Discussion of Midterm, CS2040S 2024

April 19, 2024

1 Question Q1: Sorting

This question involved executing the specified sorting algorithms on the specified array of unsorted trees.

A. MergeSort: The MergeSort is recursively executing on the first half of the array (while the second half is unchanged). The first quarter has finished sorting, while the second quarter is about to be merged.

B. HeapSort: This sort was "none of the above," which in this case was HeapSort. As a partially sorted array, it is not sorted at the beginning (like InsertionSort or SelectionSort). It cannot be BubbleSort because Maple has moved all the way from the end to the beginning, while only the last three items are in order. It cannot be MergeSort because items have moved from the first half to the second half of the list (but the array is far from sorted, i.e., this is not the final merge). It looks a bit like it could be QuickSort with Nutmeg as the pivot, but the reorganization of the first half does not reflect any pivot algorithm we saw. Given the issue with the QuickSort question, we gave full credit for choosing QuickSort here. (Other sorts were wrong.)

C. BubbleSort: The last three items in the array are in their proper place, so this is the state after three iterations of BubbleSort. Notice that no item has moved more than three slots toward the front.

D. QuickSort: Here, the first pivot chosen was Nutmeg, and the array partitioned around that. The second pivot chosen was Cedar. The partitioning algorithm used mistakenly was slightly different than the one from class, and so we gave full credit also for choosing "none of the above." However, you could recognized QuickSort by focusing on the pivots and how it was partitioned around the pivots. (Other sorts were wrong.)

E. InsertionSort: Note that the prefix from Cedar to Pine is sorted, and the rest of the array is unchanged—this is characteristic of InsertionSort.

F. SelectionSort: Note that the prefix from Ash to Date is in the correct place (while most of the rest of the array is unchanged)—this is characteristic of SelectionSort.

2 Question Q2: Asymptotics

A. The tight asymptotic bound here is $O(n^2 \log^2 n)$, but that is not an available option. The closest option (asymptotically) larger is $O(n^3)$.

B. This is a standard summation $1 + 2 + 4 + 8 + \ldots + n = O(n)$.

C. This is a standard summation $1 + 2 + 3 + 4 + \ldots + n = O(n^2)$, which we have seen frequently this semester.

D. Notice that for large n, we certainly have $\log n > 3$, so this cannot be $O(n^3)$ or smaller.

E. This is simply the definition of big-O notation, as of Week 2 of the semester.

F. This function is interesting because it is neither big-O nor big-Ω. Notice that for even n, it is $\Omega(n^2)$ and for od n it is $O(n^2)$. But as a single (discontinuous) function, it is neither: there is no n big enough that it will either always be big or always be small. Note that you can come up with continuous functions that have the same property (e.g., that oscillate between larger and larger positive and negative values).

G. The recurrence tree has n nodes, each with value 7.

H. The recurrence "tree" has depth $O(\log n)$ and is a line where each node has value 2040. Notice that if you design an algorithm that spends a constant cost (i.e., in this case, 2040) and reduces the problem by a constant fraction, then you have an $O(\log n)$ time algorithm! An example of that is binary search.

I. You might notice that this looks a lot like InsertionSort (where doSomething swaps the items)!

J. For the recurrence here, it seems like m is not very important: $T(n) = 2T(n/2)+ O(n)$. If it recursed all the way down to $n = 1$, then it would run in $O(n \log n)$ time. The tricky part here is the base case: the recursion stops when $n < m$. Thus, the leaves in the recursion tree occur when the leaf is of size m. Thus the recursion tree is of depth $\log(n/m) = \log(n) - \log(m)$. Each level of the recursion tree still sums to $O(n)$ as usual, so the total cost os $O(n \log(n/m)$.

K. The main tricky thing here was that this is a bi-recursion between two functions. However, if you focus on the recurrence relationship you can unravel it. The PartyTime function has the following properties: it does $O(n)$ work calling doParty, and then it executes some recursions via studyTime with paramter $n/3$. The studyTime function does $O(1)$ work and calls 3 recursive instances of partyTime. So the recurrence is going to be $T(n) = 3T(n/3) + O(n)$.

3 Question Q3: How fast is it?

A. InsertionSort spends $O(n^2)$ time, even ignoring comparisons because it has to swap items backward in the array.

B. SelectionSort, by contrast, only spends $O(n)$ moving things around because it can swap items directly into place.

C. This is a question about sorting an almost sorted array. Notice that each chunk of 10 items is out-of-order, but any two items that are in different chunks are ordered correctly. Unfortunately, QuickSort is no faster on an almost sorted array, and so the expected running time is still $O(n \log n)$.

D. By contrast, InsertionSort is very fast on an almost sorted array. Notice that when running InsertionSort, no items is moved more than distance 10 (since an item in a given chunk is never moved outside that chunk). So for each iteration of InsertionSort, there are no more than 10 swaps—so the total cost os $O(n)$.

E. This is a basic fact about AVL trees (notably, deletions may require $O(\log n)$ rotations).

F. This is a basic fact about (2,4)-trees. Imagine that the tree is completely full—then every node on the insertion path will need to be split.

4 Question Q4: Invariants and Algorithms

A. This is, effectively, a two-sided BubbleSort that is sorting up and down at the same time! First it sweeps up doing a BubbleSort comparison, and then it sweeps down. The only tricky part is that it is sorting in reverse (i.e., note the direction of comparison). As such, the proper invariants are similar to BubbleSort—generalized to both sides: each iteration moves the next biggest item to the front and the next smallest item to the end. So, after iteration 0, A[0] contains the biggest element in the array and A[n-1] contains the smallest element in the array.

B. A balanced tree is one that is height $O(\log n)$. There are certainly balanced trees that are not (always) height-balanced (e.g., weight-balanced trees). There are many different ways to construct a balanced tree. The converse, however, is true: every height-balanced tree is balanced.

C. This follows simply from the definition of height (i.e., the max height of a child plus one).

D. A simple counterexample here is a perfectly balanced binary tree (which is a valid height-balanced AVL tree). For example, a perfectly balanced binary tree of height 2 (where leaves are height 0) has 7 nodes, which is larger than $2^2 + 1$. (A perfectly balanced binary tree of height h has $2^{h+1} - 1$ nodes.)

E. This is exactly what we proved when showing that height-balanced trees were balanced. See the proof from class for details.

F. This question is stated incorrectly and everyone got full credit. The intent of the question was to generalize the definition of height-balanced to a degree-k tree. A better statement of the question would have been that no two children differed in height by more than k (analogous to a height-balanced tree with a difference of at most 1), where non-existant nodes count as height -1. In that case, you can follow the same analysis we did for height-balanced trees. As stated, however, the definition is not formulated clearly, and so the question has been discounted.

G. This is actually a completely correct (if somewhat silly) way to check if an array is sorted! Think about it recursively: if the left half of the array is correctly sorted and the right half of the array is correctly sorted and the largest element on the left is less than the smallest element on the right, then the whole array is correctly sorted!

H. The recurrence here is $T(n) = 2T(n/2) + O(1) = O(n)$. So this algorithm isn't even obviously (asymptotically) worse than a naive check for whether something is sorted—though it will be much slower in practice. (Why?)

5 Question Q5: Trees

A. After the insertion, the root is out of balance.

B. This is a double-rotation case. (You had to selection the option with two rotations—and the order of the two rotations matters.)

C. Each termination-sign indicates a string. So this question simply involved counting the dollar signs.

D. This was simply checking if a string is in a trie.

E. This was simply checking if a string is in a trie.

F. This was simply checking if a string is in a trie.

G. This question was intended to observe that a trie with k characters has $O(k)$ nodes. It was not clear from the question whether to count the string-terminating characters as nodes. It is true that if the sum of all the string lengths is T, then there are $O(T)$ nodes—but if you count the termination characters, then you could have more nodes than text. Hence we gave full credit for this question for either answer.

6 Question Q6: AVL tree search

This question is about constructing an augmented tree that is very, very close to a Rank-Select tree that we saw in class. When we ask for the rank of a node, we are

asking (effectively) how many nodes are smaller than a given key. Here, we invert that question to ask how many nodes are larger than a given key.

The basic idea here is easiest to think of in terms of recursion / wishful thinking:

- If the target is larger than or equal to the current node's key, then simply recurse on the right subtree. All the keys larger than the target are in the right subtree.
- If the target is smaller than the current node's key, then we need to add two values: the number of keys in the right subtree PLUS 1 (for the current node), plus all the keys in the left subtree that are larger than the target.

It turns out that there is a small issue with this question, notably that it does not check in one place whether a child node is null before referencing its size. For the purpose of this exam, we believe it is clear that the size of the subtree is the best available answer—but in practice, please do be sure to check for null before deferencing a node!

7 Question Q7: Intervals

In this question, we are augmenting an AVL tree in order to easily compute the minimum frequency gap between towers. We keep the tree sorted by distance, since what we care about is identifying adjacent towers.

The tree is augmented by the minimum gap in its subtree. The question then is what additional information is needed to recursively compute the minimum gap. The most useful additional information is the frequency of the tower at the maximum location and the tower at the minimum location. This makes it easy to recursively compute the minimum gap. (There are other possible ways to organize this, but not that work with the pseudocode given.)

Notably, the minimum gap in the subtree rooted at u is going to be the smallest of three things:

- the smallest frequency gap between adjacent towers in the left subtree
- the smallest frequency gap between adjacent towers in the right subtree
- the smallest frequency gap between u and a neighbor

Hence (A,B) are computing the minimum gap between u and its left neighbor, while (C,D) are computing the minimum gap between u and its right neighbor. We then take the minimum of those two.