Permutation Recovery

-editorial-

Author: Oncescu Costin
Preparation: Oncescu Costin & Niculae Alexandru

We first need to understand how to compute the vector Q for given P. We should compute delta[i] = Q[i] - Q[i - 1] which is the number of increasing subsequences that end in i (those ending at a position smaller than or equal to i minus those ending at a position smaller than or equal to i - 1). It is easy to compute delta[i] in time complexity O(N^2) by fixing the position j preceding i in the subsequence and adding one for the subsequence formed just from element P[i]. This way, we get delta[i] = 1 + sum of delta[j] with 1 <= j < i and P[j] < P[i].

10 points

For N <= 9, we can simply try all the N! possible permutations and compute the vector delta for them and compare it with the one computed based on Q. Time complexity: O(N!*N^2)

25 points

In order to get the second subtask, we need a much better algorithm, a polynomial one. We notice that in this subtask, all numbers have at most 18 digits. That means they fit in long long so we don't need big integers. Let's make one first observation: any 2 different permutations have different Qs, and implicitly different deltas. This is not so easy to prove. We'll prove it inductively and in the same time get the answer. For N = 1, it's obvious that the claim holds (there is just one permutation). Let's prove that if the proposition is true for some N, it is true for N + 1 too.

Let's say we have an array delta of length N + 1 and we want to prove it can't have two permutations generating it. By applying the claim for the prefix of length N, we know that there is at most one relative order among the first N elements of P. Also, from the way delta is generated, we have delta[i] > 0 for any i in any permutation.

We also know that delta[N + 1] = 1 + sum of delta[j] with 1 <= j <= N and P[j] < P[N + 1]. But, we already know the relative order of the values in the prefix of length N of P. Let's say we have an array S of length N. S[i] represents the position where we find ith smallest number from the first N values of P. We have a lot of cases:

If 1 = delta[N], then P[N] < P[S[1]].
If delta[S[1]] + 1 = delta[N + 1], then P[S[1]] < P[N + 1] < P[S[2]].
If delta[S[1]] + delta[S[2]] + 1 = delta[N + 1], then P[S[2]] < P[N + 1] < P[S[3]].
....
If delta[S[1]] + delta[S[2]] + ..+ delta[S[N]] + 1 = delta[N + 1], then P[S[N]] < P[N + 1].

Also, as delta[i] > 0, we cannot have more cases at a time. So what we did here is that we found what elements of the permutation are smaller than P[N + 1] and what elements are larger. We basically found a unique relative order of the elements and, so, a unique permutation having the given delta, which proves that the claimed statement holds for N + 1 too. To be noticed that if none of the above cases is true, then there is no permutation with the given delta. We also found how to solve the problem in complexity N^2. After each step i, we find in O(N) the relative order of all the numbers in the permutation so far.

43 points

In order to get the third subtask, we can use the same method, but this time we should handle big integers computations (sum and equality). Even though it might not be needed in this subtask, we can use larger bases (powers of 10) to speed up the time needed to compute sums and compare numbers.

60-79 points depending on the implementation

In order to get some, if not all, of the subtasks 4, 5 and 6, we need a complexity of O(TNlogN) where T is the length of the greatest number in the input and contributes to the complexity through the big numbers computations. Looking more thoroughly, we observe that we actually need just to find out what is the position that has a certain prefix sum and insert an element somewhere at that position. We can use a binary search tree (such as a treap) to achieve NlogN complexity that easily supports this kind of operations.

We could also approach the problem in a totally different way: let's roll back to the N^2 solution and try doing it in some other way. We can start by using some other, more clever observation: we can easily find out where is the 1 in the permutation. It should be on some position i with delta[i] = 1, that's for sure. But what if there are more such positions? Then, it is on the last position with the property that delta[i] = 1. Why is that? Let's suppose the last such position is j, and it actually is in some other place i with i < j. Then, P[j] > 1 (because 1 was already fixed) and that means P[j] is also greater than P[i] and that implies delta[j] = 1 + delta[i] + something (because i < j and P[i] < P[j]).

Let's generalize the approach: we fix all the values from 1 to i in order. Now we should fix where is the value i + 1 in the permutation. Let's suppose that we already have built a part of the permutation and the free places have P[i] = 0. In order for a position T to be a potential position for value i + 1, we must have delta[T] = 1 + sum of delta[j] with P[j] != 0 and j < T. Why is that? Well, P[j] != 0 is equivalent to P[j] < P[T] when assuming that P[T] = i + 1, as all fixed values are in [1, i]. But the same question arises: what if we have more such positions? The answer is the same: we take the last one and the demonstration is analogous.

So we invented a completely new approach also with complexity N^2. The advantage is that this approach does not require inserting values. Let's improve it to NlogN using a segment tree. We should first define some operations that we need to handle:

- Build: read some initial values delta[i] for 1 <= i <= N
- Update pos, val: for every i with pos <= i <= N, do delta[i] -= val
- Query: find the last position where we have an 1 in the current array delta
- Fix pos: make delta[pos] equal to INF where INF is some really big value

The problem that we encounter is how to find the last 1. Finding the last position i with delta[i] equal to something doesn't seem doable. But what if we look at it from some other perspective: what if we ask for the last appearance of the minimum? That's way easier: we could simply store for each interval in the segment tree the minimum and some value to help us propagate the lazy update and that's all we need. And it works because, thanks to the particularities of the problem, we always have delta[i] >= 1 and if 1 occurs at some point, it surely is the minimum in the array.

94-100 points depending on implementation

So far, our best solutions had really good complexities. The slow part was doing computations with big numbers. We can observe that in subtasks 7 and 8, we no longer have a maximum limit of T. We should somehow skip the computations part. What if we store the numbers just modulo some values (we basically keep a hash)? Let's consider for simplicity that we store the value modulo just some P (if we use more, the solution doesn't change). The problem now is that N^2 solutions still work but both NlogN solutions are dependent on comparisons, which can no longer be performed. We'll explain a way of adapting the segment tree solution and supporting the operations in complexity sqrt(N).

Let K be some constant. We divide the array in about N / K intervals. We also store for each interval a hashmap of all the values in the interval and a value dec[interval], initially 0.

When we do an update, we modify the values and rebuild the hash table of the maximum one piece that is partially intersecting the update segment. For the pieces completely included in the update interval, we just increase the value dec by the given val. Time complexity: K + N / K

In order to handle a query, we traverse the pieces from the last to the first. We look in the hashtable of a piece for the value 1 + dec[piece] and, if we find it, we traverse the whole piece again to get the last position having that value. Time complexity K + N / K

When handling a Fix operation, we can simply delete the value delta[pos] from the hashtable of the piece containing position pos. Time complexity O(1)

We choose K = sqrt (N), and so, the complexity of handling an operation becomes sqrt (N). The overall time complexity of this algorithm is O(inputSize + NsqrtN) with O(N) memory.
We've assumed that hashtables have O(1) complexity. We could obtain O(NsqrtNlogN) and O(Nsqrt(NlogN)) complexities without using hashtables by keeping the values sorted and doing binary searches when supporting queries.

PS: There is a way of adapting the binary search tree solution to modulos briefly explained by Xellos: http://codeforces.com/blog/entry/51171#comment-354270 .