Problem A. LaIS

Let's add 0 to the beginning of a, then we'll increase LaIS by one and so it will always start from 0. Let's look at any almost increasing subsequence (aIS) and underline elements, which are minimums in at least one consecutive pair, for example, $[\underline{0}, \underline{1}, \underline{2}, 7, \underline{2}, \underline{2}, 3]$.

Note that underlined elements form increasing subsequence (IS) and there is no more than one element between each consecutive pair. What constraints these elements have? Obviously, $a[pos_{i-1}] \le a[pos_i] \ge a[pos_{i+1}]$, but we can ease the constraints just to $a[pos_{i-1}] \le a[pos_i]$, i. e. we can allow $a[pos_i] < a[pos_{i+1}]$, since aIS will still be aIS.

Now, note that between pos_{i-1} and pos_{i+1} we can choose any j such that $a[pos_{i-1}] \le a[j]$, so we can always choose the first such j, moreover we can precalculate such j as nxt_i for each a_i. Using stacks or something similar.

Now, we can note that each a_i is either minimum in LaIS or $i = nxt_j$ for some other element a_j. And we can write the following dynamic programming: let $d[i]$ be the LaIS which finish in a_i and it's the last minimum (we can think that the last element of LaIS is minimum in the imaginary pair). To calculate it, we can iterate i from left to right and store for each value x the length of LaIS with the last minimum (not the last element) equal to x in Segment Tree (ST).

So, d_i is equal to "get maximum over segment $[0, a_i]$ in ST" plus 1. And we update ST in two moments: firstly, right now with value d_i in position a_i and secondly, after we meet the element nxt_i with value $d_i + 1$ also in position a_i.

After we process all i we'll get ST where for each x the length of LaIS with last *minimum* equal to x will be stored and the answer will be equal just to "maximum over all tree (segment $[0, n]$)".

In total, we should calculate nxt_i for each a_i (we can do it in linear time using stack) and maintain Segment Tree with two basic operations: range maximum and update in position. The time complexity is $O(n \log(n))$ and the space complexity is $O(n)$.

Problem B. Bakery

Let's look at all n days with some fixed k. Obviously, the seller works like a stack, so let's divide all days into segments in such a way, that the stack is emptied between consecutive segments. We can note that after we've got these segments — the answer is the maximum length among all segments. Why? Because the stalest bread on the bottom of the stack and we can't sell it until we empty the stack.

Now, let's set $k = 10^9$ and look at what happens when we gradually lower k. With $k = 10^9$, we sell all bread that we baked on the same day, so all segments consist of one day $[i, i+1)$. Now, if we start lowering k then at some moment segments will start to merge (and only merge), since k is not enough to sell all bread in this interval. Since no segment will split up, there will be only $n - 1$ merging.

So we can look only at moments k when either several segments merge or when we should answer the query. With what k the segment $[p_1, p_1)$ will start to merge with the next segment $[p_1, p_2)$? The answer is when $(p_2 - p_1) \cdot k < \sum\limits_{p_1 \le i < p_2} a_i$ or $k = \left\lfloor \frac{(\sum a_i) - 1}{p_2 - p_1} \right\rfloor$.

So, we can for each segment $[p_i, p_{i+1})$ maintain its value k_i when it will start merging with next segment in `set`. And if we want to calculate the answer for a given k from the queries, we should merge all segments with $k_i \ge k$, while updating the current maximum length among all segments.

Since merging two segments require two operations with the `set` then the total time complexity is $O(m + n \log n)$.

Problem C. Berpizza

The hardest part of this problem is to efficiently implement the third query, i. e. finding the customer with the greatest value of m and erasing it. Simply iterating on all of them is too slow, since there may be up to $2.5 \cdot 10^5$ such queries and up to $5 \cdot 10^5$ customers at the pizzeria.

There are several solutions to this issue, I will describe two of them. The first solution is to treat each customer as a pair (id, m), where id is the number of the customer. Then the first query means "insert a new pair", the second query — "remove the pair with minimum id", and the third query — "remove the pair with maximum m". To maintain them, we can use two balanced binary trees that store these pairs — one will store them sorted by id, and the other — sorted by m. Then the second query means "find the leftmost element in the first tree and erase it from both trees", and the third query means "find the rightmost element in the second tree and erase it from both trees". Balanced binary trees can perform these operations in $O(\log n)$, where n is the size of the tree. Note that in most languages you don't have to write a balanced binary tree from scratch — for example, the containers `std::set` in C++ and `TreeSet` in Java already support all of the required operations.

The second solution maintains three data structures: a queue for supporting the queries of type 2, a heap for supporting the queries of type 3, and an array or set that allows checking whether some customer was already deleted by a query. Finding the customer that came first using a queue or has a maximum value of m using a heap is easy, but deleting some element from queue/heap is much harder (because we have to delete some arbitrary element, not necessarily the element at the head of the queue/heap). Instead, we can do it the other way: when we delete some customer from one of these data structures, we mark it as deleted. And while processing the following queries of type 2 or 3, we should check the element in the head of the data structure and, if it is marked as deleted, remove it before processing the query. Note that there can be multiple such elements, so it should be done in a loop. Since each element is deleted at most once, this solution works in $O(n \log n)$ amortized.

Problem D. Firecrackers

The first crucial observation that we need is the following one: it is optimal to light and drop some firecrackers, and only then start running away from the guard (that's because running away doesn't actually do anything if none of the firecrackers are lit). The hooligan shouldn't drop more than $|a - b| - 1$ firecrackers, because otherwise he will be caught before starting running away, and the last firecracker he dropped won't go off.

Okay, now suppose the hooligan wants to explode exactly f firecrackers. It's obvious that he wants to choose the f firecrackers with minimum s_i, but in which order he should drop them? If the i-th firecracker he drops goes off in s_j seconds, then it will explode on the $(i + s_j)$-th second. We have to choose an ordering of the firecrackers that minimizes the maximum of $(i + s_j)$ in order to check that the hooligan has enough time to see all the firecrackers he dropped explode. It can be shown that we can drop the firecracker with the maximum s_j first, then the firecracker with the second maximum s_j, and so on, and the maximum of $(i + s_j)$ is minimized (obviously, we consider only the firecrackers with f minimum values of s_i).

This leads us to a solution with a binary search on f in $O(n \log n)$: we can check that the hooligan can explode at least f firecrackers in $O(n)$ (after sorting the sequence s beforehand), and binary search requires to check it for only $\log n$ values of f.

Problem E. Four Segments

Suppose the values of a_1, a_2, a_3, a_4 are sorted in non-descending order. Then the shorter side of the rectangle cannot be longer than a_1, because one of the sides must be formed by a segment of length a_1. Similarly, the longer side of the rectangle cannot be longer than a_3, because there should be at least two segments with length not less than the length of the longer side. So, the answer cannot be greater than $a_1 \cdot a_3$.

It's easy to construct the rectangle with exactly this area by drawing the following segments:

- from $(0, 0)$ to $(a_1, 0)$;
- from $(0, a_3)$ to (a_2, a_3);
- from $(0, 0)$ to $(0, a_3)$;

- from $(a_1, 0)$ to (a_1, a_4).

So, the solution is to sort the sequence $[a_1, a_2, a_3, a_4]$, and then print $a_1 \cdot a_3$.

Problem F. Full Turn

Lets define for person with index i their initial vision vector as vector $(u_i - x_i,\ v_i - y_i)$. It is possible to prove that two persons will make eye contact during 360 rotation if and only if their initial vision vectors are collinear and oppositely directed. Note that the position of the persons does not matter, only their vision vectors.

E.g. lets assume that person A has initial vision vector (3, −4) and person B - (−6, 8). These vectors are collinear and oppositely directed, hence person A and B will make eye contact during the rotation.

If we try to check separately for each pair of persons if they will make eye contact, that would take too much time. For example for $n = 10^5$ that would take $\approx 5 * 10^9$ checks.

Instead we should use a different approach. First, lets normalize vision vector of each person by dividing its coordinates by GCD of absolute values of the coordinates. Here GCD stands for greatest common divisor. E.g. vector's coordinates (6, −8) should be divided by $GCD(|6|, |-8|) = GCD(6, 8) = 2$, and the normalized vector will be (3, −4). There is a special case for vectors, which have zero as one of the coordinates: (0, C), (0, $-C$), (C, 0) and ($-C$, 0), where C is some positive integer. These should be normalized to vectors (0, 1), (0, −1), (1, 0) and (−1, 0) respectively.

After normalization all collinear and co-directed vectors will have exactly the same coordinates. Lets group such vectors and count the number of vectors in each group. Then it is obvious that each person from group with vector (x, y) will make eye contact with each person from group with vector ($-x$, $-y$). If the first group has k vectors and the second group has l vectors, in total there will be $k * l$ eye contacts between members of these two groups. And also members of these two groups will not make eye contact with members of any other groups.

So fast algorithm should create a map, where key would be group's vector and value - number of persons in the group. Then the algorithm should iterate over groups in the map, for each group find the oppositely directed group, and add to the answer multiplication of these groups' sizes.

Note that the maximum possible answer is $(n/2)^2$. That would be $2.5 * 10^9$ when $n = 10^5$, which does not fit into signed int32.

Problem G. Hobbits

Let's start with the general idea of the solution. To solve the problem, we need to iterate over all relief segments and understand, which part the segment is seen by the Eye. A segment point can be hidden from the Eye by several mountains, but it is enough to track only the highest mountain. Generally speaking, the relief segments can be processed in any order, bit it would be more convenient to iterate on them backwards — i.e. in the order from the Eye to the start point. Processing the segments in reversed order will allow to recalculate the highest mountain easier.

Let's now discuss implementation details. Formally speaking, there are n relief points $p_i = (x_i, y_i)$ $(1 \le i \le n)$ and the Eye point $E = (x_n, y_n + H)$. Each relief point defines its own angle α_i, which is measured counter-clockwise between positive direction of the OX axis and the (E, p_i) vector. Now, having two relief points p_i and p_j $(i < j)$, it can be said that p_i is hidden from the Eye if $\alpha_i > \alpha_j$. When this check is implemented in the solution, to avoid the precision loss, it is recommended to use vector product and analyse its sign to understand the relation of the angles. This check is done in $O(1)$ time.

Being able to understand when a point is hidden from the Eye by another point, it is possible to calculate, which part of a segment (p_i, p_{i+1}) is seen by the Eye. First of all let's note, that if the segment left point p_i is hidden by its right point p_{i+1}, then the entire segment is hidden from the Eye. Now, we have the situation when left segment point is not hidden from the Eye by its right point. But the segment or its part can still be hidden by the highest mountain (point M), which is a point with minimal angle from all relief points, located to the right of our segment. Here three cases are possible:

- both p_i and p_{i+1} are hidden by M — in this case the entire segment is hidden and we switch to the next segment;
- both p_i and p_{i+1} are visible by the Eye (not hidden by the highest mountain M) — in this case the entire segment is visible and we should add its length to the answer;
- left segment point p_i is visible by the Eye, but right segment point p_{i+1} is hidden by M — in this case we need to find intersection point I of the segment (p_i, p_{i+1}) and the ray (E, M). What is left - add length of (p_i, I) segment to the answer.

It is very convenient to implement segment and ray intersection in a parametric way, when intersection point I is represented as $I = p_i + t_1 * (p_{i+1} - p_i) = E + t_2 * (M - E)$ and then parameters t_1 and t_2 are found as solutions of linear equation system.

Now, let's conclude the final algorithm:

1. Iterate over all relief segments from right to left, keeping the highest mountain point M.
2. Analyze how left and right points of the current segment are located relatively to each other and point M.
3. Recalculate M, taking points of the current segments as candidates for the highest mountain.

Total algorithm complexity is $O(N)$.

Problem H. K and Medians

Since after each operation we erase exactly $k-1$ element from a then if $(n-m) \not\equiv 0 \pmod{(k-1)}$ then the answer is `NO`.

Otherwise, if there is such element b_{pos} such that there are at least $\frac{k-1}{2}$ erased elements lower than b_i and at least $\frac{k-1}{2}$ erased elements greater that b_i, then answer is `YES`, otherwise `NO`.

Let's prove this criterion in two ways. From the one side, in the last step, we should choose k elements and erase them excepts its median, so the median is exactly that element b_{pos}.

From the other side, let's prove that if there is such b_{pos} then we can always choose operations to get sequence b. Let's make operations in such a way that in the last step we'll erase b_{pos}, $d = \frac{k-1}{2}$ elements lower b_{pos} and d elements greater b_{pos}. Since, it doesn't matter which d elements to take from left and right of b_{pos}, we will only consider number of such elements.

Let's denote x and y as the initial number of elements from left and right of b_{pos} to erase. We know that $x \geq d$ and $y \geq d$. and we want to make both of them equal to d.

Let $x' = x - d$ and $y' = y - d$. We can think that we have x' free elements to erase from left and y' from the right. While $x' + y' \geq k$ let's take any k free elements that should be erased and erase $k-1$ of them. Then we get situation $0 \leq x' < k$, $0 \leq y' < k$ and $x' + y' < k$. Since $(n-m)$ is divisible by $(k-1)$, then it means that $x' + y' = k - 1$.

Let's look at what we can do now. We should take one of "reserved" elements to participate in erasing last $x' + y'$ free elements but it shouldn't break the situation with d lower elements, b_{pos} and d greater elements.

If $x' \geq y'$, let's take one extra element which is lower than b_{pos}, then after erasing, the remaining median will also be lower than b_{pos}.

If $x' < y'$, let's take one extra element greater than b_{pos}, then the remaining median will also be greater than b_{pos}.

In the end, we choose d elements lower, b_{pos} and d elements greater. Erase them except its median b_{pos} and get the desired array b.

Problem I. Plane Tiling

One of the solutions is to consider an infinite 2 dimensional plane and draw a grid with lines parallel to vectors (dx_1, dy_1) and (dx_2, dy_2). By doing so we will get a tiling of the plane with parallelepipeds, one of them will have corners at $(0, 0)$, (dx_1, dy_1), $(dx_1 + dx_2, dy_1 + dy_2)$, (dx_2, dy_2).

All the parallelepipeds are same shape and all of them can be represented by translating one of them by vector $(a \cdot dx_1 + b \cdot dx_2, a \cdot dy_1 + b \cdot dy_2)$, where a and b are all possible integer pairs. Thus, if we "rasterize" one of them, we will get a correct solution.

To do so, we can represent the parallelepiped as two triangles and rasterize each of them individually. To rasterize a triangle, we need to select those cells which center is either inside of it, or located on the "top" or "left" edge of it. A "top" edge is an edge that is perfectly horizontal and whose defining vertices are above the third one. A "left" edge is an edge that is going up if we go in triangle's clockwise order.

This way, no cell is going to be selected twice in case we "rasterize" two triangles sharing an edge, and since we are tiling the plane with given parallelepipeds rasterized cells are exactly our solution.

If vectors (dx_1, dy_1) and (dx_2, dy_2) are either collinear or one of them is zero, the answer is "`NO`".

Also there is another solution. First of all we have to calculate d — absolute value of determinant of the matrix formed by the given vectors:

$d = |det((dx_1, dy_1), (dx_2, dy_2))| = |dx_1 * dy_2 - dx_2 * dy_1|$

If given value n is not equal to d then there is no solution, in particular if $d = 0$ there is no required n pairs of integers.

Now let $n{=}d$ and $d_x = gcd(dx_1, dx_2)$, $d_y = gcd(dy_1, dy_2)$. Let's go to the same problem with smaller integers — we divide dx_1, dx_2 by d_x and divide dy_1, dy_2 by d_y. Also we define $m = n/(d_x \cdot d_y)$ ($n{=}d$ so it is divisible by $d_x \cdot d_y$). So firstly we need to find such m points (x_i, y_i) that all values $(x_i + a \cdot dx_1 + b \cdot dx_2, y_i + a \cdot dy_1 + b \cdot dy_2)$ are different. It is enough for solution, because we still have m equal to the absolute value of the determinant of the new matrix.

It turns out that it is easy to find such set of points. In particular we may choose points $(0, 0), (0, 1), \cdots, (0, m - 1)$, i.e. $x_i = i$, $y_i = 0$. Let's prove that such set is correct. Assume that for some non-zero pair (a, b) and for some j we also have one of these points: $x_i = x_j + a \cdot dx_1 + b \cdot dx_2$ and $y_i = y_j + a \cdot dy_1 + b \cdot dy_2$. Considering $y_i = y_j = 0$, we have $a \cdot dy_1 + b \cdot dy_2 = 0$. Since dy_1 and dy_2 are coprime (we have divided it by d_y) $a = k \cdot dy_2$, $b = -k \cdot dy_1$ for some integer k. If we use this for x equation, we will have: $x_i - x_j = k \cdot dy_2 \cdot dx_1 - k \cdot dy_1 \cdot dx_2$. I.e. $x_i - x_j = k \cdot (dx_1 \cdot dy_2 - dx_2 \cdot dy1) = \pm k \cdot m$. Also $-m < x_i - x_j < m$ so this equation has no solution for integer non-zero k. Thus this set of points is correct.

Finally having such m points we have to create an n-points solution for the original problem. Obviously we need to multiply current answer coordinates by d_x and d_y. Then for each of these points, for each $0 \le r_x < d_x$ and for each $0 \le r_y < d_y$ we need to print a new point. So, the answer is $(i \cdot d_x + r_x, r_y)$ for each $0 \le i < m$, $0 \le r_x < d_x$, $0 \le r_y < d_y$.

Problem J. Road Reform

We will consider two cases: the road network without roads having $s_i > k$ is either connected or not. Checking that may be done with the help of DFS, BFS, DSU, or any other graph algorithm/data structure that allows checking if some graph is connected.

If the network without roads having $s_i > k$ is not connected, then we have to take several roads with $s_i > k$ into the answer. Since their speed limit is too high, we have to decrease the speed limit on them to k. Then the required number of changes is $\sum_{i \in R} \max(0, s_i - k)$, where R is the set of roads that are added to the answer. To minimize this sum, we can set each road's speed limit to $\max(0, s_i - k)$ and find the minimum spanning tree of the resulting graph.

Unfortunately, this approach doesn't work in the other case — we may build a road network having the

maximum speed limit less than k, not exactly k. We have to choose a road with the current speed limit as close to k as possible, i. e. the one with the minimum value of $|s_i - k|$. After applying $|s_i - k|$ changes to it, we can choose $n - 2$ roads having $s_i \le k$ to form a spanning tree with the chosen road (it is always possible due to the properties of the spanning tree: if we have a spanning tree and we want to add an edge of the graph to it, we can always find another edge to discard). So, in this case, the answer is $\min\limits_{i=1}^{m} |s_i - k|$.

Problem K. The Robot

It is obvious that the cell that can be the answer must belong to the original path of the robot. Thus, there are at most n candidate cells in the answer. In order to make sure that a candidate cell is indeed the answer; it is necessary to simulate the movement of the robot, taking into account this cell as an obstacle. No more than n actions will be spent on one such simulation. Therefore, to check all n candidates, the total number of actions spent is will not exceed n^2, which fits into the requirements for the running time (time limit) of a solution.

Problem L. Prime Divisors Selection

Let's find for each prime integer $p \le 10^9$ amount of integers p^q among the given integers. If there are no more than 2 such integers, this prime p is not interesting for us: if we add one such integer to the answer, we may always select exactly one divisor p in a suitable sequence so it will not be friendly.

Otherwise let's call such p *important* and find a group X_p (with some size $k_i > 1$) — all integers p^q in the given set. So we have several (t) such groups with sizes k_1, k_2, $\dots k_t$. Now there are some different cases.

Case 1: required size $k \le k_1 + k_2 + \dots + k_t$.

If k is odd and k_i=2 for each i, we have to find some integer y that has minimum possible different important primes in factorization (and does not have not important primes). If there is no such y, obviously, there is no solution. Now let there is y that has u different primes in factorization. If $u \le (k-1)/2$ we may take all u groups necessary for this integer y and add some other groups X_p in order to have exactly k numbers in the result (if $u < (k-1)/2$).

If k is even and $k_i = 2$ for each i, we may just take any $k/2$ groups by 2 integers.

If $k_j > 2$ for some j it is easy to check that we may always find the ideal sequence. Let's start to add groups with the maximum sizes while it is possible. Now we have total size $k_1 \le k$ and $q = k - k_1$ 'empty' space. If $q \ge 2$, we may just add q integers from the next group. If $q = 1$, we may remove one integer from the greatest group (because its size is at least 3) and add 2 integers from the next group.

Case 2: $k > k_1 + k_2 + \dots + k_t$.

First of all let's take all the groups X_p to the answer. Now each of the remaining integers is either represented as a product of important primes, or not. If it is represented, then we can add it to the answer and the set will remain ideal. Otherwise if we add such number, we can choose any prime divisor from it that is not important and the set will not be ideal. So, we just have to check whether there are enough integers that are represented as a product of important primes and add any k - k_1 - k_2 - $\dots$ - k_t of them to the answer. If there are not enough such integers, there is no solution.

How to find all important primes? We are interested only in groups $X_p = p^q$ with sizes at least 2. So each such group either has an integer p^2 or an integer p^q with $q \ge 3$. To find all the numbers in the first case, we may just check for each given integer whether it is p^2 for some prime p. For the second case we have $p \le 10^6$, so we may find all the possible powers for each prime $p \le 10^6$.

Problem M. Similar Sets

Let's define $D = m^{1/2}$, where m is the total amount of integers. Now let's call a set of integers 'large' if its size is at least D and 'small' otherwise.

Firstly let's check whether there is a large set that has two common integers with some other (small or large) set. Let's try to find a similar set for each large set S separately. For each integer x in S we may

set flag $w[x]$=1. Now for each other set T with size k we may calculate amount of integers x in T that satisfy condition $w[x]$=1 (i.e. amount of common integers with S) using $O(k)$ operations. So, we will use $O(m)$ operations for each large set S. Since D=$m^{1/2}$ we will have at most $m/D = D$ large sets and will perform $O(D * m) = O(m^{3/2})$ operations.

Now we have to check whether there are two similar small sets. For each small set S let's find all pairs of integers (x, y), $x < y$ in this set. If there are two equal pairs in different sets, that sets are similar. To find such pairs we may just create a separate array for each integer x and add all values y to it. It is possible to check whether there are two equal integers in array in $O(k)$ operations, where k is the size of this array. Since the size of each small set is at most D we will have at most $D * k$ pairs for each set of size k and at most $D * m$ pairs in total. So we will perform $O(D * m) = O(m^{3/2})$ operations.

Problem N. Waste Sorting

It is quite obvious that if $c_1 < a_1$, $c_2 < a_2$ or $c_3 < a_3$, the answer is `NO` because it is impossible to fit even the items that fit only into one container. Otherwise, let's get rid of a_1, a_2, a_3 by decreasing c_i by a_i $(1 \le i \le 3)$.

Now we have a problem with 3 containers and only 2 item types (4-th and 5-th), the fourth item type fits only into the first and into the third container, the fifth item type — into the second and into the third container. Since the first container accepts only items of type 4, we should fit the maximum possible number of items of type 4 there — that is, $\min(a_4, c_1)$ items. Similarly, we should fit the maximum possible number of items of type 5 into the second container ($\min(a_5, c_2)$ items). After that, we only have to check that all the remaining items can be fit into the third container.