CS 270 Homework 5

Neel Gupta

February 22, 2023

Problem 1. 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) $T(n) = 3T(n/3) - n\log n$

Answer:

(a) $a = 8, b = 2, x = \log_2 8 = 3$
$f(n) = \Theta(n^1 \log^1 n), y = 1, k = 1$
If $x > y$, invoke case 1, then $T(n) = \Theta(n^3)$

(b) $a = 2, b = 2, x = \log_2 2 = 1$
$f(n) = \Theta(n^3 \log^3 n), y = 3, k = 3$
If $y > x$, invoke case 3, then $T(n) = \Theta(f(n)) = \Theta(n^3 \log^3 n)$

(c) $a = 4, b = 2, x = \log_2 4 = 2$
$f(n) = \Theta(n^2 \log^2 n), y = 2, k = 2$
If $y = x$, invoke case 2, then $T(n) = \Theta(f(n)\log n) = \Theta(n^2 \log^3 n)$

(d) The master method does not apply here because $f(n)$ is not positive.

Problem 2. Consider the divide and conquer solution described in the discussion section to find the closest pair of points in a 2D plane. Assume that we did not have 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?

Answer: $O(n \log^2 n)$

The algorithm to find the closest pair of points after dividing and conquering is to split the search space into two, so that about half the points are on each side, then finding the closest pair on each side recursively as well as finding the closest pair of points that cross the boundary. Return the closest of these solutions. We then get the minimum distance on the left and right sides where both points are on the same, so any points further than this boundary do not need to be sorted nor searched to compare find whether a closer pair of points exist.

procedure CLOSESTPAIRUNSORTED($p_1, p_2, ..., p_n$)
    Compute a seperation line with half on each side ▷ $O(n \log n)$
    Partition points set P into P_1 and P_2
    $\delta_1 \leftarrow$ CLOSESTPAIRUNSORTED(P_1) ▷ $T(n/2)$
    $\delta_2 \leftarrow$ CLOSESTPAIRUNSORTED(P_2) ▷ $T(n/2)$
    $\delta \leftarrow \min(\delta_1, \delta_2)$
    Let $D \leftarrow$ points in one δ away from the seperation line ▷ $O(n)$
    sort(D) ▷ $O(n \log n)$
    for point p in D **do** ▷ $O(n)$
        if distance between p and any of its 11 neighbors $< \delta$ **then**
            $\delta \leftarrow$ distance(p, p_{neighbor})
        end if
    end for
end procedure

$$
\begin{aligned}
T(n) &= 2T(n/2) + O(n \log n) + O(n) + O(n \log n) + O(n) \\
&= 2T(n/2) + O(n \log n) \\
x &= \log_2 2 = 1, y = 1, k = 1. \\
&\text{If } x = y, \text{invoke case 2.} \\
\implies T(n) &= O(n \log^2 n)
\end{aligned}
$$

Problem 3. Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say two bank cards are *equivalent* if they correspond to the same account.

It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them into the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

Answer:

Creating a divide and conquer algorithm similar to merge sort, we can accomplish the task with a runtime of $O(n \log n)$. Divide the elements into two equal groups of size $n/2$. Afterwards, we can recursively test if there is a majority element. Each recursive call will have bases cases for one and two elements which either return the single element or compare equality for the two and returning that element if so. If it matches at least $\frac{n-1}{2}$, then there is a majority of bank cards that are equivalent. This algorithm involves solving 2 subproblems of size $n/2$ and then combining these results in $O(2n)$. Therefore $T(n) = 2T(n/2) + 2n = \Theta(n \log n)$ since $a = 2, b = 2$, and $x = 1, y = 1, k = 0$.

To prove correctness, we see that the algorithm checks whether a specific element x matches $\frac{n-1}{2}$ other elements, so there can exist no false positives. On the other hand, a majority element will cause over half of its elements in a bisecting split to be that element. By assuming that the algorithm is correct on up to n elements. We can use induction to show that up to $n + 1$, a majority element will remain the same and the given algorithm will output that element because the majority element was outputted at n elements, so at least $k + 1$ elements must have been that majority element and it will still be outputted even with a new element added. By induction, our algorithm will output the correct card for all input sizes.

Problem 4. You are given 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 nth Fibonacci number with $O(\log n)$ time complexity and prove its time complexity using recurrence relation.

Answer: Consider the following fact about representing the recursive structure of the Fibonacci numbers by multiplying a column vector of the first two numbers such that the resulting vector is the next set of Fibonacci numbers.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} f_{n-1} \\ f_{n-2} \end{bmatrix} = \begin{bmatrix} f_{n-1} + f_{n-2} \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix}$$

Therefore, we can get the nth Fibonacci number of our varied sequence by repeatedly multiplying the matrix.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n * \begin{bmatrix} b \\ a \end{bmatrix} = \begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix}$$

Therefore the runtime of getting f_n is the dependent on the time of how long it takes to exponentiate a matrix n times.

The nth power of the matrix can be found in $O(\log n)$ time. Thus, the overall runtime of our algorithm is $O(\log n)$.

Problem 5. You are given a sorted array consisting of $k+1$ values. Only one of the values appears once, and then 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.

Answer: Consider the fact that after the element that only appears once has been found, all elements will have been shifted one element over to the right. We can start by first looking at the middle two elements and checking if they are the same. If they are the same, then the single element has not been found yet since an even number have been split by finding the initial median. Thus, we can search the right side in this same way again if they are the same. Else, we can search left of the middle to find the point when the singleton appears or disappears within the range being searched.

$n \leftarrow 2k+1$ FINDNONDUPLICATE(A,0,n)
procedure FINDNONDUPLICATE(A,l,r)
  if $(l-r) = 1$ **then return** $A[l]$
  end if
  if $A[\lfloor \frac{l+r}{2} \rfloor] = A[\lfloor \frac{l+r}{2} \rfloor + 1]$ **then** FINDNONDUPLICATE(A,$\lfloor \frac{l+r}{2} \rfloor + 1$,$r$)
  else FINDNONDUPLICATE(A,l,$\lfloor \frac{l+r}{2} \rfloor$)
  end if
end procedure

To prove correctness, we can show via mathematical induction that our algorithm returns the singleton by showing that the base case holds for 1 element, and that if we assume that our algorithm finds the singleton for $2k-1$ elements, then for $2k+1$ elements, we can show that our algorithm returns the correct singleton. So creating a new pair means that the same singleton will be found as before, so our algorithm holds for all k.

The algorithm takes $O(k)$ time to run or $O((n-1)/2)$. Therefore, the algorithm creates one subproblem of half the size at each iteration. Therefore, $T(0) = \Theta(1)$ and $T(k) = T(\lfloor k/2 \rfloor) + \Theta(1)$. By master method, $T(k) = \Theta(\log k)$ and since n is some constant function of k: $n = 2k+1$. Thus, the runtime of the entire function is $\Theta(\log n)$.