Problem A. Anime

Let's imagine that we are watching moment t and we fix some dt as very small interval that we about to watch. Obviously, that we will increase our interest by the value $I(t) \cdot dt$. If we take a limit of an infinite sum of these values we get a Riemann sum or simply an integral: $\int_0^n I(t) \cdot dt$. Due to special properties of function $I(t)$ it is easy to calculate this integral, just find an area of trapezoids.

Now we can add the usage of buttons in our solution. Let's again imagine that we are watching a moment t and time interval dt. Notice that we can "change"our moment to $t+k$ (only if $t+k \leq n$) watch interval dt there, and come back to moment t right after. So, we used forward button once and rewind button once. Using the same idea we can change any moment t to some moment t' if $|t-t'|$ is divisible by k. Obviously that we are interested in moment t' that has the biggest value of $I(t')$.

Consider a time interval $[0, 1)$, for some moment $t \in [0, 1)$ optimal t' will be in one of the following intervals: $\{[0, 1), [k, k+1), [2k, 2k+1), ...\}$. All of these intervals is a straight line, we can find maximum value of $I(t')$ by constructing a convex hull of these lines. After that it would be easy to calculate the area under such convex hull. We can build such convex hulls by dividing our lines in to groups according to their remainder modulo k.

Time complexity of the solution is $O(n \cdot log(n))$ because we need to sort our lines in order to build convex hulls.

Problem B. Random Interactive MST Bot

Let's use Prim's algorithm. For every node outside the component store a minimum edge to a node inside the component. Find a minimal edge and add a corresponding node to our component. Then check for other nodes if the edge between new node is minimal.

Notice, since weights are random number of updates will be small and all updates are *to decrease key*.

There are multiple structures that pass limits. Like:

- $k - heap$
- Storing minimum k elements in sorted order
- Build a directed graph on queries and ask random pair of nodes with no ingoing edges.

Problem C. Nomad Camp

Let for each pair of pastures $1 \leq u, v \leq n$, determine whether we can gather this pastures. Let it be boolean array $can_{u,v}$.

Claim: It is be possible to gather all the people if and only if $can_{u,v}$ is true for each $1 \leq u, v \leq n$.

Let $d_{v,t}$ be pasture u, with type t, which is closest to pasture v. It can be calculated by Dijkstra algorithm in $O(n^2)$ or $O(n^2 * log(n))$, if for each type we run the algorithm simultaneously from each node with this type.

Now we can build directed graph, from (u, v) make edge to $(d[u][t], d[v][t])$ for each t. It is easy to see that starting from (u, v) and by jumping through this edges, and eventually we come to some posture (x, x), postures u and v can be gathered together.

To calculate $can_{u,v}$ efficiently, we can solve the task backwardly. For each gathering (starting) point x, run DFS from pair (x, x), and go through reversed edge.

Total complexity would be: $O(n^2)$ or $O(n^2 * log(n))$ for each test case.

Problem D. Data Structures Master

Build Cartesian tree. To construct the Cartesian tree, set its root to be the maximum number in the sequence, and recursively construct its left and right subtrees from the subsequences before and

after this number. Now maximum of segment is lca of nodes that correspond to segment ends and $g(A_i, B_j, C_k) = a_{lca(A_i,B_j,C_k)}$.

Count $dp_v = a_v * b_v * c_v$, where a_v, b_v, c_v is there number of elements from A, B, C that in subtree v.

Node v is lca of exactly $dp_v - dp_l - dp_r$ triplets, where l, r is childs nodes of v.

Queries now look like this:

- Add one to a_v, b_v or c_v on the path from v to the root.
- Find sum of $a_v * b_v * c_v * (a_v - a_p)$ over all nodes.

It can be done with heavy-light decomposition and segment trees.

Problem E. Poisonous Labyrinth

Let O_v be the number of poisons in vertex v.

Let's build a new graph. A directed edge from x to y will be drawn if the poison of type y lies on the path between vertices where the poison of type x is located. Now, if we visit a vertex of poison x in this graph, we must visit all vertices y that are reachable from poison x.

How to build such a graph?

To do this, you need to use *Heavy-Light Decomposition* (HLD) and Segment Tree. Build a regular HLD on this tree, but give each vertex not one index in HLD, but O_v indices. On the new array obtained from HLD, build a segment tree. Now, for each poison, you just need to draw an edge to all poisons on certain segments in the resulting array (there will be $\log(n)$ of them), this is easily done using HLD and segment tree, approximately $O(\log^2(n))$ edges will be added for each vertex (on each of the $\log(n)$ segments, $\log(n)$ edges will be spent). To better understand how this is done, read articles about Heavy-Light Decomposition.

You can also build this graph using binary lifts.

Now we have a new graph. Let's condense this graph (i.e., compress the strongly connected components). Now we have a directed acyclic graph.

Let $D(s)$ be the minimum distance that needs to be covered if the vertex s was visited in the original tree. Let there be a vertex v in the new graph, before compression it was a strongly connected component. Let $V(v)$ be the set of vertices that were in the strongly connected component, which in the new graph is vertex v. Then let's notice that for any $a, b \in V(v)$, $D(a) = D(b)$ holds, since leaving vertex a we are obliged to visit vertex b and vice versa.

Let $Deg(v)$ be the number of outgoing edges from vertex v in the new graph.

Let $deg(v)$ be the degree of vertex v in the tree.

Now let's notice that if in the new graph vertex v is reachable from vertex u, then for any $x \in V(v)$ and $y \in V(u)$, $D(x) \leq D(y)$ holds, since we are obliged to visit vertex $xifwevisitedvertexy$.

From this it follows that it is not advantageous for us to start from vertices c such that $c \in V(k)$ and $Deg(k) > 0$.

Let's list all h such that $Deg(h) = 0$, for each such h we need to calculate $D(h)$. Write down all such h in an array r.

Let $T(r_i)$ be all vertices of the original tree that must be visited for component r_i, they will obviously form a connected component inside the tree, let's call it a subtree. Then let's notice that for different r_i and r_j, $T(r_i) \cap T(r_j) = \varnothing$ holds. Therefore, for each r_i, we can find $D(r_i)$ in $O(|T(r_i)|)$, since $\sum\limits_{v \in r} |T_v| \leq n$.

Create new trees and solve the following problem on each tree:

"How many moves are needed to visit all vertices of the tree and return to the starting vertex?"

The answer to this problem will be the sum of the weights of all edges in the tree multiplied by two.

Now simply take the minimum answer among all the obtained trees, and this will be the answer to our problem.

The final time complexity is $O(m \log^2 m + n)$ or $O(n \log n)$ (depends on what you will use to build the graph).

Problem F. Geometry Enjoyer

The most important observation is the fact that we can easily restore the extension of the sides of the initial polygon. The line that goes through some two points from the input is the side of the polygon if and only if there are $k-1$ points on this line. Let's call these lines good.

We can do it in $O(n^3)$ time by trying every possible line and counting the number of points that lie on this line.

Now, we want to restore the initial vertices of the polygon. One of the simplest ways is to sort our good lines by angle and find the intersection of two consecutive lines. However, this is not possible due to large input values (calculating the cross product will result in overflow even if we use `int128`, and `long double` can have precision errors).

Another way is to notice that our lines divide the plane into convex polygons, and one of them is the initial one. We can make an edge between adjacent points on the line and then use these edges to traverse our points. The first idea that comes to mind is to traverse edges in clockwise or counterclockwise order, however, due to overflows and precision error it is hard to do. Fortunately, there are at most 20 good lines and 190 points. Thus, we can use a recursive approach and traverse edges in every possible way. At some point we will find a cycle of length k that uses every single good line, this cycle is the initial polygon. Overall complexity: $O(n^3 + n \cdot 2^k)$.

Problem G. Hocolate Hame

Let us first obtain slow solution in $O(n^3)$ with dynamic programming. Let $dp[p][l][r][k]$ be optimal answer on segment $[l, r]$ with player p taken k on the previous turn ($p = 0, 1$). Then we can calculate this value by considering $dp[p^1][l+k][r][k]$ and $dp[p^1][l+k+1][r][k+1]$ in $O(1)$.

We now optimize this approach to $O(n^2)$ as follows:

- First, observe that maximal k we can archive is $O(\sqrt{n})$, since

$$1 + 2 + \ldots + k \le n = k(k+1)/2 \implies k \le \sqrt{2n}.$$

  Thus we can reduce time complexity to $O(n^2\sqrt{n})$.

- Further, we note that the difference $r - l$ does not exceed $2x$ after x turns, since each two moves the difference can only be increased by 1.

So we can store all **really** possible pairs (l, r) (the number of them would be $O(n\sqrt{n})$) and perform above dynamic programming on these pairs only with k restricted to $\sqrt{n}$.

Time and memory complexity: $O(n^2)$.

Problem H. Lost Table

To solve the problem, we will use the "inclusion-exclusion"technique. Let's start by simplifying the problem. Suppose we need not the condition of equality of the maximum in a row or column, but the inequality: $max \le a_i$ or $max \le b_i$. That is, the maximum in a row or column should not exceed a certain constant. In this case, for each cell (i, j), we can find the maximum possible value it can take, which is $min(a_i, b_j)$. So, for cell (i, j), there are $min(a_i, b_j)$ choices for the value. The number of tables satisfying the simplified problem is: $\prod_{i=1}^{n} \prod_{j=1}^{m} min(a_i, b_j)$. Let's consider this value as the initial answer for convenience.

Now we need to use the "inclusion-exclusion"technique: let's consider in which rows the condition is NOT guaranteed to be met and in which columns it is NOT guaranteed to be met. It is not difficult to "almost correctly"count all such tables: first, subtract one from $a_i (a'_i = a_i - 1)$ if this row is in the selected set (otherwise leave it unchanged), and do the same for the columns. As a result, using the same formula as above, we can count all such tables: $\prod_{i=1}^{n} \prod_{j=1}^{m} min(a'_i, b'_j)$. Unfortunately, it is possible that the condition is not met for more than x rows and y columns. Fortunately, this can be used for the "inclusion-exclusion"technique: consider all possible subsets of rows and columns, calculate the answer for them, add it to or subtract it from the answer depending on the parity of the size of the subset $(x + y)$. This is enough to write a slow solution.

To optimize for time, let's get rid of "inclusion-exclusion"for columns. Let's introduce a function: $f(b_j)$. It will calculate the answer for column j if the maximum in it is not greater than b_j. $f(b_j) = \prod_{i=1}^{n} min(a'_i, b_j)$, then we can expand $min(a'_i, b_j)$ and write it separately for the cases when $a'_i \le b_j$ and $a'_i > b_j$. For convenience, we can sort the array a'. Then $f(b_j) = b_j^{n-k} \cdot \prod_{i=1}^{k} a'_i$ (where k is the length of the maximum prefix of the array a' where the values are less than or equal to b_j). In this case, the answer for the column is: $f(b_j) - f(b_j - 1)$. Thus, we have eliminated "inclusion-exclusion"for columns, and now we can calculate the answer using "inclusion-exclusion"ONLY for rows. But this solution is still slow.

For further optimizations, we need to notice how changing one element of the array a affects the function f. Let's consider two cases. If $a_i > b_j$, then the value of the function $f(b_j)$ does not change after decreasing a_i by one; otherwise, the value of the function is multiplied by $\frac{a_i - 1}{a_i}$. However, since the answer for the column is $f(b_j) - f(b_j - 1)$, in the end there will be three variants of how the answer for the column will change.

1) $a_i > b_j$: nothing changes

2) $a_i = b_j$: $f(b_j) - f(b_j - 1) \to \frac{a_i - 1}{a_i} \cdot f(b_j) - f(b_j - 1)$

3) $a_i < b_j$: $f(b_j) - f(b_j - 1) \to \frac{a_i - 1}{a_i} \cdot [f(b_j) - f(b_j - 1)]$

Let's introduce another function $P(x, k)$. This function calculates the answer if we choose k rows with values a_i equal to x and decrease them by one. To calculate it, we need to know the number of values b_i that are strictly greater than a_i (let it be bg) and equal to a_i (let it be eq). It can be noticed that then the initial answer needs to be multiplied by:

$$\left(\frac{x-1}{x}\right)^{k \cdot bg} \cdot \left(\frac{\left(\frac{x-1}{x}\right)^{k} \cdot f(x) - f(x-1)}{f(x) - f(x-1)}\right)^{eq}$$

Now we can understand that these changes are independent for rows with different values of a_i, and to take into account the change in several different values of a_i, it is enough to multiply the functions P. Then we can use inclusion-exclusion for one value of rows, more formally, the change in the answer for a specific value x looks as follows:

$$\sum_{k=0}^{cnt} (-1)^k \cdot P(x, k) \cdot C_{cnt}^{k}$$

Finally, it is enough to find the product of these sums for all values from the array a and multiply it by the initial answer.

P.S. All calculations are performed modulo $10^9 + 7$. The MTF is used for divisions and fractions. Also, by changing the answer, we mean not the difference, but the ratio.

Problem I. Team Training

Let's choose n positions for the first team. For fixed n positions $1 \le pos_1 < pos_2 < \cdots < pos_n \le 3n$, starting from the last one, we will delete segments $[pos_i, pos_i + 3)$ from the array. The chosen positions are valid if and only if on every iteration we can delete the segment(the segment lies in the current array

and doesn't overlap). This also means that for fixed positions for the first team, positions for second and third teams are defined unambiguously.

To choose positions for first team, starting from pos_1 we can greedily choose position with higher value so that for every $1 \le i \le n$ there are not less than i positions on prefix $[1, 3i-2]$. Fortunately, since the initial array is a permutation, every pos_i is also defined unambiguously (there is a single way to choose positions with maximal sum).

Problem J. Reachability in a Matrix

Let $S_{x,y}$ denote the set of vertices reachable from vertex v. Since this set can be large, we need something smarter

This set can be represented as two arrays R, C of size n and m, where cell (i, j) reacble if $A_{i,j} + k \le max(R_i, C_j)$ or $(i, j) == (x, y)$. To prove it build a path from (x, y) to (i, j) and look for the value of the cell before (i, j).

How can we calculate $S_{x,y}$ for all cells?

We will go in the order of increasing $A_{x,y}$ from 1 to $n \times m$ and get a merge of cells $S_{i,j}$ on the same row or column and with $A_{i,j} + k \le A_{x,y}$.

Merging two sets $S_{a,b}$ and $S_{a,b}$ is just taking maximums for every position on R and C, plus adding values of (a, b) and (c, d).

But it still $O(n + m)$ merges per cell. To optimize this part we can use two pointers and store merged sets of cells in the same row/column with value $A_{i,j} + k \le A_{x,y}$. After iterating to the next $A_{x,y}$ we will need to add only one cell to its corresponding row and column.

To summarize, we can merge in $O(n+m)$ and there will be $O(n*m)$ merges. In total $O(n*m*(n+m))$ to build sets.

Answering a query is simple in $O(1)$.

Another approach is to store $S_{x,y}$ as a bitset.

Problem K. Bitvzhuh

Let $a, b, c, \ldots, z$ be the initial elements. Then

- after 1 bitvzhuh we would have $[ab, ac, bc, \ldots]$ — i.e. all pairs;
- after 2 bitvzhuhs we would have $[ab, ac, bc, \ldots, abcd, abce, \ldots]$ — i.e. all pairs and fours;
- ...

i.e. after ℓ bitvzhuhs we would have all even combinations up to 2^ℓ. If there is a basis B of vector space $[0, 2^k)$ in initial array a then after enough bitvzhuhs we would have all even combinations in this basis (subspace of even combinations of a basis).

Claim: it is enough to initially have an element x in an array a which is an even combination of an elements of basis B. Indeed, after first bitvzhuh, this even combination x xored with any element of B will give odd combination, and it is not hard to see, that eventually all odd combinations will be attained.

But is it sufficient to find **any** basis B and find an even combination of B in a? Turns out – yes. In particular, we can show, that if some basis B from a satisfy the property

"each element of a is an odd combination of elements from B"

then **any** other basis B_0 from a satisfy this property. This can be shown with Basis exchange lemma: we iteratively take $b \in B - B_0$ and $b_0 \in B_0 - B$ and replace $B \to B - \{b\} + \{b_0\}$ which is again a basis. Then we can ensure that B satisfy this "all elements are odd" property. We repeat procedure until $B = B_0$.

To find basis we use Gauss algorithm which works in $O(nk)$ since row operations are bit operations.

Altenative solution Consider the set $A = \{a_1 \oplus a_2, a_2 \oplus a_3, \ldots, a_{n-1} \oplus a_n\}$. If dimension of a linear span of A is k then the answer is yes, otherwise – no.