

RCPCamp 2023 Iasi Day Editorial

A. Yet Another Colored Tree Problem

Instead of counting the number of paths that contain at least one node with a certain color c , we can count the number of paths that don't contain any nodes with color c .

Since the number of paths in a connected component with k nodes is k^2 , we only need to find the sizes of the connected components which don't contain any nodes with color c .

For the sake of simplicity, let's root the tree at vertex 1.

Let $dp[u]$ be the size of the connected component rooted at u containing only nodes who have a different color than $parent[u]$. Initially, $dp[u] = subtree_size[u]$.

To compute the actual dp values, we will perform a dfs traversal of the tree starting from node 1. For each node, while performing the dfs traversal, we will also keep $prev[c] = v$: the lowest node v such that $color[parent[v]] = c$.

While processing node u , we will decrease the value of $dp[prev[color[c]]]$ by $subtree_size[u]$.

Let $ans[c]$ be the answer for color c . Initially, $ans[c] = n^2$. For each node u such that $color[parent[u]] = c$, $ans[c]$ will be decreased by $dp[u]^2$.

Time complexity: $O(N)$

B. Coins

- If $n = 1$, then the first player wins by taking the last coin.
- If $n = 2$, then the first player can only take one coin, and the second player will take the last coin.
- If $n = 3$, then the first player can take one coin, which forces the second player into a losing state.
- If $n > 2$ and n is even, then the first player can take $n - 2$ coins, forcing the second player into a losing state.
- If $n > 3$ and n is a prime integer, then the first can either take one coin, which makes n even, resulting in a win for the second player, or take $n - 1$ coins, which also results in a win for the second player.
- If $n = 9$, then the first player can take 1, 6 or 8 coins, all of which result in a win for the second player.
- If n is a power of 3 greater than 9, then the first player can take $n - 9$ coins, resulting in a loss for the second player.
- If n is divisible by any prime p greater than 3, the first player can take $n - p$ coins, resulting in a loss for the second player.

Therefore:

- If $n = 9$, then the second player wins.
- If $n = 3$, then the first player wins.
- If n is a prime number and $n \neq 3$, then the second player wins.
- If n is **not** a prime number and $n \neq 9$, then the first player wins.

Time complexity per testcase: $O(\sqrt{n})$

C. Almost Tree Cut

Since there are n nodes and n edges, the graph looks like a cycle with tree subgraphs sticking out of it.

Therefore, we can split the graph into two different subgraphs in two main ways:

1. We can delete one edge from one of the tree subgraphs; or
2. We can delete two edges from the cycle.

If we ignore the edges from the cycle, let $dp[u]$ be the sum of $cost[v]$ for all v who are in the subtree of u . Additionally, let s be the sum of all costs.

We will compute this dp using topological sorting based on which nodes have a degree of 1 (aka leaves).

The cost of a cut obtained by deleting one edge from a subtree connecting u and $parent[u]$ is equal to $|(s - dp[u]) - dp[u]|$.

Next, we will replace $cost[u]$ with $dp[u]$ for every node in the cycle, as all of the subtrees can be compressed into their corresponding root from the cycle.

Now, the problem boils down to finding the subarray with the sum closest to $\frac{s}{2}$ in a circular array containing non-negative integers, which can be solved using two pointers or binary search.

Time complexity: $O(N)$ or $O(N \log N)$, depending on the implementation.

D. Doping2

First of all, we will learn how to count permutations with $f(p) = x$ without worrying about the lexicographically smaller condition. It turns out that if we look at the inverse permutation, that is p^{-1} , $f(p)$ is the number of ascents in p^{-1} . Thus we have reduced counting permutations with $f(p) = x$ to counting permutations with x ascents ($p_i < p_{i+1}$ is an ascent), which is well known as eulerian numbers.

Now to deal with the lexicographically smaller condition, one can iterate through the common prefix of p and p' and then try every single possibility to create a mismatch. Now we are left with bunch of values, let's group them into consecutive ranges and note that each such range contributes independently to $f(p')$. This way we can convolve all the ranges together, getting all possible configurations of permutations.

Time complexity: $O(n^4)$.

E. Anton Would Approve This Problem

Let's consider a binary string of length n consisting of alternating characters:

0101010101...

This string can be "fixed" by deleting exactly $\lfloor \frac{n}{3} \rfloor$ characters, as follows:

- If $n \bmod 3 = 0$, then: 010101010101 \Rightarrow 01**0**10**1**01**0**1**0**1 \Rightarrow 01100110
- If $n \bmod 3 = 1$, then: 0101010101010 \Rightarrow 01**0**10**1**01**0**10**1** \Rightarrow 011001100
- If $n \bmod 3 = 2$, then: 01010101010101 \Rightarrow 01**0**10**1**01**0**10**1** \Rightarrow 0110011001

Therefore, each alternating substring of length len will require $\lfloor \frac{len}{3} \rfloor$ deletions.

For example, the answer for $s = [010]00[01][101010101]111[1010][01010101]1[101]$ is equal to:

$$\lfloor \frac{3}{3} \rfloor + \lfloor \frac{2}{3} \rfloor + \lfloor \frac{9}{3} \rfloor + \lfloor \frac{4}{3} \rfloor + \lfloor \frac{8}{3} \rfloor + \lfloor \frac{3}{3} \rfloor = 1 + 0 + 3 + 1 + 2 + 1 = 7$$

Time complexity per testcase: $O(N)$

F. Difference in Skill

Since the order of the employees doesn't matter for now, we can sort the array a .

Using two pointers or binary search on the sorted array a , we can find for each employee a_l the range of employees $[l, r]$ who have their skill level in the range $[a_l, a_l + k]$.

Since all of these employees can be part of a balanced project, we will perform:

$$ans[initial_order(i)] = \max(ans[initial_order(i)], r - l + 1)$$

for all $l \leq i \leq r$.

The most standard way to implement this would be via a lazy segment tree.

However, since we don't have to perform these queries online, we can also use difference arrays or a monotone stack.

Time complexity: $O(N \log N)$.

G. Sign Flipping

For each $x \leq 0$, let i_1, i_2, \dots, i_k be the indices for which $|a_{i_j}| = x$.

In order to maximize the sum, it is optimal to make $a_{i_j} = -a_{i_{j-1}}$, for all $2 \leq j \leq k$.

The answer can be computed using contribution technique.

Time complexity: $O(N \log N)$ (or $O(N)$ with `unordered_map`).

H. The Binary Matrix of All Time

The matrix containing the most 1's can be obtained using the following formula:

$$a_{i,j} = [(i+j) \bmod 3 \neq 1]$$

This yields the following matrix:

```
1 1 0 1 1 0 1 1 0 1 1 0 ...
1 0 1 1 0 1 1 0 1 1 0 1 ...
0 1 1 0 1 1 0 1 1 0 1 1 ...
1 1 0 1 1 0 1 1 0 1 1 0 ...
1 0 1 1 0 1 1 0 1 1 0 1 ...
0 1 1 0 1 1 0 1 1 0 1 1 ...
...
```

The total number of 1's can be written as:

$$\lfloor \frac{2 \cdot n \cdot m + 2}{3} \rfloor$$

Time complexity per testcase: $O(1)$

I. Weird Divisibility

Let $d = \gcd(a, b)$. If $d = 1$, then $a + b$ and $a \cdot b$ have no prime divisors in common.

Proof:

Let p be a prime divisor of $a \cdot b$. Since $\gcd(a, b) = 1$, **exactly one** of the following statements is true:

1. $a \bmod p = 0$; $b \bmod p \neq 0$

2. $b \bmod p = 0$; $a \bmod p \neq 0$

Therefore, $(a + b) \bmod p \neq 0$, for all prime divisors p of $a \cdot b$, which is equivalent to $a + b$ and $a \cdot b$ not having any common prime divisors.

If $a = d \cdot x$ and $b = d \cdot y$ ($\gcd(x, y) = 1$), then:

$$a + b \mid a \cdot b \Leftrightarrow$$

$$d \cdot x + d \cdot y \mid d \cdot x \cdot d \cdot y \Leftrightarrow$$

$$d \cdot (x + y) \mid d^2 \cdot x \cdot y \Leftrightarrow$$

$$x + y \mid d \cdot x \cdot y$$

Since $\gcd(x, y) = 1$, $x + y$ and $x \cdot y$ have no common prime divisors. Therefore:

$$x + y \mid d \Leftrightarrow$$

$$d \cdot (x + y) \mid d^2 \Leftrightarrow$$

$$a + b \mid d^2 \mid a^2$$

This means that $a + b$ has to be a divisor of a^2 .

For $a \leq 10^9$, a^2 always has less than $5 \cdot 10^4$ divisors, which means that we can brute force through all of the divisors of a^2 to find the answer.

Time complexity per testcase: $O(\frac{\sqrt{a}}{\log(a)} + nrdiv(a^2))$

Another proof that $a + b$ has to be a divisor of a^2 .

Since:

$$a + b \mid a \cdot (a + b)$$

$$a + b \mid a \cdot b$$

We get:

$$a + b \mid a \cdot (a + b) - a \cdot b \Leftrightarrow$$

$$a + b \mid a^2$$

J. Triple Reverse Sort

First of all, reversing $[a_i, a_{i+1}, a_{i+2}]$ is the same thing as swapping a_i and a_{i+2} .

If we create two subsets, one containing $a_1, a_3, \dots, a_{2 \cdot k + 1}$, and the other one containing $a_2, a_4, \dots, a_{2 \cdot k}$, we can see that each operation will swap two adjacent elements from one of the sets.

Since we can sort both subsets independently, the "most sorted" permutation can be obtained by merging the sorted subsets.

The cleanest implementation would be to check if $(v[i] \% 2) = (i \% 2)$, for all $1 \leq i \leq n$. If this condition is true, then we can sort the permutation. Otherwise, the permutation cannot be sorted.

Time complexity per testcase: $O(N)$

K. Distinctness Queries

Firstly, if $n > m$, we will swap n and m and transform the matrix into its transpose.

Since $n \leq m$ and $n \cdot m \leq 10^5$, we know that $n \leq \sqrt{10^5}$, which allows us to solve the problem in $O(nm \cdot n)$.

Let $ans[(i_1, i_2, j_1)]$ be the smallest j_2 such that the submatrix $(i_1, j_1) \rightarrow (i_2, j_2)$ contains at last two equal elements. If all integers inside the submatrix $(i_1, j_1) \rightarrow (i_2, m)$ are distinct, then $ans[(i_1, i_2, j_1)]$ will be equal to $m + 1$.

For a particular query consisting of i_1, i_2, j_1, j_2 we only need to check whether $j_2 < ans[(i_1, i_2, j_1)]$.

We will compute the values of ans firstly by iterating through every i_1 from 1 to n .

For each value x , $1 \leq x \leq n \cdot m$, we will construct a linked list containing the cells (i, j) such that $a[i][j] = x$ and $i \geq i_1$, **in lexicographical order**.

The linked list can be constructed by first iterating first through j and then through i_2 .

Let $nxt[(i, j)]$ be the smallest column $j_2 \geq j$ which contains an element equal to $a[i][j]$. Similarly, Let $prv[(i, j)]$ be the largest column $j_2 \leq j$ which contains an element equal to $a[i][j]$.

Note that nxt and prv **mustn't** consider $a[i][j]$ as being equal to itself. Otherwise, $nxt[(i, j)]$ and $prv[(i, j)]$ would both be always equal to j , which isn't very useful.

After constructing the linked list, we will traverse the matrix again, first through i_2 **in reverse**, and then through j , removing one line from the matrix at a time from the linked lists.

For each cell $a[i][j]$ we will find that:

1. $prv[(i, j)] = linkedlist_prv[(i, j)].j$
2. $nxt[(i, j)] = linkedlist_nxt[(i, j)].j$

If $prv[(i, j)] = j$ or $nxt[(i, j)] = j$, then both of them must be made equal to j . Now, we'll traverse the matrix one more time, first through i_2 and then through j .

For each cell $a[i][j]$, the following things will happen:

- $ans[(i1, i, j)] = \min(ans[(i1, i, j)], nxt[(i, j)])$.
- $ans[(i1, i, prv[(i, j)])] = \min(ans[(i1, i, prv[(i, j)])], j)$.

Additionally, since $ans[(i1, i, j)] \leq ans[(i1, i, j + 1)]$, we will have to perform:

$$ans[(i1, i, j)] = \min(ans[(i1, i, j)], ans[(i1, i, j + 1)]), \text{ for all } j \text{ in reverse.}$$

Time complexity: $O(nm \cdot \min(n, m) + q)$

L. Best or Worst

Let's consider an unusual permutation a :

First of all, let $l_i = \min(a_1, a_2, \dots, a_i)$ and $r_i = \max(a_1, a_2, \dots, a_i)$. We can see that $[a_1, a_2, \dots, a_i]$ are a permutation of $[l_i, l_i + 1, l_i + 2, \dots, r_i]$.

Proof by contradiction:

Suppose that there is an integer x from the interval $[l_i, r_i]$ such that x doesn't appear in the first i elements of a . Therefore, there is some $j > i$ for which $a_j = x$.

Since $a_i = l_i$ or $a_i = r_i$, we know that $l_i < x < r_i$. Therefore, $l_i < a_j < r_i$, which means that a_j is neither a prefix minimum nor a maximum, which contradicts the fact that a is an unusual permutation.

Therefore, for each position i where $p_i \neq 0$, there are only two possible intervals which can contain a_1, a_2, \dots, a_i :

1. $[p_i, p_i + (i - 1)]$
2. $[p_i - (i - 1), p_i]$

This leads us to the following dp states:

1. $dp[i][0]$ ($p_i \neq 0$) - the number of ways to fill the first i elements such that p_i is a prefix maximum.
2. $dp[i][1]$ ($p_i \neq 0$) - the number of ways to fill the first i elements such that p_i is a prefix minimum.

Each dp state corresponds to a certain interval, which is either $[p_i, p_i + (i - 1)]$ or $[p_i - (i - 1), p_i]$.

The number of ways to transition from an interval $[l_i, r_i]$ to the next interval $[l_j, r_j]$ is equal to:

$$f(l_i, r_i, l_j, r_j) = \begin{cases} 0, & \text{if } l_i < l_j \text{ or } r_i > r_j \\ C_{(l_i - l_j) + C(r_j - r_i)}^{r_j - r_i}, & \text{otherwise} \end{cases}$$

The number of ways to fill a prefix containing i zeroes is equal to $2^{\max(0, i-1)}$, since we can fill it in reverse.

The number of ways to fill the suffix starting from the last i such that $p_i \neq 0$ is equal to $f(l_i, r_i, 1, n)$.

Time complexity: $O(N \log)$ or $O(N)$, depending on the implementation

[Feedback link](#)