

Minimums on Arcs

Problem author and developer: Alexander Babin

Subtask 1.

In this subtask, the permutation is restored as follows:

$$p = [1, f(1, 2), 2, f(2, 3), 3, \dots, n-1, f(n-1, n), n]$$

Subtask 2.

In this subtask, it was possible to make all possible queries $f(i, j)$ in advance, and then restore the permutation in $O(n^2)$ time. This can be done as follows:

- Let $g(x)$ be the number of such y ($1 \leq y \leq n$) that $f(x, y) = 1$.
- It is clear that any element $x \neq 1$ is at distance $\frac{n+1}{2} - g(x)$ from one.

Place one at position $\frac{n+1}{2}$ in the permutation and arrange arbitrarily the elements that must be at distance exactly 1 from it. This can be done because the function f does not change if we apply a cyclic shift or a reversal to the permutation.

For each element x at distance $\leq \frac{n-3}{2}$ from one, we can uniquely determine in which half of the permutation it lies: it is enough to check whether $f(p_{\frac{n-1}{2}}, x) = 1$. If this equality holds, then the element lies in the right half; otherwise, in the left half.

The positions of elements p_1 and p_n can then be brute-forced, and the correctness of all queries can be checked naively. In some cases, it is impossible to uniquely restore their relative order.

Subtask 3.

This subtask guaranteed that solutions better than quadratic could score at least some points.

Solution for 70+ points. (Randomized, complicated, but provable)

The solution can be split into three phases:

Phase 1. Splitting elements into two halves relative to one.

We will maintain 5 sets of elements: L — the set of elements to the left of one, R — the set of elements to the right of one, $L' \subset L$, $R' \subset R$, and C — the set of elements at distance $\frac{n-1}{2}$ from one. The sets L' and R' will contain some subset of the elements closest to one from the sets L and R .

Choose a random element x for which we have not yet been able to determine on which side it lies:

- If $L = R = \emptyset$, then form the set $R = R' = \{y \mid f(x, y) = 1\}$. Otherwise, it is guaranteed that $L' \neq \emptyset$ and $R' \neq \emptyset$ (see the next item), and then we can form the pair of sets $L(x) = \{y \in L' \mid f(x, y) = 1\}$ and $R(x) = \{y \in R' \mid f(x, y) = 1\}$.
 - If $L(x) = R(x) = \emptyset$, then the element x is at distance $\frac{n-1}{2}$ from one and is a *corner* element, so it is added to the set C , and the next step of the algorithm is not performed.
 - If $R(x) \neq \emptyset$, then $L(x) = \emptyset$. Therefore, we can set $R' \leftarrow R(x)$ and assign x to the left half.
 - If $L(x) \neq \emptyset$, then $R(x) = \emptyset$ and we should set $L' \leftarrow L(x)$.
 - It is not necessary to build both sets $L(x)$ and $R(x)$ simultaneously; one can first check one of them for emptiness, and only if necessary build the second set.
- Suppose we have determined that x lies in the left half (that is, in the previous item we updated R'). Then take an arbitrary random element $y \in R'$ after the update. Form the set $L(y) = \{z \notin L \cup R \mid f(z, y) = 1\} \cup \{x\} \cup L$, and set $L \leftarrow L(y)$. If $L' = \emptyset$, then set $L' \leftarrow L$.

- Proceed analogously if x lies in the right half.

Thus, if we repeat the two items above, the values $2, \dots, n$ will eventually be split into 3 groups L, R, C — left values, right values, and corner values. Note that during the execution of the algorithm, the sets $L(x), R(x), L(y), R(y)$, whenever nonempty, coincided with $\{z \mid f(x, z) = 1\}$ or $\{z \mid f(y, z) = 1\}$ (proof left as an exercise). Moreover, one can show that at each operation the quantity $n - |L(x)| - |R(x)|$ decreased by a factor of two on average, and the size of L' or R' also decreased by about a factor of two. Therefore, the first phase requires $O(n)$ queries. Of course, one can perform a more careful analysis and compute the expected number of queries rigorously.

Denote $G(x) = \{y \mid f(x, y) = 1\}$. Since for some elements we have already determined $G(x)$, and for all elements we have already determined on which side of one they lie, for such elements we can already uniquely restore their position. Moreover, the different values of the computed sets $G(x)$ for $x \in L$ allow us to split all elements y into groups of consecutive values on the cycle. Indeed, if $G(x_1) \subset G(x_2)$, $x_1, x_2 \in L$, then all elements from $G(x_1)$ are closer to one than the elements from $G(x_2) \setminus G(x_1)$.

Phase 2. Refining the positions of elements from L and R .

Split all elements from R into blocks $L \setminus G(x_1), G(x_1) \setminus G(x_2), G(x_2) \setminus G(x_3), \dots, G(x_{k-1}) \setminus G(x_k)$, where $x_i \in L$ and $G(x_k) \subset G(x_{k-1}) \subset \dots \subset G(x_1) \subset L$; let these be the blocks B_1, \dots, B_k . For each block B_i , one can determine the set of positions occupied by the values from B_i , and these sets of positions do not intersect for different blocks. Moreover, it is clear that if some block contains exactly one value $x \in B_i$ whose position is unknown, then it can be restored immediately.

Thus, the more values $G(x)$ ($x \in L$) we know, the more precisely we can place the elements from the right half. The values $x \in L$ can be split into blocks in exactly the same way, guided by the computed sets $G(y)$ ($y \in R$).

If we choose a random $x \in L$ whose position is still unknown, then it is not necessary to check all $y \in R$ in order to build $G(x)$. First, there are values $x' \in L$ for which we know $G(x')$ and which are guaranteed to be either to the left or to the right of x (based on the partition into blocks of the left half), so from this we can conclude $G(x) \subset G(x')$ or $G(x') \subset G(x)$ depending on their relative order. Second, knowing the values $G(y)$ ($y \in R$), we can also narrow the interval of values for which it is unknown whether they should lie in $G(x)$ ($x \in G(y) \Leftrightarrow y \in G(x)$ + some information is known about the relative order of blocks and already placed elements).

After these prunings, only some interval of blocks from the right half remains, B_l, \dots, B_r , for whose elements it is unknown whether they lie in $G(x)$ or not. First there will be blocks that do not lie in $G(x)$, then a block that lies partially in $G(x)$, and then blocks that are fully contained in $G(x)$. Therefore, we can use binary search and determine in $\log_2(r - l)$ the pair of blocks that may partially lie in $G(x)$. Then, in time proportional to the sum of the sizes of these blocks, we determine which block is partial and split it into a pair of blocks.

Thus, we have devised a fairly query-efficient way to split blocks into even smaller blocks. The proposed solution randomly chooses an element x for which $G(x)$ is unknown (elements from L and from R are chosen with equal probability), and then performs the described block-splitting procedure. Eventually all blocks are split into blocks of size 1, and victory is achieved. Of course, the sets $G(x)$ are not maintained directly; only the partition into blocks is computed.

This part can be implemented to run in expected $O(n \log n)$ time. For this, it is enough to work in time $O(|B_x| + |B_a|)$, where $|B_x|$ is the size of the block containing x , and $|B_a|$ is the total size of the blocks from the other half that turned out to be relevant for consideration.

Phase 3.

After the third phase, all values p_2, \dots, p_{n-1} are restored ($p_{\frac{n+1}{2}} = 1$); it remains only to place the elements from C (corner elements) into positions p_1 and p_n . This cannot always be done uniquely, but it is claimed that it is sufficient to consider the values $f(c, p_2)$ and $f(c, p_{n-1})$ ($c \in C$) and from them restore any suitable arrangement of the corner elements.

It is not easy, but more or less intuitively, one can show that the expected number of queries of such a solution does not exceed $O(n \log n)$.

Solution for 85+ points. (Information theory)

Suppose we are in the second phase of the algorithm. Essentially, we have already determined C — the corner elements (at distance $\frac{n-1}{2}$ from one), determined the positions of some elements, and split the remaining elements into blocks B_1, \dots, B_k , where each block consists of some values for which it is known that they occupy some set of consecutive positions on which no restored permutation elements have yet been placed (some already found element may lie between two possible positions for one block), and these sets of positions do not intersect.

Then, given this information, there are

$$2^J = |B_1|! \cdot |B_2|! \cdot \dots \cdot |B_k|!$$

possible permutations, and the amount of information J is therefore

$$J = \sum_{i=1}^k \log_2 |B_i|! = \sum_{i=1}^k \sum_{j=1}^{|B_i|} \log_2 j.$$

In many problems, a heuristic based on information theory is applicable — it is enough to choose a greedy strategy that at each step maximizes the ratio of the decrease in the amount of information to the number of spent queries, i.e. maximize the expected value of

$$\frac{-\Delta J}{\Delta q}$$

over all allowed steps, where ΔJ is the amount of information after the next step, and Δq is the number of queries spent on this step.

In our case, a step consists of choosing some element x with unknown position, and then finding $G(x)$ for it ($G(x) = \{y \mid f(x, y) = 1\}$), after which we learn the position of the element x and also split some block into two parts.

Let us compute the number of queries and the decrease in the amount of information that occurs if we determine the position of the element with number i :

- $-\Delta J = \log_2 |B(i)| - \log_2(X!) - \log_2(Y!) + \log_2((X + Y)!)$, where $|B(i)|$ is the size of the block containing element i , and X and Y are the sizes of the blocks into which the block is split after locating block i .
- $\Delta q \approx X + Y + \log_2 M(i)$, where X and Y are the same quantities as in the previous item, and $M(i)$ is the number of blocks that can be split by an element x' lying in the same block as x .

Accordingly, the expected value of the ratio of obtained information to spent actions $E[-\frac{\Delta J}{\Delta q}]$ for a particular block B_j is equal to the arithmetic mean of $-\frac{\Delta J}{\Delta q}$ over all possible indices $i \in B_j$ where elements from B_j may lie. And the greedy strategy will be to take a random value from the block in which this expectation is as large as possible.

It remains only to learn how to recompute all the described values. Surprisingly, all this can be done naively; the total amount of changes in ΔJ and Δq over all indices turns out not to be very large.

Solution for 100+ points. (Cartesian tree)

Suppose we have split all elements into 3 groups L, R, C (see the first phase of the 70-point algorithm). Then we can spend $n \log_2 n$ queries to build a Cartesian tree on the elements of L and on the elements of R . A Cartesian tree is a tree built for an array $[a_1, \dots, a_n]$ as follows:

- Choose the minimum element by value, a_i ; it becomes the root of the tree, and the left and right children of this vertex are the Cartesian trees built for the subarrays $a[1 \dots, i - 1]$ and $a[i + 1, \dots, n]$.

It is easy to see that if we choose two values $x, y \in L$, then $f(x, y)$ will be equal to the LCA of vertices x and y in the Cartesian tree for the subarray of the permutation where the elements from L are located.

To build the Cartesian tree for the set L , we can gradually add elements to it in increasing order. Suppose we add an element x . Then we can identify a path from the root in the already built tree that always goes into the larger subtree. If this path ends at a leaf y , then after querying $f(x, y)$ we understand the LCA of vertices x and y , that is, the moment when we need to leave this path. Once we leave the path, we enter a subtree that is twice smaller, so we can repeat the query; in $O(\log n)$ queries we can understand where the vertex x should be attached. The only nuance is that in the built tree it is unclear which children are left and which are right, but even without this information the built tree helps speed up the second phase of the algorithm.

Note that when some block is split, or when we learn the value of some element from a block and need to cut it out of the Cartesian tree, we can recompute everything without any problems. It is enough to split the original Cartesian tree into the minimum possible number of trees, and then connect the roots of the resulting trees into one tree using the same algorithm.

Now let us consider why the Cartesian tree can help at all. Suppose we have an element x and a set of blocks with built trees B_1, \dots, B_n , and at the current moment we are splitting some block B using the element x . Denote $a(v) = 1$ ($v \in B$) if $f(x, v) = 1$, and $a(v) = 0$ otherwise. Then we have the following prunings:

- If v is the root, and l and r are some vertices from its different subtrees, then if $a(v) \neq a(l)$, we have $a(v) = a(r)$.
- If v is the root, and one of its subtrees contains some y with $a(y) \neq a(v)$, then we can separate from the tree the child of v whose subtree contains the vertex y , and uniquely determine which tree is to the left and which is to the right (that is, additionally split everything into blocks).
- It is advantageous to compute the values $a(v) = f(x, v)$ in decreasing order of subtree sizes of v .

This pair of prunings works well because it splits a block not into one block, but into at least $\Omega(\log(n))$ blocks in expectation, if we split blocks at random places. Moreover, this method gives some guarantees as well (handwaving follows, but still):

- We split the tree at a random place, so most often $f(v, x) = 1$ and $f(v, x) \neq 1$ will occur with roughly comparable probability. Therefore, the pruning from the first item will work very often, and we will make few queries on a balanced tree.
- If, on the other hand, we split the tree in a very uneven proportion, then we will make many queries, but many edges will also be cut, because we will descend quite deeply.