HOMEWORK 5

1. **[20 points] Solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n. Assume that T(n) represents the running time of an algorithm, i.e., T(n) is a positive and non-decreasing function of n. For each part below, briefly describe the steps along with the final answer:**

a. $T(n) = 9T(n/3) + n^2 \log n$

Ans:

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

$n^{(\log_b a)} = n^{(\log_3 9)} = n^2$

Here, $f(n) = \Theta(n^{(\log_b a)})$

It falls in Case 2 of Master theorem. (Not Case 3 because it's not growing faster than a polynomial component).

Here, k>-1 (k = 1→power of log term)

Hence,

$T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$

$T(n) = \Theta(n^2 \cdot \log^{1+1} n)$

Hence, **$T(n) = \Theta(n^2 \cdot \log^2 n)$**

b. $T(n) = (4.01)T(n/2) + n^2 \log n$

Ans:

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

$n^{(\log_b a)} = n^{(\log_2 4.01)}$ which is approximately $n^{2.00036}$

Here, $f(n) = O(n^{(\log_b a - \epsilon)})$

It falls in Case 1 of Master theorem.

$T(n) = \Theta(n^{\log_b a})$

Hence, **$T(n) = \Theta(n^{\log_2 4.01})$**

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

Ans:

$f(n) = n\text{^}(\sqrt{6000})$ which is approximately $n^{77.46}$

$n^{(\log_b a)} = n^{(\log_{\sqrt{6000}})}$ which is approximately $n^{6.27}$

Here, $f(n) = \Omega(n^{(\log_b a + \epsilon)})$

It falls in Case 3 of Master theorem.

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

Hence, **$T(n) = \Theta(n\text{^}(\sqrt{6000}))$**

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

Ans:

$f(n) = 2^n$

$n^{(\log_b a)} = n^{(\log_2 10)}$ which is approximately $n^{3.32}$

Here, $f(n) = \Omega(n^{(\log_b a + \epsilon)})$

It falls in Case 3 of Master theorem.

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

Hence, **$T(n) = \Theta(2^n)$**

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

Ans:

Suppose we replace n with a variable 2^m.

Now, $T(2^m) = 2T(\sqrt{2^m}) + \log_2 2^m$

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

Let $T'(m)$ be $T(2^m)$

Hence,

$T'(m) = 2T'(m/2) + m$

Here,

$f(m) = m$

$m^{(\log_b a)} = m^{(\log_2 2)} = m$

This falls in Case 2.

Since k = 0,

$T'(m) = \Theta(m^{(\log_b a)} \cdot \log^{k+1} m)$

$T'(m) = \Theta(m. \log m)$

Since, $n = 2^m$, $m = \log_2 n$

Hence, now T(n) becomes

$T(n) = \Theta(\log_2 n.\log((\log_2 n))$

2. Hidden surface removal is a problem in computer graphics that scarcely needs an introduction: when Woody is standing in front of Buzz, you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody .... well, you get the idea.

The magic of hidden surface removal is that you-can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a basic speed-up that can be achieved. You are given n nonvertical lines in the plane, labeled L1 ..... Ln, with the i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is uppermost at a given x-coordinate x_0 if its y-coordinate at x_0 is greater than

the y-coordinates of all the other lines at x_0: $a_i x_o + bi > a_j xo + b1$ for all $j \neq i$. We say line L_i is visible if there is some x-coordinate at which it is uppermost--intuitively, some portion of it can be seen if you look down from "y = ’.
Give an algorithm that takes n lines as input and in O(n log n) time returns all of the ones that are visible. Figure 5.10 gives an example.

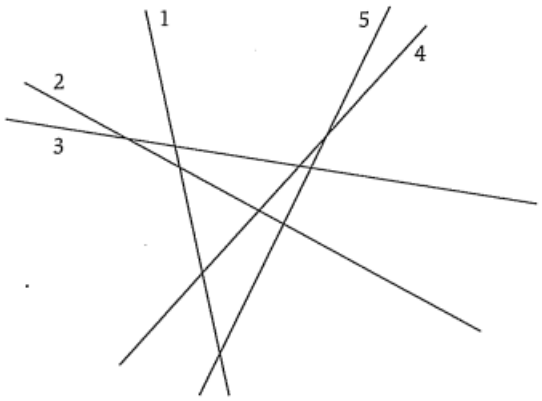


Figure 5.10 An instance of hidden surface removal with five lines (labeled *1–5* in the figure). All the lines except for *2* are visible.

Ans:
Here, we have n nonvertical lines in the plan labelled L_1, …, L_n with the i^{th} line specified by the
equation y = y = $a_i x + b_i$. We arrange these lines in ascending order of their slopes. We split these lines in 2 equal groups. After splitting them, we recursively find the visible lines on the left and right halves. This continues till there is one line available in either of the set. Then we merge the results to find the visible lines between them.

Algorithm:
Check if number of lines is 1 or not. If yes, print the line as it is obviously visible.
Divide the lines in two halves.
Split the total lines in 2 equal lists.
Recursively find the visible lines in left and right halves.
Merge the result into the function created for merging the lists.
Merge function:
Initiate a while loop till the counter is more than either of the length of list.
In this loop,
If the length of the left element is more than the right element, append the left element to the result. Else, add the right element to the result.
End if
End while
Add the remaining elements (if any) to the result at the end.

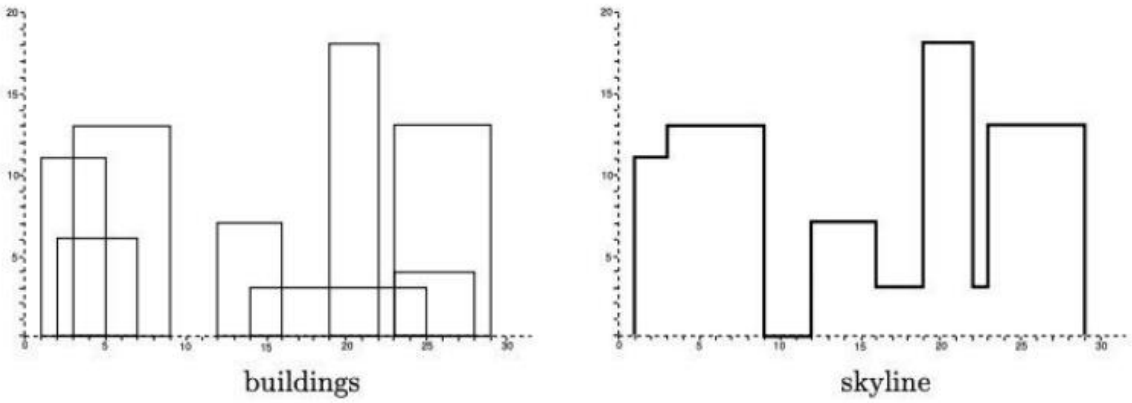
Time Complexity:
Recursive function:
Here, we divide the list into 2 halves, each with n/2 elements. It takes O(n) to merge.
Hence, the complexity becomes,
T(n) = 2T(n/2) + n.
f(n) = n
$n^{(\log_b a)} = n^{(\log_2 2)} = n$
This belongs to Case 2. Since k=0, $T(n) = n^{(\log_b a)} \log^{k+1}(n)$
Hence, T(n) = nlogn

3. **[20 points] Assume that you have a blackbox that can multiply two integers. Describe an algorithm that when given an n-bit positive integer a and an integer x, computes x^a with at most a $O(n)$ calls to the blackbox.**
Ans:
Here, we use divide and conquer algorithm to divide exponential problem into smaller subproblems, which minimize the multiplication required.
We check whether a is an odd or even number. (// →floor division)
If its an odd number: - $x^a = x^{a//2} . x^{a//2} . x$
If its an even number: - $x^a = (x^{a/2})^2$

Algorithm:
Define a blackbox function to multiple two integers
Define another function to split exponent value into smaller value till base is reached
Get the values of a and x
If a is equal to 0, return 1.
Else if a is equal to1, return x.
Else if a is even, call this function recursively using x and a/2 and store it in a variable.
Call the blackbox function with input as the variable twice. $((x^{a/2})^2)$
Else (odd number)
Call this function recursively using x and (a+1)//2 and store it in a variable. ($x^{a//2} . x^{a//2} . x$)
Call the blackbox function with input as (x, and the blackbox function with input as the variable twice).

Time complexity:
Here, $x^{a/2}$ is called atmost 3 times.
T(n) = T(n-1) + 3.
f(n) = 3
$n^{(\log_b a)} = n^0 = 1$.
Hence, f(n) = Ω(f(n))
Hence, T(n) = O(1).
This recurrence occurs n times.
Hence, the total complexity of this algorithm is O(n).

4. **[25 points] A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. A building B_i is represented as a triplet (L_i , H_i , R_i) where L_i and R_i denote the left and right x coordinates of the building, and H_i denotes the height of the building. Describe an O(n log n) algorithm for finding the skyline of n buildings. For example, the skyline of the buildings {(3, 13, 9),(1, 11, 5),(12, 7, 16),(14, 3, 25),(19, 18, 22),(2, 6, 7),(23, 13, 29),(23, 4, 28)} is {(1, 11),(3, 13), (9, 0), (12, 7), (16, 3), (19, 18), (22, 3), (23, 13), (29, 0)}. (Note that the x coordinates in a skyline are sorted)**



Ans:
Here, we use divide and conquer by dividing the set of buildings into 2 halves.
Recursively find the skyline of each half. Then we merge the result.

Algorithm:
Check if number of buildings is 1 or not. If yes, get the building's left, height and right x coordinates. Print ((left, height), (right, 0))
Divide the buildings in two halves.
Split the total buildings in 2 equal lists.
Merge the 2 lists.
Recursively find the visible lines in left and right halves.
Merge function:
Initiate a while loop till the counter is more than either of the length of list
In this loop,
Iterate though both skylines, and at each point, check the next points in both the lists.
If the next point in one skyline has a larger x-coordinate, add it to the merged skyline.
If it's a left endpoint, add its height to the current maximum height
If it's a right endpoint, remove its height.
If the x-coordinates are equal, choose the point with a larger height.
Add the remaining elements (if any) to the result at the end.

Time Complexity:

Recursive function:

Here, we divide the list into 2 halves, each with n/2 elements. It takes O(n) to merge.

Hence, the complexity becomes,

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

$f(n) = n$

$n^{(\log_b a)} = n^{(\log_2 2)} = n$

This belongs to Case 2. Since k=0, $T(n) = n^{(\log_b a)} \log^{k+1}(n)$

Hence, T(n) = nlogn