CS 161 W22: Recitation 5 Problems

February 2022

Exercise 0

Solve the following recurrences — that is, get a tight bound of the form $T(n) = O(f(n))$ for the appropriate function f. You may use the master theorem if it applies. Ignore floors and ceilings, and assume $T(0) = T(1) = 1$.

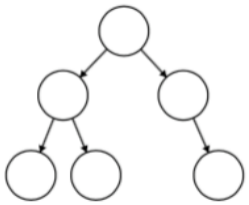
(a) $T(n) = T(n/5) + 64n$

(b) $T(n) = 4T(n/4) + n$

(c) $T(n) = T(n-2) + 9n$

Exercise 1

You are given a binary tree structure with n nodes and a set of n distinct keys (numbers). Prove or disprove: There is exactly one way to assign keys to the given tree structure such that the resulting tree is a valid binary search tree. Example: You are given the binary tree drawn below and the set of keys 1, 2, 3, 4, 5, 6. The question asks whether there is exactly one way to assign the keys to nodes such that the tree will be a binary search tree. (If you prove the statement, it should be for any input and not just this example.)



Exercise 2

(Note: This is a more difficult question from a previous offering's homework)

We say an array A of n distinct numbers is k-approximately sorted if the following property holds: for each $i = 1, 2, \ldots, n$, if $A[i]$ is the j-th smallest number in the array, then $|i - j| \leq k$. In other words, each number is at most k positions away from its actual sorted position. A k-approximate sorting algorithm takes an array of distinct numbers as input and produces an array that is k-approximately sorted.

(a) Show that given any n distinct numbers, the number of permutations of those numbers that are $\sqrt{n}$-approximately sorted is $O(n^{cn})$ for some $c < 1$. In other words, show that there are $O(n^{cn})$ possible permutations of the numbers such that each element lands within $\sqrt{n}$ distance of its sorted positions.

(b) Use part (a) to find a lower bound on the number of leaf nodes in the decision tree for any comparison-based $\sqrt{n}$-approximate sorting algorithm, and prove that any such $\sqrt{n}$-approximate sorting algorithm must have worst case complexity $\Omega(n \log n)$.

(c) Let $k > 1$ be an arbitrary constant. Can we construct an $\frac{n}{k}$-approximate sorting algorithm that does better than $O(n \log n)$? (Hint: the SELECT algorithm might help you here).

Exercise 3

You want to create a comparison-based data structure called a `FocusSet`, which stores arbitrary comparable objects. It also maintains a "focus", which is a single element that can be efficiently retrieved or deleted. The focus can be moved to the next larger or next smaller element in the data structure. More concretely, the `FocusSet` supports six operations:

- `Constructor(x)`: Creates a data structure with one element, x. The focus is initially on x.
- `Insert(x)`: Inserts element x into the data structure. Assume no duplicate elements will be added.
- `MoveNext()`: If the previously focused element was the qth largest element, then after a call to this function the new focused element will be the $(q+1)$th largest element.
- `MovePrev()`: If the previously focused element was the qth largest element, then after a call to this function the new focused element will be the $(q-1)$th largest element.
- `GetFocus()`: Returns the currently focused element.
- `PopFocus()`: If the previously focused element was the qth largest element, then after a call to this function the new focused element will be the $(q+1)$th largest element. The previously focused element will not exist in the data structure.

You can use any data structure and their operations covered in lecture. The point of this problem is to reason through different data structures on a high level, so we won't care too much about edge cases.

Part	`Insert`	`MoveNext` /`MovePrev`	`GetFocus`	`PopFocus`
(i)	$O(n)$	$O(1)$	$O(1)$	$O(n)$
(ii)	$O(1)$	$O(n)$	$O(1)$	$O(n)$
(iii)	$O(n)$	$O(1)$	$O(1)$	$O(1)$
(iv)	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n)$
(v)	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$
(vi)	$O(\log n)^*$	$O(1)^*$	$O(1)^*$	$O(\log n)^*$
(vii)	$O(\sqrt{\log n})$	$O(\sqrt{\log n})$	$O(1)$	$O(\sqrt{\log n})$
(viii)	$O(1)^*$	$O(1)^*$	$O(1)^*$	$O(1)^*$

* randomized expected runtime.

(a) For (i) through (viii), either describe how to implement FocusSet and each of the six operations in the required time, or prove that it cannot be done.

Note: (v) uses a special heap called a Fibonacci heap that can be used exactly like a binary heap, except `Insert` *runs in* $O(1)$*. This data structure will come up next week. In the mean time, how would you use a Fibonacci heap to implement this data structure? (vi) uses a data structure that some iterations of CS 106B may have covered. Read up about skip lists!*

Both are these data structures are optional and are not on the midterm. They're just here to show you how there are even cooler data structures out there, and you should take CS 166!

(b) Pepper thinks that the runtimes in (viii) are actually possible, and proposes the following implementation.

- Maintain a hash table of elements, with $O(n)$ buckets, with a hash function chosen from some universal hash family. Maintain a pointer to the current focus.
- `Insert`: insert the element into the hash table, which takes O(1) expected time.
- `MoveNext`: loop to the next element in the linked list of the current focus. If you reach the end of the list, scan forward in the hash table buckets until you find the next non-empty list and use the first node of that list as the focus. There are $O(n)$ buckets, and we showed in class that $O(n)$ of those buckets will be non-empty, so the expected time needed to scan through the buckets to find the next element is $O(1)$.
- `MoveBack`: same idea as MoveNext, but go backwards.
- `GetFocus`: return the element at the pointer.
- `PopFocus`: call `MoveNext`, and then remove the previous focus from the hash set which takes $O(1)$ expected time.

However, you proved in (a) that the runtimes in (viii) are impossible. What is incorrect about Pepper's idea? It's not the runtime analysis. Something is fundamentally wrong about this approach and the functions output wrong answers.