Self-describing Analysis

Authors: Ivan Lupov

Intro

As the subtasks suggest and is often the case with harder query problems, it is a worthwhile endeavor to understand how to solve a single query and only after that tackle the problem of solving for multiple queries.

Solving for the whole array

There are two main approaches to problems, asking for the number of subarrays that abide a given restriction – sweepline or divide and conquer. Our task is solvable by both.

Sweepline

Both sweepline solutions require the following view over the problem: say we have fixed a pointer r somewhere in the array, thus have processed the prefix $[1, r]$ – what positions $l \in [1, r]$ can be valid left endpoints of a *self-describing* subarray? That depends on the values and their position: say a value x has been seen on positions $p_1, p_2, \dots, p_k$.

- Then for all $p_k < l$, x will be irrelevant or in other words these options for l are ”valid“ as much as it concerns the constraints given by x in the definition.
- Otherwise, in the case $l \leq p_k$ and x is present in the $[l, r]$ subarray we are investigating, l should additionally be between the p_{k-x}-th and p_{k-x+1}-st occurrence of the value x.

This motivation suggests well that there are some intervals in which it is “forbidden” to put the left endpoint and some where it is allowed. One way of keeping track of them is by using a segment tree which adds a “penalty” to some interval of positions where the left endpoint of a subarray cannot be. This way the query of “how many positions can serve as the left endpoint of a subarray?” becomes “how many positions have zero penalties?” which can be answered with a segment tree keeping track of its minimum value and its frequency. This is an obvious $O(n \log n)$ solution, befitting the subtasks constraints.

Another, not so obvious solution time-wise is to keep all the candidate left endpoints in a set and iterate it for every right endpoint we examine. Keeping track of both the maximum bound for the left endpoint and the expected size of the subarray (every number

x that appears must appear exactly x times) with braking if either condition fails is also enough to give a passing solution.

The complexity analysis of the second solution is more involved – for every "major" *self-describing* subarray (that is a subarray, not contained in any other *self-describing* one), we need to traverse all of its elements and account for them. It will be proven later (alongside another key observation for the problem) that this count will stay in $O(n)$.

Divide & conquer

Another reasonable approach in such problems is to enumerate the *self-describing* subarray using some divide-and-conquer strategy. This turns out to be possible. We will focus on only the merging of two meighbouring intervals $[l, m]$ (nicknamed *left*) and $[m + 1, r]$ (nicknamed *right*).

The choosen strategy is the following: for every prefix in *right* we want to find all the suitable suffixes in *left*. This means that if a value x:

- does not appear in *right*, it can either not appear or appear x times in *left*;
- appears $y < x$ times in *right*, it must appear $x - y$ times in *left*;
- appears exactly x times in *right*, it mustn't appear in *left*;
- appears more than x times in *right*, then it is not possible to use this or any further prefixes of *right*.

Doing all these checks for every possible value would require taking care of multiple counter arrays, fortunately we can instead hash them and query for their existence/frequency over their hashes. This is possible because knowing the prefix in *right* that we are taking we can *almost* reconstruct the left part of the *self-describing* subarray. The only thing we should additionally worry about is if all of a single value's occurrences are only in either the left or the right part of the subarray.

If all of the occurrences are to the left we can simply ignore them when constructing the hash – this way $[1, 2, 2]$ hashes to the same value as $[1]$. In the case where a single value has all of its occurrences in the right part, we should throw away all hashes to the right of its first occurrence in the left part and never consider them again.

Since the merging of two intervals takes time proportional to their sizes, the classic $O(n \log n)$ complexity holds.

$O(n)$ bound on the number of self-describing subarrays

The very restrictive definition of *self-describing* subarray can suggest that, but their maximum count in a single input cannot exceed a certain reasonable threshold. The approach we will take is the the following: given any array from all possible inputs, applying some operations that help make it easier to understand and simultaneously increase the number of *self-describing* subarrays, we will never get more than $O(n)$ *self-describing* subarrays.

First, assume we pick some array with length n and all its values are in the interval $[1, n]$. There is no convinient structure to this array that could help us investigate it, however we can force some rules. It is obviously beneficial to have all occurrences of x in blocks of size x, spaced apart from eachother. For example $[1, 2, 3, 2, 1, 3, 3]$ can be transformed into $[1, 2, 2, 3, 3, 3, 1]$. Notice both ones aren't together in a block, they make two separete blocks from a single element. This way we obviously increase the number of *self-describing* subarrays, but that is not a problem since we are proving an upper bound. If the frequency of occurrences of some value x is not divisible by x we can simply throw away the remainder occurrences – they won't cause trouble anyway.

Now we will think of the blocks we established as the elements of the sequence – this way the problem boils down the "how many subarrays have no duplicate elements?". This translates to our restriction of not having two blocks size x of value x (that would be $2x$ such values). Restating the problem this way it should be fairly clear that an optimal (as in maximum number of *self-describing* subarrays) array must have the following form: multiple increasing subsequences of decreasing height (think something like $[1, 2, 3, \dots, x, 1, 2, 3, \dots, y, 1, 2, \dots, z, \dots]$ for $x \geq y \geq z \geq \dots$). Slide variations of this are possible (reordering the elements $1, 2, 3, \dots, x$ as long as we keep the order consistent with the next chunks), but we will not bother with them.

Funnily enough, without finishing the proof, we can run a dynamic programming with states $(len, last)$ keeping track of how long the array we constructed is and what is the length of the last chunk. This can run in $O(n\sqrt{n})$ and show us the exact number in the worst case.

However, in order to (hand-wavely) finish the proof, let's examine the convinient case where $n = 1 + 2 + 3 + \dots + m$ for some m (and the array of blocks looks like $[1, 2, 3, \dots]$). The number of *self-describing* subarrays is $\frac{m(m+1)}{2}$ which by definition of m is $O(n)$. If we try to extend our array to some $[1, 2, 3, \dots, m, 1, 2, \dots, p]$ for some $p \leq m$ on every step we will be adding some $m \approx \sqrt{n}$ and thus the sum will stay in $O(n)$.

Admittedly, the proof can use some more rigor, but this should be enough. At least in competitions, it is a useful skill to be able to handwavely prove some facts like this.

Answering multiple queries

After going through the pain of proving some sensible upper bound on the number of *self-describing* subarrays in a given input, answering the queries boils down to simply finding how many of the found intervals are contained in the interval defined by a query. This can be done by either merge sort tree or persistent segment tree.

The merge sort tree solution can be naively implemented in $O(n \log^2 n)$ time or be optimized to $O(n \log n)$. This is done with fractional cascading, which in simple words does the following: all nodes in the merge sort tree are sorted arrays that were create by merging the sorted arrays of the child nodes. Since we are going to binary search on each node when answering a query, we should additionally keep data of where a given element in the sorted array came from – was it from the left/right child node and what index was it in its sorted array. This way we can afford to do a single binary search in the root of the merge sort tree.

The persistent segmtree tree solution seemed to perform better while testing the prob-

lem and it gives a direct $O(n \log n)$ solution. If we find a *self-describing* subarray $[l, r]$, we will add to the r-th version of the segment tree $+1$ in the l-th position. This way queries $[u, v]$ can be answered by querying the sum in the interval $[u, v]$ for the v-th version of the segment tree. For this version of the segment tree the only *self-describing* subarrays that exist are the ones with right border $\leq v$ and since we are querying the interval $[u, v]$ we will count only those of them that have left border $\geq u$.