

# MITIT 2026 Spring Invitational Editorials: Qualifier Round 1

The MITIT Organizers

April 6, 2025

## A Tart Splitting

*Problem Idea: Yifan Kang*

*Problem Preparation: Yifan Kang*

*Analysis By: Yifan Kang*

Notice that the operation can be seen as rotating the array by  $k$ . It suffices to check if any rotation of the original string  $S$  satisfies the constraint. We can check this in  $O(N)$  by rotating  $M$  to  $S[1]$  and check if the string matches.

## B Orange Pit

*Problem Idea: Ray Law*

*Problem Preparation: Ray Law*

*Analysis By: Ray Law*

Let  $S$  be the multiset of the labels of the oranges, and let  $b_i$  be the  $i$ th smallest integer in  $S$ . Consider a graph on  $n$  nodes, numbered 1 to  $n$ . For each  $i$  and  $j$ , there is an edge between nodes  $i$  and  $j$  with weight  $|b_i - b_j|$ . The problem now reduces to finding the maximum spanning tree of this graph.

Imagine running Kruskal's on this graph. Kruskal's will first connect 1 and  $n$ . Then at every step, Kruskal will either connect the smallest unconnected vertex to  $n$  or the largest unconnected vertex to 1. In the resulting MST, 1 is connected to  $n$ , and every vertex in between is connected to the farther of 1 and  $n$ . So if  $f(b_i) = \max(b_i - b_1, b_n - b_i)$ , the answer is  $b_n - b_1 + \sum_{i=2}^{n-1} f(b_i)$ .

Time Complexity:  $O(N)$

## C Carrot Party

*Problem Idea: Ditbul Ban*

*Problem Preparation: Ditbul Ban*

*Analysis By: Ditbul Ban*

In this editorial, we restate the problem as a game where the players repeatedly choose a number from a sequence and remove it. Also, let us abbreviate Busy Beaver and Sneaky Snake as  $A$  and  $B$ , respectively.

### Subtask 2

We should first observe two important properties.

- Until the very end of the game, the choices of the two players are independent:  $A$ 's choices do not affect which options  $B$  will face, and vice versa.
- For both players, it is optimal to choose the larger number whenever they have a choice.

The first property is straightforward. Until only two numbers remain,  $A$  only interacts with the left part of the sequence, while  $B$  only interacts with the right part. Therefore, neither player's decisions can influence the options available to the other player.

Now consider what happens before the final two numbers remain. By that point,  $A$  has taken

$$P := \left\lfloor \frac{N-1}{2} \right\rfloor$$

turns and has seen  $P+1$  numbers in total. Similarly,  $B$  has taken

$$Q := \left\lfloor \frac{N-2}{2} \right\rfloor$$

turns and has seen  $Q+1$  numbers in total.

So what are the two numbers that remain at that moment? One is the single number that  $A$  did not take among the  $P+1$  numbers he encountered, and the other is the single number that  $B$  did not take among the  $Q+1$  numbers she encountered.

At this point, the optimal strategy becomes clear: each player should leave the smallest number among all the numbers they have seen. Leaving any larger number would only give the opponent a better opportunity at the end. Therefore, choosing the larger available number on every turn guarantees that the remaining number on that side is the smallest one the player has seen.

This greedy strategy can be implemented in  $O(N)$  time per test case.

### Subtask 3

From the observation above, we can compute the final scores as follows.

- Consider the left part of the sequence seen by  $A$ . Find its smallest element, and add the sum of all the other elements in that part to  $A$ 's score.
- Consider the right part of the sequence seen by  $B$ . Find its smallest element, and add the sum of all the other elements in that part to  $B$ 's score.
- Two numbers remain. Add the larger one to the player whose turn comes next when only two numbers remain ( $A$  if  $N$  is even,  $B$  if  $N$  is odd), and add the smaller one to the other player.

To support updates, we need to maintain the following information for the left and right parts of the sequence:

- the sum of the elements,
- the minimum element.

This can be done with data structures such as `std::multiset` together with running sums, or with a segment tree. Then, for each side, the contribution excluding the minimum element is simply

$$(\text{sum of the side}) - (\text{minimum of the side}).$$

Since we also know the two remaining numbers and which player moves first when they remain, we can compute the final scores directly. Each update can be processed in  $O(\log N)$  time, since it affects only one side and requires only a constant number of data-structure operations.

Therefore, the problem is solvable in  $O((N + Q) \log N)$  time per test case.

## D Binary Beaver

*Problem Idea: Yifan Kang*

*Problem Preparation: Brian Xue, Yifan Kang*

*Analysis By: Brian Xue, Yifan Kang*

### Subtask 1

This subtask satisfies a specific property: the optimal  $x$  will always be some  $a_i \oplus 2^{k-1}$ , for some  $1 \leq i \leq n$ . Intuitively, the optimal  $x$  should share as many trailing bits with  $a_i$  as possible. A value of  $x$  that does not satisfy the above property can always be improved by changing the most significant bits to match any  $a_i$  as close as possible.

One possible solution is to just enumerate every possible value of  $i$  and solve each query in  $O(N^2)$  time, which can pass with a good implementation. You can also store the answer for  $x_i = a_i \oplus 2^{k-1}$  for all  $1 \leq i \leq n$ . Note the expression

$$ans(x_i) = \sum_{1 \leq j \leq n} v_2(x_i \oplus a_j)$$

You only need to update contribution for  $(x_i, a_j)$  pairs that are changed to solve each query in  $O(N)$  time.

### Subtask 2

A very natural representation of the members of  $a_i$  that is friendly to us is a trie. Consider the trie created by inserting all  $a_i$ , keyed using the binary representation of each  $a_i$  starting from the least significant bit. The problem asks us to find a non-existent node on the lowest level of the trie that shares the maximum sum of the longest common prefix with all existing nodes on the trie.

It naturally follows to use DP — we can consider a DP on the nodes of the trie that calculates the maximum possible answer if we only consider the nodes in the subtree and the trailing bits that exist in the subtree. More formally, let  $sub_{k,p}$ , for  $0 \leq k \leq K$  and  $0 \leq p < 2^k$ , represent the set of elements  $\{\text{highBit}(a_i, k) \mid 1 \leq i \leq n, \text{lowBit}(a_i, K - k) = p\}$  where  $\text{highBit}(a_i, k)$  selects the most significant bits of length  $k$  and  $\text{lowBit}(a_i, K - k)$  selects the least significant bits of length  $K - k$  — note that we entirely prune the leading bits shared by every node in the subtree. In addition, under this formulation, the two "children" of node  $(k, p)$  are  $(k + 1, 2p)$  and  $(k + 1, 2p + 1)$ . Let  $dp_{k,p}$ , under the same conditions for  $k$  and  $p$ , represent the maximum answer given only the set of elements  $sub_{k,p}$ . The  $\text{highBit}$  trims all the values to only the most significant bits of the binary representation / suffix of the key. If no answer exists because no valid non-colliding value of  $x$  exists, then let the answer be  $-\infty$ .

Now we only need to find the transition. As we can guess with the trie structure, the DP value of a node only depends on its two children, i.e.  $dp_{k,p} = f(dp_{k+1,2p}, dp_{k+1,2p+1})$ . There are two cases: the ideal value of  $x$  lies within the left child's subtree, or the ideal value of  $x$  lies within the right child's subtree. If the former, the new dp value is computed by  $dp_{k,p} = dp_{k+1,2p} + |sub_{k+1,2p}|$ . We take the contribution within the left subtree, and then add the new contribution from the  $k+1$ th bit that was not counted before. The right side is symmetric, and this gives us the relation:

$$dp_{k,p} = \max(dp_{k+1,2p} + |sub_{k+1,2p}|, dp_{k+1,2p+1} + |sub_{k+1,2p+1}|)$$

For each query, we can simply build the entire trie and run our DP on it. You will have to compute the size of subtrees using a simple DP as well, and empty subtrees satisfy  $dp_{k,p} = 0$ . Take care to consider what happens when an entire subtree is filled and no valid value of  $x$  exists in it, as the  $dp$  value of such a subtree must be equal to  $-\infty$ . In total, this solution takes  $O(NK)$  time per query (size of trie) and  $O(NQK)$  time total.

### Subtask 3

Notice that each update on the trie only changes value on a path of  $O(K)$  nodes from the root. We can update the DP values on this path in a bottom-up manner in  $O(K)$  time. With this, we can solve the problem in  $O((N + Q)K)$ .

## E Snake

*Problem Idea: Ray Law*

*Problem Preparation: Andrii Smutchak*

*Analysis By: Andrii Smutchak*

### Subtask 1

We can consider the following dp: Let  $dp[t][i]$  be the maximum score at time  $t$ , if the last apple was eaten at time  $t$  and the length of the snake becomes  $i$ .

To define transitions, let us define the following function: Let  $c(t)$  be the cell occupied by the head at time  $t$ . Let  $f(t)$  be the maximum  $t' < t$ , such that  $c(t') = c(t)$ .

Then  $dp[t][i] = \max dp[k][i-1] + (i-1) \cdot (t-k) + 1$  for all  $f(t) + i - 1 \leq k < t$ . The lower bound guarantees that the apple can be spawned at this cell.

The final answer can be computed as  $\max dp[t][i] + i \cdot (N-t)$  over all valid pairs  $(t, i)$ .

The time complexity is  $O(N^3)$ .

### Subtask 2

We can notice that  $dp[t][i] = \max dp[k][i-1] + (i-1) \cdot (t-k) + 1 = t \cdot (i-1) + 1 + \max dp[k][i-1] - k \cdot (i-1)$ .

To do transition from  $dp[\cdot][i-1]$  to  $dp[\cdot][i]$ , let us find  $dp[k][i-1] - k \cdot (i-1)$  for every  $k$ . Then we can use data structure such as segment tree or sparse table to find RMQ over  $dp[k][i-1] - k \cdot (i-1)$  in  $O(\log N)$ .

This reduces the total time complexity to  $O(N^2 \log N)$ .

### Note

Although not required, a linear-time RMQ data structure (e.g. <https://codeforces.com/blog/entry/78931>) speeds up the solution to Subtask 2 to  $O(N^2)$ .