Code Jam 2013 - World Finals

Analysis: Let Me Tell You a Story

Dissecting the problem

The problem statement asks for the number of ways to remove elements from a sequence until we obtain a non-increasing sequence. Looking at the problem from the end, suppose we know the final non-increasing sequence, and it's of length **K**. Certain elements have been eliminated, and we removed **N**-**K** elements in total. There are (**N**-**K**)! (factorial of **N**-**K**) ways to remove those elements. So the first approximation seems to be to sum those factorials over all non-increasing subsequences of the the original sequence.

However, this is not entirely correct. Some non-increasing subsequences are not even reachable since we stop as soon as our sequence becomes non-increasing. For example, in the second example from the problem statement the sequence is non-increasing from the start, so no proper subsequence is reachable at all. For subsequences that are reachable, not every way of reaching them might be possible. For example, in the first example from the problem statement we can reach the '7 <first 6>' subsequence (note, it should be considered different from '7 <second 6>' subsequence) by eliminating the second 6, and then 4. However, it can't be reached by doing those operations in reverse order, since we'd stop right after eliminating 4.

Now instead of all (**N**-**K**)! ways to reach a certain subsequence, we need to count just the possible ways. The main trick in solving this problem is: let's count the impossible ways instead, and then subtract. The impossible ways are simply those when before the last removal, the subsequence was already a non-increasing sequence of length **K**+1. And for each such subsequence there are exactly **K**+1 ways to make an impossible removal and arrive at a subsequence of length **K**. That means the sum of numbers of impossible ways to reach all non-increasing subsequences of length **K** is equal to the total number of non-increasing subsequences of length **K**+1 times (**N**-**K**-1)! (the number of ways to reach the longer subsequence) times **K**+1.

To summarize, suppose $\mathbf{A_K}$ is the number of non-increasing subsequences of length **K**. Then the answer to this problem is sum over **K** of $\mathbf{A_K}$*(**N**-**K**)!-$\mathbf{A_{K+1}}$*(**N**-**K**-1)!*(**K**+1).

Solving the sub-problem

We've now reduced our problem to a much simpler one: find $\mathbf{A_K}$. This problem can be solved used a somewhat standard dynamic programming approach, with a twist to make it run faster.

First, let's assume that all input numbers are different, and between 0 and **N**-1. It's not hard to transform them in this way without changing the answer. If there are equal numbers, we'll slightly reduce the number to the right, so that non-increasingness of all subsequences is preserved.

Our dynamic programming problem will now be: what is the number of non-increasing subsequences of length **P** that end with number **Q**? Let's call that number $\mathbf{B_{P,Q}}$. We will find them in the order **Q**s appear in the sequence.

It's not hard to see that $\mathbf{B_{P,Q}}$ is just the sum of $\mathbf{B_{P\text{-}1,Q'}}$ for all numbers **Q'** that are greater than **Q** and appear before **Q** in the sequence. Since we process the states in the order **Q**s appear in the sequence, we just need to take the sum over all **Q'** that are greater than **Q**.

This is already a working solution for our problem, but it is a bit too slow: it runs in $O(\mathbf{N}^3)$ which is a bit too much for **N**=8000. We have $O(\mathbf{N}^2)$ states in our dynamic programming, and we need to find a sum of $O(\mathbf{N})$ numbers to process each state. However, we can compute such sum faster! We just need an array-like data structure that supports changing elements and finding the sum of its suffix (over all **Q'** that are greater than **Q**), and the Fenwick tree is a data structure that does exactly that, performing each operation in $O(\log\mathbf{N})$ time, for a total running time of $O(\mathbf{N}^2*\log\mathbf{N})$.