# Codeforces 1771F optimization

Yoshi_likes_e4

July 2025

## 1 Introduction

First, let us give an overview on what this problem is.

We are given an array $A$ of length $n \leq 200000$, each containing elements from $[0, n-1]$, and there are $q \leq 200000$ queries, each containing two numbers $l, r$. We need to find the minimum element that appears an **odd** number of times in the range $[l, r]$ or report that it does not exist. The problem is interactive, i.e. you can only get the next query after answering the last one.

## 2 Step A: Precursor

This is not the solution (yet), but we will leave it here.

Consider an array $B[i][j]$ that is the value of $[A_i = j]$.

We want to find the minimum j such that the sum of $B[l..r][j]$ is odd.

We can use a prefix sum array here: We take the cumulative sum (in the $i$-dimension) of $B[i][j]$ for each j. This allows us to check for the sum faster.

However, storing cumulative sums is expensive. We just want to check for the parity. How can we store parity only?

If we consider $B[i]$ as a binary vector, then if we want to get the parity of the sum, we just $XOR$ them together.

We can finally represent the cumulative $XOR$ of $B[i]$ as an additional array $C[i]$. Then, the answer to the query is just the position of the LSB in $C[r]\ XOR\ C[l-1]$. (Assume that the $C$ array represents the full prefix interval and $C[-1] = 0$).

We can now represent $C$ as an array of bitsets and construct $C$ from $B$ simply by $XOR$-ing.

The XOR and LSB finding operation can be done in $O(n/w)$. This gives us a $O(nq/w)$ time complexity. Yay!!!

## 3 Step B: Initial "Solution"

We consider the obvious bottleneck: The memory consumption of the $C$ array. The $C$ array takes $n^2$ bits to store, or a whopping 5 GB, far exceeding the 256 MB limit.

We can take an idea inspired by sqrt decomposition: Store only spaced values of $C[i]$, each representing a block of the array. Suppose that the block size is now $BLOCK$. Let $C'[i]$ be the cumulative XOR up to the $i$-th block (0-indexed).

Note: the / operator here represents floor division. (from now on)

Then, when answering a query, we can take $C'[l/BLOCK]\ XOR\ C'[r/BLOCK-1]$. Then, we flip the bits in the 2 sides of the interval (everything excluding the range $[(l/BLOCK+1)*BLOCK, r/BLOCK*BLOCK))$. We find the LSB of the resulting bitset normally.

Choosing an optimal block size for time complexity gives us the time complexity of $O(nq/w+(n+q)\sqrt{n/w})$ with space complexity of $O(n\sqrt{nw}+q)$.

The method described above will not pass under normal conditions. There's a lot of excessive copying, and this will be eliminated in Part C.

## 4 Step C: Initial Solution

We devise a few optimizations that can be done to the method above.

First, reflipping is better than copying (Consider the number of operations: $n/w$ compared to $\sqrt{n/w}$).

We can just directly flip over $C'[r/BLOCK-1]$ and flip it back after the LSB finding phase. This avoids an expensive copy of $C'[r/BLOCK-1]$.

We can write our own "Find LSB" function that takes 2 bitsets (as reference) then find the LSB while directly XOR-ing them. This eliminates the need of copying for both $C'[r/BLOCK-1]$ and $C'[l/BLOCK]$.

Note: To facilitate the "Find LSB" operation, we need to find a way to access the inner elements of the bitset. Such a way is possible if we either cast the bitset reference to an array reference (not recommended) or write our own bitset.

Writing your own bitset (for this problem) is rather easy. A few optimizations are listed here:

We use `__builtin_ctzll` for finding LSB of a single integer.

For maximum performance, remember to add alignment to the inner bitset data.

The time and space complexity won't change, but your code should be $3-4$ times faster.

## 5 Step D: Xor Hashing

Let's bring our most valuable friend: XOR hashes.

We assign to each number from $[0,n)$ a big 64-bit random number. Store this array as $R[x]$.

The XOR of a sequence of integers is the XOR of the odd-frequent numbers.

Then, if we take the XOR of $R[A[i]]$ for all $A[i]$ in the range $[0,v]$ and $i$ in the range $[l,r]$, we get the XOR hash of all the odd-frequent numbers $\leq v$ in the subarray $A[l..r]$.

Then, we just need to find the minimum $v$ for this hash to be non-zero (We consider the collision probability to be negligible). Easy, right?

## 6 Step E: Square Root Decomposition

Now, XOR hashing on its own doesn't provide a time complexity improvement. But, we can combine it with Sqrt decomposition to completely eliminate the $O(nq/w)$ part.

First, we decompose the bitset to $\sqrt{n/w}$ parts. Each part now contains $\sqrt{nw}$ bits.

Let $VLB = \sqrt{nw}$.

We construct an additional array $XORHASH[i][j]$ of dimensions $[n, \sqrt{n/w}]$. This will keep the 2-dimensional cumulative XOR hash of all the numbers in the subarray $A[0..i]$ up to $(j+1) \cdot VLB - 1$.

We can easily construct this array in $O(n\sqrt{n/w})$ time.

Then, if we wanted to check whether $v$ (see Part D) is in the range $[0; j * VLB)$, we can just check if $XORHASH[r][j-1] \; XOR \; XORHASH[l-1][j-1]$ is zero or not.

After constraining $v$ to a block of size $VLB$, we can limit the "Find LSB" operation to only work with $\sqrt{n/w}$ numbers. We still do the re-flipping phase like before.

The space complexity is not changed. The time complexity is now $O((n+q)\sqrt{n/w})$.

## 7 Extra talk

1. How can we eliminate reflipping?

Create a new *tmp* bitset that have only elements of 2 sides that were inside $v$'s block. Then, we wouldn't need to reflip. Instead, we can erase the block of size $VLB$ in the *tmp* bitset.

2. Are we `FindLSB` bottlenecked?

In the sqrt decomp solution, probably no. In the bitset solution, a thousand time yes.

3. What if, the problem wasn't actually interactive?

We can use Mo's algorithm here. It provides a simple $O(n\sqrt{q} + q\sqrt{n/w})$ solution that is memory-optimal, i.e. $O(n+q)$.