Homework-5:

(1) For algorithm ALG : $T(n) = 7T(n/2) + n^2$.

$a = 7, \quad b = 2, \qquad f(n) = n^2$.

$\therefore \quad n^c = n^{\log_2 7} > f(n)$

$\therefore \quad \boxed{\Theta(n^{\log_2 7})}$

For algorithm ALG' : $T'(n) = aT'(n/4) + n^2 \log n$

Now $T'(n)$ has to be smaller than $T(n)$ to make $T'(n)$ asymptotically faster.

$\therefore \quad T'(n) \leq n^{\log_2 7}$

$\therefore$ $T'(n)$ can not be $n^2 \log n$ $[f(n)]$ as $n^2 \log$ is smaller than $n^{\log_2 7}$.

Now, $c = \log_b a = \log_4 a$

$\therefore \quad \cancel{n^c} > n^{\log_2 7}$ $\qquad\qquad \therefore \log_4 a \leq \log_2 7$

$\therefore \quad n^{\log_4 a} \; \cancel{>} \; \cancel{n^{\log_2 7}}$

$\therefore \quad \cancel{n^{\log_n a \cdot \log_4 n}} \; \cancel{>} \; n^{\log_n 7 \cdot \log_2 n}$

$\therefore \quad \cancel{a^{\log_4 n}} \; \cancel{>} \; 7^{\log_2 n}$

$\therefore \quad \dfrac{\log_2 a}{\log_2 4} \leq \log_2 7$

$\therefore \quad \log_2 a \leq 2 \log_2 7$

$\therefore \quad \boxed{a \leq 49}$.

$\therefore$ maximum value of a can be $\boxed{49}$.

Scanned by CamScanner

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

$a=4,\ b=2,\ c=\log_b a = \log_2 4 = \boxed{2}$, $f(n) = n^2 \log n$

$n^c = n^2$

$\therefore n^2 = f(n) = n^2 \log n$.

So as per case-2 : $\boxed{\Theta(n^2 \log^2 n)}$ ans

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

$a=8,\ b=6,\ c=\log_6 8,\ n^c = n^{\log_6 8}$, $f(n) = n \log n$.

$c = \frac{\log_2 8}{\log_2 6} = 3\log_6 2$

$\therefore n^{3\log_6 2} > n \log n$

$\therefore$ As per case-1 : $\boxed{\Theta(n^{\log_6 8})}$ ans.

(c.) $T(n) = \sqrt{6006}\,T(n/2) + n^{\sqrt{6006}}$

$a = \sqrt{6006},\ b=2,\ c = \log_2 (6006)^{1/2}$, $f(n) = n^{(6006)^{1/2}}$

$\therefore n^{\log_2 (6006)^{1/2}} < n^{(6006)^{1/2}}$

$\therefore$ As per case-3 : $\boxed{\Theta(n^{\sqrt{6006}})}$ ans

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

$a=10,\ b=2,\ c=\log_2 10$, $n^c = n^{\log_2 10}$, $f(n) = 2^n$

$\therefore n^c < f(n)$.

$\therefore$ As per case-3 : $\boxed{\Theta(2^n)}$ ans

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

Let $n = 2^x$, $\therefore \sqrt{n} = 2^{x/2}$, $\log_2 n = x\log_2 2 = x$

$\therefore T(2^x) = 2(T(\cancel{x}\ 2^{x/2})) + x$

Let $T(2^x) = S(x)$

$S(x) = 2S(\cancel{x}/2) + x$.

now $a=2,\ b=2,\ c=\log_2 2 = 1$, $f(x) = x$.

$x^c = x = f(x)$ $\Rightarrow \Theta(x \log x)$

Now substituting $x = \log_2 n$

$\therefore \boxed{\Theta(\log_2 n \cdot \log \log_2 n)}$ ans (f.) $T^2(n) = T(n/2) \cdot T(2n) - T(n)\,T(n/2)$

$$\frac{T^2(n)}{T(n)T(n/2)} = \frac{T(2n)\cdot T(n/2)}{T(n)\cdot T(n/2)} - \frac{T(n)T(n/2)}{T(n)T(n/2)}$$

$$\therefore \frac{T(n)}{T(n/2)} = \frac{T(2n)}{T(n)} - 1$$

Let $S(n) = \frac{T(2n)}{T(n)}$

$$\therefore \frac{T(n)}{T(n/2)} = S(n/2)$$

$\therefore S(n/2) = S(n) - 1$

$\therefore S(n) = S(n/2) + 1$

$a=1,\ b=2,\ c = \log_2 1 = 0,\quad f(n)=1$

$n^c = n^0 = 1 = f(n)$

$\therefore \Theta(\log n)$

(g.) $T(n) = 2T(n/2) - \sqrt{n}$

Since the function is negative, Master theorem cannot be implemented here.

$T(n) = 2T(n/2) - \sqrt{n}$

& $T(n/2) = 2T(n/4) - \sqrt{n/2}$

$\therefore T(n) = 2\left(2T(n/4) - \sqrt{n/2}\right) - \sqrt{n}$

$= 2^2 T(n/4) - 2\sqrt{n/2} - \sqrt{n}$

Similarly substituting $T(n/4)$ and so on.

In general we can write : $2^{k-1}T\left(\frac{n}{2^{k-1}}\right) - \sqrt{n}\left(\frac{\sqrt{2^{k-1}}-1}{\sqrt{2}-1}\right)$

here, $k = \log_2 n$

$$\therefore T(n) = 2^{\log_2 n - 1}\, T\left(\frac{n}{2^{\log_2 n - 1}}\right) - \sqrt{n}\left(\frac{\sqrt{2^{\log_2 n - 1}} - 1}{\sqrt{2}-1}\right)$$

Now, $2^{\log_2 n - 1} = \frac{2^{\log_2 n}}{2} = \frac{n}{2}$

$$\therefore T(n) = n\left(\frac{T(2)}{2} - \frac{1}{2-\sqrt{2}}\right) + \frac{\sqrt{2n}}{2-\sqrt{2}}$$

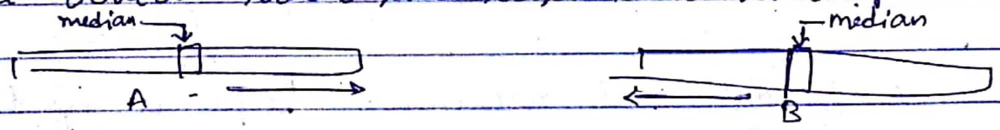
$T(2)$ can be greater than or equal to $f(n)$
So complexities can be: $\Theta(n)$ or $\Theta(\sqrt{n})$

3) (a) takes constant time
(b) scans each element for A and compares, so for n elements it takes $O(n^2)$
(c) takes linear time
(d) takes linear time
(e) divides the problem into two subproblems. and this is done for both lists B and C. So it takes $2T(n/2)$ time.
(f) append takes linear time.

$$\therefore T(n) = O(1) + \Theta(n^2) + \Theta(n) + O(n) + 2T(n/2) + O(n$$

$$f(n) = \Theta(n^2), \quad a=2, \; b=2, \; c = \log_2 2 = 1$$
$$n^c = n^1 < f(n)$$
$\therefore$ Complexity is $\boxed{\Theta(n^2)}$

4) We divide both the lists into two parts.



Then we compare both the medians.
If the Median of A is > Median of B then discard the right-most part of median in A and ~~if the med~~ the leftmost part of median in B. If median (A) < Median (B) then discard leftmost part in A and Rightmost in B. If both are equal then we found the match. Compare the remaining list of A with B and find the new median.

Here $T(n) = \cancel{2}\, T(n/2) + \text{constant time}$

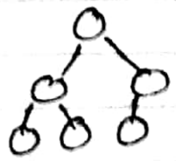
So $f(n) = O(1) = n^c = n^0 = 1$

$\therefore$ Complexity : $\boxed{\Theta(\log_2 n)}$

(5) First we divide the list into two equal parts.
~~Randomly pick~~ Pick the first card in first list.
If it has its equivalent in the list, return the card. EndIf.
If it has more number of equivalents in the list then check the other list as well.
If there are half cards at least equivalent to the current card, then return the card
Else go to next card.
EndIf
EndIf

In this algorithm n cards are compared for $\log n$ times (height of tree). So the complexity is $\boxed{O(n \log n)}$

(6) complete binary tree eg:
$n = 2^d - 1$



To find local minimum, take a node, compare the node with its left and right child.
If the parent node is smaller than both then it is the local minimum.
If the left child is smaller then the parent and right child, then repeat the algorithm for left subtree.
If the right child is smaller then the parent and left child, then repeat the algorithm for right subtree.

The function here will be called as much the height of tree is. So the complexity is $\boxed{O(\log n)}$

Proof: If the local minima is at the root node, it is returned at that point.
If it is at the left child, then the left subtree is further explored, repeating the same algorithm everytime. At any point the comparison will be held in a node and its' left and right child. So for any three nodes, the minimum value is going to be chosen as the local minima and will be explored further leaving no chance of considering a large node as local minima.
In the case where root is not the local minimum, it will explore maximum to the leaf node.
If a particular node doesn't have any children then there is no comparison, leaving that particular node as local minimum, which is true.

7) Divide the list into two parts.
Now compare.
If $A[i] \leq A[i-1]$ && $A[i] \leq A[i+1]$
then return $A[i]$ as local minimum.
Else If $A[i] > A[i-1]$, then choose the list of A from element 1 to i^{th} element and repeat the algorithm.
Else ~~If~~ repeat the algorithm for rest half of list → i to n^{th} element.

Complexity : $O(\log n)$

Proof:
Let us assume it gives local minimum for 3 elements. [Just the way it is true for 1 or 2 elements]
A[1] > A[2] < A[3] — Base case.
where A[2] is the local minimum.
For a list with size > 3
first the list will be divided into half and then the algorithm will be run.
Everytime the ~~situation~~ problem size will be reduced and at some point it will be same as the base case.
Hence, in no condition the algorithm will give the wrong value.

(8) > Considering to start counterclock wise from the minimum x-coordinate.
> First we divide the polygon into two parts. As per the property of the convex polygon, the maximum point of x-coordinate will be somewhere near the ~~mid~~ center line.
> Exploring in the counter clockwise direction [the polygon is divided into two parts where x=0, y=middle part] So the upper and lower case will be similar where the value of x increases proportionally.
> The same process can be done for y's max. value by dividing from (x=midpoint, y=0) coordinates. The maximum value will be again ~~near~~ the center part of upper half.
→ For both the cases (x & y - finding maximum) the ~~values~~ process remains the same, and the comparison ~~is done as per the~~ in O(logn) time.

(9) → Find the minimum element and maximum element in the array.

→ If the first element is minimum and last is maximum or vice-versa (as we do not know if the sorting was in ascending or descending order), then ~~the~~ no rotations are required.

→ Let there be a rotation count 'r', now check for minimum value A[1 to r] and A[r+1 to n] (Considering ascending order)

If r has some value, then repeat the algorithm for 1 to r elements.

If the element is not found, then repeat the algorithm for r+1 to n elements.

→ Both the parts can be searched in O(log n) time.

When there is no rotation, the value of r becomes '0' and when there is a rotation, minimum value from where the rotation is started is searched till the end or vice-versa to get the correct count of rotation and value.

Thus it finds the element in array.

Scanned by CamScanner