CSCI270 Homework 4

Jacob Ma

September 29, 2022

1 Problem 1

Problem 1

Solve the following recurrences. In all cases, assume $T(1)$ is some positive constant. You should be able to find a matching upper and lower bound on $T(n)$. In other words, your answer should be of the form $T(n) = \Theta(\cdots)$. You do not need to provide formal proofs, just a brief justification for your answer. As we often due in class, you may sacrifice formality by ignoring issues with rounding integers; in all these recurrences, those issues will not matter.

(a) $T(n) = 9T(n/3) + c_1 \cdot n^2$

(b) $T(n) = 10T(n/5) + c_2 \cdot n^{1.5}$

(c) $T(n) = 3T(n/4) + c_3 \cdot \sqrt{n}$

(d) $T(n) = T(n/3) + T(\frac{2}{3}n) + c_4 \cdot n$

(e) $T(n) = T(n/4) + T(n/2) + c_5 \cdot n$

Solution: 1 – (*a*)

According to the problem, we have $a_1 = 9, b_1 = 3$, and $f(n) = c_1 \cdot n^2$. Thus, $n^{\log_{b_1} a_1} = n^{\log_3 9} = n^2$.
Because $f(n) = \Theta\left(n^2\right)$, according to Master Theorem, we apply case 2.
$T(n) = \Theta(n^{\log_{b_1} a_1} \cdot \log n) = \Theta\left(c_1 \cdot n^2 \cdot \log n\right) = \Theta(n^2 \cdot \log n)$.

Solution: 1 – (*b*)

According to the problem, we have $a_2 = 10, b_2 = 5$, and $f(n) = c_2 \cdot n^{1.5}$. Thus, $n^{\log_{b_2} a_2} = n^{\log_5 10}$.

Because $5^{1.5} = \sqrt{125} > \sqrt{100} = 10$, so $f(n) = \Omega\left(n^{\log_5 10+\epsilon}\right)$ for some small $\epsilon > 0$. And we have

$$\begin{aligned} a_2 f\left(\frac{n}{b}\right) &\leqslant c \cdot f(n) \\ 10 f\left(\frac{n}{5}\right) &\leqslant c \cdot f(n) \\ 10 \cdot c_2 \cdot \left(\frac{n}{5}\right)^{1.5} &\leqslant c \cdot c_2 \cdot n^{1.5} \\ \frac{2\sqrt{5}}{5} &\leqslant c \qquad \text{This will hold for any } c \in \left[\frac{2\sqrt{5}}{5}, 1\right) \end{aligned}$$

Thus, according to Master Theorem, this satisfies the condition to apply case 3.
$T(n) = \Theta(f(n)) = \Theta\left(c_2 \cdot n^{1.5}\right) = \Theta\left(n^{1.5}\right)$.

Solution: 1 – (c)

According to the problem, we have $a_3 = 3, b_3 = 4$, and $f(n) = c_3 \cdot \sqrt{n}$. Thus, $n^{\log_{b_3} a_3} = n^{\log_3 4}$.
We are trying to compare $f(n) = c_3 \cdot \sqrt{n}$ with $n^{\log_3 4}$. Because $4^{\frac{1}{2}} < 3$, thus $\frac{1}{2} < \log_4 3 \implies f(n) = O\left(n^{\log_4 3-\epsilon}\right)$ for some $\epsilon > 0$.
Thus, applying Master Theorem Case 1, $T(n) = \Theta\left(n^{\log_{b_3} a_3}\right) = \Theta\left(n^{\log_4 3}\right)$.

Solution: 1 – (d)

Let's consider a certain step inside the recurrences, let's call it k^{th} step, and we firstly consider $k = 1$, while we need to calculate $T(n)$ in this step. Apart from the recurrence step, $f(n) = c_4 \cdot n$, thus during this step, $T(n)$ took $\Theta(n)$ time to run through the $f(n)$, and then do the recurrence by dividing $T(n)$ into $T\left(\frac{n}{3}\right)$ and $T\left(\frac{2}{3}n\right)$.
If we consider $T\left(\frac{n}{3}\right)$, we would find it takes $\Theta\left(c_4 \frac{n}{3}\right) = \Theta\left(\frac{n}{3}\right)$ time during this step; consider $T\left(\frac{2}{3}n\right)$, it takes $\Theta\left(c_4 \frac{2}{3}n\right) = \Theta\left(\frac{2}{3}n\right)$ time during this step. Thus, the two recurrence step actually took $\Theta\left(\frac{n}{3}\right) + \Theta\left(\frac{2}{3}n\right) = \Theta(n)$ during the next step, 2^{nd} step. Thus, we know that during the $k+1^{th}$ step, the runtime is equivalent to the runtime at the previous k^{th} step.
Thus, we found that during each iteration of the recurrence, if there are still $T(\frac{n}{3})$ and $T(\frac{2}{3}n)$ fully left to execute, each iteration would take $\Theta(n)$ time.
We also know that $\log_3 n < \log_{\frac{3}{2}} n$ (in other words, after certain amount of execution, $T\left(\frac{n}{3}\right)$ would reaches 0 while $T\left(\frac{2n}{3}\right)$ has not), thus the lower bound of number of total iterations where none of $T\left(\frac{n}{3}\right)$ and $T\left(\frac{2}{3}n\right)$ reaches 0 is determined by $\log_3 n$, while the upper bound determined by $\log_{\frac{3}{2}} n$. Thus, we have

$$\log_{\frac{3}{2}} n \cdot \Theta(n) \geqslant T(n) \geqslant \log_3 n \cdot \Theta(n)$$

Thus, $T(n) = \Theta(n \cdot \log n)$

Solution: 1 – (e)

Similar to the question $1\,(d)$ above, we consider a certain step inside the recurrence, let's call it k^{th} step. And we take $k = 1$ as an example. During this step, the algorithm takes $f(n) = \Theta(c_5 \cdot n) = \Theta(n)$ runtime, and divides the works into $T\left(\frac{n}{4}\right)$ and $T\left(\frac{n}{2}\right)$.
Consider $T\left(\frac{n}{4}\right)$ and $T\left(\frac{n}{2}\right)$, they individually gives $\Theta\left(c_5 \cdot \frac{n}{4}\right)$ and $\Theta\left(c_5\left(\frac{n}{2}\right)\right)$ runtime individually before they distribute the works into the next recurrence step. Thus, during this step, it takes $\Theta\left(c_5 \cdot \left(\frac{n}{4} + \frac{n}{2}\right)\right) = \Theta\left(\frac{3n}{4}\right)$ in this step. Thus, we have such function, in the k^{th} step, it will take $\Theta\left(\left(\frac{3}{4}\right)^{k-1} n\right)$ amount of runtime. We also know that $\log_4 n < \log_2$, meaning it will at most going through $\log_2$ iterations. Thus, the total runtime would be

$$\begin{aligned}
T(n) &= \sum_{k=1}^{\log_2 n} \Theta\left(\left(\frac{3}{4}\right)^{k-1} n\right) \\
&= \Theta\left(\sum_{k=1}^{\log_2 n} \left(\frac{3}{4}\right)^{k-1} n\right) \\
&= O\left(\sum_{k=1}^{\infty} \left(\frac{3}{4}^{k-1}\right) n\right) \\
&= O\left(n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^{i}\right) \\
&= O(n \cdot 3) \\
&= O(n)
\end{aligned}$$

We also know that the algorithm would take at lest c_5^n because it needs to do so in the first recurrence, so $T(n) = \Omega(n)$. Thus, $T(n) = \Theta(n)$.

2 Problem 2

Problem 2

Consider an array of integers $A = [a_1, \ldots, a_n]$ with $a_1 < a_n$.

(a) Show that there exists an index $i \in \{1, \ldots, n-1\}$ such that $a_i < a_{i+1}$. In other words, there exist two consecutive entries of the array of which the first is smaller than the second.

(b) Give and analyze an algorithm which finds such a pair of entries in $O(\log n)$ time. Your algorithm should output $i \in \{1, \ldots, n-1\}$ such that $a_i < a_{i+1}$. You may assume that the array A has already been loaded into memory. You may also assume that comparing two entries of the array is a primitive operation (i.e., takes constant time).

[Hint: The first part may be useful here.]

Proof. We prove $2\,(a)$ by contradiction.
Assume there does not exist two consecutive entries of the array of which the firs is smaller than the second, meaning for every two consecutive index i and $i+1$, $a_i \geqslant a_{i+1}$.

Thus, applying this assumption, we have

$$a_1 \geqslant a_2 \geqslant a_3 \geqslant \cdots \geqslant a_{n-1} \geqslant a_n$$

In this case, $a_1 \geqslant a_n$, contradicting $a_i < a_n$.

The conclusion follows the contradiction. □

Solution: 2 – (*b*)

Here is the algorithm:

```
1   int findIndex(int l, int r, int[] A){
2       int m = (i+j)/2;
3       if (l == r){
4           return null; // Invalid input of size 1, no 2 consecutive index could be found
5       }
6       else if (A[l] < A[l+1]){
7           return l; // Directly return if A[l] < A[l+1]
8       }
9       else if (A[l] < A[m]){
10          return findIndex(l, m, A); // If not, we perform finding such index in a new sub-array
                where the last element > first element
11      }
12      else{
13          return findIndex(m, r, A); // Perform the function on a sub-array where last element >
                first element
14      }
15  }
```

Proof. We first prove this algorithm terminates. Because this is a finite array of size n, and in the worst case, we are performing the recurrence on the next function of the sub array of size $\frac{n}{2}$, and return with the base case $n = 1$ or $n = 2$, thus the algorithm would absolutely terminate with finite recurrence calls, which is $\log_2 n$ at most.

We than prove this algorithm is correct. For the base case, we know that there must exist two consecutive entries of the array if the array has the last element > the first element. Thus, if we have a sub-array of size 2 with such condition, we can come up with the solution.

In each recurrence call, we are creating a new sub array $A[l]$ to $A[r]$, where the last element is guaranteed to be greater than the first element, $A[l] < A[r]$. For such sub array, we have proven that there must exist a solution in such array. Thus, the algorithm will return a correct value and the algorithm will terminate.

As for the runtime of the algorithm, we now that in each call of the function, every comparison and operations are linear $O(1)$, and than call the recurrence function. We also know that, this algorithm would call at most $\log_2 n$ times, since we are calling the recurrence function on a sub array of $\frac{1}{2}$ original size each time and terminate at size = 1 (invalid input) or 2 (worst case for valid input). Thus, we have

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

Applying Master Theorem Case 2, $\left(1 = \Theta\left(n^0\right)\right)$, we have $T(n) = \Theta(\log n)$. □

3 Problem 3

Problem 3

Consider the following situation. At the end of the semester, David has two arrays a, b of weighted average scores of the students in the two sections of CSCI 270. Let's say that both sections have the same size n, which you may assume is a power of 2 if it simplifies things for you.

He has already sorted the students by score, so $a[1] \leq a[2] \leq \cdots \leq a[n]$, and $b[1] \leq b[2] \leq \cdots \leq b[n]$. However, he has not yet done any comparison of students between the two sections: it is possible that all numbers in the a array are larger, or all numbers in the b array are larger, or they are mixed in any possible way.

David now wants to know how the "most middle" student performed. That is, he wants to find a student i (out of the $2n$ total) such that at least n students had scores no higher than i, and at least n students had scores no lower than i. There may be multiple such students; for instance, if all students achieved the same score, then any student would do. Give an algorithm that always runs in time $O(\log n)$ and finds one such student, by outputting the array (a or b) and the index i where the student occurs in that array.

Consider the algorithm below:

```
int findMedian(int[] a, int[] b, int l1, int h1, int l2, int h2){
      if (l1 == h1 || l2 == h2){
            return a[l1] // We got a input where with int[] a and int[] b with only size 1, return the first
                 number, which is median
      }
      if (h1 - l1 == 1 && h2 - l2 == 1){ // Base Case: We only find the median among 4 numbers
            int[] basicArray = new int[4];
            basicArray = {a[l1], a[h1], b[l1], b[h1]};
            Arrays.sort(basicArray); // Sort the last four elements, which takes O(nlogn) = O(4log4) = O(8) =
                 O(1), constant time
            return basicArray[1]; // Return the second element, which is the median out of 4 elements
      }
      int m1 = (h1-l1)/2;
      int m2 = (h2-l2)/2; // Get median number of both array
      if (a[m1] < b[m2]){
            return findMedian(a, b, m1, h1, l2, m2); // Find the median number of two sub array by ignoring the
                 numbers smaller or equal to the smaller median out of the two and greater or equal to the bigger
                 median out of the two
      }
      else{
            return findMedian(a, b, l1, m1, m2, h2); // Find the median in the sub array similar to above
      }
}
```

Proof. We first proof the correctness of the algorithm.

Lemma 3.0.1

The algorithm terminates.

Proof. If the array is of size 1, the algorithm will directly terminates.
If the size of both a, b are arrays with finite size $n \geqslant 2$, and each time $\frac{n}{2}$ will be taken into the next step of recurrence, with a base case of size 2 for each array. Thus, the algorithm must terminate in finite number of recurrence, which is $\log_2 n$ steps at most. □

Lemma 3.0.2

The median of the two sub array, get by discarding the numbers smaller than the smaller median out of the two and the numbers larger than the larger median out of the two, equals to the median of the two complete array in this problem.

Proof. We know that if we have a sorted array of size n, the medium exist at index $\left(\frac{n}{2}\right)$, meaning there will be definitely $\frac{n}{2} - 1$ numbers smaller or equal to it, and $\frac{n}{2}$ numbers greater or equal to it.
We also know that, because we are comparing the medium of two sorted array, every number smaller or equal to the smaller median, which have $\frac{n}{4} - 1$ in total, are also the $\frac{n}{4} - 1$ smallest numbers of the 2 arrays, thus must smaller or equal to the medium of the two arrays. Same applies to the $\frac{n}{4} - 1$ numbers bigger the larger medium out of two. Thus, we are now safely discarding the smallest $\frac{n}{4} - 1$ smallest and $\frac{n}{4} - 1$ largest numbers of the array, and since there are $\frac{n}{2} - 1 > \frac{n}{4} - 1$ numbers smaller or equal to the medium and $\frac{n}{2} > \frac{n}{4} - 1$ numbers greater or equal to the medium, the medium must still exist in the sub arrays remained. □

Thus, combining the two lemmas, we know that the algorithm will terminate, and we are able to find the median by constantly finding the median of the sub array get by the algorithm, thus, the algorithm properly gives the correct output. □

Proof. Now we look at the runtime.
Every operation from line 2 to 13 are constant time operations. Though there exist a sorting algorithm on line 8, but since we know the size of the array is 4, the sorting will give $O(n \cdot \log n) = O(4 \log 4) = O(8) = O(1)$ time.
As for line 13 to 18, according to a constant time caparison, we will choose to step into one of the occurrence, which transfers two arrays originally both with size n into sub arrays into $\frac{n}{2}$. Thus, we have the following runtime:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Based on Master Theorem, we have $a = 1, b = 2, f(n) = O(1)$. Since $n^{\log_2 1} = n^0 = 1 = \Theta(1)$, we apply the case 2 of the Master Theorem. Thus, the runtime will take $T(n) = \Theta(1 \cdot \log n) = \Theta(\log n)$. □