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 $O((n+q)\log n)$.

Task Homework

Prepared by: Krešimir Nežmah and Dominik Fistrić

We can represent a valid expression by a binary tree with n leaves and $n-1$ inner nodes. The leaves correspond to question marks, and each inner node corresponds to one of the functions `max` or `min`. The problem can now be rephrased as writing a permutation of numbers from 1 to n in the leaves, and propagating these values to the root. The solution for every subtask involves parsing the string from the input and converting it to such a binary tree.

The first subtask can be solved in $O(n! \cdot n)$ by trying out all the different permutation of $\{1, 2, \ldots, n\}$, substituting it in the expression and evaluating it.

The second subtask can be solved with a bitmask dp. The state is `dp[node][mask]`, which represents the set of all possible values obtainable in this node if the allowed values are from the mask. For the transition, we try to partition the mask into two submasks (one for each child), in all possible ways. The time complexity is $O(3^n \cdot n^2)$, but in practice it is much faster because all states for which the number of ones in the mask don't match up with the number of nodes in the subtree can be discarded.

There is also a randomized solution which solves the second subtask. We make a guess that the set of obtainable numbers forms an interval (this guess will later turn out to be true). Then we try to evaluate as many different permutations as possible, and store the smallest and largest number we've come across. We declare our final output to be the length of that interval. Challenge for the reader: prove that the probability of success of such an approach is sufficiently high.

The third subtask is solved by looking at the longest sequence of nodes, starting from the root, which are of the same type (all `min` or all `max`). Let's say that there are k of them. The solution is then $n-k$. The proof is left as an excersize to the reader.

Let's look at a node and its subtree. Let S be a set of distinct numbers whose size equals the number of leaves in the subtree. The full solution requires the observation that the set of number obtainable in this node, using the numbers from S as the values for the leaves, forms an interval in S. Additionally, we must figure out a way to combine the intervals of the left and right child to get the interval for the node itself. The proof is inductive/recursive and at the same time describes a way to obtain the mentioned intervals, allowing us to solve the problem with a tree dp.

Suppose that a node has a left child with L leaves, and a right child with R leaves. Also, let the interval of possible values for the left child be $[a, b] \subseteq [1, L]$, and $[c, d] \subseteq [1, R]$ for the right child.

If the node is of type `max`, the lower limit for the interval of the node turns out to be $a + c$. Indeed, let's call the smallest $a + c - 1$ numbers from $[1, L + R]$ *small*, and the rest of them *large*. Let's say that the left tree contains x small numbers, and the right subtree contains y small numbers. Since $x + y = a + c - 1$, at least one of $x < a$ and $y < b$ must hold. With out loss of generality assume that $x < a$. Now the a-th smallest number in the left subtree is large, so the maximum of the left and right subtrees will also be large. This shows that the node value is at least $a + c$. This value is also obtainable: put the numbers $[1 + c, L + c]$ in the left subree, and the rest of them in right subtree. We can make it so that the value of the left subtree turns out to be $a + c$, and the value of the right subtree is c.

If the node is of type `min`, the lower limit will be $\min(a, c)$. This can be shown in a similar way to what is described above. This time, the small numbers will be the ones smaller than $\min(a, c)$, and the rest will be large. The same type of argument shows that the node value is at least $\min(a, c)$. If $a < c$, this can be obtained by putting $[1, L]$ in the left subtree, and $[L + 1, L + R]$ in the right subtree. The case where $a \geq c$ is similar.

We can show a similar thing for the upper limit. If the node is of type `max`, the upper limit is $\max(b + R, d + L)$, and if it is of type `min`, it is $b + d - 1$. To prove that the values in between are obtainable, we just have to slightly alter the way of distributing the numbers in the left and right subtrees, similar to the constructions for achieving the bound. The total time complexity is $O(n)$.

Task Prize

Prepared by: Ivan Paljak and Josip Klepec

Setup

Summarization of the task is somehow choosing the K node labels and then $K-1$ queries such that they uniquely determine a tree.

Algorithm 1 Choose subset of K node labels

$N \leftarrow$ size of trees T_1 and T_2
$K \leftarrow$ size of subset od node labels to choose
$P_1 \leftarrow$ preorder of T_1
$S \leftarrow$ first K node labels with respect to preorder P_1 in tree T_1

The subset S of node labels forms a connected subgraph in tree T_1. In tree T_2 it is just some arbitrary subset of nodes.

Algorithm 2 Asking the queries

$P_2 \leftarrow$ preorder of T_2
$s \leftarrow$ array of length K which values are the values from S sorted with respect to P_2
for $i = 1, 2, \ldots, K-1$ **do**
    Ask query for node labels s_i and s_{i+1}
end for

Each query may also give some *lowest common ancestor* which is not in S.

We define S_1 and S_2 as subset of nodes in respective trees as union of S and *lowest common ancestors* that appear in queries.

Lemma 0.1. *S_i can be formed into a tree structure. Furthermore, S_1 forms a connected subgraph in tree T_1 but it isn't relevant for the rest of the algorithm.*

We will describe the algorithm for making such small tree. It can be applied to both T_1 and T_2.

Algorithm 3 Forming trees

$V \leftarrow S_i$ sorted with respect to preorder of respective tree.
nodes $\leftarrow$ empty stack
for each $v \in V$ **do**
    while stack is not empty AND $lca(v, stack.top()) \neq stack.top()$ **do**
        pop from stack
    end while
    if stack is not empty **then**
        make v child of the node on top of the stack
    end if
    push v to top of the stack
end for

Now for this trees we want to reconstruct weights of edges. For each tree we have $2 \times (K-1)$ paths we know the length of. Some may be redundant.

Lemma 0.2. *Those paths are such that they uniquely determine the tree.*

The proof is not too hard so we won't go into too much detail. Firstly, imagine we construct a graph as described in section for Full Points (see section couple lines below)..

Now, all we need to prove is that that graph is connected. There is no need to prove that the given set of queries don't give contradiction because we can relly on the authentication of the interactor.

We can prove the connectivity by analyzing how we constructed the queries. The connectivity becomes very obvious very quickly after we realize that we first sorted the node labels and then asked of adjecent ones. Every pair of adjecent node labels being connected means the whole set is.

Subtask 3

Even thoough it is not necessary for full points we can use Gaussian elimination to solve this subtask. Every query forms some equation (with variables being edge weights in respective trees). Now we get two linear systems, one for each tree and we just solve each independently.

`https://cp-algorithms.com/linear_algebra/linear-system-gauss.html`

Subtask 4

It is not hard to modify algorithm for constructing the small tree to reconstruct the weights as well if we our queries are also sorted by the preorder. But it only holds for the second tree. To achieve this for both trees we need to find a set of node labels such that they can be ordered in a way that they form a monotonic sequence with respect to the preorder in the first tree, but also in the second tree.

One could always choose such subset because $K^2 \leq N$ holds.

If we order node labels with respect to preorder in tree T_1 and look at the sequence of their preordering positions in the tree T_2. We can find a monotonic subsequence of length K.

Theorem 0.3 (Erdos-Szekeres)**.** *For given positive natural numbers s,r, any sequence of distinct real numbers whose length is at least N = s r + 1 contains a monotonically increasing subsequence of length s + 1 or a monotonically decreasing subsequence of length r + 1 (or both).*

Full Points

Method 1

For a rooted tree we denote $d(x)$ as length of the path that starts at the root of the tree and ends at x.

Then our queries give equations of the form

$$d(x_i) - d(y_i) = w_i$$

Now, we construct a weighted directed graph by adding an edge from y_i to x_i with weight w_i and an edge a reverse edge with negative weight $-w_i$.

Now to find $d(x)$ we just find the length of the path from root to x in our new graph using dfs. Such path is guaranteed to exist because of the structure of queries.

When we reconstructed $d(x)$ for every x it is easy to output the answers.

This gives us an algorithm with complexity $\mathcal{O}(K + N \log N)$

Method 2

Alternatively, one could solve the system of equations by using Gaussian elimination faster by observing that each equations forms a path in a tree. Further more a path that from some node to the root of the tree (meaning *lca* of endpoints of the path is always on of them).

Algorithm 4 Algorithm

while Tree has more than one vertex **do**
  $v \leftarrow$ arbitrary leaf that is not the root.
  $eq \leftarrow$ path from that leaf (equation) that is the shortest.
  substract equation eq from every other equation that also has leaf v as endpoint.
  after substraction equations still form a path, just a different one.
end while
After we have finished the elimination we backtrack to get the desired weights

At first this gives us complexity $\mathcal{O}(K^2 + N \log N)$, but it can be sped up to $\mathcal{O}(K \log^2 K + N \log N)$ by keeping the set of equations that begin at every nodes and merging those sets when substraction equations. If we merge the two sets such that we move everything from smaller set to larger we get the desired complexity. It is convinient to keep equations in stl Set structure to get the minimum path.