ASSIGNMENT 1

ESO207

Name: Aaryaman Aggarwal
Roll No: 230020

August 2025

1 Maximal Product Sum

1.1 (a)

To obtain the maximum inner product, we take the approach that multiplying elements of larger magnitude would yield a higher product. To achieve this, we sort both arrays A and B and multiply elements of the maximum order.

Algorithm 1 Maximum dot product

```
1: procedure COMPUTE(a[n],b[n],n)
2:     sorted_a ← MeregeSort(a)
3:     sorted_b ← MergeSort(b)
4:     P ← 0
5:     for i ← 1  to  n do
6:         P ← P + sorted_a[i] * sorted_b[i]
7:     end for
8:     return P
9: end procedure
```

Time complexity analysis:
The **Merge sort** function has a time complexity of **cnlogn**.Therefore Step 2 and Step 3 both have a time complexity of $cn \log n$
Step **6** has a time complexity of **1** and the loop runs for n steps. So, the total time complexity of this step is **n**.
Step **4** and **8** have a time complexity of **1** each.
Total time complexity $= 2cn \log n + 2 + n = \mathbf{O(n \log n)}$

1.2 (b)

Given we already have our sorted arrays *sorted_a* and *sorted_b* in addition to the original arrays a and b and we are given (o, x) for each operation. q such operations are given and each operation is of the following manner and we have already computed P as earlier:

- $1 \leq x \leq n$
- If $o = 1$ increment a_x by 1
- If $o = 2$ increment b_x by 1

Algorithm 2 Adjusting for q queries

```
1: procedure UPDATE AND COMPUTE(a[n], b[n], n, o, x P)
2:     if o[i] = 1 then
3:         result ← BINARY_SEARCH(sorted_a[n],a[x-1])
4:         a[x-1] ← a[x-1]+1
5:         sorted_a[result] ← sorted_a[result]+1
6:         P ← P + sorted_b[result]
7:         Return P
8:     else
9:         result ← BINARY_SEARCH(sorted_b[n],b[x-1])
10:        b[x-1] ← b[x-1]+1
11:        sorted_b[result] ← sorted_b[result]+1
12:        P ← P + sorted_a[result]
13:        Return P
14:    end if
15: end procedure
16:
17: procedure Q-SITUATIONS(a[n], b[n], n, q, queries[q], x[q])
18:    for i ← 0 to q do do
19:        P = UPDATE AND COMPUTE(a[n],b[n], n, o[i], x[i], P)
20:        Output P
21:    end for
22: end procedure
```

1.3 Proof of Correctness

First let us prove that the maximum inner product is obtained on multiplying the corresponding elements of the sorted arrays a and b.

Suppose we have two arrays $(a_1, a_2 \cdots a_n)$ and $(b_1, b_2 \cdots, b_n)$ and let's say both are sorted in non decreasing order:

$$a_1 \leq a_2 \cdots \leq a_n \ , \ b_1 \leq b_2 \cdots \leq b_n$$

Now consider any permutation of b, say $b'_1, b'_2 \cdots b'_n$. If this permutation is not sorted, then there exists two indices i and j such that indices $i < j$ and $b'_i > b'_j$.

Now consider the following contribution to P due to a_i, a_j, b'_i, b'_j and we know that $a_i \leq a_j$ as a is sorted

$$a_i b'_i + a_j b'_j \tag{1}$$

Now consider the multiplication of the larger elements in both arrays

$$a_i b'_j + a_j b'_i$$

The difference between these two sums is,

$$(a_j - a_i)(b'_i - b'_j)$$

Since, $a_j \geq a_i$ and $b'_i > b'_j$ this difference is strictly positive. This implies that the product of larger multiples is greater than the other permutation and hence we can say that multiplication of the two sorted arrays would yield the maximum inner product

So, now since we have proved that if the array is sorted the inner product is maximum.
Assertion: P(x) : Updating the last occurrence of a_x ensures the array $a[0, \cdots, n-1]$ is sorted.
Let the array be,

$$a_1, \cdots a_x, a_x, a_{x+1} \cdots a_n$$

Since the array is sorted, $a_{x+1} > a_x$(The equal to is not there since we have assumed this is the last occurrence of a_x) So,

$$a_{x+1} = a_x + 1 \text{ or } a_{x+1} > a_x + 1$$

which is the same as,

$$a_{x+1} \geq a_x + 1$$

On updation the array a becomes,

$$a_1, \cdots a_x, a_x + 1, a_{x+1} \cdots a_n$$

The remaining array was sorted already, and we have that $a_{x+1} \geq a_x + 1$. Hence, proved the array is sorted.
So, we maintain the array to be sorted after each query.

Before the array is updated the contribution of the value changed is,

$$a_x \cdot b_x$$

So, after updating the value it becomes

$$(a_x + 1) \cdot b_x$$

So, the change in the value of p is,

$$(a_x + 1) \cdot b_x - a_x \cdot b_x = b_x$$

So, if the array a is updated then the change in the value of the inner product is the corresponding element in b and the vice versa is true as well.

Time complexity analysis:
Time Complexity of Update and compute: In the procedure of Update and Compute, to compute result, where we use Binary Search in an array of size n which has a time complexity of $c \log n$. The rest of the steps in this procedure take only constant time(c_2) and hence, the time complexity of the procedure Update and Compute is $c_1 \log n \ + c_2$
Time complexity of Q-Situations: In the procedure Q-Situations, there is a loop that iterates q number of times, and it calls the function Update and Compute within the loop. The other step in the loop is to output P which will take constant time only. Therefore we can say that the time complexity of this procedure will be
$q * (c_1 \log n + c_2) \implies$ The time complexity for this is of order $\mathcal{O}(\amalg \log \setminus)$.
This is the time complexity only for the updation procedure, given that P has already been computed. If we include the time taken to compute the initial P, the time complexity would be $c_0 n \log n + c_1 q \log n + c_2 q \implies$ the overall time complexity of the program will be of the order $\mathcal{O}((n + q) \log n)$

2 Game Score Maximization

The first step is to find the number of **distinct prime factors** of each number to maximize the score. We do this using the *Sieve* algorithm as given below. Using this, we pre-compute an array which stores the distinct factors of each number in the given array.

Algorithm 3 Distinct Prime Factors

```
1: procedure SIEVE(arr[n])
2:     M ← max(arr[n])
3:     Create SPF[M]
4:     // An array of size M to hold the smallest prime factor of the number i.
5:     All elements are initialized to 0
6:     SPF[1] ← 1
7:     for i ← 1 to n-1 do
8:         if SPF[i] = 0 then
9:             SPF[i] ← i
10:            j ← i * i
11:            while j ≤ M do
12:                if SPF[j] =0 then
13:                    SPF[j] ← i
14:                    j ← j + i
15:                end if
16:            end while
17:        end if
18:    end for
19:
20:    Initialize an array factors[n] to 0
21:    // This is the array which stores the numbers of distinct factors
22:    for i ← 1 to n do
23:        count ← 0
24:        while x ¿ 1 do
25:            f ← SPF[x]
26:            count ← count + 1
27:            while x mod f = 0 do
28:                x ← x / f
29:            end while
30:        end while
31:        factors[i] ← count
32:    end for
33:    //factors[n] now stores the number of prime factors for each number in arr[n]
34: end procedure
```

So, now that we have calculated the number of distinct prime factors for each element in the array, to maximize our score, we wish to use the largest number the maximum number of times, and then proceed to use the second largest number maximum number of times and so on. To do this, we compute the maximum number of subarrays of $arr[n]$

which would contain a_i such that a_i has the maximum number of prime factors in that subarray

To do this we find the first left and right index from i which has more prime factors than a[i]. This is done using 2 stacks. The algorithm for this is shown below. This will be stored in $right[i]$ and $left[i]$. For any i, $left[i]$ and $right[i]$ store the maximum number of indexes to the left and right from $a[i]$ in which $a[i]$ will be the number with the maximum number of prime factors. Product $left[i] * right[i]$ will be number of sub-arrays which will contain $a[i]$ as the element with maximum distinct prime factors. Hence we can use $a[i]$ that many number of times.

Algorithm 4 Maximize Game Score

```
1: procedure MAXIMIZE SCORE(arr[n],factors[n])
2:     Create Stack a
3:     Create Stack index
4:     Create two arrays: left[n] and right[n]
5:     //To compute the left array
6:     for i ← 0 to n-1 do
7:         if a.empty() then
8:             a.push(factors[i])
9:             index.push(i)
10:            left[i] ← 0
11:            //This is only for the case where the first element is being entered
12:        else
13:            while not a.empty() and a.top < factors[i] dodo
14:                a.pop()
15:                index.pop()
16:            end while
17:            if index.empty() then
18:                left[i] ← i
19:            else
20:                left[i] ← i − index.top() − 1
21:            end if
22:            a.push(factors[i])
23:            index.push(i)
24:        end if
25:    end for
26:    for i ← n-1 to 0 do //Similarily compute right[n]
27:        if a.empty() then
28:            a.push(factors[i])
29:            index.push(i)
30:        else
31:            while not a.empty() and a.top ¡ factors[i] do
32:                a.pop()
33:                index.pop()
34:            end while
```

```
        if index.empty() then
            right[i] ← i
        else
            right[i] ← index.top() − i − 1
        end if
        a.push(factors[i])
        index.push(i)
    end if
end for
// Both left[n] and right[n] are now computed.

sort(arr[n]) and apply the same permutation to right[n] and left[n]
kcount ← 0
i ← n-1
score ← 1
while kcount ≠ k do
    usecount ← (left[i] + 1) ∗ (right[i])
    if ( thenusecount + kcount > k)
        usecount ← k − kcount
    end if
    kcount ← kcount + usecount
    score ← score ∗ power(arr[i], usecount)
end while
end procedure
```

Proof of Correctness:
It is trivial that to maximize our score, we would wish to multiply the larger elements the maximum number of times. Hence, our algorithm hinges on the step that $(left[i] + 1) * (right[i] + 1)$ is the number of subarrays in which $a[i]$ would be selected.
Assertion: $P(i)$: $left[i]$ and $right[i]$ are the maximum values such that

$$\text{In the sub-array } (i - x, i + y)$$
$$\text{where } 0 \leq x \leq left[i]$$
$$\text{and } 0 \leq y \leq right[i]$$

$a[i]$ will always be the number with the highest number of prime factors and

$$prod = (left[i] + 1) * (right[i] + 1)$$

is the maximum number of such sub-arrays

First we shall prove that $left[i]$ is accurate and by symmetry the same will be true for $right[i]$
In our algorithm, we use a stack and either of the following happens:

- If stack, a is empty, we push $factors[i]$ and store $left[i]$ as 0, as this will only occur in the first iteration when the stack is empty. For the first element of the array, it is obvious that the only value of L which will contain the first element in a subarray would be 0. The vice-versa would be true for the value of R

- If $a.top() > factors[i]$ then it is simply pushed on to the stack, and $left[i]$ will again be stored as . The reason behind this is that the element before i has a greater number of factors. Therefore, any sub-array which contains the element to the left of i will always choose that element over i. Hence only if the sub-array begins with i, then $arr[i]$ will be chosen. The converse is true for R as well.

- If $a.top < factors[i]$ we pop both the stack until the top element is greater than $factors[i]$. We look for the element(say x) on the left which has a greater number of factors than $arr[i]$. Any sub-array that starts after the element x and ends at $right[i]$ will always choose $arr[i]$ to maximize the score. Hence
  $left[i] \leftarrow i - index.top() - 1$ which is the length till the previous element which has greater number of factors than $arr[i]$. Hence for any index greater than equal $i - left[i]$ to $i + right[i]$ $arr[i]$ would always be chosen.

To create a sub-array (L, R) which has $arr[i]$ as the element with the maximum number of distinct factors, we can choose L from the set
$\{i - left[i], i - left[i] + 1, i - left[i] + 2, \ldots, i\}$ and R from the set
$\{i, i + 1, i + 2, \ldots i + right[i] - 1, i + right[i]\}$.

Therefore, the number of choices for L is equal to $left[i] + 1$(1 extra as the case $L = R$ is valid, hence choosing i for L or R would also be valid), and for R is equal to $right[i] + 1$. Now from combinations, the number of subarrays are the total possible combinations of (L, R) which is equal to $(left[i] + 1) * (right[i] + 1)$. Therefore, just knowing the extremes will give us the total number of sub-arrays in which we would select the element $arr[i]$.

The last part of the algorithm is trivial as we sort $arr[i]$ and apply the same permutation on $left[n]$ and $right[n]$ to maintain the order. We begin from the largest element and use it as many times $(= (left[i] + 1) * (right[i] + 1))$ to maximize our score.

Time Complexity

- To compute the total number of prime factors for all the numbers of the array using *Sieve* take time $c_1 n \log M$

- The loops in *Algorithm* 4 to calculate $left[n]$ and $right[n]$ iterate n number of times, and all the operations with the loop are of constant time. Therefore the time complexity of this step is $c_2 n$
- Sorting the arrays will take $c_3 n \log n$ time.
- The last loop is dependent on $c_4 k$ which would be lower than $n \log n$

The total time complexity of the algorithm would be $= c_1 n \log M + c_2 n + c_3 n \log n + c_4 k$. Hence, the time complexity would be of $\mathcal{O}(n \log n + n \log M + k)$

3 Wealth accumulate

This problem is a clearly a problem of recursion. We flatten our BST such that we write all the nodes of each level sequentially from left to right. Let the name of such a traversal be level traversal.
So that we get a vector of the following order.

$$X_o \leftarrow \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

Where a_1 is the root node of the family tree. a_2 is the left child and a_3 is the right child of a_1 and this continues. We can also say that after this flattening, the children of a_i will be $2 * i + 1$. Hence for any element j it's parent will be $\frac{j-1}{2}$. We do not need to take separate cases due to the floor nature of integer division. We use two matrices which are alternatively multiplied, once where half of the wealth is given to the left child and in the next year, it is given to the right child. The two matrices are,

$$A \leftarrow \begin{pmatrix} 2 & 0 & 0 & 0\cdots \\ 2 & 2 & 0 & 0\cdots \\ 0 & 0 & 2 & 0\cdots \\ 0 & 2 & 0 & 2\cdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} \quad \text{and} \quad B \leftarrow \begin{pmatrix} 2 & 0 & 0 & 0\cdots \\ 0 & 2 & 0 & 0\cdots \\ 2 & 0 & 2 & 0\cdots \\ 0 & 0 & 0 & 2\cdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

$$A_{i,j} = \begin{cases} 4, & i = j \text{ and } 2j+1 > n \\ 2, & i = j \text{ and } 2j+1 \leq n \\ 2, & i = 2j+1 \text{ and } 2j+1 \leq n \\ 0, & \text{otherwise} \end{cases} \qquad B_{i,j} = \begin{cases} 4, & i = j \text{ and } 2j > n \\ 2, & i = j \text{ and } 2j \leq n \\ 2, & i = 2j \text{ and } 2j \leq n \\ 0, & \text{otherwise.} \end{cases}$$

Note that the last diagonal elements of the matrix will be just 4 since they don't have any children to give their wealth. The recurrence relation can be written as follows:

$$\text{for i} = 1 \qquad a_1(n) = 2 * a_1(n-1)$$
$$\text{for i} \geq 2 \qquad a_i(n) = 2 * a_i(n-1) + 2 * a_j(n-1) \quad \text{where} \quad i = 2j+1$$
$$\text{i.e i is the child of j and i has children itself.}$$
$$\text{if i has no children then} \qquad a_i(n) = 4 * a_i(n-1) + 2 * a_j(n-1) \quad \text{where} \quad 2i+1 > n$$

Note that the relation in line 2 is general and will alternate based on the year which is taken care of by considering two matrices. Combining the above recurrence relation

using the matrices given we can say that

$$X(k) = \begin{cases} (AB)^{k/2} X_o & if \;\; k \;\; is \;\; even \\ (AB)^{k/2} A X_o & if \;\; k \;\; is \;\; odd \end{cases}$$

Algorithm 5 Wealth Accumulate

```
1: procedure POWER(n,A)
2:     if n = 1 then
3:         return A
4:     end if
5:     if k mod 2 = 0 then
6:         return power(n/2,A) * power(n/2,A)    ▷ the multiplication here is a matrix multiplication
7:     else
8:         return power(n/2,A) * power(n/2,A) * A    ▷ the multiplication here is a matrix multiplication
9:     end if
10: end procedure
11:
12: procedure COMPUTE(n,T,k)
13:     Create a n x 1 array arr which stores the Binary Tree
14:     arr ← LevelTraversal(T)
15:     if k mod 2 = 0 then
16:         result ← power(k/2,A*B) * arr    ▷ the multiplication here is a matrix multiplication
17:     else
18:         result ← power(k/2,A*B) * A * arr    ▷ the multiplication here is a matrix multiplication
19:     end if
20:     return result
21: end procedure
```

Time complexity Analysis:

The power function includes the multiplication of 2 $n * n$ matrices takes $c_0 n^3$ steps. The Power function using binary exponentiation takes $c_1 \log k$ steps and in each step $c_o n^3$ operations are required. Hence, the total time for the Power function is $cn^3 \log k$.
The level traversal takes cn time which will be of a lower order then $\mathcal{O}(\backslash^{\ni} \log \|)$ and all the other steps take constant time. The recursive relation for the time can also be

written as follows:

$$T(k) = c_1 n^3 + 2 * T(k/2)$$

$$T(k) = c_1 n^3 + c_1 n^3 + 4 * T(k/2^2)$$

$$\vdots$$

$$T(k) = c_1 n^3 logk$$

So, the total time complexity is given by $cn + cn^3 logk = \mathcal{O}(n^3 \log k)$ since we have $k >>> n$.

4 The King's Punishment

An efficient $O(n \log n)$ algorithm for this can be derived of the problem of counting inversions. Instead of counting the total number of inversions, we count the number of inversions for each individual element and store it.

Algorithm 6 King's Punishment, Inversion Count

```
1: struct Point {h, pu} // A structure which will simultaneously store the height and punishments for the guard.
2: procedure MergeAndCount(Arr, i, mid,k, C)
3:     p ← i; j ← mid + 1; r ← 0;
4:     while p ≤ mid and j ≤ k do
5:         if A[p].h ¡ A[j].h then
6:             C[r] ← A[p]; r++; p++
7:         else
8:             C[r] ← A[j]; j++
9:             C[r].pu ← C[r].pu + (mid - p + 1); r++;
10:            //It increments the total number of elements in (i, mid) that are larger than A[j].
11:        end if
12:    end while
13:    while p ≤ mid do
14:        C[r] ← A[p]; p++; r++;
15:    end while
16:    while j ≤ k do
17:        C[r] ← A[j]; j++; r++;
18:    end while
19: end procedure
20: procedure SortAndInvCount(A[n], i, k)
21:    if i = k thenreturn 0;
22:    else
23:        mid ← (i + k) / 2
24:        SortAndInvCount(A, i, mid)
25:        SortAndInvCount(A, mid + 1, k)
26:        Create a temporary Arr C[0, . . . , k - i]
27:        MergeAndCount(A, i, mid, k, C)
28:        Copy C[0, . . . , k - i] to A[i, . . . , k]
29:    end if
30: end procedure
```

```
31: procedure MAIN(G[n])
32:     //Arr G[n] is the original array on which we need to work
33:     //First construct an Array A[n] which stores the Data Structure Point using G[n]
34:     Initialize A[n]
35:     for i ← 0 to n-1 do
36:         A[i].h ← G[i]
37:         A[i].pu ← 0
38:     end forSORTANDINVCOUNT(A, 0, n-1)
39:     //The final state of A will be sorted in order of the heights of each element and
    contain the total number of punishments as well
40:     for i ← o to n-1 do
41:         Outpute A[i].pu
42:     end for
43: end procedure
```

This above algorithm will return an array with the number of punishments each guard receives.

(a) A brute force algorithm for this question would be that for each index we iteratively check how many elements before it is greater than the element at the current index. So, for each index i we check i-1 elements.

Time complexity:

$$T(n) = \sum_{i=0}^{n} i - 1$$

$$T(n) = n * (n+1)/2 - n$$

$$T(n) = n * (n-1)/2$$

$$T(n) = O(n^2)$$

(b). So, we once we have computed the sorted array as in *Algorithm* 6 we can convert it into a Binary Search Tree in $\mathcal{O}(n)$ time. We use a BST and insert the heights of the knights sequentially into this and aslo keep a track at each node the size of subtree of each node so whenever we insert a new height we keep a track of all the nodes on the right hand side of it as all these nodes have a height greater than the current knight and were inserted before.

Another approach is that at each node during insertion of the new knight while creation of the BST from the given array. Every time we move to the left node we update our count.

Algorithm 7 King's Punishment, BST Approach

```
1: procedure INSERT(T,x)
2:     p ← T
3:     found ← False
4:     count ← 0
5:     while found = False do
6:         size(p) ← size(p) + 1
7:         if value(p) > x then
8:             if left(p) = NULL then
9:                 left(p) ← createnode(x)
10:                found ← True
11:            else
12:                count ← count + size(right(p)) + 1
13:                p ← left(p)
14:            end if
15:        else
16:            if right(p) = NULL then
17:                right(p) ← createnode(x)
18:                found ← True
19:            else
20:                p ← right(p)
21:            end if
22:        end if
23:    end while
24:    return count
25: end procedure
```

So, the required operations for this data structure is insert and size. So, we can obtain the size of the subtree of each node while inserting the new heights into the BST.

(c). Now, one issue clearly is maintaining the nearly balanced BST so that the insert operation still takes place in O(logn) time. For this we keep a safeguard that after every insertion if the height of the subtree has become more than $\frac{3}{4}^{th}$ of the height of the tree then we remake our BST which will simply take $\mathcal{O}(n)$. Here, is the algorithm with the self balancing part added when the tree gets unbalanced.

Algorithm 8 Balancing Insert

```
1: procedure INSERT(T, x)
2:     p ← T
3:     found ← False
4:     count ← 0
5:     while found = False do
6:         size(p) ← size(p) + 1
7:         if value(p) > x then
8:             if left(p) = NULL then
9:                 left(p) ← createnode(x)
10:                found ← True
11:            else
12:                count ← count + size(right(p)) + 1
13:                p ← left(p)
14:            end if
15:        else
16:            if right(p) = NULL then
17:                right(p) ← createnode(x)
18:                found ← True
19:            else
20:                p ← right(p)
21:            end if
22:        end if
23:    end while
24:    if IsUnbalanced(T) then
25:        T ← Rebuild(T)
26:    end if
27:    return count
28: end procedure
29: procedure IsUnbalanced(u)
30:    if max(size(left(u)), size(right(u))) > 3/4 · size(u) then
31:        return True
32:    else
33:        return False
34:    end if
35: end procedure
```

(d). The rebuilding of tree takes place very rarely during the program so on an average we can ignore the time complexity as the rebuilding happens only a few number of times.

For inserting a node, the time taken will be proportional to the height of the BST which has to be traversed at the given moment, which will be $\log n$ at max. Therefore, the time complexity will be to insert all n nodes would be a sum of the time taken for each

insertion.

$$\mathrm{T(n)} = \sum_{i=1}^{n} log(i)$$

$$T(n) = \sum_{i=1}^{n} log(i) \leq \sum_{i=1}^{n} log(n) = nlogn$$
$$\implies T(n) = \mathcal{O}(n \log n)$$

The rebuilding of the tree will also be a rare occurence, but it will be an operation of $\mathcal{O}(n)$ as we can get the sorted array through an in-order traversal in $\mathcal{O}(n)$ and creating a BST from the array will also be $\mathcal{O}(n)$.
Therefore the total time complexity would be of $\mathcal{O}(n \log n)$.

Even updation of any single node would take $\mathcal{O}(\log n)$ as we would have to traverse till the element to update which is just the height of the BST.

Even for *Algorithm* 6 the time complexity is of $\mathcal{O}(n \log n)$ as the recurrence relation is the same as $MergeSort$ which is also of $\mathcal{O}(n \log n)$ with some extra arthimetic steps of constant time to count the Inversions.

The Binary Search tree would be better as it would allow us to insert and update elements in a much easier manner with better time complexity than an array based approach.