HackerEarth August Circuit 2018 Editorial

Hasin Rayhan Dewan Dhruboo

September 2019

1 Problem 1 : Distributing Toys

There are $\binom{n-1}{2}$ ways to distribute n toys among 3 people. So you can brute-force all possible ways in $O(n^2)$ and try to see that if a specific distribution meets the condition in the statement.
You can find the setter's solution here.

2 Problem 2 : Health of a person

You can see that i^{th} person will only be affected by the walks in j^{th}days when j is a divisor of i. So for i^{th}person, you need to find the smallest divisor(let's call it x) of i, so that $A_i \leq B_x$.

We can find such smallest divisor x for every i using sieve of Eratosthenes in $O(nlogn)$ time complexity.
You can find the setter's solution here.

3 Problem 3 : Moving People

There are two main observations:

- The Left-Right move and the Up-Down move is independent of each other.
- After any number of moves, the present people in the grid form a rectangle. Let's call it $CurrentRectangle$.

So, we will keep track of four variables $Left, Right, Up$ and $Down$ which will denote the rectangle's current form. Initially, $Left = 1,\ Right = m,\ Up = 1,\ Down = n$.

Now let's say we need to move the people k steps left, So some people in the left side of the current rectangle will be out of the grid now. So,the $Left$ and $Right$ pointer will need to move k steps right. So we will do the following operation : $Left = Left + k,\ Right = Right + k$.

But after some observation you can see that, this $left$ doesn't point the actual left side of the $CurrentRectangle$, because maybe the $Left$ pointer has already got past this point in some of the previous moves. That's why we need to keep track of the maximum position where $Left$ pointer has already been, which we will call $maxLeft$ (Initially 1). We will update it after every update : $maxLeft = max(maxLeft, Left)$

For moving people k steps right, the $Right$ pointer will move k steps left and in the same manner we will keep track of $minRight$(initially m).

For moving people up and down, we can use the same logic and keep track of Up, $Down$, $maxUp$, $minDown$.

For query operation we can pre-process and make a cumulative sum on the initial grid and answer query in O(1). See setter's code for details. If after any move $maxLeft > minRight$ or $maxUp > minDown$, the answer will be 0.
You can find the setter's solution here.

4 Problem 4 : Maximizing Expressions

We can see that, the operation on each index is independent. So for each index we will do the following operations and try to find the best D value to maximize our answer. Let's ignore the non-zero condition on our D value for now.

Now let's try to maximize the value $cur_i = A_i \oplus B_i$. The answer will be $S = \sum_{i=1}^{n} cur_i$.

For every j^{th} bit of C_i,

- if it's 1, then our D may have 1 or 0 in that bit.
- otherwise that bit for D must be 0.

Let's define a function $getBit(val, j)$ which returns 1 if j_{th} of val is set, otherwise it returns 0.

Now for every j^{th} bit, let

- $X = getBit(A_i, j) \oplus getBit(B_i, j)$
- $Y = getBit(A_i, j) \oplus getBit(B_i, j) \oplus getBit(C_i, j)$

Now, we can see that we just need to take $max(X, Y)$ for every bit of cur_i. In other words, for j^{th} bit of cur_i,if $max(X, Y) = 1$, then we will make that bit 1, otherwise it will be 0.

But in this way, maybe for some i^th position, we will take $D = 0$, So in

that case, we just need to forcefully take the Last Set Bit of C_i, and (XOR) it with B_i.
Time Complexity : $O(n * log_2(max(A_i)))$

See the setter's code here for implementation details.

5 Problem 5 : GCD Problem

We will process the queries (L, R) offline in the non-increasing order of their L. We will keep an array end, where end_i = sum of all $F(i, j)$ where $i \geq currentPosition$ and $y = j$.
Let's say we have processed all the queries whose $L > x$ and now we are x^{th} position. So, currently end_y = sum of all $F(i, j)$ where $i > x$ and $y = j$.

So we need to add $F(x, y)$ to end_y for all y such that $y \geq x$. In other words, we need to find the $F(i, j)$ value for all the sub-arrays which starts at x^th position and add this value to end_j.

The main observation is that there are at most distinct $\log_2(A_x)$ values for all the F values of the sub-arrays starting at x^th position. Another observation is that for a fixed F value, the right-point of the sub-array is in a continuous range (y_1, y_2). So we need to add this F value to all end_y such that $y_1 \leq y \leq y_2$. We can find these $log_2(A_i)$ distinct values and their corresponding ranges using binary search and sparse table in $O(logn * log(A_i))$ time complexity.

Now we will process queries for whose $L = x$. It's easy to see that for these queries(starting at x^{th} position) the answer is just range sum of end array from their corresponding L *to* R.
As we can see that there is only range sum updates and range sum queries in end array, so we can use a segment tree to maintain the end array.

Total Time Complexity : $O(n * logn * log(max(A_i)) + n * logn)$

See the setter's code here for implementation details.

6 Problem 6 : Constructing Building

Lets solve the counting version first. We want to calculate the number of ways that satisfies our condition.

First, We build palindromic tree on each of the strings.

The main observation is : there can be at most $O(L)$ distinct palindromes in a string of length L. We maintain a map c_i for each string , where we store

the occurrence count of each palindrome sub-string against a hash-value for the palindrome.
A palindrome is a candidate for our answer, if it's hash-value is in all the map.

Now, We initialize Ans = 0.
Then for each candidate palindrome X , we add $\prod_{i=1}^{i=N} c_i[X]$ to Ans.
Calculating the probability is left as an exercise.

Complexity : $O(\sum_{i=1}^{i=N} |S_i|)$
See the setter's code here for implementation details.

7 Problem 7 : Permutation

Build a segment tree and keep a trie in each node.
For a node denoting $[l..r]$, insert each a_p such that $l \leq p \leq r$ in the trie of this node.
Complexity: $O(N * log\ N * log\ a_i)$

For update operation, update a_x by traversing each node from the root to the leaf denoting $[x, x]$ in the segment tree, and deleting a_x from their trie. For inserting a_y in those trie. You can update a_y in the same manner.
Complexity: $O(log\ N * log\ a_i)$

For query operation, find the $O(logn)$ nodes which represent range $[l, r]$ in the segment tree. We need to find $K - mex$ in the union of the tries of these nodes.

Let's review the process of finding K-mex in a single trie. We begin from the root of the trie and always try to move to the child through '0' edge. If we find that number of missing elements in '0' edge child is equal or greater than K, then we can move to that child. If we can't, then we move to the child through '1' edge and also subtract the number of missing elements in '0' edge child from K. Repeating these steps, we find K-Mex as we reach a leaf.

To solve our current problem, instead of beginning from a single root, we travel from all the O(log n) trie roots in parallel. In each step, we check if the sum of missing elements in '0' edge child of all the trie is equal or greater than K, then we move to that '0' edge child in all the trie. Otherwise, we move to the '1' edge child.

Complexity : $O(log\ N * log\ a_i)$

Overall Complexity : $O((N + Q) * log\ N * log\ a_i)$

See the setter's code here for implementation details.