Kick Start 2019 - Round D

Analysis: X or What?

Test set 1 (Visible)

Let's define a new array S. We set $S_0 = 0$, $S_1 = \mathbf{A}_1$ and $S_i = S_{i-1}$ xor $\mathbf{A}_i$ for i = 2 to **N** (Note that S is zero-indexed while **A** is one-indexed). We can see that once we've calculated this, $\mathbf{A}_l$ xor $\mathbf{A}_{l+1}$ ... xor $\mathbf{A}_r$ is simply given by S_r xor S_{l-1}.

With this, Test set 1 can be solved just by calculating the xor sum of every sub-interval of **A** and checking if it's *xor-even*. After each update, we need to recompute S which only takes O(**N**) time. So each query can be handled in $O(\mathbf{N}^2)$ time with an overall complexity of $O(\mathbf{QN}^2)$.

Test set 2 (Hidden)

Let's extend the definition of *xor-even* to mean any number having even number of 1s in it's binary representation, similarly for *xor-odd*. Now, notice that if we xor two *xor-even* numbers or two *xor-odd* numbers (numbers having an odd number of 1s in their binary representations), we get a *xor-even* number and, similarly, if we xor a *xor-even* number with a *xor-odd* number, we get a *xor-odd* number. Hence, if there are a even number of *xor-odd* numbers in an interval then that interval is going to be *xor-even* and vice versa.

This means that if there are even number of *xor-odd* numbers in our array, the whole array is *xor-even*. Otherwise, we consider the subarray starting just after the first *xor-odd* number and going till the end and the subarray starting from the first element in our array and ending just before the last *xor-odd* number. Both are *xor-even* intervals and the larger of them should be the largest *xor-even* interval in our array.

We can do this by keeping a set of all positions of *xor-odd* numbers. Every time we update a number, we simply do an insertion or a deletion or leave the set unchanged. If the size of the set is even, then the whole array is *xor-even*, otherwise we get the left most and the right most positions from this set and output the answer as discussed above.

Since we will have an O(**N**) elements in the set and there will be **Q** queries, each of O(log(**N**)) time, this solution has a complexity of O((**N**+**Q**)log(**N**)).

We can also solve this problem using [van Emde Boas trees] in O(**Q**log(log (**N**))). Or we can use offline algorithms as well to achieve even better asymptotic solutions. Figuring out the details of these approaches is left as exercises to the reader.