Solutions to 15-295 Contest of March 20, 2024
`https://vjudge.net/contest/617627`

Problem A

Count the number of indices where $a_i = i$. Let the count be cnt.

- If this number is even, we can just swap pairs of them to achieve the answer in $\frac{cnt}{2}$ operations. Note that we need at least this many, since each operation only touches 2 elements, and with fewer than $\frac{cnt}{2}$ operations there would be some bad index we don't touch.
- If this number is odd and greater than 1, then we can again swap pairs of them until we get to the last three. Then we just do $[1, 2, 3] \to [2, 1, 3] \to [2, 3, 1]$ in two swaps, achieving the answer in $\frac{cnt+1}{2}$ operations. We can use the same argument from above to show that this is optimal.

If $cnt = 1$, let i be the index where $a_i = i$. If all other $a_j = i$, then it is clearly impossible, because we will always have $a_i = i$. Otherwise, we swap a_i with some other $a_j \neq i$, and achieve the answer in one swap.

This gets us a $O(n)$ solution.

Problem B

If $k > n^2$, the answer is clearly `NO`. From now on assume $1 \leq k \leq n^2$.

If n is even, then note that the board will be split into $\frac{n^2}{4}$ orbits under rotations by multiples of 90 degrees, each one consisting of 4 squares.

In each orbit, all squares must have the same color. So we must have k divisible by 4.

The case for odd n is similar, just that we have an extra orbit of size 1. So the answer is `YES` iff $k \equiv 0$ or $k \equiv 1$ modulo 4.

Implementing this check is just $O(1)$ per test.

Problem C

If n is odd, then we can do the following construction.

- For each $i = 1, 2, 3, \ldots, n$ in sequence, fill in the cell $(i, 1)$.
- For each $i = 1, 3, 5, 7, \ldots, n$ in sequence, and for each $j = 2, 3, 4, 5, 6, \ldots, m$ in sequence, fill in the cell (i, j).
- For each $i = 2, 4, 6, 8, \ldots, n-1$ in sequence, and for each $j = 2, 3, 4, 5, 6, \ldots, m$ in sequence, fill in the cell (i, j).

We are basically making a kind of repeated "E" shape, then filling in the gaps.

If m is odd, we can swap n and m, use the same construction, then rotate the grid.

If both n and m are even, the goal is impossible. The proof is by looking at the perimeter of our black region. Let P be the perimeter of our black region. After the first operation, $P = 4$. Each subsequent operation either increases P by 2 (if we perform the operation on a square with 1 black neighbor) or decreases P by 2 (on a square with 3 black neighbors). At the end, we need P to be $2(n+m)$, which is divisible by 4. In other words, at the end, $\frac{P}{2}$ must be even.

But, we start with $\frac{P}{2}$ even, and each operation changes the parity of $\frac{P}{2}$. Since we perform $nm-1$, an odd number, of operations, the perimeter at the end must be odd. Contradiction.

So we can construct the answer, or report that it is impossible, with $O(nm)$ work per test.

Problem D

The most important thing to note is that for any integers x, y, z where $x \leq y$, we will have $f(x,z) \leq f(y,z)$.

This motivates the following. Root the tree arbitrarily. We will split any candidate path into two parts: one going up the tree, and one going down.

To compute the "up" part, for each vertex v, compute the path with the longest weight starting in some subtree of v and ending at v. Also compute the path with the longest weight which goes down a different subtree of v (if v is a leaf or has no children, this might be weight $-\infty$).

Then, we can use the “up” part to compute the “down” part. Run a DFS on the tree; when we are at a vertex v, we keep track of the greatest weight of a path which starts outside the subtree of v and ends in v. If we want to move from v to u, where u is a child of v, then we check what the maximum weight of a path is that starts in a subtree of v. If that subtree is the subtree containing u, we instead use the second-greatest weight. Max our current greatest-outside-subtree weight with this new subtree weight (denote the result as x), and we can proceed with our DFS into u, with a greatest-outside-subtree weight of $f(x, a_u)$.

Complexity is $O(n)$.

Problem E

Note that since q is small, we can compute the value of a after the v_i-th swap (this can be done in $O(n+m)$, for example, by cyclically shifting the indices swapped, then cyclically shifting a at the end). Denote this value as a_{v_i}, and count the number of positions where a_{v_i} differs from b.

- If this count is 0, the answer is `YES`; we can just do nothing and we will have $b = a_{v_i}$.
- This count cannot be 1; two permutations cannot differ at exactly one index.
- If this count is 2, then the parity of a_{v_i} and b are different. In effect, by changing at most one swap operation, we are performing two swaps on the permutation, so we cannot make $a_{v_i} = b$. So the answer is `NO`.
- If this count is 5 or more, then the answer is `NO`. Note that we are in effect performing two swaps, and each swap touches at most two new elements. So we can change the values of 4 elements in the permutation at most.

This leaves the case where 3 or 4 positions differ. In this case, we can just brute-force which two swaps we want to do. Note that both of these swaps can only touch indices where the positions differ — this is true when we have 4 differing positions because we can touch at most 4 indices and we need to touch 4 indices. And when we have 3 differing positions, if one of the swaps touches some non-different index, the other swap must as well, and then we would only touch 2 or fewer of the differing indices.

When we are checking if a pair of swaps works, we only care about the values that we are swapping. Check if one of the swaps we need was performed before, and if it was, we can replace that one, and the answer will be `YES`.

The overall complexity will be $O((n+m)q)$ per test.

Problem F

The minimum spanning tree has weight $n-1$, and is achieved by connecting each vertex $2 \leq v \leq n$ to vertex 1.

The maximum spanning tree can be computed in $O(n \log(n) \alpha(n))$ as follows. Iterate from $i = \left\lfloor \frac{n}{2} \right\rfloor$ down to 1. For each $1 \leq j \leq \left\lfloor \frac{n}{i} \right\rfloor$, check if adding an edge between vertices i and ij would create a cycle (for example, with a path-compression DSU). Add the edge if it would not.

This is just Kruskal's algorithm. We are exploiting the fact that we can iterate over edges with the same weight in any order to skip over the edges that we know would create a cycle.

If $k < n-1$ or k is greater than this maximum spanning tree weight, the answer is clearly `-1`.

Otherwise the answer always exists. A constructive proof:

Run Kruskal's as described above, until we get to an edge that would put us over the limit. More specifically, suppose that we are about to add an edge (i, ij), that the edges we have added so far have total weight W, and that we still have x more edges to add after this one. If we add in this edge, the minimum possible weight of the spanning tree after that will be $W + i + x$. If this value does not exceed k, we proceed as usual. If this value is strictly greater than k, then we need to do something else.

Let $y = k - W - x$. Note that $W + i + x > k \Rightarrow i > k - W - x$, so we have not added any edges with vertex y as an endpoint yet. We will add an edge from $(y, 2y)$ and then connect all the remaining x components with edges to vertex 1.

The total weight of this spanning tree will then be $W + y + x = W + k - W - x + x = k$. Yay!

The overall complexity is just $O(n \log(n) \alpha(n))$, for computing the max spanning tree's weight, and constructing the answer itself.

Problem G

First note that if, for any $1 \leq i < n$, b_i is not a supermask of b_{i+1} (in other words, there is some set bit in b_{i+1} that is not set in b_i), then the answer is unachievable.

To see why, convince yourself that when we perform an operation on an index $1 \leq i \leq n$, the following becomes true:

- For all pairs of indices (x, y) where $1 \leq x \leq i$ and $i \leq y \leq n$, a_x will be a supermask of a_y.

It follows that, since we perform an operation on every index, we must have any element of b be a supermask of any other element that follows it. From now on assume that all b_i are supermasks of b_{i+1}.

The other important observation is the following:

- After we perform an operation on an index $1 \leq i \leq n$, no other operation can change the value of a_i.

This is because after performing the operation on i, any operation on some $1 \leq j < i$ will `AND` a_i with a supermask of itself. Similarly, any operation on some $i < k \leq n$ will `OR` a_i with a submask of itself. So if at some point in time, we have $a_i \neq b_i$, we cannot perform the operation on i.

Note also that if $a_i = b_i$ and we perform the operation on i, then this will not hurt us in the future. Since all elements of b are supermasks of those that come after it, when we the perform the operation on some $a_i = b_i$, it will not erase any important bits in the elements that come after it, or add any unnecessary bits to those that come before it.

So we have a greedy strategy: while there exists some $a_i = b_i$, perform the operation on it. There are many ways to maintain this efficiently. One, as an example, is to store (in deques), for each bit, of which elements have that bit set to 0 and which have it set to 1. Suppose we perform an operation on some element where that bit is 0 — then we just pop off all the elements who have bits set to one, and modify them. Similarly, when we perform an operation on some element where the bit is 1, we pop off all the 0s that come before it. Note that the only thing other than these deques we need to maintain is a set of all a_i that are equal to b_i. So keeping everything up to date isn't hard, just add elements to the set as they become equal.

How can we prove this works fast enough? Note that whenever some bit in an element gets changed, it can never change back — if it got changed to a 1, it would need to be `AND`ed with some element that comes before it. But since it got changed to a 1, that means that everything else before it also got `OR`ed into a 1. The case for 0 is similar.

So every element changes at most $\log(10^9)$ times, and this solution will work in $O(n \log(n) \log(a_i))$. As a bonus, if we use some BST-based set to store the indices where $a_i = b_i$, then we can also find the lexicographically minimal solution (if it exists).

Problem H

We can identify a valid starting position with two things:

- The point currently being used as a pivot.
- The point that will next be used as a pivot.

It's not hard to verify that two valid starting positions are equivalent from the definition in the problem statement, if and only if these two points are the same for them.

So we need to consider n^2 valid starting positions. Note that every starting position will always evolve into a unique other starting position, and that every starting position evolves from a unique other starting position.

So if we think about the digraph of starting positions and which evolves into which, we will have a permutation graph. We can compute this permutation graph in $O(n^2 \log(n))$ with an angular sweep over the points to find the next new pivot for each starting position.

So the problem reduces to the following. We are given a sequence $e_1, e_2, \ldots, e_\ell$ of edges, and we want to compute the spanning tree generated by greedily taking edges, for each cyclic rotation of e.

If you want to cause yourself pain, you could try doing this with a link-cut tree in $O(n^2 \log(n))$ (and in fact as I write this, rainboy has managed to make one such implementation pass in 3.7 seconds), but there is a much faster way.

The crucial observation is that every two adjacent edges in this sequence e will share an endpoint. This means that for any prefix of edges in e, the spanning tree over that prefix will have exactly one nontrivial connected component.

So, as we iterate from left to right in e, we should add an edge into our spanning tree if and only if it introduces a new vertex, i.e. iff one of the edge's endpoints hasn't appeared before (otherwise it will connect two vertices which are already in the same component).

This motivates the following. For each vertex, we will store the indices where it appears in e in a deque. Add in all the edges in e that introduce a new vertex.

As we iterate through the cyclic shifts, we will repeatedly be popping an edge from the front of e and pushing it to the back. For each endpoint of the edge we just popped, we can just find the next place where it appears in e using our deques, and update the spanning tree accordingly.

This is linear in the total number of starting positions. Since we still need the angular sweep at the beginning, this is $O(n^2 \log(n) + n^2) \in O(n^2 \log(n))$ overall, the same as the LCT solution, but the constant factor should be much lower.