CSOR W4231: Midterm Practice Solutions
Fall 2019

These problems are ungraded, and are intended as a study aid. Solutions will also be posted on canvas. They are very similar to the problems that will appear on the midterm. The midterm is closed book, closed notes; you are allowed to have . It will cover up material up to and including augmenting data structures, lecture 10 (i.e., up to and including homework 3).

Problem 1 Solve the following recurrences. Give your final answer in Θ notation. You may use any method you like and show your work.

- $T(n) = 2T(n/2) + 5$
- $T(n) = 2T(n-1)$
- $T(n) = 2T(n/3) + \log(n)$

Solution For $T(n) = 2T(n/2) + 5$, we can apply the master theorem and compare $f(n) = 5$ with $n^{\log_b(a)} = n$, and we see that $f(n) = \mathcal{O}(n^{1-\epsilon})$ (with $\epsilon = 1/2$, for example), so we have $T(n) = \Theta(n)$.

For $T(n) = 2T(n-1)$, we can simply apply this formula iteratively to see that:

$$T(n) = 2T(n-1),$$

$$T(n) = 4T(n-2),$$

$$\vdots$$

$$T(n) = 2^n T(0),$$

so $T(n) = \Theta(2^n)$. (You may always assume that $T(0) = O(1)$.)

For $T(n) = 2T(n/3) + \log(n)$, we can again apply the master theorem and we see that $f(n) = \log(n) = \mathcal{O}(n^{\log_3(2)-\epsilon})$ for some positive ϵ (anything strictly between 0 and $\log_3(2)$ will do), so we have $T(n) = \Theta(n^{\log_3(2)})$.

Problem 2 Describe how to modify the QUICKSORT algorithm to achieve $O(n \log n)$ runtime *deterministically* in the worst-case. You may assume all elements are distinct. Briefly justify this running time. You may use the PARTITION subroutine and its running time bound as described in class.

Solution Based on the observation, we can see that the worst cases occurs when we partition the array into unbalanced parts with sizes $n-1$ and 0, in which case, the algorithm may take $O(n^2)$ time. In RANDOMIZED-QUICKSORT, there is still the possibility that the randomly picked element is either the minimum or the maximum in the array. To reach $O(n \log n)$ time deterministically, we should divide the array more evenly. We know that the median of an unsorted array can be found in linear time by using SELECT algorithm (with median of medians). So in PARTITION, we can find the median first and then use it as the pivot to do the partition. The other part of the algorithm is unchanged so the correctness can be guaranteed.

```
funcion Modified-Partition(A,left,right)
    i = Select(A, left, right, ceil((left + right) / 2))
    swap(A[i], A[r])
    Partition(A, left, right)
```

Running Time. Since the pivot in partition is always the median, the partition would be balanced all the time. And the recursive formula can be write as $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + \Theta(n)$, which is $T(n) \leq 2T(n/2) + \Theta(n)$. By master method case 2 the solution is $O(n \log n)$.

Problem 3 Suppose you are given n integers between 0 and n^d. Give the running times for sorting them using each of the following algorithms (as functions of n and d in asymptotic notation). You do not need to justify your answers.

- Mergesort
- Counting sort
- Radix sorting using expressions of each number as d digits in base n

Solution Mergesort will take time $\Theta(n \log n)$. Counting sort will take time $\Theta(n^d + n)$. Radix sort will take time $\Theta(dn)$.

Problem 4 Consider a uniformly random permutation of 1 through n. Compute the expected number of pairs i, j $(i < j)$ such that i still appears before j in the permutation. For example, if we permute 1 through 4 as 3214, the pairs $(1,4)$, $(2,4)$, and $(3,4)$ are the pairs that satisfy this property. Justify your answer.

Solution For each pair $i < j$, we let $I_{i,j}$ denote the indicator random variable that $= 1$ if i still appears before j, and $= 0$ otherwise. Then the number of pairs that maintain their relative ordering is a random variable X that can be written as the sum over all $i < j$ of $I_{i,j}$:

$$X = \sum_{i<j} I_{i,j}.$$

The expectation of this sum is the sum of the expectations, so we consider $\mathbb{E}[I_{i,j}]$, which is simply the probability that $I_{i,j} = 1$, which is $\frac{1}{2}$. This is because exactly half of all permutations switch the relative order of i and j, while exactly half maintain it. Thus, $\mathbb{E}[X]$ is equal to:

$$= \sum_{i<j} \mathbb{E}[I_{i,j}] = \sum_{i<j} \frac{1}{2} = \frac{1}{2}\binom{n}{2}.$$

Problem 5 Prove that, in order to find the *median* of an unsorted array A, one needs to perform at least $n-1$ comparisons (for any comparisons-based algorithm).[1]

Solution First, we claim that, in order to correctly determine if any particular element x is the median, it must participate in at least 2 comparisons.

To prove this claim, suppose that we claim that some element x is the median but it only participated in one comparison with another element y. Then, even if we know the rank of every other element in the array, including y, we cannot know the rank of x with any certainty. For example, if we learn that $x > y$, then we can only know that x's rank is bigger than y's, but there is no way to conculde that x is the median.

Now assume that a correct algorithm performs $n-1$ comparisons. With those $n-1$ comparisons, there must be at least $2n-2(n-1)=2$ elements that are only compared at most once. Thus, by the claim above, we cannot correctly compute the median.

As a concrete example, consider three numbers a, b, c, with $n-1=2$ comparisons, we might learn $a < b$ and $a < c$. From this, we can conclude that a is the minimum, but we have no idea if b or c is the median.

Problem 6 Consider a *min*-heap that stores $n \geq 100$ distinct integers. What are the nodes of the min-heap that may contain the 3rd smallest integer (i.e., the rank 4 element in increasing order of the elements in the min-heap)? Deduce an exact number of such possible nodes. How much time would it take to find this rank-4 integer ($O(\cdot)$ answer is ok)? No need to justify your answer.

Solution The 4th smallest integer can only be in the first four levels of the heap (we treat the root as level 1 here). In a min-heap, since the root node is the smallest, the 2nd smallest node must be in level 2 since there can be only one element smaller than it (but an important thing to note here is that you don't know before hand which element on the 2nd level is the 2nd smallest element and so, you'll need to check both of them). The 3rd smallest element could then either be in level 2 as a child of the root, or it could be a child of the 2nd smallest node, in which case, it resides in level 3 and so on. Thus, the maximum number of nodes to check is 14 (eight in level 4, four in level 3 plus two in level 2 - we needn't check the root since it is definitely the smallest element in the heap).

It thus takes $O(1)$ time to find this integer. The most basic/intuitive way in which you can achieve this time complexity is the following. Let $A[1], A[2], \ldots A[n]$ be the min-heap. As specified above, you only need to check the elements from $A[2]$ to $A[15]$. So, you can take these 14 elements, sort them in time $O(14 \log 14) = O(1)$ and then return the third minimum element from the sorted list.

[1]For fun, you can try to prove a higher lower bound, $\geq n$.