Theoretical Assignment 1

Divyansh Mathur

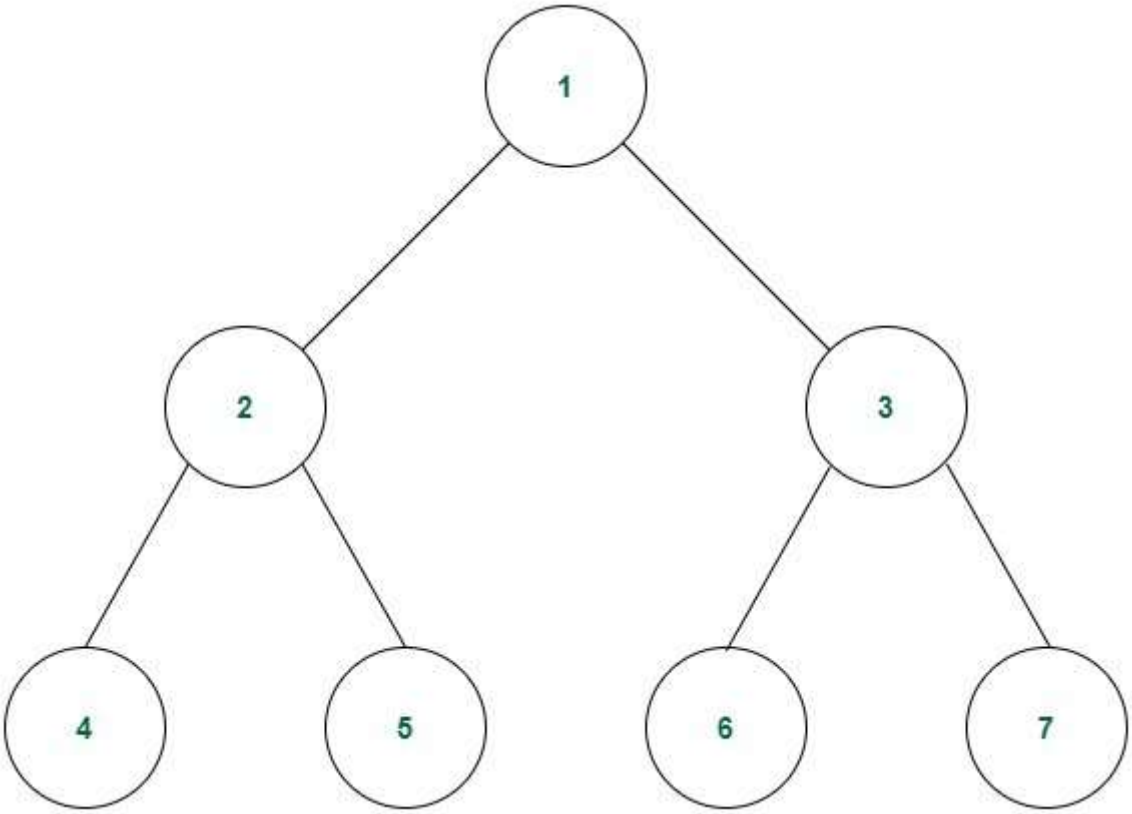
210357

Question 1

Divyansh Mathur, 210357

a.) To design an algorithm in $O(n^3 log(m))$ complexity to find the final wealth of each company after m months.

Company hierarchy:



Where 1 is root company and 4,5,6,7 is leaf company.

Observation:

Consider a node which is neither root nor leaf company i.e. it has both parent and child. Let its wealth after certain years be **W** and its parent wealth be **W_parent**. After 12 months or 1 year **W_new** will be $\mathbf{2^{11} W + 2^{10} W\_parent}$ because it gives half of its wealth to its child and receives quarter of its parent wealth.

{after 12 years parents wealth $=2^{12} \times W\_parent$

After 12 years node's wealth $=2^{12} \times W$

After distributions $W\_new=2^{11} W + 2^{10} W\_parent$}

For root node it will be simply $2^{11}W$.

And for leaf node it will be $2^{10} W\_parent + 2^{12}W$

So if we are given m months it will have y=floor(m/12) years whose wealth is given by above and k=m%12 months still remaining. Therefore after we have wealth after y years just multiply each node wealth by 2^k to get the final wealth.

Designing Algorithm: Since the new wealth of any node depends either only on itself or on parent and itself, we define a column matrix W_y of N rows such that i^{th} row depicts wealth of **i+1** node after y years.

#numbering is starting from 1

$$\boldsymbol{W_{y+1}}_{\mathbf{N \times 1}} \qquad \boldsymbol{W_y}_{\mathbf{N \times 1}}$$

$$\begin{vmatrix} W_{y+1}[1] \\ W_{y+1}[2] \\ W_{y+1}[3] \\ W_{y+1}[4] \\ . \\ . \\ W_{y+1}[n] \end{vmatrix} \qquad \begin{vmatrix} W_y[1] \\ W_y[2] \\ W_y[3] \\ W_y[4] \\ . \\ . \\ W_y[n] \end{vmatrix}$$

Since we know the relationship between W_{y+1} and W_y for each node we can set a relationship between these column matrices.

NOTE: There will be L=(N+1)/2 leaf nodes and 1 root node

$$\boldsymbol{W_{y+1}}_{\text{N x 1}} \quad \boldsymbol{M}_{\text{N x N}} \quad \boldsymbol{W_y}_{\text{N x1}}$$

$$\begin{vmatrix} W_{y+1}[1] \\ W_{y+1}[2] \\ W_{y+1}[3] \\ W_{y+1}[4] \\ . \\ . \\ W_{y+1}[n] \end{vmatrix} = \begin{vmatrix} 2^{11} & 0 & 0 & 0 & . & . & 0 \\ 2^{10} & 2^{11} & 0 & 0 & . & . & 0 \\ 2^{10} & 0 & 2^{11} & 0 & . & . & 0 \\ 0 & 2^{10} & 0 & 2^{11} & . & . & 0 \\ . & . & . & . & . & . & 0 \\ . & . & . & . & . & . & 0 \\ 0 & 0 & 0 & 0 & . & . & 2^{12} \end{vmatrix} \times \begin{vmatrix} W_y[1] \\ W_y[2] \\ W_y[3] \\ W_y[4] \\ . \\ . \\ W_y[n] \end{vmatrix}$$

Since 1based indexing, parent index of any node i will be given as floor(i/2)

Last L nodes will not have 2^{11} but they'll have 2^{12} because they are leaf nodes and they don't distribute their wealth.

$$\mathbf{W_{y+1} = M \times W_y}$$

Therefore,

$$\mathbf{W_y = M^y \times W_0}$$

(Where W_0 denotes the initial wealth of each company)

Now, W_y can be found by Matrix exponentiation.

Pseudocode:

```
MULT( matrix a, matrix b){
    for i from 1 to n
        for j from 1 to n
            for k from 1 to n
                ANSWERMATRIX+= a[i][k] * b[k][j]

    return ANSWERMATRIX
}
Power(M, y){
    if (y=0) return 1;
    else {
        temp = Power(M, y/2);
        temp = MULT(temp , temp);
        if (y mod 2=1 ) temp = MULT(temp , y) ;
        return temp;
    }
}

Wealth_final
{
    Matrix M ^ (m/12) calculated as above and let it be P matrix
    now multiply it with 2^(m%12) * initial wealth column matrix W0
    Ans= column matrix showing wealth after m months
    for i from 1 to n
        Ans[i]=0
        for j from 1 to n
            Ans[i]+= P[i][j]*W0[j]*2^(m%12) // this multiplication takes O(1)
time since m%12<12

    Ans column matrix has final wealth // this whole step takes O(n^2) time

}
```

b.) To analyse the time complexity of your algorithm and briefly argue about the correctness of your solution.

Time Complexity:

Matrix exponentiation: Power function is called almost log(y) {y is years} times and each time it is called it calls for 1 or 2 matrix multiplications of 2 matrices, each matrix multiplication takes N^3 time since the size of matrix M is N x N. This takes N^3 log y time order of N^3 log m, { *m=12*y}*

After the above step we also multiply $2^{m\%12}$ to each node, this takes $O(N^2)$ time.

Therefore total time complexity is $O(N^3 \log m + N^2)$ **$= O(N^3 \log m)$.**

Proof of Correctness:

Theorem: At the end of y years wealth of a non-leaf, non- root node is given by

$$W_y = M^y \times W_0$$

Where W_0 is the initial wealth of each node, and M is the Matrix discussed above.

Proof:

For y=0 →

$$W_0 = M^0 \times W_0$$

$$W_0 = W_0$$

Assume W_y is correctly given as $M^y \times W_0$, then using the discussed Observation {

Consider a node which is neither root nor leaf company i.e. it has both parent and child. Let its wealth after certain years be **W** and its parent wealth be **W_parent**. After 12 months or 1 year **W_new** will be $\mathbf{2^{11} W + 2^{10} W\_parent}$ because it gives half of its wealth to its child and receives quarter of its parent wealth.

}

We can say ,

$$W_{y+1} = M \times W_y$$

Replacing W_y with $M^y \times W_0$ we get

$$W_{y+1} = M^{y+1} \times W_0$$

Hence Proved.

c.) Consider the case of a single company (i.e. only root) in the tree. Give a constant time solution to find the final wealth after m months.

Given single node root. It has no parent, no child. Therefore every month its wealth just doubles. Therefore, after m months its wealth would be 2^m times its original wealth. Assuming 2^m or pow(2,m) can be calculated in constant time we can simply return $2^m \times W_0$ where W_0 is the initial wealth.

Question 2

Divyansh Mathur, 210357

Simplifying the question: Given an array of n integers we have to find the minimum value of j-i+1 where 0<=i<=j<n and A[i]+A[i+1]….A[j] >=P

The cost will be C x ans from above question.

a.) To Design an algorithm in O(n) time complexity for determining the minimum cost room allocation

Designing Algorithm: We want the smallest length subarray such that its sum is greater than P. Since the capacity of a room cannot be negative, **if we take a sum of subarray, it will always be greater than if we left some front or rear elements**.

We first set our pointers low and high to 0. So, we first find an index high such that sum of all elements from A[low] to A[high] >= P. Once we find the high, we move the low forward and reduce the sum until this sum value becomes less than P. At each iteration we check if the condition is valid, and we check if this (high-low) is the minimum of all plausible cases. Once the sum is less than P we again increase high until sum exceeds or equals P.

We break our iterations once high==n.

Overall Time Complexity: We are approximately visiting every element almost 2 times. T(n)= 2n

Therefore O(n) time

Pseudocode:

```
low =0, high=0, sum=0, ans=infinity {a very high value, maybe INT_MAX}
while(high<n){
    while(sum<P && high<n){
        sum+=A[high]
        high++
    }
    while(sum>=P){
        C[low]=high-low
        ans= min(ans,high-low)
        sum-=A[low]
        low++
    }
}
```

b.) To Design an algorithm in O(nlog(n)) time complexity for determining the maximum number of contiguous rooms they can get which satisfy the beauty constraints

Simplifying the Question: We are given an array of n positive integers, we need to find length of maximum contiguous subsequence such that GCD of that subsequence is >=k.

Designing Algorithm: If we are able to find GCD for a given sequence A(i,j) in constant time then we can apply binary search on answer.

Binary Search: So the maximum length of such subsequence can be n, and the minimum length can be 1 (assuming at least one integer is >=k if not simply return 0) , so we need to apply binary search between 1 to n, this takes O(log n) time.

To check whether a length is valid length or not: We check every possible index whether it is the starting point of such contiguous subsequence. Now to suit our overall time complexity this should take maximum O(n) time. So, we need to find GCD of a subsequence (contiguous) in O (1) time.

Find GCD of a subsequence (contiguous) in O(1) time: To do this we can use concept of Range minima Query and range minima data structure. We make a n x log (n) matrix M, such that M[i][j] gives the GCD of all the elements from i to $i+2^j$ (maximum value of j will be log n, concepts of range minima).

Preprocessing: To make the M matrix we need extra space of n log n and extra time of O(nlogn) but it is a onetime thing and our overall time complexity will still remain O(nlogn). To preprocess in O(nlogn) time we need to use the fact that

$$M[i][j] = \text{GCD of } (M[i][j-1] \text{ and } M[i+2^{j-1}][j-1])$$

Such that none of these terms goes out of bounds.

***we are using the fact that GCD of an array from i to j = GCD of GCD of array from i to k and k+1 to j**

Pseudocode for preprocessing:

```
GCD (no1, no2){
    //blackbox function which returns GCD of 2 input
}

POW[logn]{// creating array such that POW[j]=2^j
    POW[1]=2
    for i from 2 to logn:
        POW[i]=2*POW[i-1]
}

LOG[n+1]// LOG[m]: such that the greatest integer k such that 2^k  ≤ m+1

for i from 0 to n-1
{
    M[i][0]=A[i] // since GCD of a no. is the no. itself
}

for j from 1 to logn
{
    for( int i=0; i+ POW[j] -1 < n; i++){
        M[i][j]= GCD( M[i][j-1], M[i+POW[j-1]][j-1])
    }
}
```

Binary Search:

We set low =1 and high =n, for a mid we check is it possible or not, if it is then low=mid+1 else high=mid-1. Checking involves comparing GCD of every contiguous subsequence of that length with k

Pseudocode:

```
low =1, high=n, ans
while (low<=high){
    mid= (low+high) /2
    if(check(mid)==true) {
        ans=mid
        low=mid+1}
    else high=mid-1
}

check(mid){
    for i from 0 to n-mid
        if( findgcd (i,mid) >= k) return true
        else return false
}
findgcd(i, mid){
    j=i+mid-1
    t=POW[mid];
    h=LOG[mid];
    if (t == mid)  return  M[i][h];
    else  return GCD(M[i][h], M[j-t][h]);
}
Print(ans)
```

c.) To Give proof of correctness and time complexity analysis of approach for part (b)

Theorem: C[i] gives the length of the smallest subarray such that its sum >=P if possible otherwise gives no value

Proof: We increment j until we encounter sum>=P for the first time. Once we encounter sum>=P. We get to know the smallest length starting from i where sum is >=P. Point to note is that

Sum_Subarray [i+1,j] < Sum_Subarray [i ,j],

because Sum_Subarray [i ,j]-Arr[i]= Sum_Subarray [i +1,j] and Arr[i] >0

Now if Sum_Subarray [i+1][j] is also >=P then again j-(i+1) +1 gives the length of smallest subarray.

Before the initial loop iteration, i, j, and sum are set to 0, while ans is initialized with a high value symbolizing infinity. Since the subarray arr[0:0] is empty, sum correctly represents the sum (0), and ans is suitably set for minimum cost.

Assuming the loop's validity before each iteration, we consider two scenarios:

1. If sum < P, we increment j and update sum with arr[j]. The loop's validity remains intact as sum stays the subarray sum, j advances, and i remains unchanged.

2. When sum >= P, we enter an inner loop. It repeatedly increases i and decreases sum until sum < P. The loop's validity persists as sum adjusts by subtracting elements, i increments, and j stays constant.

The loop concludes when j exceeds n due to incremental j and finite n. Post-termination, the loop's validity still holds. Thus, ans indeed stores the minimum cost for a subarray sum of at least P.

Hence Proved.

Time Complexity Analysis:

We are approximately visiting every element almost 2 times. T(n)= 2n

Worst case scenario: Take all elements equal to P. all room capacity =P.

Therefore O(n) time

Question 3

Divyansh Mathur, 210357

This question revolves around binary search trees (B.S.T.)

Assumption made: Given BST has all distinct elements

Condition of BST:

All children node VALUE to the left of a particular node are smaller than the node value and children node value to the right of a particular node are greater than the node value.

a.) To identify the swapped nodes

To identify the swapped nodes we need to traverse the BST and find the node which disturbs the condition of BST.

Designing Algorithm:

Therefore, if we devise a traversal in which we write the nodes on the left and then the node and then the nodes on the right then we will get an increasing sequence.

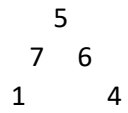
In other words, if we do this traversal and store it in an array the array will be sorted.

What to expect:

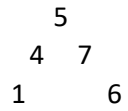
If the given BST is disturbed, then the traversal will not give increasing sequence such that there will be **at least** one node whose value will be smaller than the prev node encountered.

Why at least:

When a parent child pair is swapped it MAY NOT create 2 invalid nodes it may create only one invalid node, consider the case:



CASE- I Traversal here: 1,7,5,6,4 (7 and 4 invalid case)



CASE-II Traversal here: 1,4,5,7,6

Note here only 6 is invalid node!

In This case we will just swap the invalid node with its parent

Algorithm:

We define null node pointer prev ,first_distortion , second_distortion ,first_next such that:

Prev represent just previous node visited

first_distortion represent the previous node to the first node whose value < its prev val

first_next represents the first node referred above

second_distortion represents the node where the distortion occurs for the second time

If Second_distortion remains NULL at the end that means we are having Case I then simply swap first_distortion and first_next,

Else Case II, swap first_distortion and second_distortion.

Pseudocode:

```
Nodes prev,first_distortion, second_distortion,first_next all declared
NULL inititally
traversal(root)// declared later
if(first_distortion and second_distortion both not NULL)
swap(first_distortion val, second_distortion val)
else swap(first_distortion val, first_next val)

traversal(node){
    if(node == NULL) return;
    traversal(left child of node);
    if (prev is not NULL and (node val < prev val))
    {
        if ( first_distortion is NULL )
        {
            first_distortion = prev
            first_next = node
        }
        else
            second_distortion = node
    }
    prev = node;
    traversal(right child of node)
}
```

To Determine Common ancestors:

We use the following principle in BST:

If you are standing on a node then to find a value smaller than this node in its subtree you need to go to the left child and for greater value you need to go to the right child.

Once we have corrected the BST, We do traversal to find its common ancestor. On our traversal 4 case may arise:

1. Both node values is greater than the current node's value: move to right child of this node
2. Both node values is smaller than the current node's value: move to left child of this node
3. One node is > and other is < than the current node' value: This was the last common ancestor, their path part ways now.
4. One of the node value == current node's value: This was the last common ancestor, path of one node ends here.

Pseudocode:

```
Common_ancestor(node)
{
    node is one of common ancestor

    if(node1 val > node val and node2 val > node val){
        Common_ancestor(right child of node)
    }else if(node1 val < node val and node2 val < node val){
        Common_ancestor(left child of node)
```

```
    }else{
        return;
    }

}
```

b.) To determine the value of k and which nodes were rearranged and design an algorithm of complexity O(min(G + n, nlog(n))) for the same

Primitive Algorithm:

We do inorder traversal (LEFT-NODE-RIGHT) of the BST once and compare it with the sorted version of this array and the nodes where the values differ count them. The count value will be K.
It will take T(n)= n log (n) + 4* n time which is O(nlogn) time.

Pseudocode

```
inorder[n]

inorder_traversal(root) //declared below takes n time
dup[] = duplicate(inorder) //takes n time
sort(dup) //takes nlogn time
k=0
for i from 0 to n-1
    if (dup[i]!=inorder[i]) k++ // takes n time
print k

inorder_traversal(root)
{
    if(root is NULL) return;
    inorder_traversal(left child of root)
    push root in inorder[]
    inorder_traversal(right child of root)
}
```

Algorithm Using 'G' the maximum value:

Create a Boolean array of length G, let it be called Present_array. Next we do inorder traversal (LEFT-NODE-RIGHT) of the BST once and store it in an array named inorder, and also mark Present_Array[node val] = true.

Now we traverse the Present_array and inorder array simultaneously (we use 2 pointer one for earch array) .

If an element i is present i.e. Present_array[i]==true, then we see whether was this the node which was meant to come, i.e. we check whether inorder[j] <?> i , if they are equal then simply move ahead i++ and j++ else store i and move ahead i++ j++.

It will take T(n)= G+n. Ovreral: O(n+G)

Pseudocode

```
inorder[n]
Present_array[G] // all false by default
inorder_traversal(root)
{
    if(root is NULL) return;
    inorder_traversal(left child of root)
    Present_array[node val]=true
    push root in inorder[]
    inorder_traversal(right child of root)
}
j=0,k=0, ans_arr[]
for i from 0 to G-1
{
    if(Present_array==true){
        if(inorder[j]!=i ){
            store i in ans_array
            k++
        }
        j++
    }
}
print k
```

Overall Time Complexity:

Min(O(nlogn), O(G+n))

Question 4

Divyansh Mathur, 210357

Simplifying the question:

The problem gives us an array such that it a shifted array version or a sorted array. We need to find the index of the last element in the given array in the sorted array.

Given Array: A[0], A[1]……. A[i]…..A[n-2],A[n-1]

We need to find i such that ***A[i]<A[n-1] and A[i-1]>A[n-1].***

Our answer would be ***n-i-1***

#Assumption made: " he picked a random number k between 0 and n" {0 and n excluded}{

a.) To design an algorithm of complexity O(log(n)) for Joker to find the value of k

The algorithm we would use be a modified version of binary search.

We check for middle element. There are three possibility:

i.) A[mid]<A[n-1] and A[mid-1]>A[n-1]: in this case we return mid

ii.) A[mid]<A[n-1] and A[mid-1]<A[n-1]: in this case our ans will be on the right half of our segment so l=mid+1

iii.) A[mid]>A[n-1] and A[mid-1]>A[n-1]: in this case our ans will be on the left half of our segment so r=mid-1

*no other case exists because the array is otherwise sorted

NOTE:

1. Shifted array version: Given a sorted and rotated array **A[]** of size **n** and a **key**, the task is to find the position of the key
2. All elements are distinct

Pseudo Code:

```
low= 0, high = n-1
found= false
while( found is false and low <= high){
    mid =(low + high)/2
    if( A[n-1]> A[mid] && A[n-1]< A[mid-1]){
        i=mid
        found=true
    }else if( A[n-1] > A[mid] && A[n-1] > A[mid-1]){
        r=mid-1
    }else{
        l=mid+1
    }
}
print( n-i-1)
```

b.) <u>To provide time complexity analysis for this strategy</u>

At each stage we are checking whether which condition should be followed in constant time and after each iteration we are dividing the array in 2 equal segments of which we are considering only 1 segment.

Length of array =>

N -> N/2 -> N/4……1

Therefore time complexity is O(log(N))

Question 5

Divyansh Mathur, 210357

a.) <u>To design a strategy to find number of palindromic substrings in the hidden string so your crew can safely escape from this region in less than or equal to O(n*log^{2}n)</u>

<u># in whole question 0 based indexing is used</u>

<u>Brute force:</u>

The brute force approach is easily visible, viewing every substring and then counting ++ if this substring is palindrom. But this will take O(n^2) time.

<u>Observation:</u>

1. Odd sized palindrome: Consider an index i, if we assume the length of the largest palindrome such that i is its middle element to be k, then there will be (k+1)/2 such palindrome centered at i.
2. Even sized palindrome: Similar approach works here also. If i, i+1 makes a palindrome. Then check for the longest length (front from i+1 and rear for i) such that i,j makes a palindrome.

<u>Designing Algorithm:</u>

First, we will count palindrome for odd length and then for even length and then sum their answer. Now, to do this in optimal time we would use **binary search on answer**. Let we are standing on an index i, the minimum odd sized palindrome will be of length 1 where extent on right/left=0 (we are counting the extent of palindrome to right and left) and the maximum we can go right or left will be M=min{ i, n-i-1} **(0 based indexing)** now we need to find maximum length k such that S(i-k,i+k) [0<=k<=M] is palindrome also we will add k to the ans.

Then we will check if i, i+1 can be middle element of a palindrome {By ispalindrome(i,i+1)}. If they are then again do binary search minimum again being 0 and maximum being M = min{i,n-i-2} (0 based indexing) now we need to find maximum length k such that S(i-k,i+1+k) [0<=k<=M] is palindrome.

<u>Time Complexity:</u>

In the worst case we might end up applying both the binary search (even and odd) on a index, and there are n index so it may make up to 2*n*logn operations. Which is less than n*log^{2}n operations.

<u>Pseudocode:</u>

```
ispalindrome(i,j){
    blackbox function which return true or false whether S(i,j) is palindrome
in O(1) time
}
sum=0
for i from 0 to n-1
{
    low=0, high= min{i,n-i-1}, ans
    while(low <= high){
        mid=(low+high)/2
```

```
        if(ispalindrome(i-mid,i+mid)== true){
            ans=mid
            low=mid+1
        }else{
            high=mid-1
        }
    }
    sum= sum+ ans+1 // +1 because length 0 also counts as palindrome
    if( i<n-1 and ispalindrome(i,i+1)==true){
        low=0, high= min{i,n-i-2}, ans=0
        while(low <= high){
            mid=(low+high)/2
            if(ispalindrome(i-mid,i+mid+1)== true){
                ans=mid
                low=mid+1
            }else{
                high=mid-1
            }
        }
        sum= sum+ans
    }
}
print(sum)
```