Editorial

Tasks, test data and solutions for CEOI 2022 were prepared by: Dominik Fistrić, Josip Klepec, Krešimir Nežmah, Ivan Paljak and Paula Vidas. Implementation examples are given in attached source code files.

Task Abracadabra

Prepared by: Krešimir Nežmah and Dominik Fistrić

Preliminary observations

By looking at the numbers written on the face of each card from bottom to top, we can represent the state of the deck at any time by a permutation of the numbers from 1 to n. Denote by `riffle(L,R)` the function which takes as input two arrays of integers L and R, representing the cards in each hand, and returns a new array representing the result of Tin's riffle shuffle on L and R. This procedure can easily be implemented in $O(|L| + |R|)$ by adding the cards one by one from L and R to the result. Therefore, performing one riffle shuffle on a permutation representing the deck can be done in $O(n)$.

After writing out a few examples, one can notice that it seems like the process of shuffling the deck stabilizes at some point. That is, after some number of shuffles we reach a point where shuffling the deck no longer changes it. As we will later see, the number of shuffles needed to reach this point is bounded by n. Therefore, the answer for a query wont change if we set $t = min(t, n)$. The intended solution for the first subtask is to repeatedly shuffle the deck until we reach this point and store the state of the deck at each point in time. After that, we can answer each query in $O(1)$ by looking up the answer. The total time complexity of such an approach is $O(n^2 + q)$.

Block structure

Let's find a better way to describe how the function `riffle(L,R)` computes the result. We will divide L and R into blocks in the following way: the first block starts at the first element of the array and ends right before the smallest element that is larger than the first one. The second block then starts at this element and ends right before the smallest element larger than it. This process continues until we reach the end of the array. Notice that the maximum value in each block is precisely the front element of the block, and the sequence of maximums of the blocks forms an increasing sequence.

The key observation is the following: instead of computing the result one element at a time, we can compute it block by block, and the final result consists of these blocks placed next to each other, ordered by the front element of the block. Indeed, if the front element of some block is larger than the front element of some other block, then it is also larger than the rest of the elements in that block. Also, the blocks will be sorted in the end because they are sorted initially and at each point the block with the smaller front element is appended to the result. Note that the blocks will never merge together to form a larger block, i.e. if we split the resulting array into blocks, we obtain precisely a permutation of the starting blocks.

Now let's look at a single shuffle operation with this view in mind. Divide the initial permutation into blocks in the same way as described above. After splitting the permutation in half, some of the blocks will be in the left half, some of them will be in the right half, and there will be at most one block which is partially contained in both. If such a block exists, we'll call it the *middle block*, and we'll talk about its left and right parts. Let L and R be the arrays containing the elements of the left and right halves, respectively. If the middle block does not exist, the blocks of L are precisely the blocks of the permutation which are in the left half, and the blocks of R are precisely the blocks from the right half. All blocks from L have a smaller front element than all blocks from R, so after the shuffle the permutation will not change. On the other hand, if the middle block exists, L will have one more additional block, namely, the left part

of the middle block. R might have multiple additional blocks, since the right part of the middle block need not be a block itself. Still, all elements from the right part of the middle block are smaller than the front element of the next block, so the additional blocks in R will form a subdivision into multiple smaller blocks of the right part of the middle block. Once we perform `riffle(`L`,`R`)`, the blocks of L and R will become sorted by their front elements. This wont affect the blocks that were fully contained in the right half to begin with. It will, however, affect the blocks which make up the right part of the middle block, because all of their front elements are smaller than the front element of the left part of the middle block. Consequently, these blocks will be moved over somewhere to the left of that left part.

To summarize, we have the following:

- The permutation will stay the same after a shuffle if and only if the middle block does not exist.
- The positions of the blocks which are to the right of the middle block will not change.
- The left part of the middle block is its own block, the right part might have to be split into multiple smaller blocks.
- These smaller blocks will move to the left of the left part of the middle block.

In particular, the middle block splits into at least two smaller blocks, so the total number of blocks increases by at least one after each shuffle. Initially, there is at least one block, and in the end there are at most n blocks, so the total number of shuffles is bounded by $n - 1$.
Challenge for the reader: find a case which achieves the maximum number of shuffles until stabilizing.

Implementation

It is possible to efficiently implement the procedure described above. We store each block using a triplet of integers (v, l, r), where v represents the value at the front of the block, while l and r are the indices of the ends of the block, from the point of view of the initial permutation. We keep all of the blocks in an `std::set`, to ensure they are sorted according to v at all times. We also keep track of the total length of all the blocks that are currently in the set. Note that if at any time there is a block which is completely contained in the right half, we can remove it from the back of the set, since this block will never again change its position.

When performing a shuffle we do the following:

- While there is a block completely in the right half, remove it.
- If the total length of the blocks in the set is $\frac{n}{2}$, there is no middle block and we can stop.
- Otherwise, the back of the set now contains the middle block. Remove it from the set, split it into smaller blocks and insert them back in the set.

What is left is to figure out how to obtain the initial set of blocks and how to efficiently split the middle block into smaller blocks. For this we precompute for each position i in the initial permutation the position `nxt[`i`]` representing the smallest index whose corresponding value is larger than the value at position i, or $n + 1$ if there is no such position. This can be done in a standard way in $O(n)$ using a stack. The sequence i, `nxt[`i`]`, `nxt[nxt[`i`]]`, ... determines the starting indices of the blocks starting from i. Using this we can decompose the right part of the middle block in $O(\text{number of new blocks})$.

For the second subtask, all the queries have the same t value, so it is sufficient to run this process until time t, at which point we can iterate over all the blocks in order and obtain the whole array. It is easy to show that the total number of blocks that were in the set at one point or another is at most $2n$, so the total time complexity of obtaining what the array looks like after t shuffles is $O(n \log n)$. After this we can answer all the queries in $O(1)$.

Supporting queries

We can input all the queries, sort them by their t value, and answer them offline. To do this we need to maintain a data structure which keeps track of the currently active blocks, and is able to determine for an arbitrary index i in which block is it currently contained in. This can be done directly with a balanced binary search tree like splay or treap, but we'll describe an easier way involving only a segment tree or a fenwick tree.

First we run the process described above from the second subtask. That is, using a set we repeatedly shuffle the permutation until we obtain the final ordering. We create a list of all the blocks that were contained in the set at some point or another. We order this list according to the front value of each block. Then we create a range-sum, point update segment tree on top of this array of blocks. Each node of the segment tree will store the total length of all active blocks in its range.

We then run this process a second time, starting from the beginning again, but this time we additionally keep track of the lengths of the blocks using the segment tree. Initially, each node of the segment tree stores the value zero, because there are no active blocks. Every time a new block appears in the set, or is deleted from the set, we make it active/inactive in the segment tree as well. That is, we update the point at the corresponding index by adding the length of that block to that position.

To answer a query we have to be able to do the following: for a given value i, determine the index of the first active block such that the prefix sum of the lengths of blocks up to that point is at least i. This is a standard problem which can be solved either in $O(\log^2 n)$ per query using a binary search along with using the segment tree for querying the prefix sums. A better way to do it is to start from the root of the segment tree and directly walk down to the desired index in $O(\log n)$. This solves subtasks 3 and/or 4, depending on the efficiency of the implementation. The total time complexity is $O((n+q)\log n)$.