5️⃣

HW5

▼ **Question 1**

Using extended master theorem

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$$

① if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

② if $a = b^k$, then

- (a) $p > -1 \rightarrow T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
- (b) $p = -1 \rightarrow T(n) = \Theta(n^{\log_b a} \log \log n)$
- (c) $p < -1 \rightarrow T(n) = \Theta(n^{\log_b a})$

③ if $a < b^k$

- (a) $p \geq 0 \rightarrow T(n) = \Theta(n^k \log^p n)$
- (b) $p < 0 \rightarrow T(n) = \Theta(n^k)$

Q1 $T(n) = 4T(n/2) + n^2 \log n$

$f(n) = n^2 \log n = \Theta(n^2 \log n)$

$a = 4,\ b = 2,\ k = 2,\ p = 1$

$a = 4$ & $b^k = 2^2 = 4$

$a = b^k$, case 2 & $p > -1$ so, $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

$T(n) = \Theta(n^{\log_2 4} \log^2 n)$

$\boxed{T(n) = \Theta(n^2 \log^2 n)}$

(b) $T(n) = 8T(n/6) + n\log n$

$f(n) = n\log n = \Theta(n\log n)$

$a = 8,\ b = 6,\ k = 1,\ p = 1$

$a = 8,$
$b^k = 6^1 = 6.$

$a > b^k$, (case 1) $T(n) = \Theta(n^{\log_b a})$

$T(n) = \Theta(n^{\log_6 8}) = \Theta(n^{1.16})$

$$\therefore T(n) = \Theta(n^{1.16})$$

(c) $T(n) = \sqrt{6000}\,T(n/2) + n^{\sqrt{6000}}$

$f(n) = n^{\sqrt{6000}} = \Theta(n^{\sqrt{6000}})$

$a = \sqrt{6000},\ b = 2,\ k = \sqrt{6000},\ p = 0$

$b^k = 2^{\sqrt{6000}}$

$a < b^k \rightarrow$ (case 3) & $p = 0 \geq 0$ so, $T(n) = \Theta(n^k \log^p n)$

$$T(n) = \Theta(n^{\sqrt{6000}})$$

(d) $T(n) = 10T(n/2) + 2^n$

$f(n) = 2^n = \Theta(2^n)$

$a = 10,\ b = 2,\ n^{\log_b a} = n^{\log_2 10}$

This can be case 3, so,

$a \cdot f(n/b) \leq c \cdot f(n),\ c < 1\ \forall\ n$

$10\left(2^{n/2}\right) \leq c \cdot 2^n$ for $c \leq 10$ this condition holds true.

So,

$T(n) = \Theta(f(n))$

$\boxed{T(n) = \Theta(2^n)}$

(e) $T(n) = 2T(\sqrt{n}) + \log_2 n$

$n = 2^m$

$T(2^m) = 2T(2^{m/2}) + m$

$S(m) = T(2^m)$

$S(m) = 2S(m/2) + m$

$= 2\left(2S(m/4) + m/2\right) + m$

$= 4S(m/4) + 2m$

$= 2^i S(m/2^i) + im$

$i = \log m \quad = m \cdot S(1) + m \log m$

$= m \cdot T(2) + m \log m$

$m \cdot T(2) + m \log m$

Initially, $n = 2^m$, so, $m = \log_2 n = \log n$.

$T(n) = T(2) \cdot \log n + \log n \cdot \log \cdot \log n$.

$T(n) = O(\log n \cdot \log \cdot \log n)$

$\boxed{T(M) = O(\log n \cdot \log \cdot \log n)}$

▼ **Question 2**

2. Consider an array A of n numbers with the assurance that n > 2, A[1] ≥ A[2] and A[n] ≥ A[n−1]. An index i is said to be a local minimum of the array A if it satisfies 1<i<n, A[i − 1] ≥ A[i] and A[i + 1] ≥ A[i].

    (a) Prove that there always exists a local minimum for A.
    (b) Design an algorithm to compute a local minimum of A.

    Your algorithm is allowed to make at most O(log n) pairwise comparisons between elements of A.

Solution

a. According to the constraints provided, we see that first 2 elements are in decreasing order and the final 2 elements are in increasing order. For an array for size > 3, lets assume that the series constantly decreases, then when we reach the second last element of the array, A[n-1] should be less than A[n - 2] but which is not the case. This violates our assumption, hence the contradiction and hence, there always exists a local minima in the given array, A.

b. We can make use of Binary search to find the local minima → we calculate the middle index, i and check whether element present at that index is smaller than either of its neighbors, $arr[i-1]$ and $arr[i+1]$. If yes, we return that element as the local minima. Else, if the middle element is smaller than left neighbor, we visit the left subaray, else if the middle element is greater than the right neighbor, we visit the right subarray.

Time Complexity Analysis

$T(n) = T(n/2) + 3$, one for left comparison, one for right comparison and one for self comparison

$f(n) = \theta(1)$ as the Divide and Conquer step takes constant time (arithmetic operation to calculate mid)

a = 0 (since we only consider one subarray depending on the value of the middle element → as this is a binary search

$log_b a = log_2 1 = 0$

Case 2 for the Master's Theorem, which gives us $T(n) = \theta(log(n))$

▼ **Question 3**

$m \cdot T(2) + m \log m$

Initially, $n = 2^m$, so, $m = \log_2 n = \log n$.

$T(n) = T(2) \cdot \log n + \log n \cdot \log \cdot \log n$.

$T(n) = O(\log n \cdot \log \cdot \log n)$

$\boxed{T(M) = O(\log n \cdot \log \cdot \log n)}$

ALG $\Rightarrow T(n) = 7T(n/2) + n^2$ ——— ①

ALG' $\Rightarrow T'(n) = aT'(n/4) + n^2 \log n$. ——— ②

Solving ① & ② using Master Theorem

① $\Rightarrow T(n) = 7T\left(\frac{n}{2}\right) + n^2$

$$T(n) = 7\left[7T\left(\frac{n}{4}\right) + \frac{n^2}{4}\right] + n^2 = 49T\left(\frac{n}{4}\right) + \frac{11n^2}{4}$$

to make, bases same $a = 49, b = 4, f(n) = 11n^2/4$

$n^{\log_b a} = n^{\log_4 49} \approx n^{2.81}$

Clearly, $f(n) = O(n^2) = O(n^{\log_4 49 - \epsilon}) = O(n^{2.81 - 0.81})$

$\epsilon = 0.81$

Case ① of Master Theorem $\rightarrow$

$\boxed{T(n) = O(n^{\log_b a}) = O(n^{\log_4 49})}$

According to the question, ALG' needs to be asymptotically faster than ALG1.

for the second algo,

$$T'(n) = aT'\left(\frac{n}{4}\right) + n^2$$

$$f(n) = n^2 = O(n^2) = \Theta(n^2)$$

$n^{\log_b a} = n^{\log_4 a}$, if $a > 16$, we will have Case ① of master theorem. Hence.

$$T'(n) = \Theta(n^{\log_4 a})$$

now, we know that

$$T'(n) < T(n)$$

$$\Theta(n^{\log_4 a}) < \Theta(n^{\log_4 49})$$

for this to be true, $\boxed{a < 49}$

so, greatest value of 'a' that is possible is

$$\boxed{a = 48}$$

▼ Question 4

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 a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that 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 in to the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

Solution

We can divide the list into 2 smaller lists and then deploy a mechanism in place that calculates the account number and the number of time that this account number is present in this list recursively, for example → the mechanism F returns a pair (m, c) from the smaller list where m = account number and c = number of times this account number is present in this list, is c is greater than half of the size of the list. If there is no such account number that repeats more than half of the list size times, then return (-1, 0)

Let's say F returns (m1, c1) from the first list and (m2, c2) from the second list. If m1 == m2, then return (m1, c1 + c2). Else, in the first list, search for m2 and update c2 accordingly. Similarly, in the second list search for m1 and update c1 accordingly. If $c1 > c2$ return $(m1, c1)$ else if $c2 > c1$ then return $(m2, c2)$. If we have returned any positive values of c1 and c2, the resulting set will have a solution where we will have the account list where size is greater than $n/2$.

To consider for the base cases, we keep on dividing the list to get smaller subproblems, the answers to which will need to be combined to form the final answer. While, dividing, if the list is empty, then return (-1, 0). If the list has a

single account number, m, return $(m, 1)$ and if the list has 2 account numbers, a and b and $a == b$ then return $(a, 2)$. Once we have values from these 2 sub-problems, we combine these solutions to solve the big problem and so on.

Time Complexity Analysis

To devise a recurrence pair, we divide the problem into 2 subproblems with half the size of the initial problem, and the divide step takes $O(n)$ time as their are n elements in the list and all need to be traversed. Hence, we have

$$T(n) = 2T(n/2) + n$$

Solving this using Extended Master's Theorem, we get →

a = 2, b = 2, $f(n) = n$ and $log_b(a) = log_2(2) = 1$ and k = 1 and $b^k = 2$ and $p = 0$

> 💡 This is case 2, and using the formula we get → $T(n) = \theta(n^{log_b a} \cdot log^{p+1} n)$. Plugging in values, we get $T(n) = n \cdot log(n)$ which is the same as merge sort

---

▼ **Question 5**

5. Given a binary search tree T, its root node *r*, and two random nodes *x* and *y* in the tree. Find the lowest common ancestry of the two nodes *x* and *y*. Note that a parent node *p* has pointers to its children *p.leftChild()* and *p.rightChild()*, a child node does **not** have a pointer to its parent node. The complexity must be $O(\log n)$ or better, where *n* is the number of nodes in the tree.
Recall that in a binary search tree, for a given node *p*, its right child *r,* and its left child *l*, $r.value() \geq p.value() \geq l.value()$. Hint: use divide and conquer

Solution

We use the classic property of a binary search tree and traverse the tree based on the following conditions -

- **DIVIDE STEP:** We traverse the tree and check if the value of x and value of y is less than that of root. If yes, we recursively solve for the left subtree. If the value of x and the value of y is greater than the root, we recursively solve for (traverse) the right subtree.

- If value of x is less than that of root and value of y is greater than that of root, we return the current node that we are present on.

> 💡 Since the tree is balanced, the time complexity of the above algorithm is - $O(log(n))$

Proof using **Master Method**

$f(n) = D(n) + C(n) = 1$ dividing the tree takes constant time. So, $f(n) = \theta(1)$

So, $T(n) = T(n/2) + 1$

Solving above using Master method, we get $a = 1$ because we only consider half of a tree at any given point in time. And $b = 2$

So, $log_b a = log_2 1 = 0$

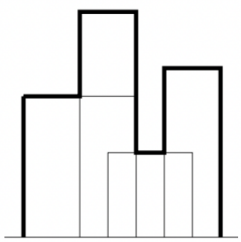
Using case 2 of master's theorem → we get $T(n) = log(n)$

▼ Question 6

6. Suppose that you are given the exact locations and shapes of several rectangular buildings in a city, and you wish to draw the skyline (in two dimensions) of these buildings, eliminating hidden lines. Assume that the bottoms of all the buildings lie on the x-axis. Building B_i is represented by a triple (L_i, H_i, R_i), where L_i and R_i denote the left and right x coordinates of the building, respectively, and H_i denotes the building's height. A skyline is a list of x coordinates and the heights connecting them arranged in order from left to right.
For example, the buildings in the figure below correspond to the following input

(1, 5, 5), (4, 3, 7), (3, 8, 5), (6, 6, 8).

The skyline is represented as follows: (1, **5**, 3, **8**, 5, **3**, 6, **6**, 8). Notice that the x-coordinates in the skyline are in sorted order.



a) Given a skyline of n buildings and another skyline of m buildings in the form (x_1, $\mathbf{h_1}$, x_2, $\mathbf{h_2}$, ….x_n) and (x'_1, $\mathbf{h'_1}$, x'_2, $\mathbf{h'_2}$, ….x'_m) , show how to compute the combined skyline for the m + n buildings in O(m + n) steps.

b) Assuming that we have correctly built a solution to part a, give a divide and conquer algorithm to compute the skyline of a given set of n buildings. Your algorithm should run in O(n log n) steps.

Solution

a. The idea is to divide the buildings into 2 and once we have the skylines for each of those buildings, we merge these 2 skylines in the following manner -

   For each set of buildings, we check for the smallest x-coordinate and add the x-coordinate and its corresponding height, h11, to the output set → here, it is assumed that skyline1 has the smaller x-coordinate) Similarly, we will get a scenario where skyline2, with height h21 will be added to the output set.

   Consider the following →

   Skyline 1 → (x1, h11, x2, h12, ….xn)

   Skyline 2 → (x1, h21, x2, h22, ….xn)

   The height of the resultant set is max(h1, h2)

   Initially, h11 and h21 are initialized to 0.

   - h11 is updated when output is from Skyline 1
   - h21 is updated when output is from Skyline 2

b. The skylines are divided into 2 groups and we compute the skylines recursively, the divide and conquer step takes $\theta(n)$ time.

   Recurrence relation is $T(n) <= 2T(n/2) + \theta(n)$

   Using the master method, $f(n) = \theta(n)$ and $log_b a = log_2 2 = 1$

   Using the generalized formula, we get a = 2, b = 2, k = 1, p = 0, $b^k = 2^1 = 2$ and hence $a = b^k$

   This is case 2 of the extended master theorem. Now, $p > -1$, so $T(n) = \theta(n^{log_b a}.log^{p+1} n)$ = $T(n) = \theta(n \cdot log(n))$