ESO207 THEORY ASSIGNMENT 1

Submitted by: Aditi Khandelia

5th September 2023

1
(a)

This question can be solved using matrix exponentiation encountered in the fibonacci series problem.
At the end of each year W_i is multiplied with 2^{11} whilst also being added to 2^{10}*W_{i-1}, where W_i is the wealth of the node at the i^{th} level.
Thus, the recurrence relation is W_i= 2^{11}*W_i + to $2^{10}*W_{i-1}$.
This operation is carried out $log_1$2m times.
Additionally, we must create a matrix that stores the initial wealths of each family chain. The length of each chain will be log_2(n+1). The number of family chains will be $\frac{n-1}{2}$. Let this matrix be W[log_2(n+1)][$\frac{n-1}{2}$].
We assume multiplying by a multiple of 2 takes unit time using right shift operation.

```
findwealth(W,m,n)
{

    y=$\frac{m}{12}$;
    //initialise the matrix that solves the recurrence relation.
    M = $\begin{bmatrix} 2^{11} & 0 & \cdots & 0 & 0 \\ 2^{10} & 2^{11} & 0 & \cdots & 0 \\ 0 & 2^{10} & 2^{11} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 2^{10} & 2^{12} \end{bmatrix}$
    M=power(M,$log_2$(n+1),y);
    W=matrixmultiply(M,W,$log_2$(n+1),$\frac{n-1}{2}$,$log_2$(n+1));
    return W;

}
```

Finally all wealths can be multiplied by $2^{m mod 12}$ in O(n) time using right shift operation.

matrixmultiply(A,B,rows,common,columns):

```
    //Create an empty matrix C[rows][columns]
    for(i=0;i<rows;i++)
        for(j=0;j<columns;j++)
            for(k=0;k<common;k++) C[i][j] += A[i][k]*B[k][j];
    return C;
```

The pseudocode to calculate the matrix raised to a particular value power(M,n,y)

```
{
    if(y==0) return M;
    else:
        M'=matrixmultiply(power(M,n,y/2),power(M,n,y/2),n,n,n);
        if(y is odd) M=matrixmultiply(M',M,n,n,n);
        else M=M'
        return M;
}
```

(b)
(1)Time Complexity:
Since we have followed the algorithm of raising a matrix to a particular power using matrix exponentiation, time complexity of the step is n^2*log_2(m), where n is the size of the matrix. Since our matrix has size log_2(n), time complexity of this step is O($(logn)^2$*log_2(m)).

We also multiply the matrix of wealths with the matrix created above. The time complexity of this is O($(logn)^2 * \frac{n-1}{2}$).

Final time complexity = O($(logn)^2 * log_2$(m)) + O($(logn)^2 * \frac{n-1}{2}$)= O($n^3 log_2$n)

(2)Correctness
The correctness argument lies in the fact that the recurrence relation is able to capture the wealth of each company in a particular level using the wealths of the company in the previous level.Using PMI:
P(1) is true since the wealth of the parent company is first multiplied by 2^{12} and then halved since wealth is distributed amongst children. So the first row of W stores correct wealth.
Suppose P(i-1) is true. Thus, the recurrence relation stores correct wealths for the $(i-1)^{th}$ row of W.
Then, P(i) is obtained by adding 2^{10} of the values in row i-1 to 2^{11} of the values in row i, which is the correct mathematical expression for calculating the wealth. Additionally since leaf nodes don't distribute wealth, their recurrence relation has 2^{12} replacing 2^{11}.
Thus, proved.

(c)
For a single company, the entire operation is effectively multiplying the wealth with 2^m since the wealth is never halved(the company has no children).

```
calculatewealth(wealth,m)
{
    wealth = wealth << m;
    return wealth;
}
```

2.
(a)

This problem can be solved using a two-pointer approach.
We start by calculating the subarray with minimum no. of rooms beginning from a particular index i till the occupancy of the subarray is greater than or equal to p, or till we reach the end of the array.
If we are able to find such a subarray, we set the min. cost for that index to be c*num, where num is the number of rooms in that subarray. If we can't find any such subarray, cost is set to be c*n + 1.
This is done for all i<n where n is the size of the input array.Additionally, when we move i to the right, we use the subarray calculated before to find the subarray with min. no. of rooms.

```
calculateminrooms(arr, n, c, p)
    left=0;
    right=0;
    currsubarraysum=0;
    currocc=0;
    currmincost=c*n+1;
    while left<n:
        currocc=currsubarraysum;
        while currccc < p and right < n:
            currocc+=arr[right];
            right++;
        currsubarraysum = currocc - arr[left]; //to be used in the
        next iteration
        if left==right and currocc>=p return c; //allocation with
        a single room
        else if currocc >= p arr[left]=c*(right - left); //min cost
        subarray beginning from i
        else arr[left]=n*c+1 //no possible allocation
        if arr[left]<currmincost currmincost=arr[left]; //this will
        store the minimum cost possible uptil now.
        left++;

    return currmincost;
```

If the program returns c*n+1, no room allocation is possible. Otherwise, the cost returned is the minimum possible cost.

b.
Suppose the input array is arr of size n where arr[i] contains the capacity of room i+1.
First part of the problem involves the computation of a table similar to the one encountered in range-minima problem. This is going to be a 2-dimensional table whose index [i][j] stores the gcd of all elements beginning from the i^{th} to the $(i+2^j - 1)^{th}$ index.
Let the input array be arr of size n and the created 2-dimensional array be arr1.
Let the blackbox gcd algorithm be gcd(a,b), where a and b are the input integers.

```
create(arr,arr1,n)
{

    i=0;
    for(;i<ceil(log2(n));i++)

        j=0;
        for(;j<n;j++)
            k = j + $2^{i-1}$ //done using right-shift operator,
            takes constant time
            if i==0: arr1[j][i]=arr[j][i];
            else if k<n: arr1[j][i] = gcd(arr1[j][i - 1], arr1[k][i
            - 1]);
            else: arr1[j][i] = arr1[j][i - 1];

}
```

Next, we write a function that calculates the max no of contiguous rooms possible by applying a binary search approach on all possible no. of room allocations(0 to n)

```
findmaxrooms(subarraygcd, n, k)
{

    left = 0;
    right = n;
    maxrooms=0;
    while (left<=right):
        num = (left+right)/2;
        if contiguousrooms(arr1, n, k, num):
            maxrooms=num; left = num+1;
        else:
            right=num-1;
    return maxrooms;
}
```

Lastly, we need to write the function that uses the arr1 matrix to see if there is a section of the given capacity array of a given size that has gcd >= k

```
contiguousrooms(arr1, n, k, num):
{
    for (int i=0; i<=n-num; i++)
        int j = log2(num);
        int gcdval =gcd(arr1[i][j],arr1[i+num-$2^j$][j]);
        if gcdval>=k: return 1;

    return 0;
}
```

Upon execution of findmaxrooms, if we get 0, we conclude that no room allocation is possible.Otherwise we print the no. returned by the function.

(c)

(1)Proof of Correctness:
We can prove correctness of the algorithm in (a) using the Principle of Mathematical Induction.
This is done by applying PMI on i such that when left==i, the termination of the loop inside the function results in the storage of the min cost allocation possible for all subarrays beginning from j such that 0<=j<=i in the variable "currmincost". Since the loop executes till i=n-1, the final currmincost stores the minimum cost allocation possible after having considered all subarrays.

P(1): left=0.
Since in the first iteration right is incremented till we reach the end of the array or till we get a room allocation with the required capacity, we are able to check every subarray beginning with the index 0.
P(1) is true.

P(n-1): left=n-2 .
Assume this to be true.
Thus, at the end of the loop currsubarraysum will store the capacity of the room allocation with the min. cost brginning from the index (i-1) - capacity of room (i-1).Right will store the index of the end point of this subarray(proven in time complexity analysis).

P(n): left=n-1
Since right is already at the end point of the subarray with min. cost allocation uptil now without the capacity of index (i-1).
In case the subarray we have right now has capacity>=p, there is no need for us to increment right since any more additions top the subarray will uselessly increase cost.
If however capacity<p, we can begin incrementing right till we again get a subarray with the min. cost of allocation. Thus, at the end of this iteration, we will have checked all possible subarrays beginning with index<=i.

As we check the subarrays, we keep updating the min. cost. Thus, the function finally gives the right answer.

(2)Time Complexity Analysis:
The loop begins to execute with left==right and executes till left¡n.
We can prove that at the end of each iteration, right>=left using PMI.
P(1):
In the beginning, right==left and <n. Thus, the inner loop is executed and right is incremented.
P(1) is true.
Suppose P(i-1) is true. I.e, at the end of the $(i-1)^{th}$ iteration, right>=left.
P(i):
At the beginning of the iteration, right>=left. If right-left==1, P(i) becomes true since left is only incremented once and right is not decremented anywhere.
Suppose right-left==0. If currocc<p, right is incremented and right-left==1. Thus, P(i) becomes true.
If currocc>=p, loop is not executed. However, the instruction: if left==right and currocc>=p return c; //allocation with a single room then terminates the loop.
Thus, P(i) is true.

Hence, at the end of each iteration, right>=left. Thus, right inner loop is executed at most n times. Similarly the outer loop is executed at most n times.

$$T(n)=O(n)$$

3.
(a)
This problem requires inorder traversal of a binary tree.
In inorder traversal, all the nodes present in the left-subtree of a given node are listed first, followed by the given node, followed by all the nodes in the right sub-tree of the given node.

For a binary search tree, this results in a sorted ascending array of all the keys present in the BST.

algorithm for inorder traversal:

```
inorder(p, arr):
{
    while(p≠NULL):
        inorder(p→left, arr);
        append(arr,p.val);
        inorder(p→right, arr);
}
```

Here p is the root of the BST and arr stores the keys in ascending order.
Since two nodes have been swapped in our BST, we'll get two indices a, b such that arr[a]<arr[a-1] and arr[b]<arr[b-1].
In the case where two adjacent nodes are swapped, we'll get one such index only.
The nodes that are swapped are indicated by a-1 and b.
We can write a code to traverse the array arr to initialise the values of two swapped nodes as r and q.

```
traverse(arr, n, r, q)
{
    i=1;
    flag=0;
    while(i<n):
        if arr[i]<arr[i-1]:
            if !flag:
                r=arr[i-1];
                q=arr[i];
                flag=1;
            else
                q=arr[i];
        i++;
}
```

To identify the common ancestors, we can implement a level-order search on BST. The nodes storing value in the middle will be the common ancestors.

```
findancestors(p,r,q)
{
    if p.val>r and p.val>q:
        findancestors(p→left,r,q);
    else if p.val<q and p.val>q:
        findancestors(p→right,r,q);

    else:
        cout<<p.val<<endl;
        findancestors(p→left,r,q);
        findancestors(p→right,r,q);
}
```

(b)

First in-order traversal produces an array storing the node values. The algorithm for this has been specified in the problem above.

For this problem, we need to consider two cases:
Case 1: $G>nlog_2(n)$
In this case, merge sort is used on arr(array storing the nodal values) to craete a sorted array. arr2[n]=mergesort(arr,n);
Case 2: $G<nlog_2(n)$
We will use count sort to sort our array arr. The time complexity of count sort is O(n+G). Since for large n, O(n + G) <= O(nlog_2(n)),we use count sort instead of mergesort.

Count Sort:
countsort(arr,n,G)

```
    count[G+1];//frequency array initialised to 0 in time O(G).
    for(i = 0; i<n; i++) count[arr[i]]++;
    //count array is changed to store the positions of each element
    for (i=1; i<=G; i++) count[i]+=count[i - 1];
    int arr2[n];
    for (i=n-1; i>=0; i–)
        arr2[count[arr[i]] - 1] = arr[i];
        count[arr[i]]–;
```

We now have two arrays, arr and arr2, one storing the original nodal values' sequence and the other storing the sorted sequence.
We can now run a linear time algorithm to compare the two arrays at each index. The no. of dissimilarities is equal to k. Also, the values at the points of dissimilarities are the swapped nodal values.

```
findswappednodes(arr,arr2,n)

    k=0;
    for(i=0;i<n;i++)

        if arr[i]!=arr2[i]

            cout<<arr[i];
            k++;

    cout<<k;
```

4.
(a)
The question requires us to determine the index of the maximum element in the rotated array, which can then be subtracted from n-1 to give the number of elements that underwent rotation.

Given an input array arr of size n, the function that returns the index of the maximum element is:

```
find(arr, left, right) //initial values of left and right: 0, n-1
{

    //specify the base case
    if (left==right):

        return left;

    else:

        mid=(left+right)/2;
        if(arr[mid]>arr[mid+1]) return mid;
        else if(arr[mid]>arr[0]) return find(arr,mid+1,right);
        else return find(arr,left,mid);

}
```

We can subtract the output of this function from n-1 to give the final answer.

(b)
The algorithm developed is:

```
find(arr,left,right){

    //specify the base case
    if (left==right):

        return left;

    else:
        mid=(left+right)/2;
        if(arr[mid]>arr[mid+1]) return mid;
        else if(arr[mid]>arr[0]) return find(arr,mid+1,right);
        else return find(arr,left,mid);

}
```

Suppose the input size is n, then running find first requires checking against base statement that requires constant time.
Thus, the section of the code requiring constant time is:

```
if (left==right):

    return left;

else:

    mid=(left+right)/2;
    if(arr[mid]>arr[mid+1]) return mid;
```

Let the no. of instructions here be c.
In the else block, we make recursive calls that effectively pass an array of size (n/2).
Thus, the recursive relation is

$$T(n) = c + T(n/2)$$
$$\rightarrow T(n) = c + c + T(n/4) = c + c + c + T(n/8)$$

This process continues till we reach the base case. The base cases consists of an array of size 1 which is tested against the if statement, the execution of which requires constant time.
Thus, T(1) = O(1).

$$\rightarrow T(n) = c+c+c+...+c(log_2(n) \text{ times}) + T(1) = c^* \, log_2(n) + O(1)$$
$$\rightarrow T(n) = O(log_2(n))$$

5.
(a)
The approach to this problem makes use of the fact that any palindrome of odd length has a single centre, while any palindrome of even length has two centres.

```
countoddpalindromes(count, arr, n)

    i=0;
    while i<n:

        l=binarysearch1(arr,i,0,n);
        count+=l;
        i++;

countevenpalindromes(count, arr, n)
```

```
    i=1;
    while(i<n):
        l=binarysearch2(arr,i,0,n);
        count+=l;
        i++;

binarysearch1(arr, i, low, high)
    if(low==(high-1) and (i-low+1)>=0 and (i+low-1)<n) return palindrome(arr,i-
    low+1,i+low-1);
        mid=(low+high)/2;
        if(i-mid+1>=0 and i+mid-1<n and palindrome(arr,i-mid+1,i+mid-
        1))
            if (i+mid-1)==(n-1) return mid;
            else return binarysearch1(arr, i, mid, high);
        else return binarysearch1(arr, i, low, mid-1);


binarysearch2(arr, i, low, high)
    if(low==(high-1) and i-low>=0 and i+low-1<n) return palindrome(arr,i-
    low,i+low-1);
        mid=(low+high)/2;
        if(i-mid>=0 and i+mid-1<n and palindrome(arr,i-mid,i+mid-
        1))
            if (i+mid-1)==(n-1) return mid;
            else return binarysearch1(arr, i, mid, high);
        else return binarysearch2(arr, i, low, mid-1);
```

The time complexity of this code is of the order O(n*log(n)), which is much smaller than n*$(logn)^2$ for large inputs.