Basic Algorithms CSCI-UA.0310

Homework 3

Due: February 22nd, 4 PM EST

Instructions

Please answer each **Problem** on a separate page. Submissions must be uploaded to your account on Gradescope by the due date and time above.
Please note that no late submission will be accepted for this homework.

Problems to submit

Problem 1 (5+12+5 points)

Let A be a sorted array with n distinct elements. Consider the following algorithm that finds an integer $i \in \{j \dots k\}$ such that $A[i] = x$, or returns **FALSE** if no such integer i exists.

```
1 FIND($A, j, k, x$)
2     If $j > k$ Return FALSE
3     Define $i := \dots$
4     If $A[i] = \dots$ Return $\dots$
5     If $A[i] < \dots$ Return FIND($\dots$)
6     Return FIND($\dots$)
```

(a) Fill in the blanks to complete the algorithm.

(b) Write the recurrence for the running time of the algorithm. Draw the corresponding recursion tree and find the explicit answer to the recursion.
Your answer must use $\Theta(.)$ notation, i.e., you should obtain both an upper bound and a lower bound (proof by strong induction is NOT needed).

(c) We assumed that the elements of A are distinct. Does the algorithm still work if the elements of A are not necessarily distinct? Justify your answer.

Problem 2 (5+15+5 points)

Given an array $A[1, 2, \dots, n]$, recall that the PARTITION function in QUICKSORT takes the last element $A[n]$ as the pivot and partitions A into two parts: the left part consisting of the elements which are less than or equal to the pivot and the right part consisting of the rest.

(a) Assume that the elements in A are all the same. What is the running time of QUICKSORT on A? Explain your answer.

You should be convinced by part (a) that the running time is not good! To overcome this issue, we modify the PARTITION function so that after running it, any input array A is partitioned into three parts: the left part consisting of all elements strictly less than the pivot, the middle part consisting of all elements equal to the pivot, and the right part consisting of the rest (i.e., all elements strictly larger than the pivot).

(b) Write down the pseudo-code for this modified version of the PARTITION function. Note that your algorithm must be an in-place algorithm (no auxiliary array should be used) with $\Theta(n)$ time complexity.

Hint: Define three indices: i as the end of the left part, k as the end of the middle part, and j denoting the current index which is used in the for-loop. Modify the PARTITION function discussed in the lecture for building both the left part and the middle part iteratively.

(c) Use the modified version of the PARTITION function obtained in part (b) in the QUICKSORT algorithm. Let A be an array consisting of n identical elements. What is the running time of this version of QUICKSORT on A? Explain your answer.

Problem 3 (20 points)

Let A be an array of n integers, where n is divisible by 4. Describe an algorithm with $O(n)$ time complexity to decide whether there exists an element that occurs more than $n/4$ times in A.

(a) Show that if such an element exists in A, it must be either the $(n/4)$-th, or the $(2n/4)$-th, or the $(3n/4)$-th smallest elements.

(b) Use part (a) to derive your algorithm.

Problem 4 (13 points)

Consider the recursion $T(n) = 2T(n/2) + n$ for $n \geq 2$, and $T(1) = 1$ that was solved in the lecture. What is the answer to $T(n)$?
Below, you are provided a reasoning showing that $T(n) = O(n)$. Of course, it is not correct (as seen in the lecture). Find the mistake and justify why the following reasoning is not correct. For simplicity, you may assume that both n and k (the variable used in the inductive step) are powers of 2.

We use strong induction to show that $T(n) = O(n)$, where n is a power of 2:

- Base case: One can easily check that the statement holds for the base case $n = 1$; We have $T(1) = 1$, thus, we get $T(1) = O(1)$.
- Inductive step:
  1. Assumption: Assume that the statement holds for $n = k/2$, i.e., we have $T(k/2) = O(k/2)$.
  2. Conclusion: Based on the assumption, we prove that the statement holds for $n = k$, i.e., we show that $T(k) = O(k)$:

$$\begin{aligned} T(k) &= 2T(k/2) + k && (1)\\ &= 2O(k/2) + k && (2)\\ &= 2O(k) + k && (3)\\ &= O(k) + k && (4)\\ &= O(k). && (5) \end{aligned}$$

For full credit, you have to provide a full justification for your arguments.

Problem 5 (20 points)

Given a positive integer n, we can easily compute 2^n by using $\Theta(n)$ multiplications. Explain an algorithm that computes 2^n using $\Theta(\log n)$ multiplications. Justify why your algorithm uses $\Theta(\log n)$ multiplications.

(a) Consider the case where n is a power of 2, i.e., $n = 2^k$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k} = \left(2^{2^{k-1}}\right)^2$.

(b) Consider the case where $n = 2^k + 2^{k-1}$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k+2^{k-1}} = 2^{2^k} \times 2^{2^{k-1}}$.

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

Bonus problem

The following problems are optional. They will NOT be graded and they will NOT appear on any of the exams. Work on them only if you are interested. They will help you to develop problem solving skills for your future endeavors.
You are highly encouraged to think on them and discuss them with your peers on a separate thread on the course page on Campuswire.

Bonus Problem 1

(a) Let A be an array of n integers, where n is divisible by 2. Use an idea similar to Problem 3 of this homework to develop an algorithm with $O(n)$ time complexity to decide whether there exists an element that occurs more than $n/2$ times in A.

(b) Develop the algorithm for part (a), i.e. deciding the existence of an element occurring more than $n/2$ times in A, but do not use the idea of Problem 3. In other words, your algorithm must NOT deal with the smallest k elements of the array. Your algorithm must run in $O(n)$ time with $O(1)$ space complexity.

(c) Generalize your idea in part (b) to solve Problem 3 of this homework, i.e. deciding the existence of an element occurring more than $n/4$ times in A, without using the smallest k elements of the array. Your algorithm must, again, run in $O(n)$ time with $O(1)$ space complexity.