Editorial Stelele Informaticii 2025

November 6, 2025

Jimbo

First, we observe that all card types are actually special cases of type 4, namely $C = C + x$ and $M = (M + y) \cdot z$.

Let's assume we have already determined the 5 cards we will use. How can we quickly determine in what order to play them? A useful method in such problems is to compare 2 cards (x_1, y_1, z_1) and (x_2, y_2, z_2) and hope we obtain a transitive comparator. For C the order in which we play the cards doesn't matter. Let's see when the order $1 \to 2$ is better than $2 \to 1$ for M:

$$((M + y_1) \cdot z_1 + y_2) \cdot z_2 > ((M + y_2) \cdot z_2 + y_1) \cdot z_1$$
$$Mz_1z_2 + y_1z_1z_2 + y_2z_2 > Mz_1z_2 + y_2z_1z_2 + y_1z_1$$
$$y_1z_1z_2 + y_2z_2 > y_2z_1z_2 + y_1z_1$$
$$z_1y_1(z_2 - 1) > z_2y_2(z_1 - 1)$$
$$\frac{z_1y_1}{z_1 - 1} > \frac{z_2y_2}{z_2 - 1}$$

We have some special cases when $z_1 = 1$ or $z_2 = 1$. You can verify that in these cases, the card with $z = 1$ should be played before the other card. Otherwise, it is optimal to order the cards in descending order by $\frac{zy}{z-1}$. From now on we will consider that we have sorted the N cards by this criterion.

For subtasks 4 and 5, there were simpler orderings to find.

Subtask 3

For this subtask the value of C will always be $5x_1$ regardless of what cards we choose, so we only need to maximize M. We can define $dp[i][j]$ as the maximum possible M if we consider the first i cards and play j of them ($j \leq 5$). Time complexity is $O(N \cdot 5)$

Subtask 6

We saw in the previous subtask that it was very useful that C was fixed, because we had to optimize a single value. Since $x_i \leq 1\,000$ and we will play at most 5 cards, we observe that C is at most $MAXC = 5\,000$. We can define $dp[i][j][k]$ as the maximum possible M if we consider the first i cards, play j of them and $C = k$. The transitions are similar to a knapsack-type DP. Time complexity is $O(N \cdot 5 \cdot MAXC)$.

Negotiable

Subtask 1

For each x we can generate all 2^{N-1} partitions and check by brute force if the partition is x-malleable. Time complexity: $O(2^{N-1} * N^3)$

Subtask 3-4

Let's say we want to calculate the answer for a number x. For a fixed l, it is clear that $MEX(A_l, A_{l+1}, \ldots, A_r) < MEX(A_l, A_{l+1}, \ldots, A_r, A_{r+1})$. We can think of a greedy strategy. Specifically, we take the elements in order from left to right and add them to the current sequence as long as it is x-negotiable. When adding an element would cause the MEX to increase above x, we start a new sequence. This partition is optimal.

For Subtask 3, $a_i \leq 10$, the answer for $x > 10$ is 1. We can directly run the algorithm above in $O(N)$ for each $x \leq 10$. Time complexity: $O(10 \cdot N)$

For Subtask 4, the answer for $x > 5000$ is 1 (because the maximum length of a sequence is $N = 5000$, the MEX cannot exceed 5000). Similarly we can run the algorithm above in $O(N)$ for each $x \leq N$. Time complexity: $O(N^2)$

Subtask 6

We can generalize the idea from Subtask 4. Let $ans(x)$ denote the answer for x. We observe that, when x increases, the size of the minimum partition decreases. For this reason we try to think of an algorithm for a fixed x that would work in $O(ans(x))$.

Let $next_w(i) = \min_{l \leq r \leq n, a_r = x} r$.

We optimize the naive greedy algorithm using data structures. For a fixed x and l, we need to find the maximum r such that $MEX(A_l, A_{l+1}, \ldots, A_r) \leq x$. We denote this r by $f_x(l) = \max_{0 \leq y \leq x} next_y(l)$. We observe that $f_x(l) = \max(f_{x-1}(l), next_x(l))$. This leads us to the following solution: we maintain a segment tree over the function f. When we move from $x-1$ to x, assuming that x appears in the array at positions $p_1, p_2, \ldots p_k$, we need to perform the following operations:

- $f_i = \max(f_i, p_1), 1 \leq i \leq p_1$
- $f_i = \max(f_i, p_2), p_1 < i \leq p_2$
- $\ldots$
- $f_i = \max(f_i, p_k), p_{k-1} < i \leq p_k$

Once we maintain this function we can go by brute force through $1, next_x(1), next_x(next_x(1)), \ldots$.

Let $S = \sum_{x=1}^{N} ans(x)$. If we use the method above, the time complexity will be $O(S \log N)$. Now we need to find a convenient bound for S. A first attempt would be to observe that for an x, we need at least x elements in a sequence

to have the MEX greater than x, so $ans(x) \leq \frac{N}{x}$, which gives us $S \leq N \log N$. Thus we already have a solution in $O(N \log^2 N)$.

In fact, it can be proven that $S \leq N$. Let f_i denote the frequency of element i in the array. For an x, we have $ans(x) \leq \min_{i=0}^{x-1} f_i$, because each element from 0 to $x-1$ must appear at least $ans(x)$ times. So S is the sum of prefix minimums of f. In the worst case, f will be a descending array, in which case $S = \sum_{i=0}^{N} f_i = N$. Time complexity: $O(N \log N)$.

Permutations

Solution 1

Subtask 1

If there exists a position i such that $A_i = N$ and $B_i = N$ for all permutations C the winning card will be card i. Since in the subtask such a position exists (position N), the answer will be 0 for cards $1, 2, \ldots N-1$ and $N!$ for card N. Time complexity: $O(N)$.

Subtask 2

We can try all $N!$ permutations and run the algorithm for each. Time complexity: $O(N! \cdot N)$.

Subtask 3

We define the following dynamic programming:

$dp_{mask,winner}$ = the number of ways to order elements from mask such that if we run the algorithm for this subset of positions, the winner is $winner$. Recurrence:

$dp_{mask \cup winner2,winner2} = \sum_{mask \cap winner2 = \varnothing} dp_{mask,winner}$

where $A_{winner2} > A_{winner}$ and $B_{winner2} > B_{winner}$.

Final time complexity: $O(2^{N-2} \cdot N^2)$

Subtasks 4-7

Initially we calculate f_i = how many j's dominate i. We say that j dominates i if $A_j > A_i$ and $B_j > B_i$.

Let's assume we want to calculate the answer for a certain i. If we calculate the probability p of randomly generating a good permutation (which has $winner = i$ at the end of the algorithm) and multiply it by $N!$ we get the desired answer, because $p = \frac{p_{good}}{p_{total}}$ where $p_{total} = N!$.

Thus we can derive the following dynamic programming: dp_i = the probability of generating a permutation such that $winner = i$. The recurrence is simple:

$dp_i = \sum_{i \ dominates \ j} dp_j \cdot \frac{1}{f_j}$. We can implement this recurrence in $O(N^2)$. This complexity is sufficient to pass subtasks 4 and 6.

This formula is correct because after $winner = j$ the probability that the next value of the variable $winner$ is i is $\frac{1}{f_j}$ because any of the f_j elements that dominate j can follow equiprobably.

If we sort the elements in increasing order by A_i the recurrence above can be optimized using a BIT (Binary Indexed Tree). Final time complexity $O(N \log N)$.

Solution 2

Initially we calculate d_i = how many j's dominate i. We say that j dominates i if $A_j > A_i$ and $B_j > B_i$.

We will consider the permutation as having n empty positions, which we will fill gradually. We observe that, if we want to have $pos = i$ at a certain point in the algorithm, we are not allowed to have placed anyone who dominates i yet. We define dp_i = the number of ways to place all numbers that do not dominate i in the array, such that there exists a moment when $pos = i$. This means that up to position i in the permutation we must have filled everything.

Now, the transitions: to calculate dp_i, we look at the previous j with $pos = j$; j must be dominated by i; we have already filled positions in the permutation with all elements that do not dominate j, therefore, to have $pos = j$ right before $pos = i$, among the uncompleted elements the first must be i, and the rest that dominate j but not i (thus $d_j - d_i - 1$ elements) we can place however we want on the remaining positions, so we have $A(d_j - 1, d_j - d_i - 1)$ ways to transition. Thus, we obtain a solution in $O(n^2)$.

We have two things to optimize: - calculating the array d - the transitions

To calculate d_i, we can sort the elements in descending order by x, traverse them and maintain any data structure (for example, a BIT) that supports "point update, range sum".

To optimize the transitions, we sort the elements in increasing order by x, for i we will transition only from already processed elements with $y_j < y_i$, and the transition will be $\frac{1}{d_i!} \cdot (d_j - 1)! \cdot dp_j$, so we can keep a BIT over the sum (modulo $10^9 + 7$) of $(d_j - 1)! \cdot dp_j$ at indices based on y_i, we only transition from a prefix sum (which we multiply by $\frac{1}{d_i!}$). Thus, we obtain a solution in $O(n \log n)$.

Athens

In the following we will use $MAXC = 1e6$, $MINR = 1e4$

First, the radius of the hidden circle, R, can be easily found using a single "covering" query, such as $(0, 0, 4 * MAXC)$.

Now we will take 2 points that are guaranteed not to be covered by the hidden circle (not strictly necessary, just eliminates some cases) and we will find the distance between them and the center of the hidden circle. In the official solution these are $(-2 * MAXC, 2 * MAXC)$, $(-2 * MAXC, -2 * MAXC)$. Thus, the center of the hidden circle will have two variants (since it is located at the intersection of two circles) and we can find which one is the true one using a single query.

Therefore, we have reduced the problem to finding the distance from a fixed point to the center of the hidden circle using as few questions as possible. The 100p solution manages to do this in $\log_2\left(\frac{MAXC}{MINR}\right)$ operations using the method below.

The procedure begins with a binary search in which we maintain the minimum and maximum possible distance (which respects previous answers) from the fixed point. If we receive an answer 0 it means we need to increase the distance and if we receive an answer πR^2 it means we need to decrease it.

The optimization comes from the fact that if we receive an intermediate answer (between 0 and πR^2) we can find the distance without sending additional questions!

Let's say the fixed point is (x_0, y_0) and that $query(x_0, y_0, R_{mid}) = A \in (0, \pi R^2)$. We want to find d such that the intersection between the disks (x_0, y_0, R_{mid}) and $(x_0 + d, y_0, R)$ equals A. When the area of the intersection above is less than A we decrease d and when the area is greater than A we increase d.

Using this method twice we obtain a solution that uses at most $2 + 2 \cdot \log_2\left(\frac{MAXC}{MINR}\right)$ questions.

Caterpillar 3

There exist solutions in $O(n^3), O(n^2 \log), O(n^2), O(n \log^2)$ which we will not present.

For solutions in $O(n), O(n \log n), O(n \log V)$ we can use the following idea: we will fix a position p, such that p is included in the subsequence and a_p is the AND of the subsequence. In case there are, within the same subsequence, multiple possible indices p, we will consider valid (or representative for the subsequence) the leftmost one. Now, we can try to count, for each p, in how many subsequences it is representative. In fact, this reduces to being able to calculate two arrays, L_p and R_p, such that $L_p \leq p \leq R_p$ and for any $L_p \leq i \leq R_p$, we have $a_i \ \& \ a_p = a_i$. In addition, for any $L_p \leq i \leq p-1$, $a_i \neq a_p$. If we have these two arrays calculated, the number of self-descriptive subsequences with p as representative is exactly $(p - L_p + 1) \cdot (R_p - p + 1)$.

The problem has been reduced to calculating these two arrays efficiently. We will focus on calculating the array R_p, because L_p is equivalent. The only difference is that we will need to do $L_p = \max(L_p, prev_p + 1)$ where $prev_p$ is the previous occurrence of a_p before p. The array $prev$ can be calculated in $O(n)$ with a hashmap, or better yet, we can use the observation that, for a set of numbers S, $and(S) \leq \min(S)$. That being said, if we only calculate with a stack the first element a_l to the left $\leq a_p$, we can just do $L_p = \max(L_p, l + 1)$. To calculate R_p, we can calculate for each bit the next position where it doesn't appear as we go with p from right to left and take the minimum from the bits of a_p, we can even keep a data structure (RMQ type or segment tree) for AND on an interval and see the first prefix from p that doesn't have AND a_p, but these solutions have a log factor and should obtain 81 points.

To obtain a solution in $O(n)$, we need to calculate R in $O(n)$. Let's imagine the following solution: we traverse the array from left to right, at step i calculating all R_p with $R_p \leq i$. Let $1 \leq x_1 \leq x_2 \leq \ldots \leq x_k \leq i$ be the positions to the left for which we don't have R calculated. Then, $a_{x_j} \ \& \ a_{x_{j+1}} = a_{x_j}$. Thus, if an element appears to the right, a suffix of these elements will not be a "binary submask" of the element to the right, thus finding their R. After that, an element is added to the right to our array x. Therefore, we can maintain these elements in the form of a stack, obtaining an amortized complexity of $O(n)$.

Impostors

Subtask 1

For this subtask, a brute force solution in $O(n! * n * k)$ is sufficient.

Subtask 2

This subtask is designed to admit solutions in exponential complexities better than $O(n!)$, up to $O(3^n * n)$.

Subtask 3

This subtask is crucial for understanding the problem. Let p be an initial permutation, q the final permutation (defined in the statement). We can look at a cycle of length l in q, $a_1, a_2, \ldots, a_l$. We observe that $p_{p_{a_i}} = a_{i+1}$, where we considered $a_{l+1} = a_1$. Let $b_i = p_{a_i}$, for each i. Then, $p_{b_i} = a_{i+1}$, so $q_{b_i} = b_{i+1}$. Then, b is also a cycle of length l in q. We have two cases: b is a rotation of a or not.

Case 1: b is a rotation of a. There is a single way to rotate a if l is odd, and 0 ways otherwise. A more general proof of this fact exists later in the editorial.

Case 2: b is another cycle. Then, we can choose any rotation of it (basically do $p_{a_1} = b_i$ for any i, so we have l variants).

Therefore, let's count the solutions: for each cycle length l independently, let there be f cycles of this length. We will take some single cycles, and pair the rest two by two. We can take single cycles only if l is odd. Let $k =$ the number of pairs (which must satisfy $2 \cdot k = f$ if l is even). Then, we have $C(f, 2 \cdot k)$ ways to choose the elements in pairs, $1 \cdot 3 \cdot 5 \ldots (2 \cdot k - 1) = \frac{(2k!)}{k! \cdot 2^k}$ ways to pair them, l^k ways to choose the relative rotations, so in total $C(f, 2 \cdot k) \cdot \frac{(2k!)}{k! \cdot 2^k} \cdot l^k$ ways for this k. We sum for each k, and multiply these sums for all cycle lengths (which work independently), and we have the answer. The complexity is bounded by the sum of the number of cycles of length l for all l, that is $O(n)$.

Subtasks 4-6

We need to generalize the solution from $k = 2$. For simplicity, we consider the edges $i - p_i$ as directed. Let C_1 be a cycle of length l in q. Its elements have edges to the elements of a cycle C_2 of length l in q. In general, for $1 \leq i \leq k$, cycle C_i has edges to cycle C_{i+1} with $C_{k+1} = C_1$. Let's consider the point where $C_{p+1} = C_1$ is minimal, possibly this being $p = k$. Then, it's easy to see that C_x is a rotation of C_{x-p} for $x > p$. First, we need $p|k$. Let's look at the number of ways to choose the rotations between C_i and C_{i+1} (the product over all i). For $i > p$, the rotation between i and $i + 1$ is predetermined by the rotation of $i - p$ relative to $i - p + 1$. For i from 1 to $p - 1$, the rotation between i and $i + 1$ is chosen randomly (thus l^{p-1} variants). Now, how do we choose the rotation between C_p and C_{p+1}, that is equivalently the rotation r between C_1 and C_{p+1}?

We look for a $0 \leq r \leq l-1$ such that $r \cdot \frac{k}{p} \equiv 1$ (modulo l). This is impossible if $\gcd(\frac{k}{p}, l) \geq 2$, and has a unique solution otherwise. Therefore, we can consider the valid p's as those that satisfy the conditions above.

Furthermore, we can say we have reduced to the following problem: we have f cycles of length l, and we need to group them into groups of sizes part of a subset S of the set $1, 2, 3, \ldots, f$. In how many ways can we do this? We will consider the cycles as unitary elements. There are multiple methods based on dynamic programming and combinatorial elements to find the answer. The solution that obtains 100 points is based, initially, on the following observation: the size of set S is not very large. In fact, the elements of S are certainly divisors of k smaller than n. For $K = 10^{18}$ and $N = 10^5$, the maximum number of divisors of a number $\leq K$ that are smaller than N is on the order of a few thousand. We will use this maximum number from now on under the name m.

We can formulate the dynamic programming dp_i = the number of ways to divide i elements (our answer will be dp_f). Then, if we choose g = the size of group 1, we transition from $dp_{i-g} \cdot A(i-1, g-1) \cdot l^{g-1}$. Finally, the final result will be the product of the answer to the subproblems for each cycle length l. We thus obtain a solution bounded by $O(n * m)$.