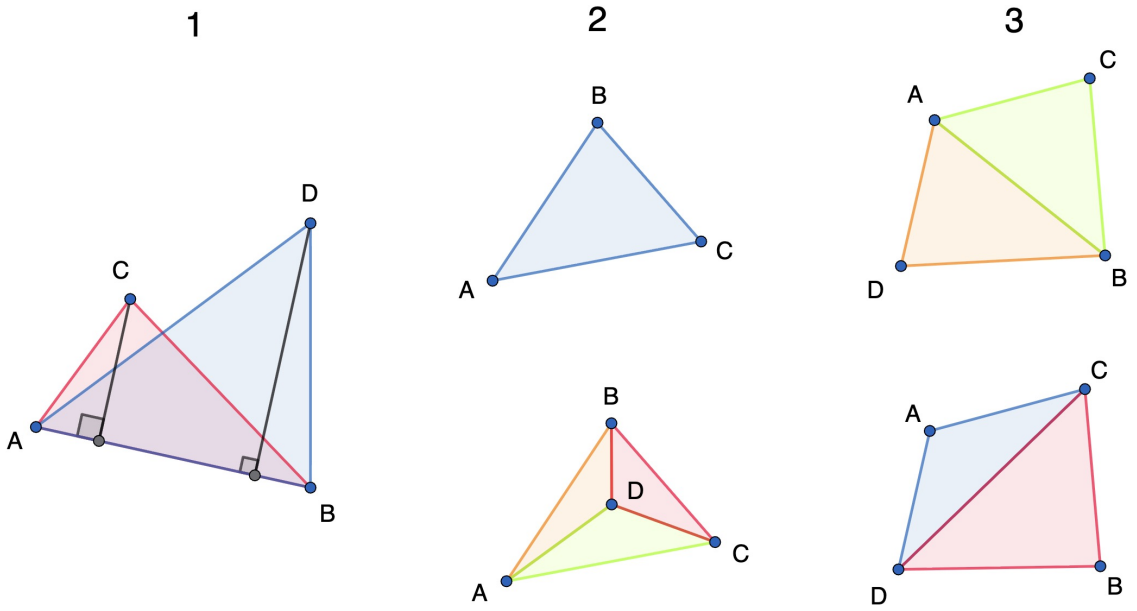
Problem Tutorial: "Cutting the Cake"

Author and problem developer: Ivan Safonov

Note that the problem asks to construct a partition of the convex hull of the set of points into parts. For the groups without maximization, it is enough to find the convex hull; for the groups with maximization, one needs to find a triangulation of the set of points.

Note that with our query we can perform the following operations:

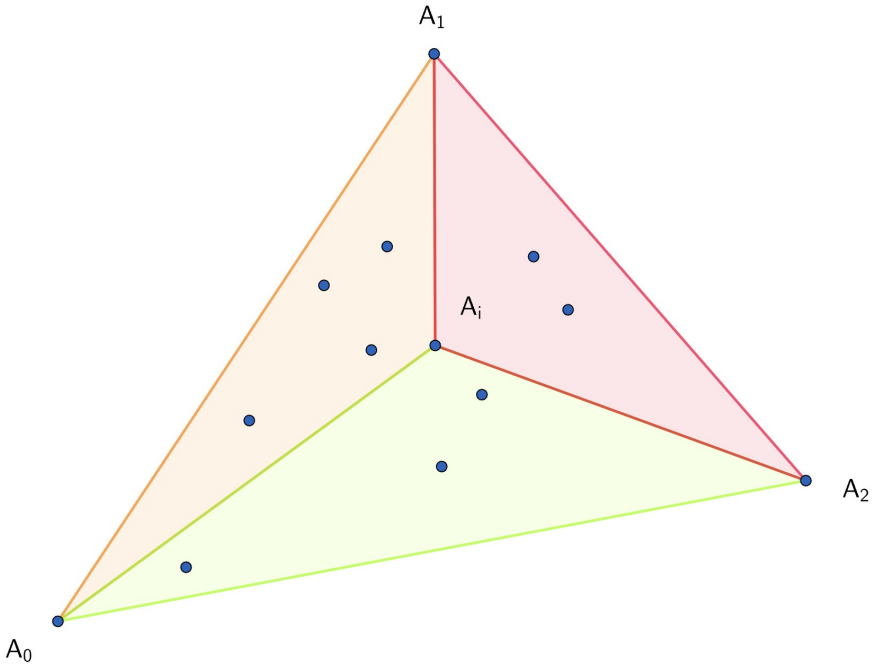
1. Given points A, B, C, D, determine which of the distances $\text{dist}(C, AB)$, $\text{dist}(D, AB)$ is larger (the distance to the line). To do this, it is enough to compare $S(ABC)$ and $S(ABD)$.
2. Given points A, B, C, D, determine whether point D lies inside triangle ABC. To do this, it is enough to check whether $S(ABC)$ equals $S(ABD) + S(ACD) + S(BCD)$.
3. Given points A, B, C, D, determine whether AB and CD intersect. To do this, it is enough to check whether $S(ABC) + S(ABD)$ equals $S(ACD) + S(BCD)$. It is exactly in this point that we use the fact that $ABCD$ is not a trapezoid, because for trapezoids equality may hold without AB and CD intersecting.



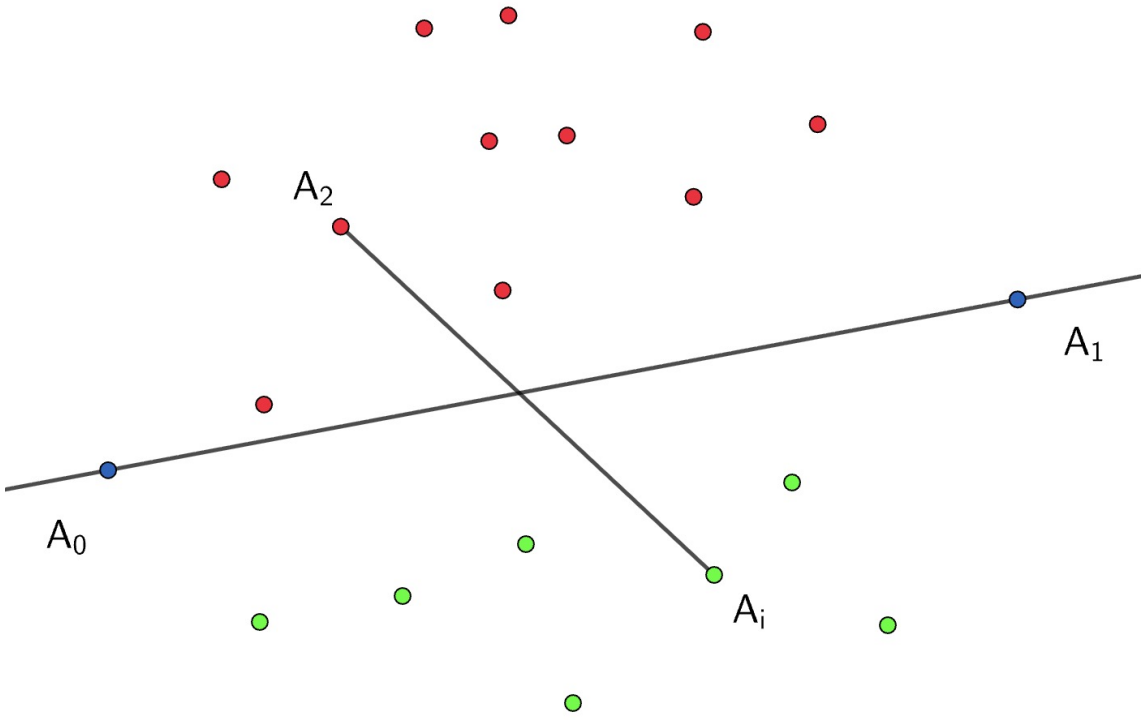
To solve groups 1 and 2 ($n = 4$), one can simply check whether each point lies inside the triangle formed by the other points. If so, the triangulation contains 3 triangles. Otherwise, the points form a convex quadrilateral. Its partition into 2 triangles can be found, for example, by checking which two segments intersect (they will be the diagonals of the quadrilateral).

Note the following idea: one can find a point lying on the convex hull in $n-2$ queries. To do this, take any two points, for example, A_0 and A_1, and among all the remaining points find the triangle with maximum area $S(A_0A_1A_i)$. Then A_i is definitely on the convex hull, because the distance $\text{dist}(A_i, A_0A_1)$ is maximal.

Thus, one can find three points on the convex hull in $\leq 3n$ queries. In groups 3 and 4, this will be the entire convex hull. To find a triangulation in this triangle in group 4, one can, for example, choose a random point inside it; it splits the triangle into 3 smaller triangles. Then, using queries of type 2, for each of the remaining points one can determine which of them it belongs to, and solve the problem recursively. Since the points inside the triangle are random, this will require $O(n \log n)$ queries.



To solve the next groups, first find two points A_0, A_1 (WLOG their indices are such) that lie on the convex hull. This will require $\leq 2n$ queries. Next, all remaining points can be split into two sets of points lying in different half-planes with respect to the line A_0A_1. To do this, one can take a third point A_2 and check each remaining point A_i by testing whether segments A_0A_1 and A_2A_i intersect. This will require another $\leq n$ queries. Then we will solve the problem for each of the half-planes independently. Therefore, now we assume that all points lie on one side of the line A_0A_1.



Sort the points by their distance to the line A_0A_1. This will require $\leq n \log_2 n$ queries.

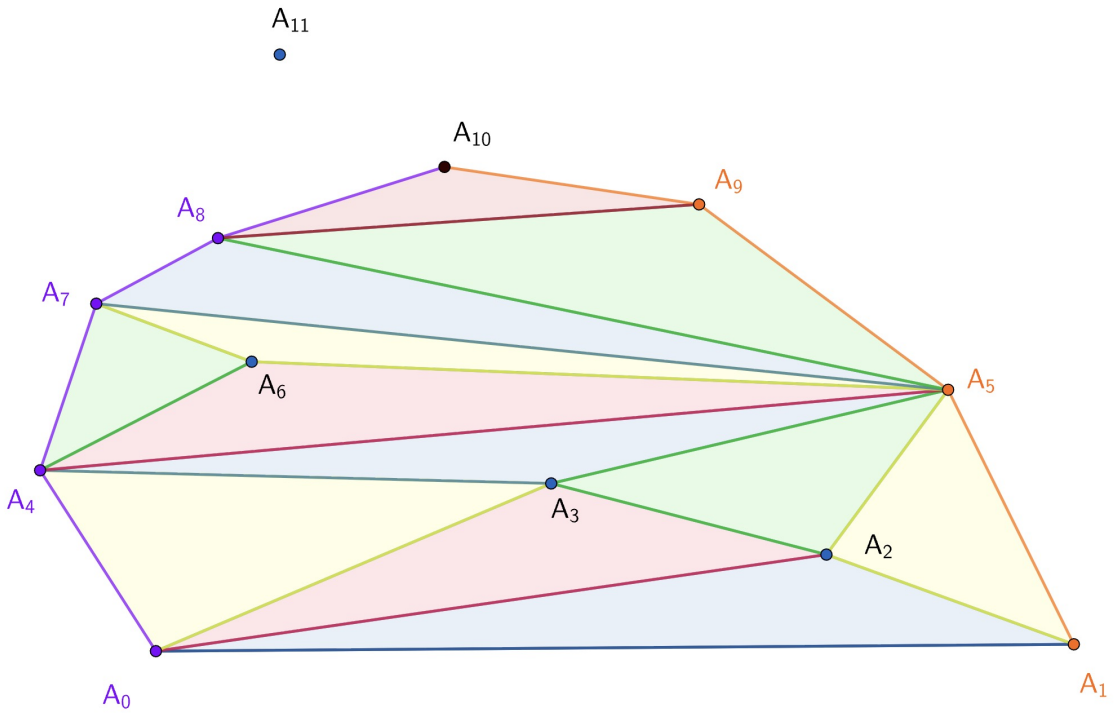
In group 5, one can take the farthest point A_i and again split all the remaining points into those lying on different sides of the line A_0A_i in $\leq n$ queries. Note that now the points in each part are monotone, and

we already know their order by height. Thus, the answer will be found in $\leq n \log_2 n + 4n$ queries, which fits comfortably.

Let us discuss the full solution. After sorting by height, apply an algorithm similar to Graham's scan, but adapted for points sorted by distance from a line rather than by angle. It turns out that during this algorithm we can also construct the triangulation.

Let the points be sorted by increasing distance from A_0A_1 in the order $A_2, A_3, \ldots, A_{n-1}$. After processing the prefix of this order $A_2, A_3, \ldots, A_{i-1}$, we will maintain the following state:

- We store the convex hull of the points $A_0, A_1, A_2, \ldots, A_{i-1}$ as the union of two chains, call them the left and the right one. The left one consists of points $[A_0, \ldots, A_{i-1}]$, the right one consists of points $[A_1, \ldots, A_{i-1}]$.
- We build a triangulation of this convex hull.

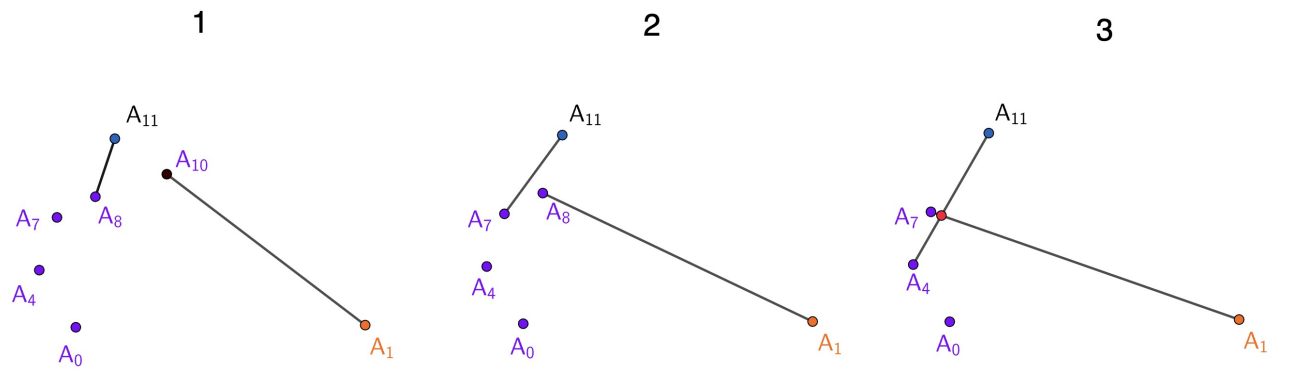


In this example, $i = 11$. The left chain consists of points $[A_0, A_4, A_7, A_8, A_{10}]$, the right chain consists of points $[A_1, A_5, A_9, A_{10}]$. We are adding point A_{11}.

Thus, in the end we will find the answer. For initialization (the only point A_2), build the left chain as $[A_0, A_2]$, the right one as $[A_1, A_2]$. The triangulation will contain a single triangle $A_0A_1A_2$.

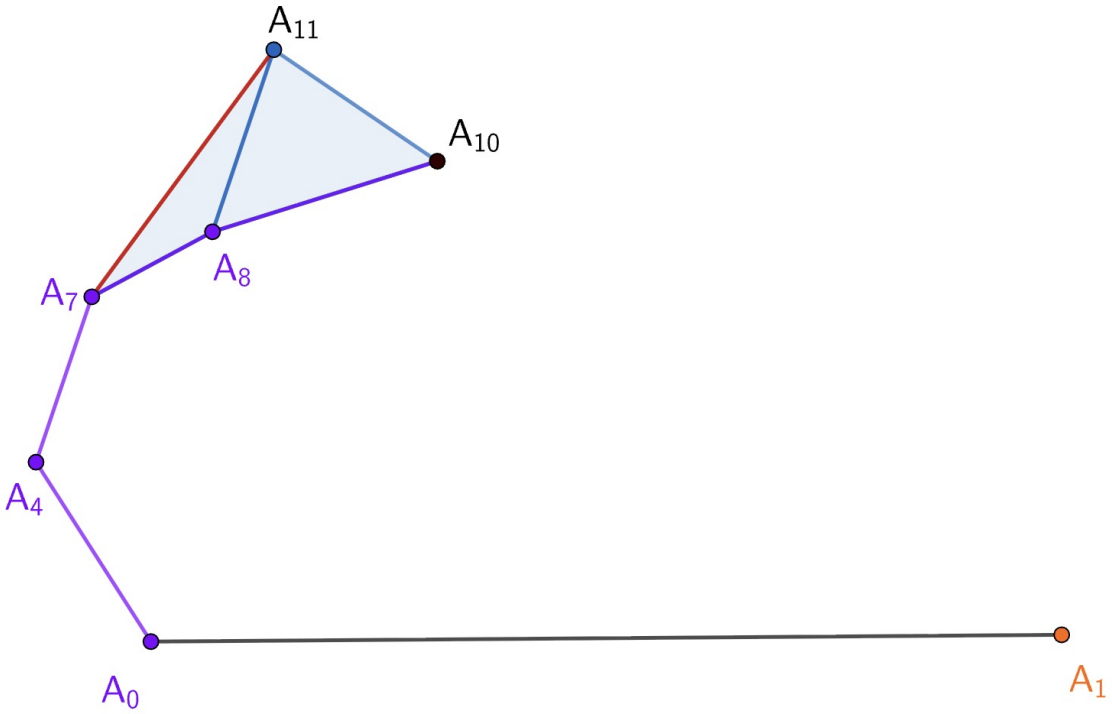
Next, let us learn how to add a new point A_i to this construction. Note that we need to add this point independently to the left and right chains.

Suppose the left chain is $[A_{l_0}, A_{l_1}, \ldots, A_{l_k}]$, where $l_0 = 0$, $l_k = i-1$. Note that now we need, as in the usual Graham scan, to remove some suffix of the chain and append A_i to the end. To remove the needed suffix, we should take the last two points A_{l_j}, $A_{l_{j+1}}$ and remove the last point $A_{l_{j+1}}$ if $\overrightarrow{A_{l_j}A_{l_{j+1}}} \times \overrightarrow{A_{l_{j+1}}A_i} > 0$ (cross product). Note that this is equivalent to saying that segments $A_{l_j}A_i$ and $A_{l_{j+1}}A_1$ do not intersect, which we know how to check using a type 3 query. Then let us remove the needed suffix of the chain using such checks.



When adding A_{11} to the left chain, we will do the following steps: do A_8A_{11} and $A_{10}A_1$ intersect? no, so remove A_{10}; do A_7A_{11} and A_8A_1 intersect? no, so remove A_8; do A_4A_{11} and A_7A_1 intersect? yes, so stop and append A_{11} to the end.

Moreover, whenever we remove the last point $A_{l_{j+1}}$ from the chain, let us add the triangle $A_{l_j}A_{l_{j+1}}A_i$ to the triangulation. Then these triangles will exactly cover the region formed between the tangent from point A_i and the old chain.



When updating the left chain, we add triangles $A_8A_{10}A_{11}$ and $A_7A_8A_{11}$ to the triangulation.

The same algorithm should be repeated for the right chain, but using point A_0 in the checks instead of A_1.

Thus, we will construct the convex hull and its triangulation. Let us note that this algorithm will require $\leq 2n$ queries for one chain, because each point can be added to and removed from the chain at most once. Therefore, this part of the solution will take $\leq 4n$ queries.

Thus, the entire solution requires $\leq n \log_2 n + 7n$ queries. If we estimate the actual sorting constant a bit more carefully when using merge sort, we can see that such a solution fits within $20n$ queries for $n \leq 10^4$. Note that the sorting estimate can be reduced a little more if, for sorting, we use an algorithm that spends $\leq \sum_{i=1}^{n} \lceil \log_2 i \rceil$ queries, by inserting a new index into the sorted order n times using binary search.

Problem Tutorial: "March of the Sheep"

Problem author: Alexander Babin

Problem developer: Yuri Fedorov

1 $T = 2$

Notice that every second the parity of each sheep's position changes. Therefore, if there exist two sheep with different position parity, then the answer is obviously `No`, since there will always be a sheep on both an even and an odd position. Otherwise, it is claimed that the answer always exists. For example, the following algorithm works:

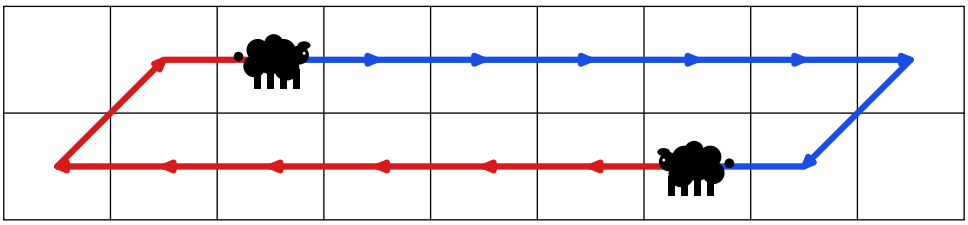
If the parity of the current last column does not match the parity of the sheep positions, then it is always possible to cut off a segment of length 1 (since there is guaranteed to be no sheep in the last column, due to parity). Thus, we can reduce the number of columns to 1, where all sheep are guaranteed to move in the same way.

2 $n \le 3, m \le 4$

If we solve $T = 1$, then there are very few remaining cases, and they can be checked by hand, or we can do brute force. As the state, we can take the positions of each sheep, as well as the length of the segment. Then there are two kinds of transitions: we simulate one second, or we decrease the segment. This can be implemented using any graph algorithm. There are also other brute-force approaches.

3 $n = 2$

In the case $a_1 = 1, a_2 = m$, we can draw it and see that the answer is $\frac{m+1}{2}$, and it is enough to cut down to this size after this same amount of time.



Now let us look at the two distances marked in blue and red. This is the distance from the first sheep to the second, and from the second to the first in the order of their path. Then notice two facts:

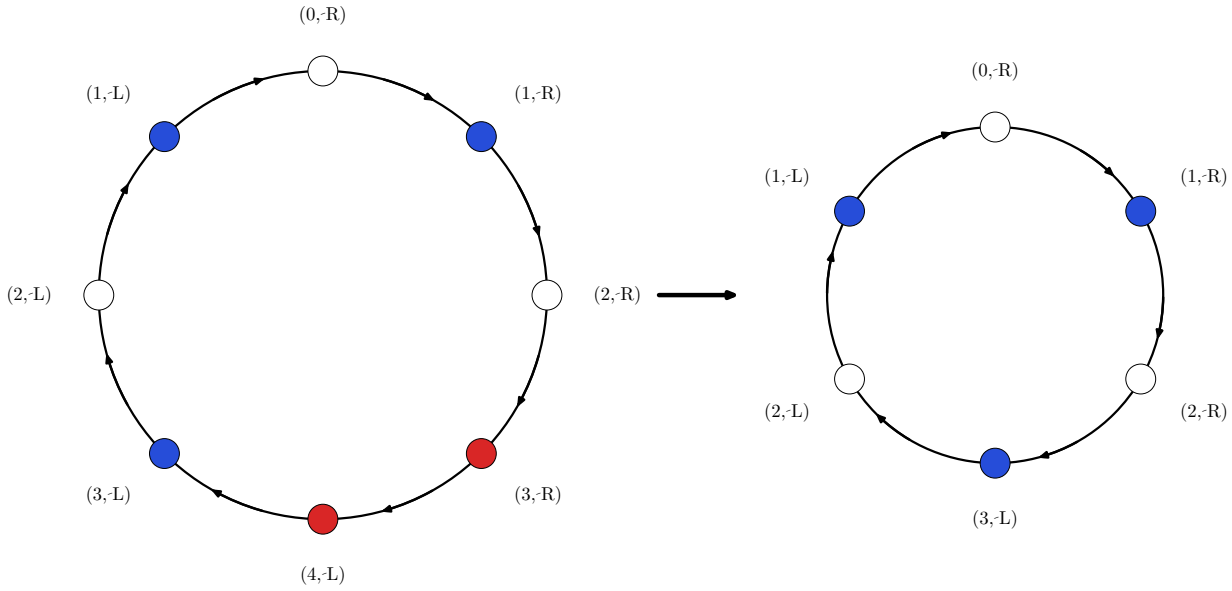
1. The sheep will move in the same way if and only if the distance from one to the other is exactly zero.
2. After decreasing the field size by 1, one of these distances decreases by 2.

From this, an $O(m)$ solution follows immediately: we can simulate their movement, and if cutting decreases the smaller of the distances, then we cut the segment by 1. In $4 \cdot m$ operations, this process is guaranteed to finish.

Now let us learn how to solve it in $O(1)$: notice that if we keep the larger of the distances, then if this length is x, the answer will be $\frac{x}{2} + 1$, since if we kept length x, then the distance from one sheep to the other would be $2 \cdot x - 2$. The details of reconstructing the answer will be considered in the full solution.

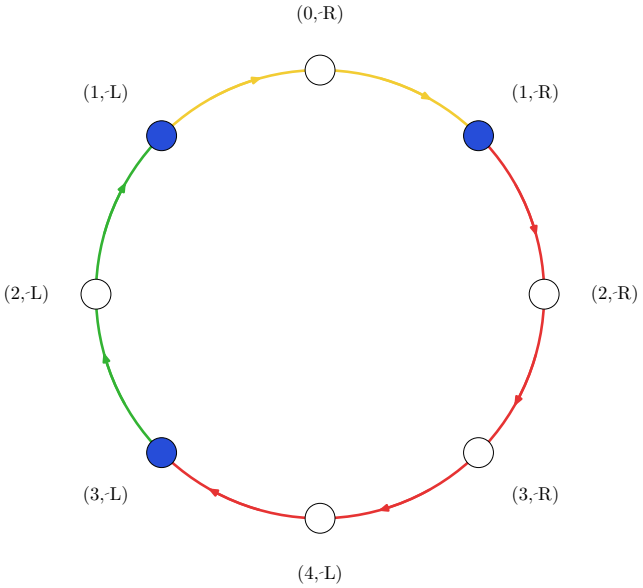
4 $T = 2$

Let us slightly develop the ideas from $n = 2$. Notice that the segment can be represented as a cycle of length $2 \cdot m - 2$. And decreasing the segment length by 1 is equivalent to decreasing the cycle length by 2:



This picture shows an example where 3 sheep are in positions $(2, 4, 2)$, and their movement directions are `RLL` (that is, the blue points are the sheep). For convenience of the further explanation, the positions are shifted to 0-indexing. The red vertices mark those that are removed after decreasing the cycle.

Then the distances to neighboring sheep can be represented as the number of arcs from a sheep to the next sheep along the cycle (here different colors denote different arcs, which represent distances to neighboring sheep):



Then notice that this construction is not fundamentally different from the $n = 2$ situation. We are still only interested in the maximum distance between neighboring sheep on the cycle. And, as in the $n = 2, m \le 2 \cdot 10^5$ version, we can simulate the movement and cut at moments when the current length is not the maximum; in this case the complexity is $O(n \cdot m)$. Or we can say that the answer is $\frac{x}{2} + 1$, where x is the maximum distance between neighboring sheep.

5 Full solution

Now we only need to learn how to reconstruct. In the case $n, m \leq 1000$ we already know how to solve it, so it remains to learn how to reconstruct under the full constraints.

Let us sort all sheep by their position on the cycle. And let i be the position of a sheep such that the arc from sheep i to the next sheep on the cycle is the largest. Then reconstruction can be done as follows:

- Renumber the array so that sheep number i becomes the last sheep.
- Let us wait long enough so that the last sheep is at position $m-1$ on the cycle in 0-indexing (in other words, so that it stands in the last column).
- Then, to cut the arc from sheep $n-1$ to sheep n, it is enough to "rotate"the cycle so that the sheep is at position $m-1+\frac{d}{2}$, where d is the length of the arc from sheep $n-1$ to sheep n. Then decrease the segment length by $\frac{d}{2}$, after which sheep number n will stand at position $m-1$ on the cycle.
- After that, we can ignore sheep number n and process sheep number $n-1$, and so on.

And it is easy to notice that this indeed removes all arcs except the maximum one.

Problem Tutorial: "Decoration"

Problem author: Alexey Mikhnenko

Problem developer: Viktor Romanenko

Formal Problem Statement

Given n segments on a line. The following operation is allowed: select two segments (l_i, r_i) and (l_j, r_j) and arbitrarily form two new segments from their four endpoints.

There are q queries k_i. For each query, it is required to determine the minimum number of operations needed to ensure that there exists a subset of k_i pairwise intersecting segments.

Main Idea

Let all segments of the desired set intersect at some point x. Assume that:

- c is the number of segments already covering point x,
- l is the number of segments located strictly to the left of x,
- r is the number of segments located strictly to the right of x.

Note that with one operation, one can take one left segment and one right segment and rearrange them so that both intersect point x. Therefore, with one operation, the number of segments intersecting x can be increased by two.

Thus, the maximum number of segments that can be made to intersect at point x is equal to

$$c + 2 \cdot \min(l, r).$$

To achieve this value, $\min(l, r)$ operations will be required. If an intermediate number k_i is needed, it is sufficient to perform

$$\left\lceil \frac{k_i - c}{2} \right\rceil$$

operations.

Subgroup 3. Segments Do Not Intersect

In this case, for any point, $c \leq 1$. Therefore, it is convenient to consider a point inside the central segment.

For any point inside such a segment, $c = 1$, hence the answer for any k_i is calculated using the formula

$$\left\lceil \frac{k_i - 1}{2} \right\rceil.$$

If the number of segments is even and $k_i = n$, then the intersection point must be located between the two central segments, where $c = 0$. In this case, the answer is equal to $n/2$. Since n is even, this value is equivalent to the formula above.

Subgroup 4. $k_i = n$

If it is required to make all segments intersect, the answer point must be located at a position where the number of segments is strictly equal on both the left and right.

Such a point can be found using a sweep line algorithm. After finding the corresponding values of c, l, and r, the answer is calculated using the previously mentioned formula.

Solution for Subgroups 1, 2, 4, 5

We will perform a sweep line. For each opening boundary of a segment and each closing boundary, we will fix the values:

- c — the number of segments covering the current point,
- $\min(l, r)$ — the minimum number of segments strictly to the left and strictly to the right.

When processing a closing boundary, the corresponding segment is no longer counted in c, but is considered to be strictly to the left, since the intersection point may be outside its boundaries.

For each such position, the minimum number of operations can be calculated using the formula

$$\left\lceil \frac{k_i - c}{2} \right\rceil,$$

provided that

$$k_i \leq c + 2 \cdot \min(l, r).$$

The minimum across all such positions gives the answer for subgroups 1, 2, 4, and 5.

Complete Solution

It remains to optimize the computation of answers.

For each k from 1 to n, we will precompute the maximum value of c such that it is possible to obtain k intersecting segments. We denote this quantity as $best[k]$.

During the sweep line, for each position, we compute the value

$$c + 2 \cdot \min(l, r),$$

which shows the maximum number of segments intersecting at this point after operations. Then we perform the update

$$best[c + 2 \cdot \min(l, r)] = c.$$

After completing the sweep line, we will traverse the $best$ array from right to left and propagate the maximum:

$$best[i] = \max(best[i], best[i + 1]).$$

Now $best[k]$ contains the maximum number of segments already covering the point from which k intersecting segments can be obtained.

The answer to the query k_i is computed using the formula

$$\left\lceil \frac{k_i - best[k_i]}{2} \right\rceil.$$

In total, the complexity is $O(n \log n)$ for the preprocessing due to sorting and $O(1)$ for each query

Problem Tutorial: "Monsters and Swords"

Author and problem setter: Renat Karimov

Subgroup $k = 1$

To solve the subgroup $k = 1$, first introduce a new notation: w_i = the minimum cost of a sword with which it is possible to kill the i-th monster. If there is no such sword for some monster, then the answer is immediately NO.

In this subgroup, we have to kill the monsters one by one in order, so each time we need to find the cost of the cheapest sword that can kill the next monster. This is exactly w_i. Therefore, to solve the problem, we just need to check n times that the number of coins we have left is greater than or equal to the next w_i. The solution works in $O(n \log n + m \log m)$

Subgroup $k = n$

To solve the subgroup with $k = n$, we need to notice that when buying a new sword, we always need to choose a sword with strength greater than the previous one. This is because we have no restriction on sword durability. Then from this fact it follows that in an optimal solution we can always use a sword until it is exhausted. That is, we only need to change swords in the situation when it cannot kill the next monster.

Using these facts, we can write a solution using the dynamic programming technique. Let $dp[i]$ = the maximum number of coins that can remain with the knight after killing the first i monsters (while it does not matter which swords were used to kill the monsters).

For convenience, let $w(i, j) = \max\limits_{i \le l \le j} w[l]$. Then the initial state is: $dp[0] = x$ and the transition is $dp[i] = \max\limits_{j} dp[j] - w(i, j-1) + r(i, j-1)$. At the same time, it must hold that $dp[j] \ge w(i, j-1)$.

To optimize this DP, we can notice that we are not interested in all states of this DP. For each sword, we can identify a prefix of monsters that it can defeat. Then, based on the facts above, we can recompute the DP only through these prefixes, and thus obtain a solution in $O(m^2 + n \log n)$.

For the full solution, we just need to optimize the dynamic programming above; this can be done, for example, using a Cartesian tree structure, then the final solution will work in $O(n \log n + m \log m)$

Subgroup k - arbitrary

To solve the subgroup with arbitrary k, we need to examine the resulting formula in more detail. Let the value i be fixed; then, as j decreases, the value of $w(i, j)$ will increase. Because of this, we would like to use the maximum stack technique. With its help, we can implement the same DP as in the case $k = n$, but with the additional condition that $i - j \le k$. This gives a solution in $O(n \log n + m \log m)$ using a Cartesian tree, HLD, or a sweep with a set.

However, we will discuss a slightly different DP in more detail. Let $dp[i]$ = the minimum amount of gold we need in order to be guaranteed to kill all monsters with indices $i...n-1$ (in increasing order of indices). Then the answer is YES only if $dp[0] \le x$.

Initially: $dp[n-1] = w[n-1]$. Transition: $dp[i] = \min\limits_{i<j\le i+k}(w(i,j-1) + \max(0, dp[j] - r(i,j-1)))$. Let $pr[i] = r(0,i)$ and rewrite the resulting formula a bit; we get $w(i,j-1) + max(0, dp[j] - pr[j-1] + pr[i])$. Then we are interested in the sign of $dp[j] - pr[j-1] + pr[i]$. It is not hard to notice that if now j is fixed and i decreases, then the value of $dp[j] - pr[j-1] + pr[i]$ only decreases, so the sign changes at most once.

Such a DP can be quickly recomputed using a maximum stack and sets, noticing the fact that $dp[i] - pr[i-1] \ge dp[i+1] - pr[i]$.

The final solution will have complexity $O((n+m)\log(n+m))$