

Theoretical Assignment 1 Solutions

Question 1

Solution Sketch:

(a) The maximum dot product between two sequences $a = [a_1, \dots, a_n]$ and $b = [b_1, \dots, b_n]$ is obtained when both are sorted in the same order. Thus, sort a and b in ascending order and compute

$$P = \sum_{i=1}^n a_i \cdot b_i.$$

This takes $O(n \log n)$ time due to sorting.

(b) We now allow q operations where an element of a or b is incremented by 1. The challenge is to maintain sorted order of a and b after each update without resorting to brute-force re-sorting in $O(n)$ time per operation.

Idea. Suppose an element c_x (from either a or b) is incremented:

$$c_x \leftarrow c_x + 1.$$

Two cases arise:

- If $(c_x + 1) \leq c_{x+1}$ (or $x = n$), the array remains sorted. so we just have to update the value of P
- Otherwise, the array becomes unsorted. A naive fix is to repeatedly swap c_x with its right neighbor until the condition $c_x \leq c_{x+1}$ holds. However, this swap operation are unnecessary when

$$c_x = c_{x+1}.$$

Instead, we directly set

$$x' = \max\{k \mid c_k = c_x\},$$

and increment

$$c_{x'} \leftarrow c_{x'} + 1.$$

After this update, the array remains sorted. Since c is always maintained in sorted order, the index x' can be found using a binary search in $O(\log n)$ time.

Updating the dot product P . Suppose the incremented element at index i was originally paired with b_i , and after adjustment moves to position j . The change in dot product is:

$$\Delta P = -a_i b_i - a_j b_j + (a_i + 1)b_j + a_j b_i.$$

Since $a_i = a_j$ (or $b_i = b_j$) during this process, the expression simplifies to

$$\Delta P = \begin{cases} +b_j & \text{if an element of } a \text{ was incremented,} \\ +a_j & \text{if an element of } b \text{ was incremented.} \end{cases}$$

(c) *Correctness:* The idea is to observe that product of two maximum number will contribute toward the maximum sum of the product. Since both a and b are initially sorted, and after each increment we either keep the element in place or move it to the last position among equal values, the arrays remain sorted. This ensures the array remains sorted after each operation, allowing the use of binary search. And we can simply update the value of P after that.

Time Complexity: Each update requires:

- A binary search to find x' in $O(\log n)$ time,
- A constant-time update of P .

Therefore, the total complexity for q operations is $O(q \log n)$. If we also include the preprocessing step from part (a), namely sorting a and b , an additional $O(n \log n)$ is incurred.

Question 2

Solution Sketch

Let us define the *prime score* of a number as the number of distinct prime factors it has. We aim to maximize the score by choosing subarrays

where the element with the highest *prime score* (and smallest index in case of ties) contributes to the score. Instead of considering every possible subarray, we determine for each element the number of subarrays in which it can be selected. Then, by greedily picking elements with higher values, we ensure the maximum possible product.

Approach

Prime Factorization. To calculate the prime score for each number up to M (the maximum value of an element), we iterate through the numbers from 1 to M . Whenever we encounter a prime number, we consider all of its multiples and increase their count by one. This ensures that each number accumulates the number of distinct primes dividing it. The primes up to M can either be precomputed in a list, or identified dynamically by repeatedly marking the multiples of each prime as non-prime, starting from 2.

Monotonic Stack. We perform two passes with a monotonic stack to calculate the left and right boundaries for each element. These boundaries define the range in which the element is the maximum based on prime score. We use a monotonic stack to compute the boundary arrays as follows:

- **Left boundary:** Index of first greater element on the left.
- **Right boundary:** Index of first greater-or-equal element on the right.

The number of subarrays where the element can be chosen is the product of distances to these boundaries.

We pair each element with its count of subarrays and sort these pairs in descending order by the element's value. Then, we multiply the score by the element raised to the power of the minimum of its count or the remaining operations k , using fast modular exponentiation (binary exponentiation).

Complexity

Time Complexity:

- Computing the prime score for all numbers up to m (the maximum element in `nums`) takes

$$O(m \log \log m).$$

This comes from the process of iterating over each prime p and marking all of its multiples. The total number of operations is

$$\sum_{p \leq m} \frac{m}{p} = m \cdot \sum_{p \leq m} \frac{1}{p}.$$

Since the sum of reciprocals of primes up to m is $O(\log \log m)$, the complexity follows.

- Sorting the n elements of the array in descending order of their values takes

$$O(n \log n).$$

- The monotonic stack computations for left and right boundaries run in linear time,

$$O(n).$$

Thus, the overall complexity is

$$O(n \log n + m \log \log m).$$

Here we can assume that $m \leq n$, and therefore the complexity is bounded by

$$O(n \log n).$$

Space Complexity:

$$O(n + m).$$

Question 3

Solution Sketch:

1. **Representing the Tree as a Vector:** Perform a level-order traversal of the binary tree and index the nodes from 1 to n in this order. Let the wealth of node i at time t be $w_i(t)$. Collect all of them into a column vector:

$$W_t = [w_1(t), w_2(t), \dots, w_n(t)]^T.$$

2. **Wealth Update as Linear Transformation:** In each year, every person's wealth is quadrupled, and half of it is donated to exactly one child (left or right depending on the year). This can be written as a linear update:

$$W_{t+1} = M_t \cdot W_t,$$

where M_t is an $n \times n$ transformation matrix.

There are two such matrices:

- **Left-donation year (M_L):** For each node i :
 - Self-retained wealth: $2 \cdot w_i(t)$ (quadruple and give half away).
 - Left child receives: $2 \cdot w_i(t)$.
- **Right-donation year (M_R):** Same as above, but the donation goes to the right child.

Thus, M_L and M_R are sparse $n \times n$ matrices where each row encodes the contribution of a node to itself and its child with appropriate case handling for leaf nodes.

3. **Combining Alternating Transformations:** Since donations alternate every year, we can combine two consecutive years into one transformation:

$$W_{t+2} = (M_R \cdot M_L) \cdot W_t.$$

Define:

$$M = M_R \cdot M_L.$$

Then:

$$W_k = \begin{cases} M^{k/2} \cdot W_0 & \text{if } k \text{ is even,} \\ M_L \cdot M^{\lfloor k/2 \rfloor} \cdot W_0 & \text{if } k \text{ is odd.} \end{cases}$$

4. **Efficient Computation:** Direct simulation year by year is infeasible for $k \gg n$. Instead, use fast matrix exponentiation:

- Matrix exponentiation requires $O(\log k)$ multiplications.
- Each multiplication of two $n \times n$ matrices costs $O(n^3)$.

5. Time Complexity:

- Fast exponentiation: $O(n^3 \log k)$.

- Final multiplication with W_0 : $O(n^2)$.

Hence, the overall time complexity is:

$$O(n^3 \log k).$$

Question 4

Part (a). Brute Force Check all pairs (i, j) with $i < j$; if $h[i] > h[j]$, punish knight j . Runs in $O(n^2)$ time.

Part (b). Data Structure Use a Binary Search Tree (BST) with subtree sizes to count taller knights efficiently.

Part (c). Pseudocode

```

1 // BST node: (value, left, right, size)
2
3 function query(T, x):
4     // number of elements in T greater than x
5     if T is null: return 0
6     if x < T.value:
7         return 1 + size(T.right) + query(T.left, x)
8     else:
9         return query(T.right, x)
10
11 function update(T, x):
12     // insert x and update subtree sizes
13     if T is null: return newNode(x)
14     if x < T.value:
15         T.left = update(T.left, x)
16     else:
17         T.right = update(T.right, x)
18     T.size = 1 + size(T.left) + size(T.right)
19
20     if not nearlyBalanced(T):
21         T = rebuildBalanced(T)    // rebuild subtree
22     return T
23
24 procedure solve(n, h[1..n]):
25     ans[1..n] = 0
26     T = null

```

```
27     for j = 1 to n:  
28         ans[j] = query(T, h[j])  
29         T = update(T, h[j])  
30     output ans
```

Part (d). Efficiency Keep the BST nearly balanced; rebuild any unbalanced subtree to ensure logarithmic depth.

Part (e). Complexity Each query and update is $O(\log n)$, giving total $O(n \log n)$.