CSCI 270 - Spring 2023 - HW 5

Due: February 22, 2023

Graded Problems

1. [20 points] For the following recurrence equations, solve for $T(n)$ if it can be found using the master method (make sure to show which case applies and why). Else, indicate that the master method is not applicable and explain why.

   (a) $T(n) = 8T(n/2) + n \log n - 2023n$

   (b) $T(n) = 2T(n/2) + n^3(\log n)^3$

   (c) $T(n) = 4T(n/2) + n^2(\log n)^2$

   (d) $T(n) = 3T(n/3) - n \log n$

   Solution

   (a) Case 1: $n^{\log_2(8)}$, $f(n) = \mathcal{O}(n \log n)$, so $T(n) = \Theta(n^3)$

   (b) Case 3: $n^{\log_2(2)} = n$, $f(n) = n^3(\log n)^3$, $2(n/2)^3(\log(n/2))^3 \leq c \cdot n^3(\log n)^3$ for $c = 1/4 < 1$, so $T(n) = \Theta(n^3(\log n)^3)$

   (c) Case 2: $n^{\log_2(4)} = n^2$, $f(n) = n^2(\log n)^2$, so $T(n) = \Theta(n^2(\log n)^3)$

   (d) Not Applicable: $f(n) = -n \log n$ is not an asymptotically positive function, the master method does not apply.

   Rubric

   - -2 points: Case is incorrect
   - -3 points: $T(n)$ is incorrect

2. [10 points] Consider the divide and conquer solution described in the discuss section to find the closest pair of points in a 2D plane. Assume that we did not have a driver routine to sort the points. So our recursive function did not receive the points in sorted orders of their X and Y coordinates and the sorting had to be done for each subproblem (at every level). What would be the worst-case complexity of this algorithm assuming that the rest of the algorithm remains the same?

   Solution

Divide step and combine step will now take $\Theta(n \log n)$ for sorting, so

$$T(n) = 2T(n/2) + \Theta(n \log n).$$

Since $n^{\log_b a} = n^{\log_2 2} = n$, case 2 of the master theorem applies, so

$$T(n) = \Theta\left(n(\log n)^2\right).$$

Rubric

- -2 points: No sorting
- -2 points: Time to conquer is incorrect — $2T(n/2)$
- -2 points: Time to divide and combine (sorting) is incorrect — $\Theta(n \log n)$
- -2 points: No explanation or apply the master theorem incorrectly — case 2
- -2 points: Worst-case complexity is incorrect — $\Theta\left(n(\log n)^2\right)$

Common mistakes

- By case 3, $n \log n = \Omega(n)$, so $T(n) = \Theta(n \log n)$.
  - One of the conditions for applying case 3 is $f(n) = \Omega(n^{\log_b(a)+\epsilon})$, for some $\epsilon > 0$. In this case, there is **NO** $\epsilon > 0$ such that $n \log n = \Omega(n^{1+\epsilon})$, so case 3 doesn't apply here. Please refer to pages 8-9 of Lecture Notes 6.
- It takes $\Theta(n \log n)$ for each level of the recursion tree and there are n levels in total, so $T(n) = n \cdot \Theta(n \log n) = \Theta(n^2 \log n)$.
  - There are only $\log n$ levels in total. Please refer to page 12 of Lecture Notes 6.

3. [10 points] Solve Kleinberg and Tardos, Chapter 5, Exercise 3.

Solution

In a set of cards, if more than half of the cards belong to a single user, we call the user a majority user.

Divide the set of cards into two roughly two equal halves, (that is one half is of size $\lfloor n/2 \rfloor$ and the other, of size $\lceil n/2 \rceil$). For each half, recursively solve the following problem, "decide if there exists a majority user and if she exists, find a card corresponding to her (as a representative)".

Once we have solved the problem for the two halves, we can combine them to solve the problem for the whole set, i.e. finding the global majority user. We can do that as follows.

If neither half has a majority user, then the whole set clearly does not have a majority user.

If both the halves have the same majority user, then that user is the global majority user. We can pick either one of the output cards returned by the halves as a representative for the whole set.

If the majority users are different, or if only one of them has a majority user, we need to check if any of these users is a global majority user. We can do this in a linear manner by comparing the representative card of the majority user with every other card in the whole set, counting the number of cards that belong to the same majority user.

If $T(n)$ denotes the number of comparisons (invocations to the equivalence tester) of the resulting divide and conquer algorithm, then

$$T(n) \leq 2T(\lfloor n/2 \rfloor) + \mathcal{O}(n) \implies T(n) = \mathcal{O}(n \log n)$$

Rubric

- -7 points: Incorrect or missing algorithm
- -3 points: The algorithm fails to output the optimal solution or is partially
- -3 points: time complexity is missing or incorrect

4. [10 points] You are given with two integers a and b, and a variation of Fibonacci series, with $f(0) = a$ and $f(1) = b$. Recall that the Fibonacci sequence is $f(n) = f(n-1) + f(n-2)$. Devise an efficient algorithm to find the n^{th} Fibonacci number with $O(\log n)$ time complexity and prove its time complexity using recurrence relation.
(Hint: You can represent the calculation of Fibonacci series using matrix multiplication as follows

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix} = \begin{bmatrix} f(n-1) + f(n-2) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix}$$

You can repetitively multiply the resultant matrix with $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ to get subsequent Fibonacci numbers.)

Solution
The idea behind this problem to make use of binary exponentiation in matrices.
Let

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}, F_n = \begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix}$$

With this we can say that

$$A \cdot F_1 = F_2$$

where

$$F_1 = \begin{bmatrix} f(1) \\ f(0) \end{bmatrix}$$

Expanding on this to obtain subsequent equations

$$A \cdot F_2 = F_3$$

$$A \cdot A \cdot F_1 = F_3 \implies A^2 \cdot F_1 = F_3$$

The generic version of this repeated multiplication would then be

$$A^{n-1} \cdot F_1 = F_n$$

This problem essentially boils down to multiplying the matrix A by itself for $n-1$ times to obtain F_n which would have the n^{th} Fibonacci number. Computing this can be done recursively as follows:

If $n-1$ is even

$$A^{n-1} = A^{\lfloor \frac{n-1}{2} \rfloor} \cdot A^{\lfloor \frac{n-1}{2} \rfloor}$$

If $n-1$ is odd

$$A^{n-1} = A^{\lfloor \frac{n-1}{2} \rfloor} \cdot A^{\lfloor \frac{n-1}{2} \rfloor} \cdot A$$

Once A^{n-1} is calculated, perform another multiplication operation with A^{n-1} and F_1 to obtain F_n which would contain the n^{th} Fibonacci number. The recurrence relation for this algorithm would be

$$T(n-1) = T(\lfloor \frac{n-1}{2} \rfloor) + O(1)$$

(Either this or same recurrence relation with $T(n)$ and $T(\frac{n}{2})$ is also acceptable)
Since there is only 1 unique subproblem and as the matrix multiplications for 2×2 matrices and 2×1 matrices are constant, The overall time complexity for this algorithm would be $O(\log n)$

Rubric

- -7 points: Algorithm is incorrect / Missing critical details
- -3 points: Recurrence relation or time complexity analysis is incorrect or missing
- -5 points: Algorithm is correct but deviates from the expected time complexity

5. [10 points] You are given a **sorted** array consisting of $k+1$ values. Only one of the values appears once, and the rest of the k values appear twice. That is, the size of the array is $2k+1$. Design an efficient Divide and Conquer algorithm for finding which value appears only once. Partial credit (at most 6 points) will be given for non-Divide and Conquer algorithms. Discuss the runtime for your algorithm. Here are some example inputs to the problem:

   1, 1, 2, 2, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8

   10, 10, 17, 17, 18, 18, 19, 19, 21, 21, 23

   1, 3, 3, 5, 5, 7, 7, 8, 8, 9, 9, 10, 10

   Solution

   Algorithm: Let A be our array and its length be $n = 2k+1$ (since it consists of k pairs and one singleton). Array A starts at position 0. If $n = 1$, return $A[0]$. If $n > 1$, compare $A[2\lfloor k/2 \rfloor]$ and $A[2\lfloor k/2 \rfloor + 1]$. If the elements are equal, recursively apply this algorithm to the subarray beginning at position $2\lfloor k/2 \rfloor + 2$ and extending to the end. Otherwise, recursively apply this algorithm to the subarray starting at the beginning of the array and extending to $2\lfloor k/2 \rfloor$, inclusive.

Algorithm 1: Pseudo-code for Q5

```
Function FindSingleValue($A[0,\dots,n-1]$):
    $n = A.size()$              // $n$ is the number of elements in A
    $k = (n-1)/2$                     // $k$ is the number of pairs
    $m = 2 * floor(k/2)$                     // $m$ is always even
    if $n == 1$ then
        return $A[0]$                                // Base case
    if $A[m] == A[m+1]$ then
        // $A[0,\dots,m+1]$ are all pairs, so we ignore.
        return FindSingleValue($A[m+2,\dots,n-1]$)
    else
        // $A[m+1,\dots,n-1]$ are all pairs, so we ignore.
        return FindSingleValue($A[0,\dots,m]$)
    return
```

   Runtime: The following recurrence gives our algorithm's runtime in terms of k:

   $T(0) = \Theta(1)$

   $T(k) \leq T(\lfloor k/2 \rfloor) + \Theta(1)$

   By the Master Theorem, this solves to $T(k) = O(\log k)$. Since $n = 2k+1$, this means that the runtime as a function of n is $O(\log n)$.

Rubric

- -3 points: Wrong/missing time complexity
- -4 points: Incorrect recurrence relation
- -2 points: Missing explanation
- -2 points: minor errors