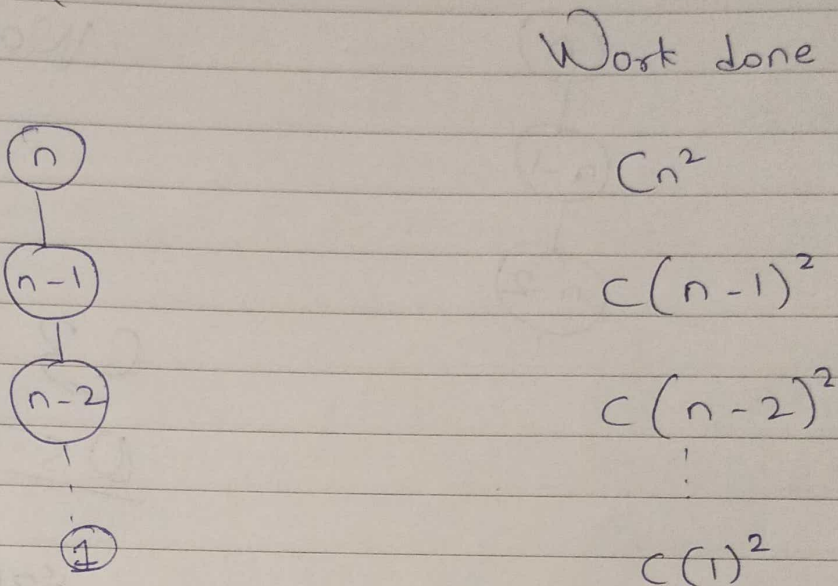
Group Members :-

1) Chetna Ajay Nainani
2) Rachana Vimal Gandhi
3) Harsh Pradeep Nahar

CSS 660
Homework - 1

Q1 Solution:

1) $T(n) = T(n-1) + n^2$



Work done will be

$$C \sum_{1} n^2 + (n-1)^2 + (n-2)^2 + \dots 1$$

$$\sum_{n=0}^{\infty} n^2 + n^2 + n^2 + \dots + n^2 + 1 + 2^2 + \dots$$

$$2n - 4n + \dots$$

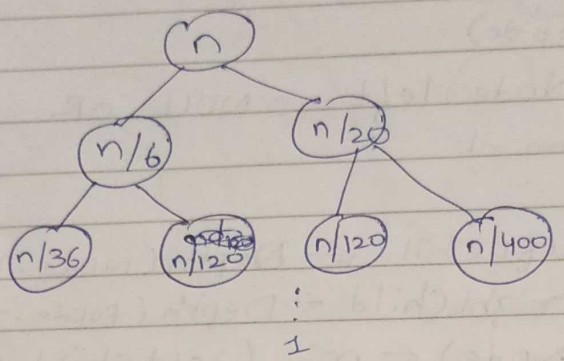
$$\frac{n(n+1)(2n+1)}{6}$$

Big O T(n) will be $O(n^3)$

(2)

2) $T(n) = T(n/6) + T(n/20) + O(\sqrt{n})$

Work Done
$c\sqrt{n}$



$c\sqrt{n/6} + \sqrt{n/20} = c\sqrt{n}\left(\frac{1}{\sqrt{6}} + \frac{1}{\sqrt{20}}\right)$

$c\sqrt{n}\left(1/6 + 2/\sqrt{120} + 1/20\right)$

$= c\sqrt{n}\left(1/\sqrt{6} + 1/\sqrt{20}\right)^2$

So, Work done at i^{th} level will be
$c\sqrt{n}\left(1/\sqrt{6} + 1/\sqrt{20}\right)^i$
As the work done is dominated by the 1st level, hence it will be $O\sqrt{n}$

3) $T(n) = 4T(n/9) + O(n)$
By using Masters theorem,
$a = 4, b = 9, d = 1$
$d > \log_b a$

$1 > 0.6$
$T(n) = O(n)$

4) Show that $2^{2n} \neq O(2^n)$
Let's assume $2^{2n} = O(2^n)$
Using log we get
$2n \lg 2 <= \lg c + n \log 2$
$2n - n <= \lg c$
$n <= \lg c$
We can see, this is false since c is fixed and n is not bounded
$\therefore 2^{2n} \neq O(2^n)$

Q2

Solution:

Show that $\sum_{k=1}^{n} \log^2 k = \Theta(n \log^2 n)$

Expanding $\sum_{k=1}^{n} \log^2 k$

$$\sum_{k=1}^{n} \log^2 k = \log^2 1 + \log^2 2 + \cdots + \log^2 k \rightarrow (1)$$

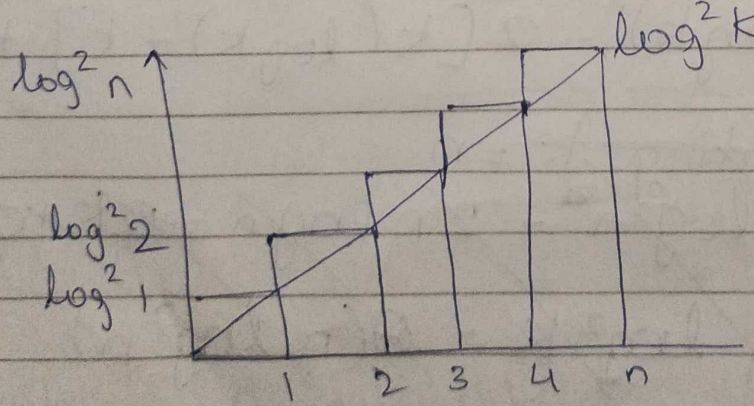
Hence,

$$\log^2 1 + \log^2 2 + \cdots + \log^2 n < \underbrace{\log^2 n + \log^2 n + \cdots \log^2 n}_{n \text{ times}}$$

$$\therefore \sum_{k=1}^{n} \log^2 k < n \log^2 n$$

$$\sum_{k=1}^{n} \log^2 k = O(n \log^2 n) \rightarrow (2)$$

Using Approximation by Integral,



Area under the bones - $\sum_{k=1}^{n} \log^2 k$

Area under the curves = $\int_{1}^{n} \log^2 k \, dk$

Area under the bones $\geq$ Area under the curve

$$\sum_{k=1}^{n} \log^2 k \geq \int_1^n \log^2 k \, dk$$

$$\int_1^n \log^2 k \, dk = \int_1^n (\log k)^2 \cdot 1 \, dk$$

$$\int u\,dv = uv - \int v\,du \qquad (\text{by } \int u\,dv \text{ rule})$$

$$= (\log k)^2 \int 1\,dk - \int \frac{d}{dk}(\log^2 k) \int 1 \cdot dk\,dk$$

$$= k(\log k)^2 - 2 \int \log(k)(1)\,dk$$

$$= k(\log k)^2 - 2\left[(\log k)\int 1\,dk - \int \frac{d}{dk}(\log k) \int dk\,dk\right]$$

$$= k(\log k)^2 - 2[k(\log k) - \int 1/k \cdot k\,dk]$$

$$= k(\log k)^2 - 2[k(\log k) - k] + C$$

classmate
Date
Page

$$= [k(\log k)^2 - 2k(\log k) + 2k]_1^n + C$$

$$= n(\log n)^2 - 2n(\log n) + 2n - 1(\log 1)^2 + 1\log(1) - 2 + C$$

$$= n(\log n)^2 - 2n(\log n) + 2n - 2 + C$$

$$\therefore \sum_1^n \log^2 k = \Omega(n \log^2 n) \rightarrow (3)$$

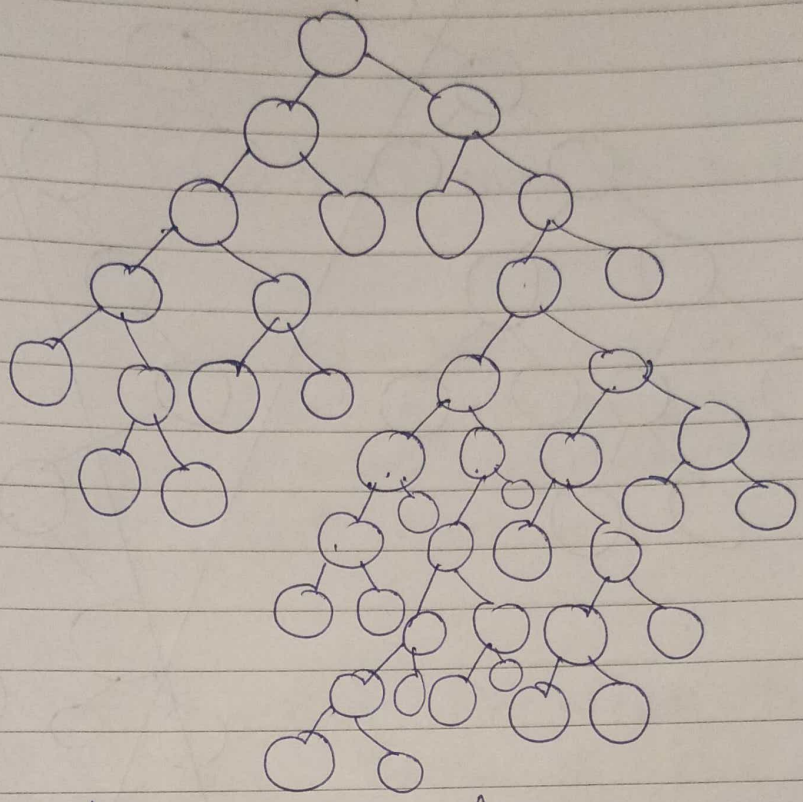
(Because area under the curve < area under the boxes)

Now, if we examine (2) and (3) we conclude that both functions grow at the same rate

$$\therefore \sum_1^n \log^2 k = \Theta(n \log^2 n)$$

Q3 Problem 37 on page 64 of Erickson

Solution:-



In this figure, the largest complete subtree of this binary tree has depth 3

Algorithm: →

```
DepthCalculation (Node)
If Node.left == NULL OR Node.right == NULL
    return -1
// Checking left and right children
else
    leftNode ← DepthCalculation (Node.left)
// Recursion call for left
    rightNode ← DepthCalculation (Node.right)
// Recursion call for right
DepthCalculation (Node) ← min( leftNode, rightNode) + 1
return DepthCalculation (Node), Node
```

Understanding:→

Depth Calculation is a recursion-based method that evaluates the depth of all binary trees. The algorithm analyzes the best case to verify if the node is a child; if not, it iteratively performs DepthCalculation on the provided node's left and right subtrees. Choosing the smallest of the left and right children results in a full binary tree with maximum depth, because all of the nodes have been visited, the runtime will be $T(n) = O(n)$

Q4) Problem 32 on Page 61 of Erickson

To prove that a local minimum must exist in the array using induction on n (base case n=3

Solution:

Proof using induction :→

for base case n=3

Given: A[1 ... n]

condition to be local minimum for any element x, such that

$A[x-1] \geq A[x]$

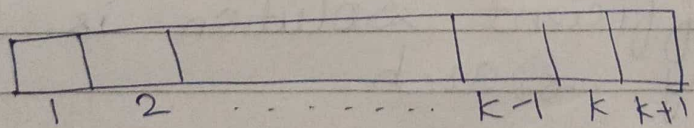
and $A[x+1] \geq A[x]$

Here,

$A[1] \geq A[2]$ & $A[n-1] \leq A[n]$

$A[1] \geq A[2] \leq A[3]$

let $n = k+1$



if $A[1] \geq A[2]$ & $A[k] \geq A[k-1]$

local minima in array

if $A[n-1] \leq A[n]$

and $A[k] \leq A[k+1]$

Comparing these two if $A[1] > A[2]$, $A[k+1] > A[k]$ then $A[k]$ is a local minima then we check in $A[1..k]$ for $A[k] > A[k-1]$

if A[k-1] > A[k] then A[k] will be the local minima

or A[k-1] will be the local minima by induction

Algorithm :→

```
minima (array, Start, end)
  mid = (Start + end) / 2

  if (array[mid-2] ≥ array[mid-1] AND
      array[mid-1] ≤ array[mid])
      return array[mid-1]

  else if (array[mid-1] ≥ array[mid-2])
      return minima (array, Start, mid)

  else
      return minima (array, mid, end)
```

// This efficient Solution is based on binary Search

Analysing :→

The minima function is going to return the minima using binary Search, to iterate through n elements, we take O(n) time in binary Search, the call can go to either half of the

array which would cost $T(\frac{n}{2})$ and have the work done is constant, which is $O(1)$

So,

$$T(n) \leq T(n/2) + O(1)$$

Comparing with Master's theorem -

$a = 1$, $b = 2$, $d = 0$

$$T(n) = O(n^d \log n)$$

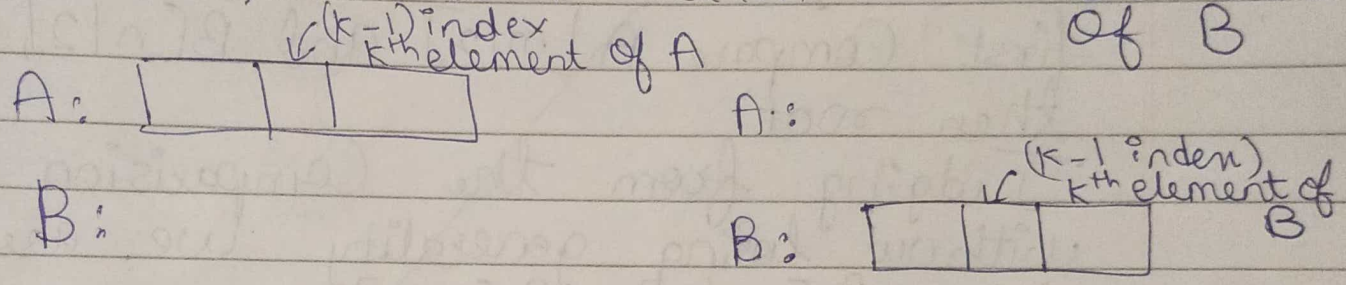
$$T(n) = O(n^0 \log n)$$

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

Solution:

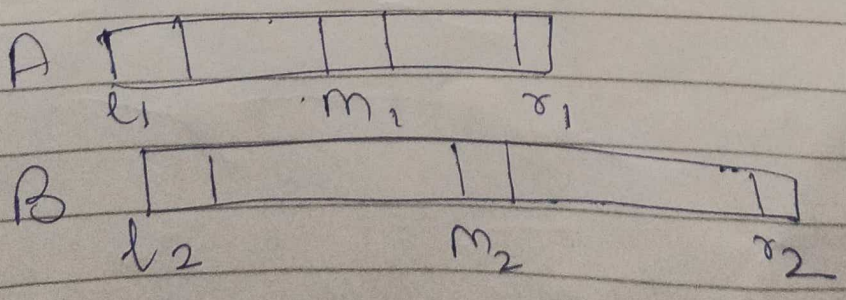
Q5 ★ Base Case :→

Let's say that 'A' our first array had some elements but 'B' is empty. So, if we merged these two arrays, we would just get 'A' and we will return k-1 index (kth element) of A Similarly, if A is empty and B had elements, we will return kth element of B



★ General Case :→

If we have two arrays and they both have values in them then we will keep along a left index and right index and we will consider medians of these two. Left index is denoted with l_1 and l_2 and right index is denoted with r_1 and r_2.



If we look at constraints of input both arrays are sorted but there's no promise of any ordering between them.

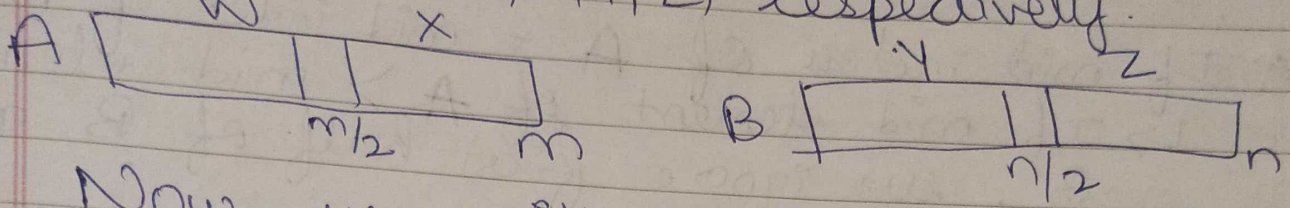
* If mid index of A + mid index of B < k:
  - ↳ If mid element of A > mid element of B, we ignore first half of B and we will adjust k.
  - ↳ else, we will ignore first half of A & we will adjust k
* If k < sum of mid indices of A and B:
  - ↳ If mid element of A > mid element of B, we ignore second half of A
  - ↳ else we ignore second half of B

Algorithm :→ length(arr1) = n , length(arr2) = m

```
kthSmallest (arr1, arr2, k)
if length(arr1) = 0                    // base case
        return arr2[k]
else if length(arr2) == 0
        return arr1[k]
mid1 = length(arr1)/2                  // calculate mid1 & mid2 of
mid2 = length(arr2)/2                  // arr1 & arr2 respectively
if mid1 + mid2 < k
    if arr1[mid1] > arr2[mid2]
        return kth Smallest (arr1,
        arr2[mid2+1 ... n], k-mid2)
    else
        return kth Smallest (arr1[mid1 ... m],
        arr2, k - mid2)
else
    if arr[mid] > arr2[mid]
    return kthSmallest (arr1[1 ... mid1], arr2, k)
    else return kth Smallest (arr1, arr2[1 ... mid2], k)
```

Complexity

Let A and B are two sorted arrays and they are divided into 2 halfs (W, X, Y, Z) respectively.



Now, we will consider all four cases:

1 :→ $k < m/2 + n/2$

$A[m/2] > B[n/2]$

$A[n/2] < B[n/2]$

2 :→ $k > m/2 + n/2$

$A[m/2] > B[n/2]$

$A[m/2] < B[n/2]$

Correctness :→

We can see that in every iteration either A or B is halved & this process is repeated until kth smallest element is found.

So, the time complexity is $O(\log m + \log n)$