Problem Tutorial: "Assignment Problem"

Let us fix the matrix A. Let's then take a matching such that we did not take the best candidate for any position. This matching can be strictly improved because there is an alternating cycle (or chain) which uses only better edges. So, there must be a position for which we choose the best candidate. Let's try all the positions, remove it and the corresponding best candidate, then apply the argument recursively.

Basically we are choosing the permutation of positions, and then take the best candidate for this position which is still available, in order. That is $O(m!poly(m))$, with appropriate precalculations it can be implemented in $O(m!\frac{m(m+1)}{2\cdot 17})$.

There is also DP[used positions][used candidates] with the same transitions (basically, memorize the same states in recursive solution), but we were not able to prove that it has better complexity.

Problem Tutorial: "Lockout vs tourist"

The set of available cards determines the game completely, so we will do a DP on subset of initial cards. To make a transition, we will use two numbers for each i: a_i (what will be the result if we choose this problem and tourist doesn't, given in input) and b_i (what will be the result if both of us choose this problem, calculated as DP for subset without this problem).

This should be solved as independent game. From the underlying logic how b_i are calculated, we know that if $a_i < a_j$ then $b_i > b_j$ and there is at least one pair with $a_i \geq b_i$. Let's use Minimax theorem. tourist's strategy is a vector of probabilities to choose each problem, let's denote the probability to choose i-th problem as p_i. According to Minimax theorem, value of the game is equal to $\min_p \max_i (a_i \cdot (1-p_i) + b_i \cdot p_i)$. The limitations are $p_i \geq 0$ and $\sum_i p_i = 1$. We can see that if we want the value to be at most C, we should choose p_i such that $(a_i \cdot (1-p_i) + b_i \cdot p_i) \leq C$. Let's start with $p_i = 0$, then decrease C and look how p_i should change. We'll stop when either $\sum_i p_i = 1$ or $C = a_i \leq b_i$. This can be implemented in $O(n)$, thus whole solution works in $O(n2^n)$ time.

Problem Tutorial: "Multiple?"

Let's find a number which appears the most number of times in the sequence. Let this number be x and total number of occurrences of all other number be t. It is well known that the maximal number of pairs one can construct from elements of the sequence such that numbers in each pair are different is $\min(t, \lfloor\frac{n-k}{2}\rfloor)$. Let's construct that many pairs, then write all the elements from our sequence such that each pair is two consecutive numbers. Let's then calculate prefix sums modulo n in this order. All of them should be distinct, otherwise we could find a subsegment with sum divisible by n. Let's now swap two elements from one of the pairs. This way we'll change exactly one prefix sum (that was between this numbers). So we have $n - k + 1 + \min(t, \lfloor\frac{n-k}{2}\rfloor)$ numbers which are the candidates for being prefix sums, and all of them should be distinct. Thus $t < k$. That means that number x occurs $n-k-t > n-2k \geq n-2\frac{n}{4} = \frac{n}{2}$ times. Then x should be coprime with n. Let's choose x, there are $\phi(n)$ ways to do that, and now divide all the elements by x.

Since we have at least $\frac{n}{2}$ ones, all other elements should be strictly less than $\frac{n}{2}$. Let their sum be at least $\frac{n}{2}$. Then we can choose their prefix such that sum on this prefix lies in $(\frac{n}{2}; n)$, then with correct number of ones we could make the sum equal to n. Now it is easy to see that sum of all the elements should be less than n. Obviously any such sequence will be good, and it automatically has more than $\frac{n}{2}$ ones. The number of such array can be calculated via stars and bars method and is equal to $\binom{n-1}{k-1}$.

Thus the answer to the problem is $\binom{n-1}{k-1}\phi(n)$.

Problem Tutorial: "Output Limit Exceeded"

Let's first solve the problem for $k \leq n/2$. If there are at least two prime numbers in the left part then we can't build the perfect matching, as both of them could be matched only to 1. For integers up to 10^{18} prime gap does not exceed 1500, thus the k theoretically could be provable only for $k \leq 3000$. We can iterate over all k, maintaining the graph and running additional dfs from Kuhn's algorithm. After k iterations graph will have $O(k)$ vertices and $O(k \log k)$ edges, thus Kuhn's algorithm will work in $O(k^2 \log k)$ time. Since the actual bound for k is much smaller, even $O(k^4)$ should be fine.

For $k > n/2$ it is always ok to match the numbers which occur in both parts with each other. Let's say we have matched $x - y$ and $y - z$, that means that x is divisible by z and we could match x with z instead. Thus the answer for k is the same as the answer for $n - k$.

Now we have to output the answer which has a form of [something][000 . . . 000][something]. Just generate strings of zeroes with lengths equal to powers of 2, and then combine them to construct the long string of zeroes in the middle, while prefix and suffix can be just written by hand.

Problem Tutorial: "Smol Vertex Cover"

First, we have to find maximum matching using Edmonds Blossom ($O(n^3)$) or any unproven algorithm of your choice. Any vertex cover should cover all the edges, that includes the edges of found matching. These edges are disjoint, which means that $M \le C$.

To achieve $C = M$ we have to choose exactly one vertex from each edge of the matching, so we have M choices with 2 options each. For each edge we should choose at least one of its endpoints, so every limitation has a form of disjunction. Therefore, it is an instance of 2-SAT problem which can be constructed and solved in $O(n^2)$.

In $C = M + 1$ case we can choose one more vertex in the cover. So, the cover will be either [one vertex not covered by matching + exactly one vertex from each edge of the matching] or [both vertices from one edge of the matching + exactly one vertex from all other edges of the matching]. We can try $n - 2M$ options for special vertex and M options for special edge, after that it will be 2-SAT instance again. Each instance can be constructed and solved in $O(n^2)$, thus whole solution works in $O(n^3)$.

Problem Tutorial: "Thanks to MikeMirzayanov"

In one operation we reverse the whole array, and then reverse the subsegments we have divided it into. Let's forget that we reverse the whole array and fix the parity afterwards.

Let's now assume that we have to sort array of 0s and 1s. Let's find the groups of consecutive 0s and 1s, starting with 0s (and group of size 0 in the beginning if needed). Let's reverse 2nd and 3rd groups together, then 6th and 7th groups together, then 10th and 11th and so on. We will actually swap these groups, thus we'll make the number of groups two times smaller. In $\lfloor \log_2 n \rfloor$ operations we'll sort the array.

Now let's return to sorting the permutation. Let's change all the elements smaller than $n/2$ to 0, and all others to 1. Now sort this array in $\lfloor \log_2 n \rfloor$ operations. Now we have all small numbers in the left part, while all big ones in the right part. Sort them recursively, and we can actually do it in parallel, thus getting the sorting done in at most $\lfloor \log_2 n \rfloor + (\lfloor \log_2 n \rfloor - 1) + \ldots + 1 = \lfloor \log_2 n \rfloor \cdot (\lfloor \log_2 n \rfloor + 1)/2 \le 120$ operations.

Problem Tutorial: "Remove the Prime"

The game is the sum of games for all primes on all maximal by inclusion segments that have this prime. If we find out the segments, we just have to calculate the xor of grundy values for all the segments lengths. The grundy value is just the length, but even if you have not noticed it, you can still calculate them in $O(n^2)$ time by using the fact that possible moves from segment of length L is a subset of possible moves from segment of length $L + 1$.

The problem is that we can't just factorize all the numbers. Let's do a scanline, maintaining all the primes whose segments go through the border. We will also maintain at most one additional segment which is actually two segments for two different primes, but we only know their product.

For each new number we can check if it is divisible by primes we already have, thus for each of them deciding if its segment should end here or should be continued. For the additional number which is a product of 2 primes we can calculate gcd with our new number and see if we can deduce the 2 primes (we can if gcd is not 1 and not the whole product). If we can, then we'll just replace this special segment by two segments for primes and proceed normally. If the gcd is 1, then both segments should end now, and that won't change the xor as we knew that these were two different primes with the same segments. In the case that is left we just continue this special segment. Now we have to almost factorize the number, possible leaving product of two different primes. To do that we'll check all the primes up to $C^{1/3} = 10^6$, and now we have either 1, p, p^2 or pq. Let's check for 1 and p^2 trivially, and check for p using Miller-Rabin test.

In every moment we have at most $O(\log C)$ open segments, so the solution works in $O\left(n\left(\frac{C^{1/3}}{\log C} + \log^2 C\right) + C^{1/3}\log\log C\right)$.

Problem Tutorial: "Excluded Min"

Let's define a function $f(S, x) = |\{z \in S : z < x\}| - x$. Then $F(S) = \max_{f(S,x)\geq 0} x$.

Idea: solve the problem offline by doing scanline by decreasing value of x, while maintaining $f(Q_i, x)$ for queries in some data structure. Ask the maximum value in that data structure, and if it is non-negative, we have found the answer for that query, we can now delete it from data structure and repeat.

To maintain such a data structure we need to add on rectangle, delete the element and take max. That's hard. But from the meaning of function f we know that if $S_1 \subset S_2$, then $f(S_1, x) \leq f(S_2, x)$. That means, that we can maintain data structure only for the segments which are not subsegments of any other not deleted segments, because they can't be a maximum.

Let's sort the queries by the left border. On active segments (which are not subsegments of any not deleted segments) the operation of removing an element is -1 on segment, we can do it using Segment Tree. With one more Segment Tree we will be able to maintain the set of active segments after deleting one segment in $O((cnt + 1)\log q)$ where cnt is the number of segments that became active. And when we insert new active segment, we have to calculate the actual value of $f(Q_i, x)$ at that moment, to do that we will also maintain Fenwick tree over the whole array (1 if element is active).

Whole solution works in $O((n + q)\log(n + q))$.

Problem Tutorial: "Trade"

If we fix the set of items to buy, it will be optimal to buy them in order from highest p_i to lowest because of Rearrangement Inequality. So we'll sort the items in advance.

If we have bought k items then we have paid at least $\sum_{i=0}^{k-1} i \cdot (k - 1 - i) = (k - 2)(k - 1)k/6$. Thus we know that $k \leq 2 + \sqrt[3]{6C}$.

So we can do $dp[p][k]$ and it will work in $O(n\sqrt[3]{C})$.

Problem Tutorial: "Increasing or Decreasing"

Let's sort the initial permutation in increasing order. Now we will go right to left, in each operation setting the right element to the current position. After $n - 1$ iterations whole permutation will be correct and we will be done in exactly n operations.

We'll maintain the invariant that after each operation all the suffices of not yet fixed elements contain the segment (consecutive elements) of yet unused elements. In other words, if we'll go right to left starting from current position, each element will be either minimum or maximum. If this is true, then it is easy to see how to set the right element on the current position: find it in the sequence, and then if it is minimum on its suffix, then sort this suffix in decreasing order, and if it is maximum, sort it in increasing order instead. It is easy to see that invariant is maintained thus the solution works.

Problem Tutorial: "Rectangle Painting"

In a nutshell, we want to solve the problem using the usual Segment Tree (ST). When we process change queries, we want to go down the tree in a normal fashion, do something in the nodes we have reached, and that's all. To make it work, we have to somehow define what is the answer for the segment, and learn how to merge the answers from two segments. The problem is that lower nodes might not know that they are fully covered by some segment with y coordinate since this query happened on bigger segment and when we processed that query we did not reach this node. Usually we deal with it by propagating the information down the tree, but in this problem information can have linear size, and it will be too slow.

We are going to resolve this issue in the following manner. Let's say that height y is important for node v of ST if we have reached node v when processing some change query on height y. Then in node v we will try to maintain answer (which is the highest black column on corresponding segment) assuming that we only have important for this node heights. For node 1 (root of the ST) all the heights up to q (as it is the maximum answer) will be important. We will also change the queries to the ST such that they do not intersect. To do that, for each height we will maintain the set of black segments, then after q queries number of queries to ST + number of black segments on all levels is bounded by $2q$.

Let's say that segment, corresponding to node v, is $[v.l, v.r]$. We will now introduce the system of colors for pairs (node, height).

- Height y is *white* for node v if all the cells from $[v.l, v.r]$ on height y are white, so we didn't have any queries on height y intersecting with segment $[v.l, v.r]$.
- Height y is *black* for node v if there is a query on height y which completely covers the segment $[v.l, v.r]$. Note that it is not the same as all the cells in corresponding segment being black. They could be covered by several segments side-by-side, in which case we don't call this height black.
- Height y is *gray* for node v if there is a query on height y which intersects with the segment $[v.l, v.r]$, but does not cover it completely. In other words, the height is gray if it is not white and not black.
- Height y is *bright white* for node v if it is white for this node and gray for its parent. For the root all white heights are bright white.
- Height y is *bright black* for node v if it is black for this node and gray for its parent. For the root all black heights are bright black.

Lemma 1. Height y is important for node v if and only if it is either gray, bright white or bright black.

It should be clear from how Segment Tree works. If we have arrived at some node v and $[v.l, v.r]$ is either doesn't intersect with query segment or completely covered by it, then its parent should be gray as its segment should intersect with query's segment but not be completely covered by it, otherwise we wouldn't go down from the parent.

In each node we will maintain two sets of heights: bright white heights and gray or bright black heights.

Lemma 2. Bright white height can only become gray or bright black (cannot become just white or just black).

Or in other words, height cannot stop be important.

Lemma 3. Standard procedure of processing query in Segment Tree visits all the vertices, for which we have to change its color for given height.

Follows from Lemma 1.

Lemma 4. The height of black column in position x is equal to minimum over all heights y which is bright white for any node on the vertical path from the leaf corresponding to position x in ST to the root.

It is obvious that the answer should be the minimum height y which is white for that leaf, but every such height is also bright white in some ancestor of that leaf.

Thus answer on segment should be maximum on the segment over the minimums of bright white vertices on vertical paths corresponding to x from segment. Let's now look at the segment corresponding to some node. All of the paths will go through this node and all its ancestors, so instead of calculating minimums on each path and then taking maximum of them, we can calculate minimums on part of each path below this node, take maximum of them, and then update it with minimums from this node and its ancestors. After a closer look we can understand that we can define answer for the node as maximum of minimums on paths to all the leaves just from this node, and it will be exactly what we wanted in the beginning: answer assuming that only important heights exist. It is easy to recalculate — take maximum of answers from sons, and then update it with minimal height which is bright white for this node. We have learned how to define answer for a segment and merge two such answers, now we will be able to write a Segment Tree in a usual fashion.

That's the solution, it works in $O(q \log^2 q)$ time and $O(q \log q)$ memory.

Problem Tutorial: "Extreme Wealth"

The answer is $\frac{2^{B+R}}{\binom{B+R}{B}}$. To see that one can either write down the DP transitions and prove the formula by induction, or use the general theory: It is written in the statement that we can't bet on both Red and Black, but it actually doesn't matter since equal bets cancel each other. So instead we can bet all the money in some proportions. Now we can see the process in this way: split all your initial money into $\binom{B+R}{B}$ parts by the sequence of plays, and for each part independently bet it on the right outcome for this sequence of plays. In the end the correct sequence will be multiplied 2^{B+R} times, while all the other will be lost. The best we can do is to split the money equally thus getting $\frac{2^{B+R}}{\binom{B+R}{B}}$ in the end.

Now the real problem starts: how to calculate this number precise enough. Let's first try to do it for $B = R$. For small B we can calculate it naively, while for big B we'll use Stirling's approximation for factorials, because all the inconvenient factors will cancel out for $B = R$:

$$\frac{2^{B+B}}{\binom{B+B}{B}} = \frac{2^{2B}(B!)^2}{(2B)!} = \frac{2^{2B}(\sqrt{2\pi B}B^B e^{-B}(1+\frac{1}{12B}+O(B^{-2})))^2}{\sqrt{2\pi 2B}(2B)^{2B}e^{-2B}(1+\frac{1}{12\cdot 2B}+O(B^{-2}))} = \sqrt{\pi B}(1+\frac{1}{8B}+O(B^{-2}))$$

Now let's assume $B \le R$, start with answer for (B, B) and then increase second parameter until reaching R:

$$\frac{2^{B+(R+1)}}{\binom{B+(R+1)}{B}} = \frac{2(R+1)}{B+R+1} \cdot \frac{2^{B+R}}{\binom{B+R}{B}}$$

If we skip first $\sqrt{B}$ operations, each next $\sqrt{B}$ operations will multiple the answer by at least 2, which means that after $\sqrt{B}\log 10^9$ operations the answer will be larger than 10^9, so the solution works in $O(\sqrt{B}\log C)$ time.

Problem Tutorial: "Discrete Logarithm is a Joke"

$a_{n+1} = \log_g a_n$, which means that $a_n = g^{a_{n+1}}$. If only some kind soul would tell us $a_{1000000} \ldots$