Theoretical Assignment 1 (ESO207)

September 2023

Pragati Agrawal: 220779 (Question 1)

Q1.

(a) First, we convert the given BST into an array $arr[N]$ using **level order traversal**. We will implement it using a queue, and assume the following operations are defined on queue: isEmpty(Q), Front(Q), CreateEmptyQueue(Q), Enqueue(x, Q), Dequeue(Q). The pseudo code for level order traversal is:

```
int arr[N];
CreateEmptyQueue(Q):
void level_order_traversal(Node* root){
    if (root == NULL)
        return;
    Enqueue(root, Q);
    int i=0;
    while (isEmpty(Q)!=TRUE) {
        Node* node = Front(Q);
        arr[i++]=Dequeue(Q);
        if (node->left != NULL)
            Enqueue(node->left->data,Q);
        if (node->right != NULL)
            Enqueue(node->right->data,Q);
    }
}
```

Now, we will perform changes in the values of the elements according to time given: m months.
Analysing the recurrence relations among the nodes, at the end of any year y, the wealth of the companies, if their wealth after $(y-1)$ years is known, will be :
if $N = 1$, then $RootNode(y) = 2^{12} * RootNode(y-1)$.
otherwise, for the root node: $RootNode(y) = 2^{11} * RootNode(y-1)$, if $N > 1$.

for any leaf node, its value will be:
$LeafNode(y) = 2^{12} * LeafNode(y-1) + 2^{10} * ParentOfLeafNode(y-1)$.
for any other node (not root and not leaf node):
$OtherNode(y) = 2^{11} * OtherNode(y-1) + 2^{10} * ParentOfOtherNode(y-1)$.
where $ParentNodes$ denote the wealth of their parent companies after $(y-1)$ years.
Now, after m months, number of years passed= $[m/12]$, where $[a]$ represents the floor of a. Let this be called y.
Now, the remaining number of months will be $(m - 12[m/12])$. Let this be called $remMths$. Now, the recurrence relations can be simplified as:

1. For a root node and $N \neq 1$, we will have 2^{11} times the previous value for the number of years passed, and for the remaining months, we need to multiply by $2^{remMths}$.
   $RootNode(m) = 2^{11*y} * 2^{remMths} * RootNode(0)$.
   $RootNode(m) = 2^{11*[m/12]} * 2^{m-12[m/12]} * RootNode(0)$
   $RootNode(m) = 2^{m-[m/12]} * RootNode(0)$.

2. For a leaf node, we will have 2^{12} times its previous value, and 2^{10} times the previous value of its parent node, for the number of years passed, and for the remaining months, we need to multiply this wealth by $2^{remMths}$.
   $LeafNode(m) =$
   $2^{remMths} * [2^{12*y} * LeafNode(0) + 2^{10*y} * ParentOfLeafNode0)]$.
   $LeafNode(m) = 2^{m-12[m/12]} * [2^{12*[m/12]} * LeafNode(0) + 2^{10*[m/12]} *$
   $ParentOfLeafNode(0)]$
   $LeafNode(m) =$
   $2^{m} * LeafNode(0) + 2^{m-2*[m/12]} * ParentOfLeafNode(0)$.

3. For a node other than a leaf or the root node, we will have 2^{11} times its previous value, and 2^{10} times the previous value of its parent node, for the number of years passed, and for the remaining months, we need to multiply this wealth by $2^{remMths}$.
   $OtherNode(m) =$
   $2^{remMths} * [2^{11*y} * OtherNode(0) + 2^{10*y} * ParentOfOtherNode0)]$.
   $OtherNode(m) = 2^{m-12[m/12]} * [2^{11*[m/12]} * OtherNode(0) + 2^{10*[m/12]} *$
   $ParentOfOtherNode(0)]$
   $OtherNode(m) =$
   $2^{m-[m/12]} * OtherNode(0) + 2^{m-2*[m/12]} * ParentOfOtherNode(0)$.

For any node whose index in the array formed above is i, then its parent node will be $floor((i-1)/2)$. The array we had created can be used as a column vector of size $n * 1$, containing the original values of wealth of all the compaines.
Now, for the months part, we create a matrix of size $n * n$, say $M[n][n]$ whose all indices we initialise to 0. Now, using the following algorithm we construct the matrix: $M[i][j]$, such that $i, j \in \{0, n-1\}$.

1. if $i = j < (n-1)/2$, then we initialise these $M[i][j] = 1/2$.

2. if $i = j \geq (n-1)/2$, then we initialise these $M[i][j] = 1$.
3. if $i > j$ and $j = (i-1)/2, i > 0$, we make it $M[i][j] = 1/4$, because for any node i,

for example, if the tree has 7 nodes, the matrix so formed looks like:
1/2 0 0 0 0 0 0
1/4 1/2 0 0 0 0 0
1/4 0 1/2 0 0 0 0
0 1/4 0 1 0 0 0
0 1/4 0 0 1 0 0
0 0 1/4 0 0 1 0
0 0 1/4 0 0 0 1
The pseudocode looks like:

```
    //create a matrix of size M[n][n], and initialize all values to 0.
    for (int i=0;i<n;i++){
        for (int j=0;j<n;j++){
            if (i==j and i<(n-1)/2){
                M[i][j]=1/2;
            }
            if (i==j and i >= (n-1)/2){
                M[i][j]=1;
            }
            if (i>j and i>0 and j=(i-1)/2){
                M[i][j]=1/4;
            }
        }
    }
```

Now, to find the values after m months, we will find the power of this matrix, raised to y ($y = [m/12]$). And then we can multiply this matrix by 2^m, to get the final matrix after m months. This matrix when multiplied to the initial wealths in the array, will give wealth after m months.
The pseudo code to find matrix raised to y is:

```
//function to multiply two matrices
void mat_mul(mat1,mat2){
    long long int sum;
    long long int mat3[n][n];
    for (long long int i=0;i<n;i++){
        for (long long int j=0;j<n;j++){
            sum=0;
            for (long long int l=0;l<n;l++){
                sum+=(mat1[i][l])*(mat2[l][j])3;
            }
            mat3[i][j]=sum;
        }
    }
    for (long long int i=0;i<n;i++){
        for (long long int j=0;j<n;j++){
            new_mat[i][j]=mat3[i][j];
        }
    }
    return;
}

//function to find the a matrix raised to power x
and store it in matrix new_mat
void pow_mat (mat[n][n],int x,new_mat[n][n]){
    if (x<=1){
        return;
    }
    pow_mat(mat,x/2,new_mat);
    mat_mul(new_mat,new_mat);
    if (x%2==0){
        return;
    }
    else{
        mat_mul(new_mat,mat);
        return;
    }
}


//function call in main(), used to find the
matrix M raised to power y,
and store in in different matrix X.
int X[n][n]={0};
pow_mat(M,y,X);
```

Now, we need to multiply all the entries of the matrix X so formed by 2^m, so that we can find wealth after m months.
The pseudo code is as:

```
num=pow(2,m);             //typically takes O(logN) time
    for (int i=0;i<n;i++){
        for (int j=0;j<n;j++){
            X[i][j]*=num;
        }
    }
```

Now, the matrix X finally so formed will be mutiplied by the column vector of arrays as:

```
//the arrar "arr" stores the initial wealths.
int final_wealth[n]={0};
int sum=0;
if (n==1){               //the tree contains only the parent node.
    return 2^m*arr[0];
}
    for (int i=0;i<n;i++){
        sum=0;
        for (int j=0;j<n;j++){
            sum+=X[i][j]*arr[j];
        }
        final_wealth[i]=sum;
    }
```

Now, the array $final_wealthcontainsthewealthsofthecompaniesaftermmonths.$
(b) **Time Complexity Analysis:** First we create an array from the given BST, using level order traversal. This visits **each node exactly once**, performs some constant number of operations, so takes $O(n)$ time.
After that we create the matrix M, which is an $n * n$ matrix, so takes $O(n^2)$ time.
Then we used *powmat* to multiply the matrix M, y times. Now, a single matrix multiplication, of order $n * n$, takes $O(n^3)$ time. This is done by the *matmul* function described above. Then we call the *matmul* function $log(y)$ times, i.e. $log([m/12]) = O(log(m))$ times. (This was taught in lectures, similar to the modified "power of a number" function). So, combined this entire step of creating matrix X, takes $O(n^3log(m))$ time.
Then we find 2^m, which takes $O(log(m))$ time, and multiplying each extry of X by 2^m takes $O(n^2)$ time.
Finally, we multiply the array with this matrix and store it in another array, this step takes $O(n^2)$ time, as we are visiting each value in the matrix just once.
So the final expression for the time complexity of the algorithm is:
$T(m) = O(n) + O(n^2) + O(n^3log(m)) + O(log(m)) + O(n^2) + O(n^2)$
Since the dominating term is $O(n^3log(m))$, the final time complexity of the algorithm will be: $T(m) = O(n^3log(m))$.
Proof of Correctness:
The proof of correctness of the algorithm can be verified from the mathematical derivation of the wealths of the companies after m months.
Now, the way the algorithm is constructed, we can say that each company is being multiplied by a suitable power of 2 and its parent node also by another suitable power of 2, and then added.
(c) To find the wealth of just a single node after m months, we can just multiply it with 2^m. Now, in order to do this in constant time, we can use **left bit shift operator**. The left bit shift operator left shifts the binary notation of the given number by the given number of places.
e.g. if $a = 5, b = 2$ then $a = 5 = 101$ in binary. Now, $a << b$ will perform

$10100 = 20$. Hence $a << b$ is equivalent to $a * 2^b$. Now, if $a = 1, b = m$ we can perform $a << b$, to give $1 * 2^m$ in constant time. Then we can multiply this quantity with the original wealth of the root node to find its wealth after m months.

Pragati Agrawal: 220779 (Question 2)

Q2.

(a) In order to find the minimum cost and capacities of rooms $\geq P$, we first sum the first i elements of the array such that the sum$\geq P$. Then, we traverse the array, removing values from the left, such that the sum is maintained $\geq P$. In each iteration, we maintain the minimum cost till now in another variable, $cost$, which compares the previous cost and current cost, and updates only if there is a decrement in the cost.
If any time, the sum value on subtraction falls below the given value P, we add more elements from the right to maintain the sum. In this process, the minimum cost is maintained.

```
    int minCost(int n, int P, int C) {
    int left = 0;
    int right = 0;
    int sum = 0;
    int minCost = INT_MAX;

    while (right < n) {
        sum += arr[right];

        while (sum >= P) {
            minCost = min(minCost, (right - left + 1) * C);
            sum -= capacities[left];
            left++;
        }

        right++;
    }
    return minCost;
}
```

Since this algorithm performs only one scan of the array to find the minimum cost, it works in $O(n)$ time.
(b) To find the maximum possible length of subarray having GCD $\geq k$, we can use an approach similar to the **Range-minima** problem approach, used in lectures. We will first create a matrix B of size $n * log(n)$, storing the GCDs of similar subarrays as in lectures, i.e., for each index, we store GCDs starting from the index i, and going to lengths successive powers of 2.
We can create this matrix in $O(nlog(n))$ time. This is because for the first column, we will find GCD of $arr[i]$ and $arr[i + 2^0] = arr[i + 1]$. Now, these are two consecutive numbers, and our blackbox can return their GCD in constant time. Now, we can create the remaining columns using its previous column. This will again require GCD of just two numbers to be compared, and hence would take constant time.

Like, to find GCD of elements from index 0 to 2, we can compare GCD of elements at $(0, 1)$ and $(1, 2)$, i.e. the first two entries of first column.
To create an i column, we will find GCD of elements at $arr[i]$ and $arr[i + 1]$, and store them at index $[i][0]$. Then for value $[i][1]$, i.e., GCD of elements $arr[i], arr[i + 1], arr[i + 2]$, we will find GCD of values at $arr[i], arr[i + 1]$ and $arr[i + 1], arr[i + 2]$. Similarly, for value at index $[i][2]$, we want to find GCD from $arr[i]$ to $arr[i + 4]$, and for that we will find GCD of $i \to i + 2$ and $i + 2 \to i + 4$, from the previous column.
So we can see that after creating the first column, the remaining columns take constant number of operations, and hence, for $log(n)$ columns, we will need $O(n * log(n))$ time to create the matrix.
After this, we can create an array of size n, to store the maximum possible subarray with GCD $\geq k$ starting at that index.
The pseudo code for the algorithm is as:

```
B= matrix (size=n*log(n)) stores the GCD of array
{A[i], A[i+1], A[i+2], A[i+4], A[i+8]...}

int rows=n;
int cols=log(n);
int arr[n]={-1};
for (int i=0;i<rows;i++){
    for (int j=0;j<cols;j++){
    if (B[i][j] <k and j==0){
        break;
    }
    else if (B[i][j] <k and j!=cols-1){
            arr[i]=j-1;
            break;
    }
    else if (B[i][j]>=k and j==cols-1){
        arr[i]=j;
        break;
    }
    }
}
```

Here, the above array contains the maximum possible subarray in difference of powers of 2, whose GCD $\geq k$. Now, we check if there is a larger subarray in between these powers of two, such that the GCD remains $\geq k$. For this , at each index, we will do something similar to binary search, we will find the mid between current index, and the next index (power of 2), where GCD dropped below k. If the GCD upto this mid is also k, we will increment the left to mid+1, and check again. Otherwise, we will decrement the right pointer to mid-1, and compare again.

We assume we also have the arrays **Power-of-2** and **Log**, as shown in the Range-minima problem in the lectures.
The pseudo code is as:

```
int temp;
int final_arr[n]={0};
for (int i=0;i<rows;i++){
    left=i+pow(2,arr[i]);
    right =i+pow(2,arr[i]+1);
    power_of_2=pow(2,arr[i]);
    while (left<right){
        mid=(left+right)/2;
        L=mid-i;
        t=Power-of-2[L];
        k=Log[L];
        temp=GCD(B[i][k], B[mid-t][k];
        if (temp<k){
            right=mid+1;
        }
        else{
            left=mid-1;
        }
    }
    final_arr[i]=mid-i+1;
}

scan the final_array to get the maximum number of rooms.
```

(c) **Proof of correctness:**
Assertion *P(i)*: After each iteration of the while loop, $sum \geq P$, and for each value of $right$, $minCost$ has the minimum length of contiguous subarray visited till now, summing to atleast P.
1. $sum >= P$
Initially, we add the elements of the array from $arr[0]$ till we reach a sum atleast P. Now, the $right$ pointer points to the node up to where we have added. After this we start subtracting the values from the lefmost part of the array. If $sum \geq P$, we compare previous cost and current cost and update their minimum in $minCost$. If the $sum < P$, then we add another element from the right of the array and again follow the same process. So, $minCost$ only gets updated if $sum \geq P$ and current cost is less than the previous cost incurred.
Time Complexity Analysis:
In the function **minCost**, we start from (right=0) and go upto (right=n-1). In each iteration, we perform a constant number, say k, of operations: arithmetic, assignment and comparisons. Hence the time complexity of the algorithm can be written as:
$T(n) = knT(n) = O(n)$.

Pragati Agrawal: 220779 (Question 3)

Q3.

(a) We have a BST, whose exactly two nodes have been swapped. Now, to identify these nodes, we can first create an array of the same size as the total number of nodes, and then use an algorithm similar to the **Traversal Algorithm**, where instead of printing, we can save the value of each node in the array. Now, since the BST was sorted (except for a single swap), the values in the array will also be almost sorted in ascending order. But because of the swap, we'll encounter a value in the array greater than its succeeding value. Going further, we'll encounter a value smaller than its preceding element. We need to identify these values in the array, as these are the swapped nodes. We can do this in a single scan of the array, i.e. $O(n)$ time. The pseudo-code for the modified **Traversal Algorithm** will be:

```
n=total number of nodes in the BST;
int arr[n];
int i=0;
 Traversal_modified(T)
{ (p= T);
 if(p==NULL) return;
 else{ if(left(p)!=NULL) Traversal_modified(left(p));
        arr[i++]=value(p);
        if(right(p)!=NULL) Traversal_modified(right(p));
    }
}
```

The array so formed will look as:
$\{a_1, a_2, a_3, ..., a_{m-1}, a_m, a_{m+1}, ..., a_{k-1}, a_k, a_{k+1}, ...a_n\}$, where:
$(a_m > a_{m+1})$, for some $m < n$, and $(a_k < a_{k-1})$, for some $k \leq n$. Now, we can scan the entire array once to find these values:

```
int val_1, val_2;
int idx_1=-1, idx_2=-1;
int flag=0;
for (int i=0;i<n-1;i++){
    if (arr[i]>arr[i+1]){
        idx_1=i;
        flag=1;
        continue;
    }
    if (flag==1 and arr[i+1]<arr[i]){
        idx_2=i+1;
        break;
    }
}
```

```
if (idx_2==-1){
    idx_2=idx_1+1;
}
val_1=arr[idx_1];
val_2=arr[idx_2];
```

Now, idx_1 and idx_2 store the indices of the swapped nodes and val_1 and val_2 contain the values of the swapped nodes. The last condition (idx_2 remains -1) is the case when the two nodes swapped are consecutive. In this case, we would assign idx_2 to be the next index of idx_1.
In order to find the common ancestors, we can compare the values of the swapped elements of the BST with the head and corresponding nodes of the BST. If both the values are greater than the current head, this implies that it was a common ancestor, and now we need to compare them with the middle element of the right subtree. Similarly, if both are smaller, then we need to further compare them with the middle of the left subtree. And we can progress successively to count the number of common ancestors.

```
\\val_1 and val_2 store the values of the swapped elements.
int count=0,k=0;
int ancestors[n];
common_ancestors(T)
{ p=T;
if (p!=NULL){
     if ((val_1==val(p)) or (val_2==val(p))){     //that node has been swapped
        return count;
     }
     else if (val_1<val(p) and val_2<val(p)){     //move to left subtree
        count++;
        ancestors[k++]=val(p);
        common_ancestors(p->left);
     }
     else if (val_1>val(p) and val_2>val(p)){      //move to right subtree
        count++;
        ancestors[k++]=val(p);
        common_ancestors(p->right);
     }
     else{                                        //no other common node will be found
        count++;
        ancestors[k++]=val(p);
        return count;
     }
  }
 }
```

Now, *ancestors* contains the list of the common ancestors, and its length is k.

(b) Now, since we have rearranged k nodes, we can first use the Traversal modified algorithm shown in (a), to store all the elements in an array of size n, say it be $arr[n]$. This would take $O(n)$ time. Now, consider two cases:

1. $G > nlogn$
   If $G > nlogn$, then we know that the values are very large, hence, we can sort a copy of the original array, say "copyarr[n]" using **Merge sort** algorithm, which takes $O(nlogn)$ time. Now, we can compare at each index respectively, if the original and sorted copy arrays have the same value, just continue. Else, we can increment the count variable, and print out that value from the original array to the standard output. When the array ends, we can print finally the value of count, i.e. the total number of nodes swapped. This would take $O(n)$ time.
   Hence, the overall time complexity would be
   $O(nlogn) + O(n) = O(nlogn)$.
   The pseudo-code for the algorithm looks like:

```
Merge_Sort(copyarr,0,n-1);
int no_of_swaps(arr,copyarr){
    int count=0;
    for (int i=0;i<n;i++){
        if (arr[i]!=copyarr[i]){
            count++;
            print(arr[i]);
        }
    }
    return count;
}
```

2. $G < nlogn$
   If $G < nlogn$, this means the values are upper bounded by G, and we need not sort the array. Instead, we can improve from $O(nlogn)$ to $O(G + n)$, by using a dummy array of size G. We will create an array $arr1[G]$, and initialize all the cells with a value, say (-1). Now, we will scan the original array $arr[n]$ once, and for each value, we will store the value at the same index in $arr1[G]$. Now, since the BST contains only natural numbers, we will create another array of size n, say $arr2[n]$, and ignoring the indices having -1 as their values, we will store the remaining elements of $arr1[G]$ in the array $arr2[n]$, in the same order. Hence, we have created an array $arr2[n]$ which has all the elements in the BST in the sorted order. Now, we'll compare this array $arr2[n]$ with the original array $arr[n]$, to find out the misplaced nodes. Again, we can keep a count variable to count the total number of swaps, and meanwhile, we can print out the values that got rearranged.
   The pseudo-code for the algorithm looks like:

```
arr1[G]={-1};
arr2[n];
for (int i=0;i<n;i++){
    arr1[arr[i]-1]=arr[i];          //takes O(n) time
}
int k=0;
for (int i=0;i<G;i++){
    if (arr1[i]!=-1){
        arr2[k++]=arr1[i];          //takes O(G) time
    }
}
int no_of_swaps(arr,arr2){
    int count=0;
    for (int i=0;i<n;i++){
        if (arr[i]!=arr2[i]){       //takes O(n) time
            count++;
            print(arr[i]);
        }
    }
    return count;
}
```

The overall algorithm takes about $O(G) + 2O(n)$ time, which is equivalent to $O(G + n)$.
Hence, the algorithm has complexity $O(min(G + n, nlog(n)))$, depending upon G and n.

Pragati Agrawal: 220779 (Question 4)

Q4.

After the arrangement of cards from top to bottom, it looks like $(a_{k+1}, a_{k+2}, ...a_n, a_1, a_2..., a_k)$, where $(k+1, k+2, ..., n, 1, 2, ..., k)$ are the original indices in the sorted deck.
Now, we know that the entire array is sorted circularly, and linearly we need to find where the value becomes smaller than its left neighbour. To accomplish this, we can maintain three variables: left (l), right(r) and mid(m). We can notice, that if the middle element of the array is greater than the last element of that portion of array, then a_1 must lie between them. And if the middle element of the concerned part is less than the last element of that array portion, it means that this portion of the array is sorted, and so a_1 would not be found here. Hence move to the other side. Now, using the *l,m* and *r* variables, the pseudo-code for the algorithm can be written as:

```
l=0;
r=n-1;
int mid;
int find(arr[n],l,r){
    m=(l+r)/2;
    if (arr[m]<arr[m+1] && arr[m]<arr[m-1]){
        return (n-m);
    }
    else if (arr[m]>arr[m+1] && arr[m]>arr[m-1]){
        return(n-(m+1));
    }
    else if (arr[m]<arr[r]){
        find(arr,l,m-1);
    }
    else if (arr[m]>arr[r]){
        find(arr,m+1,r);
    }
}
```

(b) Time Complexity Analysis:
In each iteration of the while loop, we check if the smallest element, i.e. a_1, would be found on the left half array or on the right half array, and divide the array into half successively. Now we apply further checks on only this half of the array. We do this until we find the required index.
We are performing a constant number of operations (comparisons and assignments) for each size of the array. In each recursive call, the size of the array is reduced by half, i.e.
$T(n) = T(n/2) + k(O(1))$. This leads to $O(logn)$ many recursive calls, hence the time complexity of the algorithm is $O(logn)$.

Pragati Agrawal: 220779 (Question 5)

Q5.

For a substring, if it's a palindrome, then all its sub-portions will also be palindromes. We use this logic to find the maximum possible length of subarray, which is a palindrome, centered at a given index. We use **binary search** type technique to find this maximum possible length of palindromic string. We know that we have an Oracle function that returns if a substring is a palindrome or not in constant time.
So the pseudo-code is:

```
int count=0;
void odd_substrings(str, index){     //counts the number of odd palindromes
                                              centered at index i
    int left, right,result, mid;
    int n=str.length();
    left=1;
    right=min(i, n-i-1);
    if (Oracle(i-left,i+left)==FALSE){    //no palindrome possible
            return;
        }
        else if (Oracle(i-right, i+right)==TRUE){   //maximum length palindromic
            result=right;
            count+=result;
            return;
        }
        else{
            result=1;
            while (left<right){
                mid=(left+right)/2;
                result=mid;
                if (Oracle(i-mid,i+mid)==TRUE){    //a bigger palindrome possible
                    left=mid+1;
                }
                else if (Oracle(i-mid,i+mid)==FALSE){   //a smaller palindrome possible
                    right=mid-1;
                }
            }
            count+=result;
            return;
        }
    }
```

The above function finds the number of palindromic substrings of odd length, centered at any index i of the array.

Similar logic can be used for palindromic substrings of even length, just that they are not centered at any index, but have two indices as their centres. Let the lower indexed centre be used for reference. So, at index 0, we can at maximum have just one even palindrome (0,1). the pseudo code for even length substrings will be:

```
void even_substrings(str, index){     //counts the number of even palindromes
                                          centered at index i
    int left, right,result, mid;
    int n=str.length();
    left=1;
    right=min(i, n-i-1);
    if (Oracle(i,i+left)==FALSE){    //no palindrome possible
        return;
    }
    if (right==i){
        if (Oracle(0, i+i+1)==TRUE){
            result=i+1;
            count+=result;
            return;
        }
    }
    else if (right==n-i-1){
        if (Oracle(i-right+1, i+right)==TRUE){   //maximum length palindromic
            result=right;
            count+=result;
            return;
        }
        else{
            result=1;
            while (left<right){
                mid=(left+right)/2;
                result=mid;
                if (Oracle(i-mid+1,i+mid)==TRUE){    //a bigger palindrome possible
                    left=mid+1;
                }
                else if (Oracle(i-mid+1,i+mid)==FALSE){   //a smaller palindrome possible
                    right=mid-1;
                }
            }
            count+=result;
            return;
        }
    }
```

Now, we run these algorithms for each index, to count the total number of even and odd palindromes, centered at every index, and increment this in a global variable *count*.

```
for (int i=1;i<length-1;i++){
    odd_substrings(string, i);
}
for (int i=0;i<length-1;i++){
    even_substrings(string, i);
}
cout<<count;
```

Time Complexity Analysis: We are doing two traversals of the array in the main function. In each of the odd and even substrings function, we either return getting no substring or the maximum substring, or we use binary search type technique to find the highest length of substring possible. This would take atmost $O(logn)$ queries for each index. Hence, the algorithm works in time $O(nlogn)$.