INFO(1)CUP 2022 EDITORIALS

SCIENTIFIC COMMITTEE

Problem Sumex

Authors: Matteo-Alexandru Verzotti, Alexandru Luchianov

Let's define the *minimum excluded element* as mex. Also, we define a sequence a as a subsequence of a sequence s if a can be obtained by deletion of (possibly zero) elements from the beginning and end of s. In other words, $2,3,4$ is a subsequence of $1,2,3,4,5$; whereas $1,3,4$ is not.

Subtask 1

Since all the numbers are greater than or equal to 1, then the mex of all subsequences will always be 0, so the answer in this case is always 0.
Time complexity: $O(N+Q)$.

Subtask 2

For each query we will go through all the subsequences of $a_l, a_{l+1}, \dots, a_r$ and find the mex for each one of them. We will keep a boolean array *visited*, such that $visited[x] = true$ if and only if the value x appears in the sequence $a_{l'}, a_{l'+1}, \dots, a_{r'}$. The mex of the sequence is the first value y such that $visited[y] = false$.
Time complexity: $O(L^3 * Q)$, *where L is the length of the query.*

Subtask 3

Observation. *If the mex of the sequence* $a_l, a_{l+1}, \dots a_r$ *is smaller than the mex of the sequence* $a_l, a_{l+1}, \dots a_{r'}$, *then* $r < r'$.

If we fix the left boundary for the subarray, every time we increase the right boundary, the mex can only increase. In conclusion, we don't have to reset it. Since the numbers are $\leq n$, the mex can only increase a total of n times, bringing down the complexity for getting the mex of all arrays with a fixed left boundary from $O(N^2)$ to $O(N)$.

We are going to precompute, using the observation above, the mex of all subarrays of the original array in a matrix A, such that $A[i][j]$ is the mex of the sequence $a_i, a_{i+1}, \dots, a_j$. The answer for a query l, r is the sum of all values $A[l'][r']$ where $l \leq l' \leq r' \leq r$. This can be easily computed using prefix sums on the matrix A.
Time complexity: $O(N^2 + Q)$.

Date: February 2022.

Subtask 4

We are going to precompute for each number i from 0 to $n-1$, the minimum length interval $[x_i, y_i]$ that contains all the numbers from 0 to $i-1$.

Observation. $[x_{i-1}, y_{i-1}] \subset [x_i, y_i]$ *for each* i *from* 1 *to* $n-1$.

In conclusion, the intervals can be precomputed using prefix minimums/maximums. Using these intervals, we can binary search the mex of the interval query. Let's call this value m. Now, we can deduce that the answer for the query i is given by the formula

$$\sum_{j=0}^{m}(x_j - l_i + 1)(r_i - y_j + 1)$$

By expanding the brackets, we get

$$(r_i+1)\sum_{j=0}^{m} x_j \quad + \quad (l_i-1)\sum_{j=0}^{m} y_j \quad + \quad (l_i+1)(r_i+1)\sum_{j=0}^{m} x_j y_j$$

We can solve this in constant time by precomputing partial sums on the aforementioned arrays.

Time complexity: $O(Q \log N)$

Subtask 5

Let's view the problem from a different perspective. Suppose we want to calculate the sum of mex of all subsequences ending at position i. We will be using an algorithm similar to a sweeping line. Let's create an array *lastpos*, such that *lastpos[x]* marks the last occurence of x up until i; and an array *minpos*, such that $minpos[i] = \min(lastpos[0], \ldots, lastpos[i])$.

Observation. *The sum of mex of all subsequences ending at position i is the sum of the values of the array minpos.*

Let's illustrate these using a bar chart.

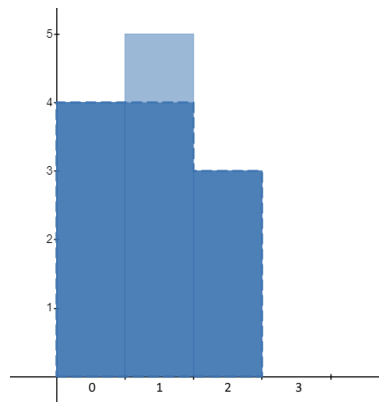

Figure 1. Chart using the array given in the problem example and $i = 5$.

The above observation can now be reformulated as: *The sum of mex of all subsequences ending at position i is the area under the partial minimums graph.*

Observation. *The sum of mex of all subsequences with their left end greater than l and ending at position r is the area under the partial minimums graph, above the horizontal line $(l-1)$.*

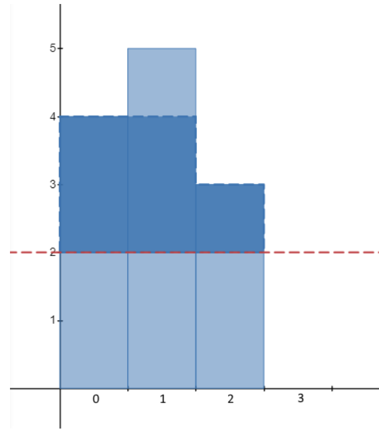

Figure 2. Sum of mex of all subarrays with left end greater or equal than 3 and right end equal to 5 for the same array is the area marked with deep blue.

Using these two observations, we can now move forward to the solution. Let's call the area above the horizontal $l-1$, at a point in time r, as $Area_{l,r}$. Formally, for the i-th query, we want to compute

$$\sum_{r'=l_i}^{r_i} Area_{l_i,r'}$$

We will solve the queries in an offline manner. We will sort them by their left end (because of the nature of the queries, they will also be sorted by their right end) and simulate the operations on the above graph using a brute-force approach. Since they are sorted by both ends, the queries can be traversed using an algorithm similar to the *two pointers* method.

Time complexity: $O(Q\log Q + QMAXVAL)$.

Subtask 6

For this subtask we'll have to optimize the way we compute the changes on the aforementioned graph. We can observe that we only really care about the minimum partial graph, so let's use that one instead, for the sake of simplicity. We will split the graph into rectangles characterised by the triplet (*left*, *right*, *height*), meaning that there is a rectangle from position *left* to position *right* with height *height*.

This time we'll traverse the array once from 1 to n to form the graph after , but the updates on it will be done in reverse. This way, when we transition from i to $i-1$, the graph flattens, which will help us in the future. We will find the value of $v[i]$, find the

next position to its left, and then update the rectangle that contains $v[i]$ accordingly. Next, we'll have to update all the rectangles to the right of $v[i]$ such that they are not taller than the one containing $v[i]$. Here's an example when transitioning from $i = 5$ to $i = 4$:

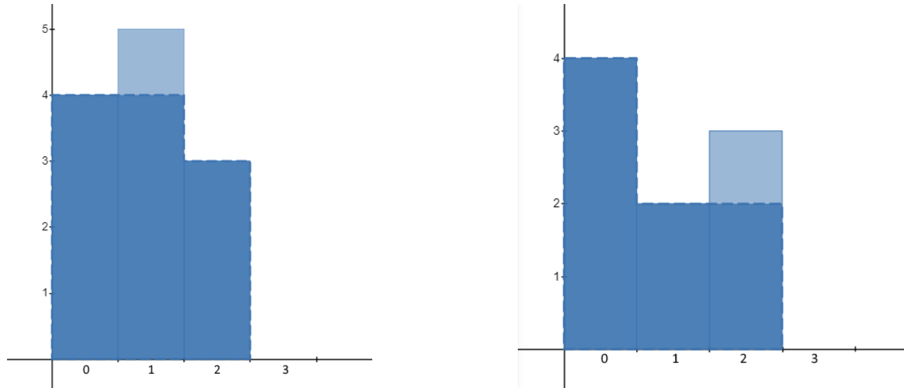

FIGURE 3. Transition from $i = 5$ to $i = 4$.

In order to achieve this, we want to keep our rectangles in a data structure that:

- Keeps them sorted.
- Supports operations of insertion/deletion.

A simple data structure that achieves this is the C++ *set*.

Time complexity: $O(N \log N)$

SUBTASK 7

We now have to optimize the way we find the area over a horizontal line. We'll keep an array *len* and an array *area*, with the following definitions:

- *len[h]* = total length of the rectangles with height less than or equal to h;
- *area[h]* = total area of the rectangles with height less than or equal to h.

Both the arrays *len* and *area* have operations of type: "Add value *val* to positions $x, x+1, \ldots, n$" and "What is the sum of values $x, x+1, \ldots, n$?". They can be processed easily using a data structure like a Segment Tree/Fenwick Tree.

Now, it's easy to see that, for a query of type l, r, we get the formula

$$Area_{l,r} = (area[n] - area[l-1]) - (len[n] - len[l-1])(l-1)$$

We can also compute for each rectangle the length of time of its existence above the horizontal line, so we can calculate its contribution to the final answer.

Time complexity: $O((N+Q)\log N)$

PROBLEM DATE

Author: Tamio-Vesa Nakajima

We will count separately the dates of form $y/m/d$ where $m \neq 2$ or $d \neq 29$ (the non-leap year case) and the dates of form $y/2/29$ where $y \bmod 4 = 0$ and either $y \bmod 400 = 0$ or $y \bmod 100 \neq 0$.

Non-leap year case: Fix the first /. To the left we need to count the number of valid years, and to the right we need to count the number of valid months and years. These values can be computed using dynamic programming: the first value by computing the number of valid years in each prefix; the second value by computing the number of times each day appears in each suffix, and then similarly for each month.

Leap year case: Fix the first /. To the left we need to count the number of valid years which also have the right value modulo 400, and to the right we need to count the number of times 2/29 appears as a subsequence. These can be computed similarly to above.

Problem Niceset

Author: Tamio-Vesa Nakajima

(In the following editorial, $[x < y]$ is 1 if $x < y$, and 0 otherwise. This notation is called an Iverson bracket.)

This problem asks for the largest subsequence of a given sequence $a_1, \ldots, a_n$ for which the sum of the absolute differences of all pairs of elements is at most S. The first observation is that, if we sort $a_1, \ldots, a_n$, then the outputted subsequence can be assumed to be continuous. The second observation is that the predicate "does there exist a subsequence of size k that satisfies the required condition" is monotone with respect to k. In other words, if it holds for $k+1$, it also holds for k. This implies that we can use binary search to find the largest possible value of k, if we can efficiently check if there exists a subsequence of size k that respects the required condition.

How can we do this? Suppose we have sorted $a_1, \ldots, a_n$. Then the sum of absolute differences of all pairs of $a_i, \ldots, a_{i+k-1}$ is exactly

$$\sum_{i \le p < q \le i+k-1} |a_q - a_p| = \sum_{p=i}^{i+k-1} \sum_{q=i}^{i+k-1} [p < q](a_q - a_p) = \sum_{q=i}^{i+k-1} a_q \left(\sum_{p=i}^{i+k-1} [p < q] \right) - \sum_{p=i}^{i+k-1} a_p \left(\sum_{q=i}^{i+k-1} [p < q] \right)$$

$$= \sum_{q=i}^{i+k-1} a_q (q-i) - \sum_{p=i}^{i+k-1} a_p (i+k-p-1) = \sum_{q=i}^{i+k-1} q a_q - i \sum_{q=i}^{i+k-1} a_q + \sum_{p=i}^{i+k-1} p a_p - (i+k-1) \sum_{p=i}^{i+k-1} a_p$$

$$= 2 \sum_{q=i}^{i+k-1} q a_q - (2i+k-1) \sum_{q=i}^{i+k-1} a_q$$

Thus if we let $S_i = \sum_{j=1}^{j<i} a_j$ and $T_i = \sum_{j=1}^{j<i} j a_j$, we can find that the sum of absolute differences of all pairs of $a_i, \ldots, a_{i+k-1}$ is

$$2(T_{i+k} - T_i) - (2i+k-1)(S_{i+k} - S_i).$$

Using this approach, we have a complexity of $O(n \log n)$.

Problem Hide and Seek

Author: Alexandru Luchianov

We will first consider a solution for when the input graph is a tree. Do a depth-first search. Now, each node has a set of children and a parent, who together constitute all of its neighbours. The sum of the neighbours of node is thus the sum of its children plus the value of its parent. This indicates the way forward: for each node, maintain the sum of its children. This allows us to find the result quickly. Then, when updating a value, we need only update itself and the sum of its parents children. This can also be done in one operation. The complexity is thus $O(1)$ per query.

Now that we know how to find the solution in this case, we can make the following observation: if we can split the edges of the graph into multiple trees (or, more generally, forests), then we can run the previous algorithm on each tree (forest). If we split into f forests, then the complexity is $O(f)$ per query. The question is then: what is the smallest number of forests we can split our input graph into?

It turns out that this is k, if we simply always remove an arbitrary forest that spans each connected component. This can be done in linear time for each removed forest, giving us a total $O(k(n+m+q))$ complexity.

Problem Ezlulu

Author: Alexandru Luchianov

First, let us consider the total result. It is the sum, for each plate, of its value times the number of plates broken by it. Another way of writing this sum is the sum, for each plate x, of the value of the plate *that breaks* x. This immediately gives us an indication of the way forward. Each plate can be broken only by the plates larger than it; and the cost is maximized whenever the plate that breaks it is the highest-value one from among the larger one. This gives us an upper bound on the total value: the sum, for each plate x, of the highest value plate larger than x. Can this cost be achieved?

In fact, this can be done. For each plate x, let p_x be the maximum value plate larger than x. We can now view the configuration of plates as a rooted tree, where the parent of plate x is p_x. All we want is that each plate is broken by its parent. If we order the children of a node in *decreasing* order of size, then it is not difficult to see that the post-order traversal of the tree gives us the correct ordering.

Problem Tennis

Author: Alexandru Luchianov

A common approach when solving counting problems is to modify the statement to an equivalent version which is easier to count. In our case, each valid sequence of balls has value equal to $count^k$, where *count* is the number of balls from the sequence with weight *at most* y. The fact that each sequence has its own value makes it hard to count, so we should reformulate the problem as follows to force each value to be 1: count the number of ways to choose a valid sequence of balls and k not necessarily distinct elements with weight at most y from the sequence. It is easy why the two formulations are equivalent.

With our new formulation, we can solve the rest using dynamic programming. Let dp_{ijh} be the number sequences of length i, whose sum, modulo w, is equivalent to j, an which contains h elements with weight y or less. To make the recurrence easier to define,

we use the convention that $dp_{i,j-w,h} = dp_{ijh}$. We then have the following recurrence relation:

$$dp_{i,j,h} = \sum_{p=0}^{w-1} v_p dp_{i-1,j-p,h} + \sum_{p=0}^{y-1} \sum_{q=0}^{h} \binom{h}{q} v_p dp_{i-1,j-p,h-q}$$

If we use this recurrence we can find the solution in $O(Nw^2k^2)$, this is unfortunately not enough to good enough to get a full score. This can be further optimized using matrix exponentiation, but there exists an even better and faster way of optimizing it. We can compute the recurrence using something similar to divide and conquer: if i is odd then we use the same formula as before; however, if i is even then we split the sequence of length i in 2 smaller sequences of size $i/2$.

$$dp_{i,j,h} = \sum_{p+p' \equiv_w j} \sum_{q+q' \equiv_w h} \binom{h}{q} dp_{i/2,p,q} dp_{i/2,p',q'}$$

(Above, $\equiv_w$ represents modulo w equivalence.) In this way, we can compute the solution in $O(w^2k^2 \log n)$. The solution can be further optimized using NTT, however this was not necessary for a full score.