height 6 before the rebalance.

8. We know that $w > y$ because a double rotation occurs. If $w < y$, then y would be left-heavy and there would be no need for a double-rotation.

8 Making a Hash of Things

1. A symbol table can be implemented with an AVL tree. An AVL tree supports all the basic symbol table operatinos (insert, delete, search, etc.) reasonably efficiently. A hint to this end is that Java has a class called TreeMap that implements the Java Map interface (i.e., the Java equivalent of a symbol table). Of course, you shouldn't really rely on Java, as it's Map interface is a lot wider than a typical symbol table (and supports inefficient operations).

2. A dictionary cannot be reasonably implemented with a hash table. A dictionary needs to support ordered operations like successor, predecessor, findMax, and findMin. None of these operations are available in a symbol table. The key point is that a dictionary is necessarily an ordered data structure, while a hash map does not maintain the order of elements.

3. This was a basic probability calculation. The simplest way to solve it is to simply draw a small event tree. Assume that Item 1 is already inserted into the hash table. Without loss of generality, you might as well it is inserted into slot 1. So the root of your event tree is: does Item 2 *also* get put in slot 1? The left branch (NO) occurs with probability $(1 - 1/m)$ and the right branch (YES) occurs with probability $1/m$. The next level of the event tree is asking: does Item 3 get put into the same slot as Item 1 or Item 2? If Item 1 and 2 are both in slot 1, then the probability is $1/m$. Otherwise, the probability is $2/m$. Now you can look at the

leaves of the event tree and multiple the probabilities. The entire right subtree (Items 1 and 2 in slot 0) occurs with probability $1/m$. In the left sub-tree, the only leaf with two items in one slot occurs with probability $(1 - 1/m)(2/m) = 2/m = 2/m^2$. Thus the total probability of a collision is $3/m - 2/m^2$. (You could obviously just write it out in terms of conditional probabilities more succinctly.)

9 3d Printing

This was an algorithm design question, where the goal was to build a data structure much like a Rank-Select tree. Let's consider what happens after a simple example:

- build 7
- build 3
- build 5

(Notice that the order of the builds does not matter.) If you go through the example, step by step, then you get some sense of how it works. After the first build, every location $\leq$ 7 has 1. After the second build, every location $\leq$ 3 has 2. Afer the third build, locations $\leq$ 3 have 3, locations 4 and 5 have 2, and locations 6 and 7 have 1.

So at location 6, there have been two builds to the left and one build to the right; that yields a height of 1. At location 4, there has been one builds to the left and two build to the right; that yields a height of 2. At location 1, there are three builds to the right, and that yields a 1.

Thus, after going through a few simple examples, we conclude that the height at location j is equal to the number of builds that end at a location $\geq j$. Thus a "trivial" augmentation would be to store at leaf j the number of builds that end at a location $\geq j$. However, that would be a very inefficient augmentation, as every new build would have to update all the leaves covered by that build. (Notice that this would be satisfied by answer 2, i.e., number of build commands that include at least one leaf in the subtree rooted at node u. For each leaf, that would be exactly equal to the number of builds that include that leaf. However, this answer would have been inefficient, and it would not have been consistent with part (c) and part (d).)

So just like the Rank-Select tree, we want to store in each node the number of builds that end in the right subtree (i.e., **answer 9** is the correct answer for 9(a)). Basically, if we weight each leaf in which a build ends as weight 1 and every other leaf as weight 0, then you can imagine that we want to store the weight of the right subtree. (Notice this is the opposite of the Rank-Select tree where we want the weight of the left subtree. This is because in Rank-Select, we want to know how many things are $\leq$ a given node, while here we want to know how many things are $\geq$ a given node.)

The second part is just about checking that the invariant makes sense. (The goal of this part was to help you understand the correct invariant by going through an example.) The root should be storing the number of builds that end in the right subtree, i.e., all the builds that end at locations $9 - -16$, thus the answer is 1 (i.e., **answer 7**).

For part (c) and (d), we need to implement an algorithm that will maintain the invariant from the first part. In part (c), you are walking up the tree from the leaf. The first step is clearly to add 1 to the leaf. Then, whenever you are the right child, you want to add 1 to the lego count at your parent, since you are increasing the number of builds in the right-subtree by 1. If you are the left child, you do not want to increase the lego count at the parent, since you are a left child! So the answer for part (c) is $(1, 0, 1)$.

Finally, for part (d), whenever we are a left child, we need to add the parent's lego count, i.e., we want to include the value from the right sub-tree. But if we are right child, then we already have the count from that sub-tree, so we do not need to add anything in. So the answer for part (d) is $(v.legos, 0)$.

If you find this question confusing, look back and make sure you undestand how and why the rank-select tree works. You also might look back at the card-flipping problem from recitation, which is similar too.