CSCI 705 Algorithms
Homework 3
Due 2/29/2024

Saad Mneimneh
Computer Science
Hunter College of CUNY

Readings
Based on Lectures 5, 6, and 7 and their assigned readings (see course website).

Problem 1: Divide an conquer with an oracle
An array contains $n \geq 3$ elements, all identical except one, say k (which is unknown). We would like to find i, such that $A[i] = k$. Of course, we can do this in linear time by examining every element. However, we have an orcale. This oracle can answer questions like the following in $O(1)$ time.

$$\text{What is } \sum_{i=a}^{b} A[i]?$$

Design a divide-and-conquer algorithm that can find i such that $A[i] = k$ in $O(\log n)$ time. Write a recurrence corresponding to your algorithm and solve it in any way you want to show your time bound.

Problem 2: Nesquiksort
Nesquiksort works exactly like randomized Quicksort, except that after the array is partitioned around a random pivot, the right partition gets sorted in $O(1)$ time by a magical bunny. Let $T(n)$ be the running time of Nesquicksort on an array of size n. As we did with Quicksort, assume that the elements are distinct and belong to $\{1, 2, \ldots, n\}$.

(a) Write a recurrence for $T(n)$ similar to the one we had for randomized Quicksort.

(b) Guess a solution for the recurrence and verify it by the substitution method. Show your work.

(c) Analyze Nesquicksort using the indicator random variable technique. Let X_{ij} be an indicator for the event that i and j are compared. Find $E[X_{ij}]$, which is equal to the probability that i and j are compared. *Hint*: For i and j to be compared, one of them must be the first to become the pivot among which set of elements? Once you find $E[X_{ij}]$, find the expected number of comparisons.

Problem 3: matching socks

There are n pairs of socks. Each pair has a distinct color. However, the $2n$ socks are randomly permuted, with every permutation being equally likely. To wear socks, we repeatedly try the next sock until we have a matching pair. How many socks do we expect to try? To answer this question, we will go through a series of steps.

(a) Show that the number of permutations that do not have any match within the first i socks is:

$$\frac{n!}{(n-i)!}2^i(2n-i)!$$

and based on that, find the probability that there is no match within the first i socks.

(c) Define the indicator random variable X_i as follows:

$$X_i = \begin{cases} 1 & \text{if there is no match within the first } i \text{ socks} \\ 0 & \text{otherwise} \end{cases}$$

Express the number of trials until we get a match in terms of indicator random variables, and find the expected number of trials. You might get a complicated expressions, but that's ok. Try to make it look like:

$$\frac{\sum_{i=0}^{n} \cdots}{\binom{2n}{n}}$$

(d) Analyze the result in (c), either experimentally or mathematically to obtain the asymptotic behavior of the number of trials needed to find a matching pair of socks. For instance, try to figure out what the numerator in the above form is for few values of n. In addition, replace the denominator by factorials and use Stirling's approximation.

Problem 4: Finding the k^{th} largest element
Given an array of n elements, we are interested in finding the k^{th} largest element (assume $n \geq k$). Obviously, if we sort the array, we can then identify that element in constant time. Therefore, this strategy based on sorting will generally require a total of $O(n \log n)$ time.

(a) Using a heap structure (think of it as a priority queue), describe how you can bring down the time complexity of the above task to $O(n + k \log n)$. *Note*: For large k, this is not better than sorting.

(b) Assume now that we are not interested in determining the exact value of the k^{th} largest element, but we would like to know if it is greater than some fixed value v. Describe an algorithm that can answer this question in $O(n + k)$ time by making use of a smart traversal of a heap.

Note 1: There is a way to find the k^{th} largest element in $O(n)$ time, but this requires a sophisticated algorithm that we will study next time.

Note 2: The k^{th} largest element can be found repeatedly for any k in $O(\log n)$ time once we build a balanced Binary Search Tree in $O(n \log n)$ time. We will study BST later.

Problem 5: Mergeable heaps (optinal)
Assume that heaps are actually implemented as nearly complete binary trees (and not arrays). Assume that given a pointer x to a node in the tree, left(x), right(x), and p(x) return the appropriate pointers. In addition, for a heap h, root(h) is a pointer to its root. Assume that all functions, such as insert, extract_max, heapify, etc..., are modified to handle the tree representation.

(a) Given two heaps h_1 and h_2 with n and m nodes respectively, where $n \geq m$, describe in pseudocode how you can merge the two heaps into one heap in $O(\log n)$ time. While your approach is not required to produce a nearly complete binary tree, it is not supposed to increase the height of the heap by more than a constant.

(b) Using the fact in part (a), namely that we can merge two tree-based heaps of size n in $O(\log n)$ time, consider the problem of merging k heaps $h_1, h_2, \ldots, h_k$, each of size n (and for simplicity assume that k is a power of 2). Consider two approaches:

- Sequential merge: We merge h_1 and h_2, then merge the result with h_3, then merge the result with h_4, and so on...
- Pairwise merge: We merge h_1 with h_2, h_3 with h_4, ..., and h_{k-1} with h_k. We recursively repeat the merge on the resulting $k/2$ heaps.

Compare the two approaches with respect to the height of the final heap and the time required to produce it. For the running time analysis, try to express it as a sum that you can easily manipulate.

Note: This problem is designed to help you capture expressions as sums and bound them.