CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Homework #1

Deadline: Thursday, September 19th, 11:59 PM

Homework Policy

- If you leave a question completely blank, you will receive 25% of the grade for that question.
- You are allowed to discuss the homework problems with other students in the class. **But you must write your solutions independently.** You may also consult all the materials used in this course (notes, textbook, further reading, etc.) while writing your solution, but no other resources are allowed.
- Do not forget to write down your name and whether or not you are using one of your two extensions. Submit your homework on Canvas.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "sort the array in $\Theta(n \log n)$ time using merge sort".
- Remember to always prove the correctness of your algorithms and analyze their running time (or any other efficiency measure asked in the question). See "Homework 0" for an example.

Problem 1. Let us revisit the chip testing problem. Recall that in this problem, we are given n chips which may be *working* or *defective*. A working chip behaves as follows: if we connect it to another chip, the original chip will *correctly* output whether the new connected chip is working or is defective. However, if we connect a defective chip to another chip, it may output *any arbitrary answer*.

In the class, we saw that if *strictly* more than half the chips are working, then there is an algorithm that finds a working chip using $O(n)$ tests. In this homework, we examine what will happen if the number of working chips is *equal* to the defective ones.

(a) Prove that even when we only have a single working chip and a single defective chip (i.e., $n = 2$), there is *no* algorithm that can find the working chip in general. **(10 points)**

(b) Give an algorithm that for any *even* number of chips n, assuming that the number of working chips is equal to the defective ones, can output a *pair of chips* such that in this pair, one of the chips is working and the other one is defective (the algorithm does not need to identify which chip is working/defective in this pair – this is crucial by part (a)). **(15 points)**

Example: As an example for part (b), suppose the input is $[W, W, W, D, D, D]$. Then the algorithm should output a pair consisting of one of the first three chips and one of the last three chips. The same type of answer is also valid if the input is instead $[D, D, D, W, W, W]$.

Hint: Recall that we already analyzed the case when number of working chips is equal to the defective ones in the main algorithm – you only need to modify that algorithm slightly for this problem.

Problem 2. This question reviews asymptotic notation. You may assume the following inequalities for this question (and throughout the course): For any <u>constant</u> $c \geq 1$,

$$(\log n)^c = o(n) \quad , \quad c^n = o((c+1)^n) \quad , \quad (n/2)^{(n/2)} = o(n!) \quad , \quad n! = o(n^n).$$

(a) Rank the following functions based on their asymptotic value in the increasing order, i.e., list them as functions $f_1, f_2, f_3, \ldots, f_{16}$ such that $f_1 = O(f_2)$, $f_2 = O(f_3), \ldots, f_{15} = O(f_{16})$. Remember to write down your proof for each equation $f_i = O(f_{i+1})$ in the sequence above. **(15 points)**

$\log\log n$	$10^{10^{10}}$	$2^{\sqrt{\log n}}$	n^2
$n^{1/\log\log n}$	$2n$	10^n	$n^{\log\log n}$
2^n	$10n$	$n!$	$\sqrt{n}$
$n/\log n$	$\log(n^{100})$	$(\log n)^{100}$	n^n

Hint: For some of the proofs, you can simply show that $f_i(n) \leq f_{i+1}(n)$ for all sufficiently large n which immediately implies $f_i = O(f_{i+1})$.

(b) Consider the following six different functions $f(n)$:

$$n! \qquad \log n \qquad 2^n \qquad n^2 \qquad 10 \qquad 2^{2^n}.$$

For each of these functions, determine which of the following statements is true and which one is false. Remember to write down your proof for each choice. **(10 points)**

- $f(n) = \Theta(f(n-1))$;
- $f(n) = \Theta(f(\frac{n}{2}))$;
- $f(n) = \Theta(f(\sqrt{n}))$;
- $f(n) = \Theta(f(\log n))$.

Example: For the function $f(n) = n!$, we have $f(n-1) = (n-1)!$. Since $\lim_{n\to\infty} \frac{n!}{(n-1)!} = \infty$, we have $n! \neq \Theta((n-1)!)$, so the first statement is false for $n!$.

Problem 3. In this problem, we analyze a *recursive* version of the *insertion sort* for sorting. The input as before is an array A of n integers. The algorithm is as follows (in the following, $A[i:j]$ refers to entries of the array with indices between i and j, for instance $A[2:4]$ is $A[2], A[3], A[4]$):

Recursive Insertion Sort:

```
1. If $n = 1$ return the same array.
2. Recursively sort $A[1 : n-1]$ using the same algorithm.
3. For $i = n-1$ down to 1 do:
     • If $A[i+1] < A[i]$, swap $A[i+1]$ and $A[i]$; otherwise break the for-loop.
4. Return the array $A$.
```

We now analyze this algorithm.

(a) Use *induction* to prove the correctness of the algorithm above. **(10 points)**

(b) Analyze the runtime of this algorithm. **(5 points)**

(c) Suppose we are *promised* that in the input array A, each element $A[i]$ is *at most* k indices away from its index in the sorted array. Prove that the algorithm above will take $O(nk)$ time now. **(10 points)**

Example. Consider the array $A = [3, 2, 4, 1, 5]$ – in this array, each element is at most 3 indices away from its correct position in the sorted order (so $k = 3$ above). In particular 3 is two indices away, 2 is zero index away, 4 is one index away, 1 is three indices away, and 5 is zero index away.

Problem 4. You are hired to help the rebels fight the evil empire in Star Wars (!). The rebels have n space ships and each space ship i ($1 \leq i \leq n$) has a certain power p_i. Moreover, the empire has m bases where each base j ($1 \leq j \leq m$) has a defensive power d_i and gold g_i. You know that each space ship can attack every base with defensive power *strictly* smaller than the ship's own power and collect its golds.

The rebels need to know that, for each of their space ships, what is the maximum amount of gold this space ship can collect. Design an algorithm with running time $O((n+m) \cdot \log m)$ for this task. **(25 points)**

Example. The correct answer for the following input with $n = 4$ and $m = 5$ is $[8, 5, 7, 11]$:

power of space ships:	$p_1 = 5$,	$p_2 = 3$,	$p_3 = 4$,	$p_4 = 9$;	
defense power of bases:	$d_1 = 3$,	$d_2 = 8$,	$d_3 = 4$,	$d_4 = 0$,	$d_5 = 1$;
amount of gold in bases:	$g_1 = 2$,	$g_2 = 3$,	$g_3 = 1$,	$g_4 = 3$,	$g_5 = 2$.

Hint: Modify the binary search algorithm so that given a sorted array A and an integer t, find the *largest index* i in A such that $A[i] < t$. Then use this algorithm to find the "strongest" base each space ship can attack (you need to preprocess the bases and their golds accordingly).

Challenge Yourself. A sorting algorithm is called *in-place* if it does not use an extra space for sorting the array. Consider the following *in-place* sorting algorithm. For simplicity, we assume the input array A consists of n *distinct* integers. **(0 points)**

1. If $n \leq 3$, sort the array by brute force (by considering all $3! = 6$ cases).
2. Partition A into three approximately equal-sized regions: $L_1 = A[1...\lfloor \frac{n}{3} \rfloor]$, $L_2 = A[\lfloor \frac{n}{3} \rfloor + 1...\lceil \frac{2n}{3} \rceil]$, $L_3 = A[\lceil \frac{2n}{3} \rceil + 1...n]$.
3. Recursively sort $L_1 \cup L_2$, *then* $L_2 \cup L_3$, and finally $L_1 \cup L_2$ *again* (note that these sorts are done in-place so the parts L_1, L_2, and L_3 are *recomputed* after each sort).

Prove the correctness of this algorithm and analyze its time complexity (write a recurrence and solve it to find a tight asymptotic bound).

Fun with Algorithms. In this problem, we briefly look at the concept of *parallelism* in algorithm design through the lens of the chip testing problem. Suppose that instead of testing the chips *sequentially* in this problem, we could test them *in parallel.* More precisely, in each *day*, we can pair of the chips together and test them all together in the same day; the only constraint is that in every day, each chip can be tested at most once (for instance, we can test both pairs (1,2) and (3,4) in one day, but we cannot test (1,2) and (2,3) in one day because that requires testing the second chip twice).

Design an algorithm that assuming the number of working chips is *strictly* more than the defective ones, finds a working in chip in $O(\log n)$ days. Note that in the context of this problem, the main measure of efficiency of the algorithm is no longer the number of the tests it performs but rather the number of days it needs for testing the chips. **(0 points)**