CodeForces Round 1021 Div. 2F / Div. 1D

omeganot

July 18, 2025

To be honest, I spent a nontrivial amount of time not realizing the operation was simply an XOR and searching for incorrect invariants. With the knowledge that you are simply performing XOR operations, an obvious invariant ends up being the basis of each "leaf" node of the tree. Here, the leaf nodes correspond to the substrings that can't be halved any further. The non-linear algebra perspective is that the set of numbers S where $x \in S$ if x is the XOR of some subset of our leaf nodes doesn't change from operation to operation.

A proof without linear algebra is relatively simple. Let a and S be the values of the leaf nodes and the set of all subset XORs, and a' and S' be the result after performing one operation. a and a' will be the same except a will have x and y while a' has $x \oplus y$ and y.

If a number v is in S, then it is also in S' since we can use the same subset, except if we need x but not y from a, we take both $x \oplus y$ and y in a'. If we need $x \oplus y$, we would take x and y from a and just $x \oplus y$ from a'. This shows that S is a subset of S', and we can use the same logic to show the reverse. Thus, $S = S'$.

The question then becomes whether identical bases is sufficient, because we've just shown it's necessary. This would mean you could take any a and turn it into any a' with the operations so long as the basis is the same. I was motivated to check if swapping was possible, since first of all, if identical bases is sufficient, then it has to be possible as swapping doesnt change the basis, but also, it felt like swapping would make the rest of the proof easy too.

If we have i, j that share the same parent swapping a_i and a_j would be easy, since we could do:

$$a, b$$

$$a, a \oplus b$$

$$b, a \oplus b$$

$$b, a$$

However, it is not as easily clear that we could turn i, j, k, l (where i, j share a parent, k, l share a parent, and the parent of i, j and the parent of k, l share a parent) into k, j, i, l. I remember trying this problem in China in my grandmother's apartment and coming across a solution, but when I came back to the states I couldn't replicate it. So I had to write some brute force code which came up with:

$$a, b, c, d$$

$$a \oplus c, b \oplus d, c, d$$

$$a \oplus c, b \oplus d, c, c \oplus d$$

$$a \oplus c, b \oplus d, a, b \oplus c$$

$$a \oplus b \oplus c \oplus d, b \oplus d, a, b \oplus c$$

$$a \oplus b \oplus c \oplus d, b \oplus d, b \oplus c \oplus d, c \oplus d$$

$$a \oplus b \oplus c \oplus d, b \oplus d, b \oplus c \oplus d, b$$

$$a \oplus b \oplus c \oplus d, b \oplus d, a, d$$

$$a \oplus c, b \oplus d, a, d$$

$$c, b, a, d$$

Generalizing this into swapping any two elements is now not so hard. Consider the tree of nodes that correspond to the operations. Say we wanted to swap a_i and a_j. We start at the root of the tree. There are two cases, either a_i and a_j are in the same child, or a different child. If they are in the same child we descend to that child and recurse. If not, we perform the above maneuver and bring them under the same parent.

We continue to do this until both a_i and a_j end up as leaves under the same parent. Now, we can simply swap the two of them and if we undo all our previous operations, we will end up with a_i and a_j swapped.

With a powerful swap operation, we are able to show that we can transform a into any a' if they have the same basis. The swap operation allows us to conclude that another simple operation is doable: setting $a_i := a_i \oplus a_j$. We can swap a_j with the leaf sibling of a_i and perform our operation, then swap back. Now, how do we turn a into a'? I found it easiest to show we can turn both a and a' into the same array, as operations are reversible.

The array I chose to transform them into was of the form:

$$b_1, b_2, \ldots, b_k, 0, 0, \ldots, 0$$

Essentially, an arbitrary basis of our numbers followed by zeros. To construct the desired array from some array a with basis b of size k, for each i from 1 to k, we first construct b_i using our universal XOR operation. This is guaranteed to be possible because b is a basis for a. We then swap the number into its corresponding place. Now, the remaining numbers are all subsets of our basis, so we can cancel them out with our XOR operation, ending up with the desired array.

Thus, we have shown that we can transform s into t if the corresponding arrays a of the leaf substrings have the same basis. To check if the basis is the same, I checked if the sizes of the basis of s equaled the size of the basis of t and the size of the basis of $s + t$. Since the basis of $s + t$ is a superset of both s and t, if all three sizes are the same then the basis of s must match the basis of t.

Computing the size of the basis was a bit more difficult because the numbers didn't necessarily fit into integers, meaning we'd have to deal with strings. XORing two strings of length k takes $O(k)$. Furthermore, the size of the basis can't exceed $\min(k, \frac{n}{k})$, since there are only k dimensions and $\mathrm{O}(\frac{n}{k})$ numbers to consider. Lastly, we must do this for every $O(\frac{n}{k})$ number. As such, our total Time Complexity ends up as $O(k \cdot \min(k, \frac{n}{k}) \cdot \frac{n}{k})$, which equals $O(n\sqrt{n})$, since $\min(k, \frac{n}{k}) \leq \sqrt{n}$. This was a bit too close for comfort with $n \leq 10^6$ in my opinion, so I used bitsets to optimize the time Complexity to $O\left(\frac{n\sqrt{n}}{32}\right)$.

I used my good friend Keys' method for "dynamically" sized bitsets alongside the standard method of computing an XOR Basis: You iterate over each number and for each number a_i, look at the current elements in basis b in order. If at any time, $a_i \oplus b_j < a_i$, you set $a_i := a_i \oplus b_j$ and continue. If a_i is nonzero at the end of this process you add it to the basis. I am too lazy to include the proof here. To check if $a_i \oplus b_j < a_i$, I maintained the most significant bit of each element of the basis, as the previous inequality depends only on the msb.