ESO 207 – THEORETICAL 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 the first array with the smallest element of the other and multiply the next smallest element of the 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 be a, b and the length of each array is n.

So we first sort(a) and sort(b). Next as the length of both arrays is the same, so we iterate in the array from 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:

Algorithm 1 Maximal Product Sum

```
1: procedure MAXIMAL_PRODUCT_SUM(int a[], int b[], int n)
2:     Sort(a)
3:     Sort(b)
4:     Sum ← 0
5:     for int i=0; i<n; i++ do
6:         Sum ← Sum + a[i]*b[i]
7:     end forreturn Sum
8: end procedure
```

(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 array again and multiply with the other one to get the maximum value. But in each sort, the time complexity required is $O(n \log n)$ and to multiply it again takes $O(n)$, so overall in q queries, the overall time complexity is $O(q * (n \log n + n) + n \log n)$. 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 $\leftarrow a_x$; $a_x \leftarrow a_x + 1$.

Now in the sorted array a, find the last index of temp using binary search and it will take $O(\log n)$ 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 do not 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 to our previous result and get the new result. Thus, in each query the time complexity is $O(\log n)$ and

the time complexity required to sort the arrays at the beginning is $O(n \log n)$. So, the overall time complexity in q queries is $O((n+q) \log n)$, which is much better than the previous one.

Pseudocode:

Algorithm 2 Maximal Sum Query

```
Int arr1[] ← a
Int arr2[] ← b
Sort(arr1)
Sort(arr2)
Int sum ← Maximal_product_sum(a, b, n)          ▷ from the previous algorithm
procedure MAX_SUM_QUERY(o, x, arr1, arr2, a, b, n)
    if o==1 then
        Temp ← a[x]
        a[x] ← a[x] + 1
        idx ← binary search(a, 0, n-1, temp)    ▷ finds the index using binary search
        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 binary search
        arr2[idx] ← arr2[idx] + 1
        sum ← sum + arr1[idx]
    end ifreturn sum
end procedure
```

(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 $x_1 < x_2 < x_3 < \ldots < x_n$ and $y_1 < y_2 < y_3 < \ldots < y_n$, then the maximum product sum is $x_1 * y_1 + x_2 * y_2 + x_3 * y_3 + \ldots + x_n * y_n$.

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,

1. Exactly one element changes.
2. The algorithm repositions it in the sorted order.
3. The new dot 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 complexity.

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

Time complexity:

As discussed earlier, in each query $O(\log n)$ 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 both arrays which takes $O(n \log n)$ time. Hence, the overall time complexity is $O((n+q)\log n)$.

Question 2: Game Score Maximization

(A)

First we have to find the number of distinct prime factors of each element 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 up to the maximum limit of the elements. And this can be done in $O(N \log \log N)$ time complexity, where N is the maximum element of the array.

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

Pseudocode for precomputation:

Algorithm 3 Sieve for Smallest Prime Factors

```
max_element ← maximum element of the array
Int arr[max_element+1]
procedure SIEVE
    for int i = 2; i ≤ max_element; i++ do
        arr[i] ← i
    end for
    for int i = 2; i * i ≤ max_element; i++ do
        if arr[i] == i then
            for int j = i * i; j ≤ max_element; j += i do
                if arr[j] == j then
                    arr[j] ← i
                end if
            end for
        end if
    end for
end procedure
```

After this precomputation, we can get the count of the number of distinct prime factors of each element of the array in time complexity $O(\log N)$ using the precomputed array where N is the element of the array.

Pseudocode for distinct prime factors counting:

Algorithm 4 Count Primes

```
1: procedure COUNT_PRIMES(x)
2:     int count ← 0
3:     int n ← −1
4:     while x > 1 do
5:         prime = arr[x]
6:         if prime != n then
7:             count++
8:             x ← prime
9:         end if
10:        x /= prime
11:    end whilereturn count
12: end procedure
```

Now we store the counts of primes of each element in an array, let it be 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 its position in the original array. Then we sort the new data structure array by its value in increasing order. And this will take $O(n \log n)$ time complexity.

Pseudocode for the data structure and sorting:

Algorithm 5 Sort Elements

```
1: Struct Node
2:    Int data
3:    Int idx
4: Node pos[n]
5: procedure SORT_ELEMENT(Node pos[], int v[], int n)
6:     for int i=0; i<n; i++ do
7:         Pos[i].data ← v[i]
8:         Pos[i].idx ← i
9:     end for
10: end procedure ▷ Now the new array pos is 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 a 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:

Algorithm 6 Find Boundary

```
1: Int left[n]
2: Int right[n]
3: procedure FIND_BOUNDARY(b, left, right, n)
4:     Stack<int>s                          ▷ first we will find the left boundary
5:     for int i=0; i<n; i++ do
6:         while !s.empty() do
7:             if b[s.top()] < b[i] then    ▷ we do not use the equality sign here because it is given in the question that if two elements have equal count of prime factors then the one who has lower index will be counted
8:                 s.pop()
9:             else
10:                break
11:            end if
12:        end while
13:        if s.empty() then
14:            Left[i] ← −1
15:        else
16:            left[i] = s.top()
17:        end if
18:        s.push(i)
19:    end for
20:    s.clear()                            ▷ erase all the elements of the existing stack
21:    for int i=n-1; i>=0; i– do
22:        while !s.empty() do
23:            if b[s.top()] <= b[i] then
24:                s.pop()
25:            else
26:                break
27:            end if
28:        end while
29:        if s.empty() then
30:            right[i] ← n
31:        else
32:            right[i] = s.top()
33:        end if
34:        s.push(i)
35:    end for
36: end procedure
```

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

Pseudocode to maximize the result:

Algorithm 7 Max Result

```
1: procedure MAX_RESULT(pos, left, right, n, k)
2:     Result ← 1
3:     for int i=n-1; i>=0; i−− do
4:         z ← (right[pos[i].idx] - pos[i].idx) * (pos[i].idx - left[pos[i].idx])
5:         if k >= z then
6:             result = result * pow(pos[i].data, z)        ▷ can be done in O(log z) time
   complexity using power exponentiation by divide and conquer method
7:             k = k - z
8:         else
9:             Result = result * pow(pos[i].data, k)
10:        end if
11:    end forreturn Result
12: end procedure
```

(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: If $k = 1$, 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 to find the precomputed array for finding the count of the prime factors in time complexity $O(M \log \log M)$, 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 $O(\log N)$ time complexity. Next we have to make a new data structure and fill it and sort it and it takes the time complexity of $O(n \log n)$. 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 the overall time complexity of the algorithm is $O(M \log \log M + n \log n)$.

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 its child.

In an odd year, it gives half of its wealth to the left child and in an even year, it gives half of its 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.

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

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

Since the pattern alternates every 2 years, we can combine: $M_2 = M(1) * M(0)$.

- If k is even: $W(k) = (M_2)^{k/2} * W(0)$
- If k is odd: $W(k) = M(0) * (M_2)^{(k/2)} * W(0)$

where matrix size $= n * n$ and matrix multiplication of matrix gives $O(n^3 \log k)$. But as the binary tree has 2 children or 0 children for every node, there are n non-zero elements in the matrix, so the matrix multiplication will take $O(n)$ time complexity.

Pseudocode:

Algorithm 8 Determine Wealth

```
1: procedure DETERMINE(Node* root, int k)
2:     W(t) = M(t (mod 2)) * W(t − 1)
3:     W(k) = M((k − 1) (mod 2)) * ... * M(1) * M(0) * W(0)
4:     M_2 = M(1) * M(0)
5:     if k (mod 2) == 0 then
6:         W(k) = (M_2)^{k/2} * W(0)                          ▷ k is even
7:     else
8:         W(k) = M(0) * (M_2)^{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(log n) time multiplication
9:     end ifreturn W[u][t]        ▷ return the wealth of u node after k years
10: end procedure
```

(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 apparently 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 $O(n^3 \log k)$. 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 kind where only the diagonal elements are non-zero and the rest are zero. So, we can multiply 2 diagonal matrices of size $n * n$ in $O(n)$ time complexity as we have to multiply the corresponding diagonal elements of the two matrices. And again the matrix exponentiation will take $O(\log k)$ times. Overall the time complexity will be $O(n \log k)$.

Question 4: The King's Punishment

(A) Brute Force algorithm:

We have to find the total number of punishments of each knight, i.e., the number of taller knights in front of each knight.

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 number of taller knights in front of that knight in the single line. Here 'arr' denotes the given inputs, 'result' is the output array with the number of punishments of each knight, and 'n' is the length of the input array.

Pseudocode:

Algorithm 9 Count Punishments (Brute Force)

```
1: procedure COUNT_PUNISHMENTS(int arr[], int result[], int n)
2:     for int i=0; i<n; i++ do
3:         Count ← 0
4:         for int j=0; j<i; j++ do
5:             if arr[j] > arr[i] then
6:                 Count++
7:             end if
8:         end for
9:         Result[i] ← Count
10:    end forreturn result
11: end procedure
```

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 number 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 a 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 get the number of taller knights in the tree before.
2. **Get_Count:** return the number 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 number of taller knights than the current knight.

Pseudocode of the above functions:

Algorithm 10 BST Functions

```
1: Struct node
2:    Int height
3:    Node* left
4:    Node* right
5:    Int count = 1
6: procedure GET_COUNT(Node* node)
7:    if node == NULL then return 0
8:    elsereturn node → count
9:    end if
10: end procedure
11: procedure UPDATE_COUNT(Node* node)
12:    if node != NULL then
13:       Node → count ← 1 + Get_count(node→left) + Get_count(node→right)
14:    end if
15: end procedure
16: procedure INSERT(Node* root, int height)
17:    Node* newnode ← new Node()
18:    Newnode→height ← height
19:    if root == NULL then return newnode
20:    end if
21:    if height < root→height then
22:       Root→left ← Insert(root→left, height) height > root→height
23:       Root→right ← Insert(root→right, height)
24:    end if
25:    Update_count(root) return root
26: end procedure
27: procedure COUNT_TALLER(Node* root, int target)
28:    if root == NULL then return 0
29:    end if
30:    if root→height > target then return 1 + Get_count(root→right) +
   Count_taller(root→left, target)
31:    elsereturn Count_taller(root→right, target)
32:    end if
33: end procedure
```

(C)

But if we insert the heights in this way, then it may be possible that the height of the binary tree becomes $O(n)$, i.e., the BST becomes skewed. Then for each insert or update operation it 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 the number of nodes of the left subtree and right subtree is at

most 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 number of nodes > 1+ number of nodes of the right subtree, or it may be right overloaded.

Pseudocode to handle unbalancing:

Algorithm 11 Balance BST

```
1: Struct node
2:    Int key
3:    Node* left
4:    Node* right
5:    Int height = 1
6: procedure FIND_HEIGHT(Node* root)
7:    if root != NULL then return root→height
8:    end ifreturn 0
9: end procedure
10: procedure BALANCE(Node* root)
11:    if root != NULL then return Find_height(root→left) - Find_height(root→right)
12:    end ifreturn 0
13: end procedure
14: procedure ROTATE(Node* root)
15:    if Balance(root) > 1 then
16:        Node* x = y→left
17:        Node* T = x→right
18:        x→right = y
19:        y→left = T
20:        y→height = max(Find_height(y→left), Find_height(y→right)) + 1
21:        x→height = max(Find_height(x→left), Find_height(x→right)) + 1 return x
   ElseIf Balance(root) < -1
22:        Node* y = x→right
23:        Node* T = y→left
24:        y→left = x
25:        x→right = T
26:        x→height = max(Find_height(x→left), Find_height(x→right)) + 1
27:        y→height = max(Find_height(y→left), Find_height(y→right)) + 1 return y
28:    end if
29: end procedure
```

(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 $O(\log n)$, where n is the number of nodes in the BST.

In case of insertion of a 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. $O(\log n)$, where H is the height of the binary search tree, i.e., $H = O(\log n)$ where n = current number of nodes in the tree.

In case of query operation, we have to check 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 the whole BST, i.e., the height of the BST. So, the time complexity of the query operation is $O(\log n)$.

Overall, there are n nodes in the nodes and from the starting of the insertion the time complexity of both query and insertion is $O(\log n)$. So the overall time complexity of the algorithm is $O(n \log n)$.