CS 3510: Homework 1B

Due on Saturday, January 29

Professor Faulkner

CS 3510 Staff

Notes:

This assignment is due on **11:59 PM EST, Saturday, January 29, 2022**. You may turn it in 1 day late for no penalty or 2 days late for a 10% penalty. On-time submissions receive 3% extra credit. Note that a late submission means late feedback, which means less time to study before an exam.

You should submit a typeset or *neatly* written pdf on Gradescope. The grading TA should not have to struggle to read what you've written; if your handwriting is hard to decipher, you will be required to typeset your future assignments.

You may collaborate with other students, but any written work should be your own. Write the names of the students you work with at the top of your assignment.

Unless otherwise noted, you should *always* justify your answer. Please give a written description for all algorithms; algorithms may NOT be specified solely in pseudocode. Whenever you are asked to give an algorithm, your answer should have three parts. First, describe the algorithm in words clearly. Second, prove the correctness of your algorithm. Finally, state and prove the runtime of your algorithm.

When the base of a log is unspecified, it is assumed to be base 2.

Problem 1

Divide and Conquer in $O(\log(n))$ (33 points)

Given an array $A[n]$ whose elements are odd numbers (positive and negative). Suppose that n is a power of 2, the elements in $A[n]$ are sorted and distinct. Design an algorithm to check whether there is at least one element in the array satisfies $A(i) = 2i + 5$, where $1 \leq i \leq n$ is the index of the i^{th} element in the array. If such element exists, your algorithm should return "yes"; otherwise, return "no". The running time of your algorithm is required to be $O(\log(n))$. You need to justify both the correctness and the running time of your algorithm.

We could create an algorithm that is simply a binary search. In each iteration of the binary search we are going to have parameters of the array $A[n]$, an upper bound, and a lower bound. We first calculate the midpoint by taking the average of the upper and lower bounds. We are going to test if $A[mid] = 2(mid) + 5$ if this is true we are going to return "yes". Otherwise we are going to test if $A[mid] > 2(mid) + 5$ or if $A[mid] < 2(mid) + 5$ if it was greater than then we are going to call the search again with bounds lower bound and mid, if it was less than then we are going to call search again with bounds mid and upper bound. Recursively perform this until we arrive at either the "yes" condition or until the lower bound equals the upper bound for which we will return "no". Our checker for the condition of $A[mid] = 2(mid) + 5$ does not affect the run time of the binary search at all so the run time of the algorithm is simply the run time of the binary search algorithm. We proved in class that the run time of the binary search algorithm is $O(\log(n))$. Therefore, the run time of the solution algorithm is $O(\log(n))$.

Problem 2

Function Analysis (33 points)

Assume n is a power of 3. Assume we are given an algorithmic routine $f(n)$ as follows:

```
function f(n):
if n > 1:
   for i in range(n):
      for j in range(n):
        print("dividing")
   f(n/3)
   f(n/3)
   f(n/3)
else:
   print("conquered")
```

Part A (10 points)

What is the running time $T(n)$ for this function $f(n)$? You don't need to justify your answer.

Part B (15 points)

How many "dividing"'s will this function print? How many "Conquered"' s will this function print? Please provide the exact number in terms of the input n. What is their relation to running time $T(n)$? You need to justify your answer.

Part A: $T(n) = 3T(n/3) + O(n^2)$ and using masters theorem we get that $3 < 3^2 = 9$, and we arrive to $T(n) = O(n^2)$

Part B: For each recursive call of the $f(n)$ the term $f(n/3)$ gets called 3 times and subsequent recursive calls within the $f(n/3)$ gets called multiples of three more times. Since we are told that n is a power of 3 so we can write it as $n = 3^i$ which we can then solve for i as $i = \log_3(n)$. We can consider a base case for when $n = 3$ then $i = 1$. If we run $f(n)$ we get that "dividing" would be printed 3^2 times which I will rewrite as $3^0(3^i)^2$. If we proceed to if $n = 3^2$ and $i = 2$, then we will see that "dividing" will be printed $9^2 = (3^2)^2$ times for $f(n)$ alone, but then the recursive calls of $3f(n)$ will print $3(3^0(3^1)^2)$ for a total of $3^2)^2 + 3^0(3^i)^2$ times. We can then generalize the amount of times "dividing" will print as $3^0(3^i)^2 + 3^1(3^{i-1})^2 + ... + 3^{i-1}(3^1)^2 = 3^0(3^{\log_3(n)})^2 + 3^1(3^{\log_3(n)-1})^2 + ... + 3^{\log_3(n)-1}(3^1)^2$. It will stop at $3^{i-1}(3^1)^2$ because if it were to continue on one more iteration it would be that $n = 1$ and it would instead print "conquered". Then for "conquered" it will only be printed once per call on the last call of all of the branches. Again if we consider when $n = 3$ then "conquered" is printed 3 times, if $n = 9$ then "conquered" will be printed 9 times. Therefore we can generalize and say that "conquered" will be printed n times. The entirety of the printing in the algorithm is consumed by printing "dividing" because it is going to be printed n^2 times for the respective call $f(n)$, printing "conquered" while happening n times will actually have no affect on the run time of the algorithm comparatively.

Problem 3

Generalized Merge Operation (34 points)

Given k sorted lists, each of size n, we want to design an algorithm that will do a k-way merge and produce one final sorted list with the elements of all of the individual lists. In other words, we want to generalize the Merge routine of MergeSort to work on k lists (not just 2). **Note:** You are allowed to use the Merge routine as a black box, and for running time analysis, you may assume k is a power of 2. Remember that the merge routine has a runtime of $O(n)$

Design an efficient Divide & Conquer algorithm for this problem. What is its running time in Big O?

We could sort the k lists into a final list if we took two groups from the k lists, let's say $[0, k/2]$ and $[k/2 : k]$. Then we could continue to split the set of lists into further groups of two and arrive at final groups that only contain two lists each given that k is a power of 2. We could then use this final group with 2 lists and perform the normal merge sort between the two already sorted lists. It is unnecessary to perform a full merge sort because the two lists in the group are already sorted themselves so we just need to merge the two lists. Then we can iterate back through the group where we will then have two lists of size 2n that will merge into a list of size 4n until we arrive at the final list of size kn. The base case for this is if there is only 1 list in which it will already be sorted, and for 2 lists then we just need to merge the two lists together. Therefore, the run time of this algorithm is $\frac{k}{2}O(n)+\frac{k}{2^2}O(2n)+\frac{k}{2^3}O(2^2n)+...+\frac{k}{k}O(\frac{k}{2}n) = 2^{\log_2(k)+1}O(n) = 2k*O(n) = O(2kn) = O(kn)$

Problem 4

Recurrence Practice (0 points)

Do not turn these in! We will not grade them. They are just to help you practice handling recurrence relations.

Solve the following recurrences by hand:

1. $T(n) = 5T(n/3) + O(1)$
2. $T(n) = T(n-2) + O(n)$
3. $T(n) = 2T(n/3) + O(n^3)$

Solve the following recurrences using the Master Theorem:

1. $T(n) = 8T(n/2) + 1001n^2$
2. $T(n) = 2T(n/2) + 2n^2 + 4n$
3. $T(n) = 2T(n/2) + O(n)$