ESO 207 – THEORITICAL ASSIGNMENT 1

CHANDRADIP KARMAKAR
240296

Name: CHANDRADIP KARMAKAR Roll No: 240296

Question 1: Maximal Product sum

(A) Here to find the maximum dot product of two given arrays, we have to multiply the smallest element of first array with the smallest element of the other and multiply the next smallest element of first array with the next smallest element of the other and so on ... and add all the products according to Rearrangement Inequality.

Let two arrays are a, b and length of each array is n

So we first sort(a) and sort(b) Next as the length of both the arrays are same, so we iteratrate in the array from the left to right and multiply the corresponding elements of the two arrays and add all the products. Thus this will give the maximum dot product of the two given arrays.

Pseudocode:

```
Maximal_product_sum(int a[], int b[], int n){
    Sort (a);
    Sort(b);
    Sum<-0;
    For(int i=0;i<n; i++){
        Sum<-sum+ a[i]*b[i];
    }
    Return sum;
}
```

(B) For each query q, we have to find the element from the original array, we have to increment the value by 1 and sort the one and again multiply with the other one to get the maximum value. But in each sort, time complexity required is O(nlog(n)) and to multiply it again takes O(n), so overall in q queries, the overall time complexity is O(q*(nlogn+n)+ nlogn). But it is not efficient for large values for q and n. so, we have to go through a different approach.

First without loss of generality, take o =1 and the element taken is a_x

Now, increment a_x by 1 in the original array and before doing it, store a_x in a separate variable, i.e temp = a_x ; a_x = a_x +1

Now in the sorted array a, find the last index of temp using binary search and it will take O(logn) time complexity. Now after this index, the elements are bigger than temp, so, the position of the incremented value a_x is also the same. so, in the sorted array a, increase the value at that index by 1, and still the array is sorted. Now, as there is a change in a particular index, we donot need to multiply the two arrays again, we have to just add the element that is at the same index in the other sorted array b in our previous result and get the new result. Thus in each query the time complexity is O(logn) and the time complexity required to sort the arrays at the beginning is O(nlogn). So . the overall time complexity in q queries in O((n+q)logn), which is much better than the previous one.

Pseudocode:

```
Int arr1[] <- a;
Int arr2[] <- b;
Sort(arr1);
Sort(arr2);
Int sum <- Maximal_product_sum(a, b, n); // from the previous algorithm
Max_sum_query(o, x, arr1, arr2, a, b, n){
    If(o==1){
        Temp <- a[x];
        a[x] <- a[x] +1;
        idx <- binary search(a, 0, n-1, temp); // finds the index using
        arr1[idx] <- arr1[idx] +1;
        sum=sum+arr2[idx];
    }
    Else{
        Temp <- b[x];
        b[x] <- b[x] +1;
        idx <- binary search(b, 0, n-1, temp); // finds the index using
        arr2[idx] <- arr2[idx] +1;
        sum=sum+arr1[idx];
    }
}
Return sum;
```

(C) **Proof of Correctness:** We know that the maximum product sum is only possible when both arrays are sorted and we multiply the corresponding elements and add the products by the rearrangement inequality theorem. i.e if x1<x2<x3<...<xn and y1<y2<y3...<yn, then the maximum product sum is x1*y1+x2*y2+x3*y3+....+xn*yn.

Proof using induction: Base case: Before any updates, the algorithm sorts both arrays and computes the maximum dot product correctly.

Inductive hypothesis: Assume after k updates, the algorithm gives the correct maximum dot product.

Inductive step: at the (k+1)th update,

(a) Exactly one element changes.

(b) The algorithm repositions it in the sorted order.

(c) The new dot product product is computed on the updated, sorted sequence, but as there is only one element change at a particular index, so we can do operation on that particular index in O(1) time complextity.

Thus, by the rearrangement inequality, this remains the maximum possible dot product and hences correctness holds in all updates.

Time complexity: As discussed earlier, in each query O(logn) is taken for searching and O(1) time for computing as there is only one change at a particular index and before the query starts, we have to sort the both arrays that takes O(nlogn) time.

Hence, the overall time complexity is O((n+q)logn).

Name: CHANDRADIP KARMAKAR Roll No: 240296

Question 2: Game Score Maximization

(A) First we have to find the number of distinct prime factors of each elements of the array. This can be done efficiently using Sieve of Eratosthenes But for that, we have to precompute smallest prime factors of all numbers upto the maximum limit of the elements. And this can be done in O(NloglogN) time complexity, where N is the maximum element of the array

Assume the given inputs are in form of array named as v

Pseudocode for precomputation:

```
max_element <- maximum element of the array
Int arr[max_element+1];
sieve() {
    for (int i = 2; i <= max_element; i++){
        arr[i] <- i;
    }
    for (int i = 2; i * i <= max_element; i++) {
        if (arr[i] == i) {
            for (int j = i * i; j <= max_element; j += i)
                if (arr[j] == j) {
                    arr[j] <- i;
                }
        }
    }
}
```

After this precomputation, we can get the count the no of distinct prime factors of each element of the array in time complexity O(logN) using the precomputed array where N is the element of the array

Pseudocode for distinct prime factors counting:

```
Count_primes(x){
    int count <- 0,
    int n <- (-1);
    while (x>1) {
        prime = arr[x];
        if (prime != n) {
                count++;
                x <- prime;
            }
            x /= prime;
        }
        return count;
    }
```

Now we store the counts of primes of each elements in a array, let b Also, as we have to maximize the result(which has value initially 1), so we try to multiply the maximum values to it as many times as possible. So, we try to sort the array. But it disturbs the initial position of the elements so, we create a new data structure that stores the value of the element and the position of it in the original array. Then we sort the new data structure array by its value in increasing order. And this will take O(nlogn) time complexity.

Pseudocode for the data structure and sorting:

```
    Struct Node{
        Int data;
        Int idx;
    }
    Node pos[n];
    Sort_element(Node pos[], int v[], int n){
        For(int i=0;i<n;i++){
            Pos[i].data <- v[i];
            Pos[i].idx <- i;
        }
    }
```

Now the new array pos in increasing order by the values (here we can use mergesort to sort the Node elements by the values)

Now we try to find the range for each element in which it has greater prime factors than other elements in that range. For that we will use stack to find the left and the right boundary for each element and store it in two different arrays. And it will take O(n) time complexity.

Pseudocode for finding the boundaries:

```
    Int left[n];
    Int right[n];
```

```
Find_boundary(b, left, right, n){
    Stack<int>s;
    // first we will find the left boundary
    For(int i=0;i<n;i++){
        While(!s.empty()){
            If(b[s.top()]<b[i]){ // we donot use the equality sign here
            // because it is given in the question that if two elements have equal count of pri
            // then the one who has lower index will be counted
                s.pop();
            }
            Else break;
        }
        If(s.empty()){
            Left[i]=-1;
        }
        Else left[i] = s.top();
        s.push(i);
    }
    s.clear(); // erase all the elements of the existing stack
    For(int i=n-1;i>=0;i--){
        While(!s.empty()){
            If(b[s.top()]<=b[i]){
                s.pop();
            }
            Else break;
        }
        If(s.empty()){
            right[i]=n;
        }
        Else right[i] = s.top();
        s.push(i);
    }
}
```

Now we will start from the largest value of the sorted array of new data structure and goes towards the lower one and find the range of that element so that it has highest prime factors in that range and multiply it with the result accordingly.

Pseudocode to maximize the result:

```
Max_result(pos, left, right, n, k){
    Result <- 1;
    For(int i=n-1;i>=0;i--){
        z <- (right[pos[i].idx]-pos[i].idx)*(pos[i].idx-left[pos[i].idx]);
        If(k>=z){
            result = result* pow(pos[i].data, z); // can be done in O(logz) time
            // complexity using power exponentiation by divide and conquer method
            k=k-z;
        }
        Else{
            Result = result * pow(pos[i].data, k);
        }
    }
    Return result;
}
```

(B) **Proof of correctness:** First of all, as we have to maximize our result so, we try to multiply the larger elements much more than the lower one and so we sort the array and start with the largest element. Also we try to find all possible ranges [L, R] so that we can multiply the larger elements of the array as many times as possible.

Proof by Induction: Base case: k=1, then only one subarray can be chosen. so, to maximize our result, we multiply with the maximum element of the array which is trivial

Inductive Hypothesis: suppose after m operations, the algorithm yields the maximum possible score.

Inductive step: For the next operation i.e. (k+1)th operation, the algorithm tries to find the next greater element and multiply it and thus the result is monotonically increasing and each time it produces the maximum result of that operation.

Thus, by induction, the algorithm produces the correct final score after all k operations.

Time complexity: First of all, we have find the precomputed array for finding the count of the prime factors in time complexity O(MloglogM), where M is the maximum element of the array and then we have to find the count of distinct prime factors of every element in O(logN) time complexity. Next we have to make a new data structure and fill it and sort it and it takes the time complexity of O(nlogn). And then we have to find the left boundary and right boundary i.e the range of each element using stack in O(n) time complexity.

Thus thus the overall time complexity of the algorithm is O(MlogM + nlogn)

Name: CHANDRADIP KARMAKAR Roll No: 240296

Question 3: Wealth accumulate

(A) Here we have given a binary tree with n nodes with each node having initial wealth In each year wealth of each node is multiplied by 4. Then it donates half of its wealth to one of his child.

In odd year, it gives half of his wealth to the left child and in even year, it gives half of his wealth to the right child. and after that it again gives to the left child and so on...

After growth in node u at year t be W[u][t] = 4*W[u][t-1] After donation, donation = W[u][t] /2 goes to child(depending on parity of t)

Remaining: W[u][t] = W[u][t] /2; Then add donations from parents.

W(t) = M(t %2) * W(t-1) where M[0] = transformation matrix for odd years (donate left) M(1) = transformation matrix for even years (donate right)

After k years: W(k) = M((k-1) mod 2) * ... * M(1) * M(0) * W(0) This is a matrix exponentiation problem with period 2.

Since the pattern alternates every 2 years, we can combine: M2= M(1) * M(0)

W(k) = (M2)^(k/2) * W(0) if k is even W(k) = M(0) * (M2)^(k/2) * W(0) if k is odd

where matrix size = n*n and matrix multiplication of matrix gives O(n^3 logk)

But as the binary tree has 2 children or 0 child of every node. So there are n non zero elements in the matrix, so the matrix multiplication will take O(n) time complexity.

Pseudocode:

```
Determine (Node* root, int k){
    W(t) = M(t mod 2) * W(t-1);
    W(k) = M((k-1) mod 2) * ... * M(1) * M(0) * W(0);
    M2 = M(1) * M(0);
    If(k%2==0){
            W(k) = (M2)^(k/2) * W(0); //k is even
        }
        Else{
            W(k) = M(0) * (M2)^(k/2) * W(0) // k is odd
            // (find the matrix multiplication using power exponential method and
            // using the divide and conquer method to get O(logn) time multiplication)
        }
        Return W[u][t]; // return the wealth of u node after k years.
    }
```

(B) **Time complexity:** Here we are using a matrix of size n*n (M(0)). and we are multiplying with the other n*n matrix (M(0)). So appearently it seems like the time complexity of the matrix multiplication is O(n^3). And also we can make the matrix multiplication exponentiation in O(log(k)) time, so overall time complexity would be O(n^3logk). But, the tree is a special type of binary tree where each node has either 0 or 2 children. So the matrix will be of a special king where only the diagonal elements are non-zero and rest are zero. So, we can multiply 2 diagonal matrix of size n*n in O(n) time complexity as we have to multiply the corresponding diagonal elements of the two matrix. And again the the matrix exponentiation will take O(logk)times.

Overall the time complexity will be O(nlogk).

Name: CHANDRADIP KARMAKAR Roll No: 240296

Question 4: The King's Punishment

(A) **Brute Force algorithm:** We have to find the total number of punishments of each knights i.e the the no of taller knights in front of each knights.

For that we can use two nested for loops where the outer loop denotes the current pointing knight and the inner loop will tell the no of taller knights in front of that knight in the single line. Here arr denotes the given inputs, result is the output array with no of punishments of each knights and n is the length of the input array.

Pseudocode:

```
Count_punishments(int arr[], int result[], int n){
    For(int i=0;i<n;i++){
        Count <- 0;
        For(int j=0;j<i;j++){
            If(arr[j]>arr[i]){
                Count++;
            }
        }
        Result[i] <- count;
    }
    Return result;
}
```

Here the outer for loop runs n times and for each i of the outer for loop, the inner for loop runs i times. And thus overall, the total no of executions is (n*(n-1))/2; and thus the brute force algorithm has time complexity of O(n^2)

(B) Here we have to insert the knight height from left to right into a data structure so that it gives the count of knights with the taller or smaller height than him. It is possible if we try to insert the height as data in a Binary Search Tree.

For that, we need the following operations required with the Binary Search Tree:

1. Insert: efficiently insert the height of the knight so that we can the no of taller knights in the tree before.
2. Get_Count: return the no of nodes of the subtree rooted at the current node
3. Update_count: it updates the count of the node while inserting a new node of height of a knight in the BST
4. Count_taller: return the no of taller knights than the current knight

Pseudocode of the above functions:

```
Struct node{
    Int height;
    Node* left;
    Node* right;
    Int count=1;
}
Get_count(Node* node){
    If(node==NULL){
        Return 0;
    }
    Else{
        Return node ->count;
    }
}
Update_count(Node* node){
    If (node !=NULL){
        Node ->count <- 1+ get_count(node->left) + get_count(node->right);
    }
}
```

```
Insert(Node* root, int height){
    Node* newnode <- new Node();
    Newnode->height <- height;
    If(root==NULL){
        Return newnode;
    }
    If(height < root->height){
        Root->left <- insert(root->left, height);
    }
    Else if(height > root->height){
        Root->right <- insert(root->right, height);
    }
    Update_count(root);
    Return root;
}
Count_taller(Node* root, int target){
    If(root==NULL){
        Return 0;
    }
    If(root->height > target){
        Return 1+ get_count(root->right) + count_taller(root->left, target);
    }
    Else{
        Return count_taller(root->right, target);
    }
}
```

(C) But if we insert the heights in this way, then it may possible that the height of the binary tree becomes of O(n), i.e. the BST becomes skewed. Then for each insert or update operation will take the time complexity of O(n) and the overall time complexity will become O(n^2) that it was earlier and then there would be no improvement in our current algorithm.

To handle this problem, we have to make a balanced binary search tree, where at each subtree the difference of number of nodes of the left subtree and right subtree is atmost 1. For that, after each insertion we have to check whether the tree is balanced or not. If the tree becomes unbalanced, then we have to balance the tree at the very instant.

There will be two possibilities that either the tree is left overloaded i.e the left subtree has no of nodes $> 1+$ no of nodes of the right subtree or it may be right overloaded.

Pseudocode to handle unbalancing:

```
Struct node{
    Int key;
    Node* left;
    Node* right;
    Int height=1;
}
Find_height (Node* root){
    If(root!= NULL){
        Return root->height;
    }
    Return 0;
}
```

```
Balance (Node* root){
    If(root!=NULL){
        Return find_height(root->left)-find_height(root->right);
    }
    Return 0;
}
Rotate(Node* root){
    If(balance(root)>1){
        Node* x = y->left;
        Node* T = x->right;
        x->right = y;
        y->left = T;
        y->height = max(find_height(y->left), find_height(y->right)) + 1;
        x->height = max(find_height (x->left), find_height(x->right)) + 1;
        return x;
    }
    Else if(balance(root)< -1){
        Node* y = x->right;
        Node* T = y->left;
        y->left = x;
        x->right = T;
        x->height = max(find_height (x->left), find_height(x->right)) + 1;
        y->height = max(find_height (y->left), find_height(y->right)) + 1;
        return y;
    }
}
```

(D) As in this algorithm we use the BST or specifically balanced Binary Search Tree. And we know the height of the balanced BST is O(logn), where n is the no of nodes in the BST.

In case of insertion of node in BST, we know that the node insertion in BST occurs in the leaf part. so for that we have to traverse from root to leaf. so the time complexity of insertion in this algorithm is O(H) i.e O(logn), where H is the height of the binary search tree, i.e H = O(logn) where n = current no of nodes in the tree

In case of query operation, we have to check in every time whether the node value is greater or less than the target value and accordingly discard the left or right subtree. Thus it may be possible that we have to traverse in the whole BST, i. e the height of the BST. So, the time complexity of the query operation is O(logn)

Overall, there are n nodes in the nodes and from the staring of the insertion the time complexity of both query and insertion is O(logn). So the overall time complexity of the algorithm is O(nlogn).