Problem A. Splits

If we can split n into k different values, then their sum is at least $1 + 2 + \ldots + k = \frac{k(k+1)}{2}$. This follows from the fact, that the first minimal unique value is at least one, the second minimal unique value is at least two and so on.

How to build the example if $n \geq \frac{k(k+1)}{2}$? Let's write down numbers from 1 to k and add to the last one what remains, in other words $n - \frac{k(k+1)}{2}$. After addition we will still have k different values and their sum will be equal to n.

Problem B. Non-triangles

The simplest solution requires $OP(n^3 \cdot q)$ time and scores 10 points. You need to store all edges (for example, in the adjacency matrix), and after each query, check all n^3 triples.

Maximum non-triangle cost — in fact, the maximum cost of three vertices where the first two are not connected by an edge, so the third vertex should be taken as much as possible, and it should not coincide with the first two. We can find such a vertex in 3 steps, find acceptable vertex of three with the largest cost, so now we can only iterate over pairs of vertices. This solution required $O(n^2 \cdot q)$ time and scored 20 points.

It was possible to store all the matching pairs of vertices in some data structure. Such solutions scored from 20 to 55 points, depending on the implementation.

In a subgroup of 20 points, the following fact could be noticed: the answer does not decrease (since the edges are only removed). At the same time, for each vertex v, we are only interested in the maximum value of the vertex u, with which v is not connected by an edge, so for each vertex we can iterate through all the others in the order of non-increasing distances, and find the first suitable one. After we have found the answer for the original problem, we need to increase it after each addition. This solution required $O(n \log n + m + q)$ time, including time to sort vertices by cost.

To get 100 points, you could combine 2 previous solutions. Let's divide all edges into two types: the edges that participate in queries, and the rest. There are no more than q edges of the first type. For edges of the second type, the answer will not change, so you can immediately calculate the answer for $O(n + m + q)$. Edges of the first type can be supported in some data structure, such as set. Total deletions, additions, and minimum queries will be $O(q)$, so the final asymptotic will be $O(n \log n + m + q \log q)$.

Problem C. Find the Path

First of all, each vertex can have n vertexes in pair. It means that first element of pair is always $v = \lceil \frac{k}{n} \rceil$. Now we need to find the second part of the answer. Problem is equal to finding $((k-1) \mod n)$-th lexicographic path starting in v (0-indexation).

Supposing, vertex v is always the same, we can make our tree rooted with root in v. Then we can sort edges from each vertex.

What about k-th path? First goes from root to root. Then range of paths use first edge in root's list, then second edge in list etc... It means that we can find all paths sorted lexicography with simple depth-first search:

```
dfs(v):
    paths.push(v)
    for (to : edges[v]):
        dfs(to)
```

For general problem, let's use the same idea. With sorted edges, we can precalculate array $count_{to}$ — number of potential finish vertices after using edge $v \to to$. Using prefix sums, we can determine range of k, which should go through edge $v \to to$.

Let's denote that if v is a parent for an answer vertex u then v's subtree is a substring of general sequence. So, if $u \in subtree(v)$, it can be found in $O(1)$ using our precalculated values.

There are cases when path goes up, then down: $v \to parent(v) \to parent(parent(v)) \to \ldots \to w \to \ldots \to u$. Let's calculate $parent[v][x]$ — 2^x-th parent of v. From now, we will solve problem $find(v, k)$ — find k-th path from vertex v. At the moment we know that we can answer $find(v, k)$ in $O(1)$, if $u \in subtree(v)$.

Let's denote $dp[v][x]$ — range of valid k values when our sequence goes up for 2^x edges. How to calculate it? For $x = 0$ we did it before with prefix sums. How to solve for bigger x?

We are going to find $dp[v][x]$ using $dp[v][x-1]$. Let's notice — k should belong $dp[v][x-1]$, so we jump to $parent[v][x-1]$. Then problem $find(v, k)$ changes to $find(parent[v][x-1], k')$ (in most cases $k' = k - left(dp[v][x-1])$). Now k' should be in $dp[parent[v][x-1]][x-1]$. Solving the equation, we get new segment for $dp[v][x]$.

For answering a query, we iterate through x. If $k \in dp[v][x]$, we do $find(parent[v][x], k')$. When we are done with it, vertex u is in the current subtree. So, we just use the general sequence to find an answer.

Important notes:

- While working with $dp[v][x-1], dp[parent[v][x-1]][x-1]$ you may get a situation where $dp[parent[v][x-1]][x-1]$ already summed v's subtree. To check this situation, you need to look at numbers of adjusted vertices of $parent[v][x-1]$. If son had number less than parent, add size of its subtree to k.
- Same situation while answering queries.
- Before going down, we want to solve $find(v, k)$. If $parent(v)$'s index is less than index of u's subtree, k includes $n - size(v)$, which should be subtracted. It can be checked with prefix sums.

Problem D. Points on the Plane

The first, the third and the fifth subtasks can be solved with naive implementation of every query with asymptotics $O(nq)$.

For the second subtask we need to build a convex hull of given points. After that we need to check if any point of this hull is in a half-plane. We can use binary search or two pointers to find the farthest point from the line and check if matches the condition. Asymptotics of this solution is $O(n \log n + q \log n)$ or $O(n \log n + q \log q)$ depending on implementation.

The fourth subtask is the classic problem on method called scanline — count the number of points in the rectangle. $O(n \log n + q \log n)$.

For sixth and seventh subtasks we need to notice, that any square can be divided into 4 right triangles with catheti are parallel to the coordinate axes and 1 square with sides which also are parallel to the coordinate axes, divide all the squares from the queries into this 5 figures. Now we have reduced our task to the following: we have a set of rectangular regions (each rectangle is defined by its own borders by x: $[lx, rx]$ and by y: $[ly, ry]$) and a half-plane for each region, for each of this query we need to understand whether at least one point from a given region lies in a given half-plane. We can solve this problem with divide and conquer algorithm($d\&c$) by x coordinate. Note, that on each level of recursion we have $O(n)$ points and $O(q)$ "new" queries. For set of queries, which $[lx, rx]$ segment covers current segment in recursion we will use $d\&c$ by y coordinate also we have $O(n)$ points and $O(q)$ "new" queries on each recursion level. Now, when we consider rectangle (by x: $[lx, rx]$, by y: $[ly, ry]$) and we can solve this task also like the second subtask. Asymptotics of this solution is $O((n+q)\log^3(n+q))$, if you build a convex hull in $O(n \log n)$ and get answers for all queries with binary searches or $O((n+q)\log^2(n+q))$, if you sort the points and queries first, and build a convex hull in $O(n)$ time, and replace binary search with a two-pointer algorithm.

The solution for the eighth subtask is based on the same idea as the last solution — we can answer the question if it is a half-plane query. Now we need to understand how to solve the general problem. Let's start from a square area and divide the area in square quarters, next divide them and so on while we can't reduce this problem to the half-plane query. On each level of the division we will have only $O(1)$ areas that are unanswered. If we're answering for all query at once, we can get asymptotics $O(n \log max\_coor)$.

Problem E. Game on the board

To solve the problem at $x \leq 10^5$ it was possible to simulate the described game.

For a full solution, note the fact: after the first round, the number 0 will appear on the board, since the number of points of the player who will lose first round will be 0.

Note that when 0 is on the board, Alice's number is always greater than 0, and Bob's number is equal to 0.

That is, all rounds except the first one are guaranteed to be win by Alice.

Thus, it remains to find out who will win the first round.

It is easy to understand that Bob will win the first round only when 0 was not on the board initially.

Problem F. Least Common Ancestor

First, let's define $f(m)$ as amount of subsets, least common ancestor of which is equal to m. Then the answer is equal to $1 \cdot f(1) + \ldots + n \cdot f(n)$. Now we have to calculate $f(m)$.

Suppose that vertex k from a set of vertexes does not lie in a subtree of a vertex m. Then m is not the least common ancestor of this set, since it does not lie on the path from 1 to k. Then we will examine subsets that are only in the subtree of m.

Instead of the original problem let's solve the opposite one, and calculate amount of "bad" sets, that are in the subtree of m with their least common ancestor is not m, and use this to calculate $f(m)$.

Let's notice that if a path between to vertexes contains m, then their least common ancestor is m. It means that every "bad" subset is contained in a subtree of a son m_i. Also let's notice that if some subset is contained in a subtree of a son m_i, then it is "bad", since the least common ancestor will be in the subtree of m_i.

For convenience, define $size(x)$ as the size of the subtree of vertex x, $m_1, m_2, \ldots, m_k$ — sons of m. Note that amount of nonempty subsets in a subtree of x is equal to $g(x) = 2^{size(x)} - 1$. Using previous observations, we can calculate amount of "bad" subsets, which is $g(m_1) + g(m_2) + \ldots + g(m_k)$.

Then other subsets give us least common ancestor m. Since we know amount of nonempty subsets is $g(m)$, then $f(m) = g(m) - g(m_1) - g(m_2) - \ldots - g(m_k)$.

We can calculate $size(x)$ by using dynamic programming on trees, which takes $O(n)$ time. In order to calculate $f(x)$ we have to visit each son once, which takes $O(n)$ time. Also let's calculate all required powers of 2, which also takes $O(n)$. The algorithm runs in $O(n)$.

Problem G. Strange Function

Note that p_i^{-1} is equal to the position of the element i in the permutation p. Therefore $|p_i^{-1} - p_{i+1}^{-1}|$ is the distance between the elements i and $i+1$ in the permutation p. This value is equal to the number of elements between them, increased by one.

Denote ats cnt_i the number of elements i in the sequence, denote as all the product of all cnt_i. If we check all possible occurrences of i and $i+1$ in the original array, and all elements j between them, this triple of elements occurs in $\frac{all}{cnt_i \cdot cnt_{i+1} \cdot cnt_j}$ permutations. In each of these permutations, element j increase the distance between elements i and $i+1$, so that is the value that should be added to the answer. In each permutation, the original distance between elemnts i and $i+1$ is 1, so the answer should be increased by $all \cdot (m-1)$. The complexity of this solution is $O(n^3)$.

Now consider j. We want to calculate how many occurrences of integers i and $i+1$ are there, such that they are not equal to j and are at different sides of j. Introduce arrays $left_i$ and $right_i$ — the number of elements equal to i to the right and to the left of the current element, correspondingly. Additionally, let us keep track of the sum of $\frac{all \cdot (left_i \cdot rigth_{i+1} + left_{i+1} \cdot right_i)}{cnt_i \cdot cnt_{i+1}}$ for all i. When we move the element j to the right, the arrays $left$ and $right$ have only $O(1)$ elements that change their values, so the sum above has only $O(1)$ terms that change. For the element j we subtract the terms that are formed by the pairs

$(j-1, j)$ and $(j, j+1)$ from this sum, and divide the rest by cnt_j. That is the number of occurrences of pairs such that j is between their elements, that is the value that must be added to the answer. Same as in the previous solution, the answer should be increased by $all \cdot (m-1)$. The final complexity is $O(n)$.

Problem H. Addition

Group 1. $n, m, q \le 100$

In the first group we can store the whole field in 2D-array and process all queries directly. For a query of type 1 we can just bypass each of t rectangles and add 1 to each cell in $O(nmt)$. For a query of type 2 we bypass the given rectangle summing the values in its cells. In total we get $O(nmqt)$ time complexity and it satisfies us (because $t \le m$).

Group 2. $n, m, q \le 700$

In the second group we still can store the whole field and bypass the rectangle in queries of type 2, but we must process queries of type 1 faster — we can't afford bypassing t rectangles. But we can use a formula to count for each cell affected by the query what value we must add: it equals to the amount of rectangle shifts (from 0 to $t-1$) that cover this cell. The formula is $\min(c_{i_1} + t - 1, c') - \max(c_{i_1}, c' - (c_{i_2} - c_{i_1})) + 1$ — try to draw this on paper to understand why it's true. Now we can process queries of both types in $O(nm)$ so the total complexity is $O(nmq)$.

Group 3. $q \le 10\,000$, $t_i = 1$

In the third group we can't store the whole field directly. Instead of it for a query of type 2 let's calculate separately for each previous query of type 1 the sum of all values added on intersection. In this group $t_i = 1$ for each change query, so the required sum equals to the area of intersection which can be calculated simply in $O(1)$. So the total time complexity is $O(q^2)$.

Group 4. $q \le 10,000$

The fourth group can be solved similarly to the third, but calculating the sum over intersection becomes harder — here we need the formula that will be used in the complete solution. Let's consider we are calculating the answer to some sum query, and let's consider a change query that we're watching at now. Its rows are same because the rectangle is being shifted horizontally, so sum over the intersection equals to the product of sum over one row of intersection with height of the intersection. Calculating the height is obvious so let's consider calculating the sum over a segment of some row of change query.

Let the columns of change query be $[l; r]$ and the amount of shifts be t. So we add 1 to each cell in segments $[l; r]$, $[l+1; r+1] \dots$, $[l+t-1; r+t-1]$. It can be represented as element-by-element addition of a sequence $(1, 2, \dots, t, t, \dots, t)$ to the suffix $[l; m]$, and then subtraction of the same sequence from the suffix $[r+1; m]$. Such sequence in turn can be represented as the difference between two equal arithmetic progressions shifted by t relative to each other: $(1, 2, \dots, t, (t+1)-1, (t+2)-2, \dots)$. Using a simple formula we can calculate the sum over any segment of an arithmetic progression, so for a given segment of intersection we can find these four points and calculate the sum over it in $O(1)$. Total time complexity is $O(q^2)$.

A technical note: unusual module 2^{31} is chosen for a certain reason. Starting from this group, we use formulas that imply division by 2. It doesn't have an inverse modulo 2^k for any k, but calculating modulo a prime would slow down the solution dramatically. In this task we can calculate everything modulo 2^{32}, i.e using unsigned integer type, and divide by 2 in the end, so the obtained number is the correct remainder of the answer modulo 2^{32}.

Group 5. The given rectangle matches the entire field in queries of type 2.

The simplest group — we can store the sum over the field in an integer variable, adding tS after each change query, where S is the area of given rectangle.

Group 6. $n, m \le 1000$, $t_i = 1$, all sum queries are after all change queries.

Groups 6 and 7 contain the key idea of complete solution. Let's call a set of cells (r', c') such that $r' \le r$ and $c' \le c$ a prefix (r, c). Similarly, let's call a set of cells (r', c') such that $r' \ge r$ and $c' \ge c$ a suffix (r, c). We will find the sum over a prefix (r, c) for each cell (r, c), and after that each sum query can be answered in $O(1)$ time complexity — sum over a rectangle can be split into four prefix sums.

A change query can be represented as adding 1 and -1 to four suffixes of the field. If we only have added 1 to the suffix (r_0, c_0) then the sum over prefix (r, c) (assuming that the suffix contains (r, c)) is $(r - r_0 + 1)(c - c_0 + 1)$. So we add a function of two variables (r, c) to a cell (r_0, c_0), and if a certain prefix (r, c) contains this cell, we add the value of this function in point (r, c) to the sum.

So we can add functions in the desired cells (in this group there are 4 of them) in change queries, and before processing the sum queries we build a table F where $F[r][c]$ equals to the sum of functions over cells of the prefix (r, c). After that sum over the prefix (r, c) of the original field equals to the value of $F[r][c]$ at point (r, c). Total time complexity is $O(nm + q)$.

Group 7. $n, m \le 1000$, all sum queries are after all change queries.

Here we do the same as in group 6, except the functions. We can use the idea from group 4: applying such operation on a segment can be represented as adding and subtracting four arithmetic progressions on suffix. As we work with rectangles here, we also should symmetrically add these progressions in row $r_{i_2} + 1$ with changed signs. So the function is $\frac{(c-c_0+1)(c-c_0+2)}{2}(r - r_0 + 1)$.

Complete solution (groups 8 and 9).

To solve the whole task we have to contain $F[n][m]$ in a compressed form allowing to add in point and get sum over prefix fast enough. We could use two-dimensional dynamic segment tree, but it works too slow and is too hard to write. Let's note that we can solve this task offline, and it allows us to use a simpler and faster data structure — a two-dimensional Fenwick tree. We'll have an outer tree, each element of which stores its own inner Fenwick tree containing the functions.

We can't afford storing it explicitly even after coordinate compression, so we should leave only interesting points. Let's choose rows of sum queries as interesting points of the outer tree. Then to process adding a function in point we should find the lowest interesting row that is not lower than the current point and lift from it. As interesting points for the inner Fenwick trees we choose columns of sum queries in the following way. For each sum query we descend from both of rows and add both columns to the list of interesting points for each visited inner tree.

Memory complexity is $O(q \log q)$, and the time complexity is $O(q \log^2 q)$ with a huge constant because to store each function we need 6 integers, and for each change query we add to 8 points in Fenwick tree. Although, due to very fast performance of Fenwick tree the solution works pretty fast.