Problem Tutorial: "Queen Placement"

Problem author and developer: Zakhar Iakovlev

Let us consider the groups one by one in increasing order of difficulty.

Queens located in the same column attack each other, so in the first test case the arrangement is correct only if it contains exactly one queen, that is, $l_i = r_i$.

In the second group, for each query and each pair of queens within the segment, we can check whether they attack each other in $\mathcal{O}(qn^2)$.

Now we need to understand how to quickly determine whether a newly added queen attacks some previously placed queen. To do this, note that it is enough to check whether there is already a queen in the same row, the same column, or on the same diagonals as the new one. We will process the queens in the order they are added, and for each row, column, and diagonal (defined by a fixed sum or difference of coordinates), store the index of the last queen on that line. In groups up to the fifth, this can be done explicitly using $\mathcal{O}(m)$ memory. Then, when adding a new queen, we will explicitly find the queen with the maximum index that it attacks, forming the array $last$.

In the third group, it is enough to iterate over all queens in the segment and check whether any of them attacks a queen from the same segment ($last[j] \geq l_i$), which works in $\mathcal{O}(nq + m)$.

In the fifth group, all segments are prefixes of the array, so it is enough to note that up to some r_i there is no pair of queens that attack each other, and after that such a pair appears. Therefore, for small r_i the answer is always «`Yes`», and for large ones it is «`No`». This boundary can be found explicitly in one pass by finding the first queen that attacks a previously placed one. The resulting complexity is $\mathcal{O}(n + q + m)$.

In the fourth group, we need to use the fact that the queries are on contiguous segments. Then it is enough to know not the last queen attacked by a given queen, but the last pair of queens that attack each other. Moreover, it is enough to know the index of the earlier queen in this pair. This index corresponds to the prefix maximum of the array $last$. Now it is enough to check whether the last pair by the time we reach the end of the segment r_i occurs after the beginning of the segment l_i. The resulting complexity is $\mathcal{O}(n + q + m)$.

Groups 6 and 7 differ from groups 5 and 4 by the absence of the restriction on m. This restriction can be bypassed if we store only the lines on which at least one queen is located. This can be done by compressing coordinates in advance or by storing the used lines in a map. The resulting complexity in these groups is $\mathcal{O}(n \log n + q)$.

Problem Tutorial: "History"

Problem author: Ilya Burkov, developer: Andrey Pavlov

Formally, the problem asks us to construct a graph on n vertices that contains all edges (i, j) such that $a_i + a_j = S$ or $a_i \oplus a_j = X$. The answer to the problem is a perfect matching in this graph, and the graph is not necessarily bipartite.

To solve the subtask with $n \leq 20$, one could enumerate all possible partitions into pairs and check whether they satisfy the conditions.

In subtask 3, the graph contains no edges satisfying $a_i + a_j = S$. Therefore, only pairs of the form $(a, a \oplus X)$ are valid. Then it is enough to count cnt_a — the number of occurrences of each value — and check that $cnt_a = cnt_{a \oplus X}$ provided that $X > 0$.

If $X = 0$, then all valid pairs satisfying the XOR condition are such that $a_i = a_j$, so for subtask 3 it is enough to check that cnt_a is divisible by 2 for every a. For subtask 4, we again have pairs of the form $a_i + a_j = S$, and we also know how to pair two equal values a, so it is enough to check that $cnt_a \equiv cnt_{S-a} \pmod 2$, while handling the case $2a = S$ separately.

From now on, we will assume that the case $X = 0$ has been handled separately and that $X > 0$.

Consider subtask 6, where all a_i are distinct. Let us understand which vertices are adjacent to vertex i: from the two equations we get that $a_j = S - a_i$ or $a_j = X \oplus a_i$. Since all numbers are distinct, such a j is unique in both cases. Therefore, each vertex i has degree at most 2 in our graph, and sometimes less, because such a j may not exist in the array. Hence, our graph consists of connected components that are simple cycles and paths. For such a graph, a matching can be found constructively using any graph traversal algorithm in $O(n)$.

Now let us move to the full solution. Here each number may occur arbitrarily many times, so we will build our graph a bit differently: we store the same array *cnt*, keep only distinct numbers in the array, build the same graph, and try to apply the solution for subtask 6. The main difference is that now each vertex additionally has a value *cnt* written on it. Essentially, for each edge we have an operation that decreases the number on both of its endpoints by 1, and using such operations we need to make all values equal to 0.

For a path, it is enough to understand the following: suppose one endpoint of the path has value x. Then we must take the only edge incident to this endpoint exactly x times. After that, the value at this vertex becomes 0, and at the other endpoint it becomes $y - x$. We can then mentally remove this vertex, after which the path becomes shorter by 1, and we can process the whole path iteratively. If at some point $y - x < 0$, or if one vertex with a nonzero value remains, then no answer exists.

Now suppose we consider a cycle. We can fix any vertex and write out the cycle from it in some direction. Then we could brute-force how many times the first edge of the cycle is taken; after that we can mentally forget about this edge and reduce the problem to the path case. However, this is too slow, so let us optimize it. Formally, suppose the numbers written on the cycle in order are $c_1, c_2, \ldots, c_k$, and we try some $x \le \min(c_1, c_k)$, after which we consider the path with numbers $c_1 - x, c_2, \ldots, c_{k-1}, c_k - x$. Let us see what conditions must hold for this path to be valid: $c_1 \le c_2$, $c_2 - c_1 \le c_3$, $c_3 - (c_2 - c_1) \le c_4$, $c_4 - (c_3 - (c_2 - c_1)) \le c_5$, ..., and $c_k - (c_{k-1} - (\ldots - (c_2 - c_1))) = 0$. Now if we insert the parameter x here, then for every even i we get some expression involving $-x$, while for odd i we get one involving $+x$. Therefore, we obtain $O(k)$ upper and lower bounds on x; denote them by *low* and *high*. Then it is enough to take any x from the interval $[low, high]$ that also satisfies the final condition, which is also easy to handle. One implementation issue is the case $2 \cdot a_i = S$, but in this solution it is handled quite simply.

The final implementation can be written using hash maps in $O(l_1 + l_2 + \ldots + l_m) = O(n)$, where l_i is the size of the i-th component, which is enough for full score.

Problem Tutorial: "Jelly Candies"

Problem author and developer: Timofey Izhitskiy

Let us learn how to solve the problem on a static array by finding its maximum lexicographic subsequence b. This can be done greedily: first, we want to make the first element as large as possible. Therefore, it is optimal to choose as the first element b_1 the maximum element of the array, and among all such elements, the leftmost one. Once we have chosen the first element of b, the rest of the problem can be solved independently on the suffix after it. Thus, we obtain a greedy algorithm that repeatedly chooses the maximum element on the current suffix. For the first subtask, this could be implemented naively, and for the second one, the greedy process could be simulated using a segment tree.

To solve the other subtasks, a different point of view is needed. Suppose we have an array, and let us understand how the maximum lexicographic subsequence changes if we append a new element to the right end of the array. Consider how the greedy algorithm described above works on these two arrays. At first, the greedy algorithms will produce the same result, since the maximum is the same, so let us look at the first difference. This difference could only appear because of the newly added element, which means that this new element becomes the maximum, after which the greedy algorithm on the new array terminates. Therefore, to obtain the new optimal subsequence, we need to remove some suffix of elements from the old maximum lexicographic subsequence and append the new element at the end. Suppose we append an

element with value x. In order for the new subsequence to be maximum, we need to remove from the old maximum lexicographic subsequence all suffix elements that are strictly smaller than x. Removing exactly these elements is optimal.

Thus, in fact, such a subsequence is simply the sequence of all suffix records. This observation is enough to solve the third and fourth subtasks, where there are no updates. We process the problem offline: fix the right boundary and answer all queries with that boundary. Then we can maintain a stack of suffix records and use binary search to find the answer.

To solve the sixth subtask, one could use sqrt decomposition, but that is not enough for a full solution. Instead, we build a segment tree that answers all queries. Each segment tree node stores:

- S — the size of the record stack in the node;
- $\mathrm{mxs}, \mathrm{mns}$ — the first and the last elements of the maximum lexicographic subsequence (in fact, this is the maximum and the last element of the node).

Let us define a recursive function $\mathrm{split}(V, x)$ — the length of the answer inside node V if we are only allowed to take elements $\geq x$. Let L, R be the left and right children of this node. Consider two cases:

- $R(\mathrm{mxs}) \geq x$, that is, the maximum on the right is at least x. Then $\mathrm{split}(V, x) = \mathrm{split}(R, x) + S - R(S)$. If the maximum on the right is at least x, then the elements on the left that entered the record stack of node V must also be at least x, so they are all valid, and their number is exactly $S - R(S)$. To count the contribution from the right, we simply recurse there.
- $R(\mathrm{mxs}) < x$. Then the maximum on the right is less than x, so no element there is suitable, and therefore we can safely recurse to the left: $\mathrm{split}(V, x) = \mathrm{split}(L, x)$.

This function works in $O(\log n)$, and with its help we can define all the others. First, let us learn how to merge two segment tree nodes. Namely, suppose we know all values in the subtree of V, and we want to compute the values for V itself. Clearly, we take the entire record stack from the right child R, and from the left child we take only those elements that are at least $R(\mathrm{mxs})$ and belong to the left record stack. This is exactly what the function split can tell us, so merging works in $O(\log n)$. Then, using a lazy segment tree, we can maintain the required values in the nodes under range updates. Moreover, each update query will take $O(\log n^2)$.

Now let us solve the subtask where $k \in \{m, m+1\}$. Essentially, there we only need to determine the size of the record stack on a segment, since the last element is easy to determine. Define a function $\mathrm{query}(l, r, x)$ — the length of the record stack on the query segment if we are only allowed to take elements $\geq x$, and also the maximum element. It works almost like a standard segment tree query. If we need to split into two segments, we first compute the record stack on the right, and from the left we take only elements greater than the maximum on the right. If a segment is fully covered, we call split. Thus, we can answer a query in $O(\log n^2)$.

For the full solution, it remains to implement descent procedures based on these functions in order to restore the actual elements. They are analogous to the previous ones: $\text{split-k}(V, x, k)$ — which element is the k-th in the record stack inside node V, if we are only allowed to take elements $\geq x$; and $\text{query-k}(l, r, x, k)$ — which element is the k-th in the record stack inside the query segment, if we are only allowed to take elements $\geq x$. It is most convenient to number the elements from the end. Their transitions are analogous to the previous ones; we only additionally need to determine in which direction to go during the descent. The final complexity is $O(n \log n + q \log n^2)$.

Problem Tutorial: "Rectangular Apartment"

Problem author and developer: Alexey Mikhnenko

First, let us assume that the turtle's apartment is infinite and contains no furniture. Consider which cells the turtle will visit if it starts its path from cell $(1, 1)$. If the turtle eventually shifts to the right by more than m, then the answer to the problem is 0. Otherwise, for each y from 1 to m, it is easy to see that the turtle either visits no cells in column y, or visits a segment of cells $[l_y, r_y]$: that is, all cells (x, y) where $l_y \le x \le r_y$.

The values l_y and r_y can be precomputed in $\mathcal{O}(|s|)$. Using these values, it is then easy to check in $\mathcal{O}(m)$ for a fixed cell (i, j) whether it satisfies the condition of the problem:

- For each y from 1 to m, if the segment $[l_y, r_y]$ is non-empty, one must verify that $j + y - 1 \le m$, and also that all cells in this column on the segment $[i + l_y - 1, i + r_y - 1]$ exist and are not occupied by furniture.

To perform this check efficiently for each column, it is sufficient to compute prefix sums in every column. This method allows checking a fixed cell in $\mathcal{O}(m)$, which gives a solution in $\mathcal{O}(|s| + nm^2)$ if we perform the check independently for every cell. This solution is sufficient to solve the problem for full score.

This problem can also be solved in $\mathcal{O}(|s| + nm \log(nm))$ using multiplication of two-dimensional polynomials, but this was not required.

Problem Tutorial: "Simple Problem"

Problem author and developer: Alexey Vasilyev

First, let us compute an auxiliary array $closest_{v,b}$ — the distance from vertex v to the nearest vertex whose number contains the set bit b ($0 \le b < k$). We solve this independently for each bit. When we want to compute *closest* for some particular bit, we run a multi-source BFS from all vertices that contain this set bit. Alternatively, this can be done using subtree DP. The complexity of this part is $O(nk)$.

Suppose we know the answer set A, and fix in it the vertices at maximum distance from each other. Let these vertices be a and b. Call the path between these vertices the diameter, and let the length of this diameter be the number of edges in it (denote the length by d).

Let us consider two cases:

1. Suppose the diameter length is even. Let vertex m — be the middle of the diameter. Then all vertices from A are at distance at most $\frac{d}{2}$ from m (if this is not the case, then it can be shown that we chose the wrong diameter). This means that if we take into the answer set all vertices at distance at most $\frac{d}{2}$ from m, it will not make things worse for us (that is, the set will not decrease, and the cost of this set will remain the same).

   We process this case as follows. Fix a vertex m. Now we want to find the minimum x such that if we take into the answer set all vertices at distance at most x from m, then they will have the OR value we need. It is claimed that x is the maximum value of $closest_{m,b}$ over all bits b. Thus, we find this x and relax the answer with $2x$ (because x is $\frac{d}{2}$).

2. Suppose the diameter length is odd. Then in the middle there is not a vertex, but an edge. Let the vertices of the middle edge be m_1 and m_2. Then, similarly to the previous item, we say that all vertices of the set A are at distance at most $\frac{d-1}{2}$ from vertex m_1 or from vertex m_2. And if we take into the answer set all vertices at distance at most $\frac{d-1}{2}$ from m_1 or m_2, it will not make things worse for us.

   Therefore, similarly to the previous item, we iterate over the edge m_1, m_2. Next, we want to find the minimum x such that if we take into the answer set all vertices at distance at most x from m_1 or m_2, then they will have the OR value we need. It is claimed that x is the maximum value of $min(closest_{m_1,b}, closest_{m_2,b})$. We find this x and relax the answer with $2x + 1$ (because x is $\frac{d-1}{2}$).

One can also avoid thinking about the second case separately as follows: add a dummy vertex with value 0 in the middle of each edge and solve the problem on this new tree. After this modification of the tree, the center of the answer diameter will always be a vertex, so one only needs to think about the first case.

The complexity of the second part is $O(nk)$, because we considered $O(n)$ candidates for the center of the diameter, and for each of them computed x in $O(k)$.

Problem Tutorial: "Minimums on Arcs"

Problem author and developer: Alexander Babin

Subtask 1.

In this subtask, the permutation is restored as follows:

$$p = [1, f(1,2), 2, f(2,3), 3, \ldots, n-1, f(n-1,n), n]$$

Subtask 2.

In this subtask, it was possible to make all possible queries $f(i,j)$ in advance, and then restore the permutation in $O(n^2)$ time. This can be done as follows:

- Let $g(x)$ be the number of such y ($1 \le y \le n$) that $f(x,y) = 1$.
- It is clear that any element $x \neq 1$ is at distance $\frac{n+1}{2} - g(x)$ from one.

Place one at position $\frac{n+1}{2}$ in the permutation and arrange arbitrarily the elements that must be at distance exactly 1 from it. This can be done because the function f does not change if we apply a cyclic shift or a reversal to the permutation.

For each element x at distance $\le \frac{n-3}{2}$ from one, we can uniquely determine in which half of the permutation it lies: it is enough to check whether $f(p_{\frac{n-1}{2}}, x) = 1$. If this equality holds, then the element lies in the right half; otherwise, in the left half.

The positions of elements p_1 and p_n can then be brute-forced, and the correctness of all queries can be checked naively. In some cases, it is impossible to uniquely restore their relative order.

Subtask 3.

This subtask guaranteed that solutions better than quadratic could score at least some points.

Solution for 70+ points. (Randomized, complicated, but provable)

The solution can be split into three phases:

Phase 1. Splitting elements into two halves relative to one.

We will maintain 5 sets of elements: L — the set of elements to the left of one, R — the set of elements to the right of one, $L' \subset L$, $R' \subset R$, and C — the set of elements at distance $\frac{n-1}{2}$ from one. The sets L' and R' will contain some subset of the elements closest to one from the sets L and R.

Choose a random element x for which we have not yet been able to determine on which side it lies:

- If $L = R = \emptyset$, then form the set $R = R' = \{y \mid f(x,y) = 1\}$. Otherwise, it is guaranteed that $L' \neq \emptyset$ and $R' \neq \emptyset$ (see the next item), and then we can form the pair of sets $L(x) = \{y \in L' \mid f(x,y) = 1\}$ and $R(x) = \{y \in R' \mid f(x,y) = 1\}$.
  - If $L(x) = R(x) = \emptyset$, then the element x is at distance $\frac{n-1}{2}$ from one and is a *corner* element, so it is added to the set C, and the next step of the algorithm is not performed.

  - If $R(x) \neq \emptyset$, then $L(x) = \emptyset$. Therefore, we can set $R' \leftarrow R(x)$ and assign x to the left half.
  - If $L(x) \neq \emptyset$, then $R(x) = \emptyset$ and we should set $L' \leftarrow L(x)$.
  - It is not necessary to build both sets $L(x)$ and $R(x)$ simultaneously; one can first check one of them for emptiness, and only if necessary build the second set.

- Suppose we have determined that x lies in the left half (that is, in the previous item we updated R'). Then take an arbitrary random element $y \in R'$ after the update. Form the set $L(y) = \{z \notin L \cup R \mid f(z, y) = 1\} \cup \{x\} \cup L$, and set $L \leftarrow L(y)$. If $L' = \emptyset$, then set $L' \leftarrow L$.

- Proceed analogously if x lies in the right half.

Thus, if we repeat the two items above, the values $2, \ldots, n$ will eventually be split into 3 groups L, R, C — left values, right values, and corner values. Note that during the execution of the algorithm, the sets $L(x)$, $R(x)$, $L(y)$, $R(y)$, whenever nonempty, coincided with $\{z \mid f(x, z) = 1\}$ or $\{z \mid f(y, z) = 1\}$ (proof left as an exercise). Moreover, one can show that at each operation the quantity $n - |L(x)| - |R(x)|$ decreased by a factor of two on average, and the size of L' or R' also decreased by about a factor of two. Therefore, the first phase requires $O(n)$ queries. Of course, one can perform a more careful analysis and compute the expected number of queries rigorously.

Denote $G(x) = \{y \mid f(x, y) = 1\}$. Since for some elements we have already determined $G(x)$, and for all elements we have already determined on which side of one they lie, for such elements we can already uniquely restore their position. Moreover, the different values of the computed sets $G(x)$ for $x \in L$ allow us to split all elements y into groups of consecutive values on the cycle. Indeed, if $G(x_1) \subset G(x_2)$, $x_1, x_2 \in L$, then all elements from $G(x_1)$ are closer to one than the elements from $G(x_2) \backslash G(x_1)$.

Phase 2. Refining the positions of elements from L and R.

Split all elements from R into blocks $L \backslash G(x_1)$, $G(x_1) \backslash G(x_2)$, $G(x_2) \backslash G(x_3)$, $\ldots$, $G(x_{k-1}) \backslash G(x_k)$, where $x_i \in L$ and $G(x_k) \subset G(x_{k-1}) \subset \ldots \subset G(x_1) \subset L$; let these be the blocks $B_1, \ldots, B_k$. For each block B_i, one can determine the set of positions occupied by the values from B_i, and these sets of positions do not intersect for different blocks. Moreover, it is clear that if some block contains exactly one value $x \in B_i$ whose position is unknown, then it can be restored immediately.

Thus, the more values $G(x)$ ($x \in L$) we know, the more precisely we can place the elements from the right half. The values $x \in L$ can be split into blocks in exactly the same way, guided by the computed sets $G(y)$ ($y \in R$).

If we choose a random $x \in L$ whose position is still unknown, then it is not necessary to check all $y \in R$ in order to build $G(x)$. First, there are values $x' \in L$ for which we know $G(x')$ and which are guaranteed to be either to the left or to the right of x (based on the partition into blocks of the left half), so from this we can conclude $G(x) \subset G(x')$ or $G(x') \subset G(x)$ depending on their relative order. Second, knowing the values $G(y)$ ($y \in R$), we can also narrow the interval of values for which it is unknown whether they should lie in $G(x)$ ($x \in G(y) \Leftrightarrow y \in G(x)$ + some information is known about the relative order of blocks and already placed elements).

After these prunings, only some interval of blocks from the right half remains, $B_l, \ldots, B_r$, for whose elements it is unknown whether they lie in $G(x)$ or not. First there will be blocks that do not lie in $G(x)$, then a block that lies partially in $G(x)$, and then blocks that are fully contained in $G(x)$. Therefore, we can use binary search and determine in $\log_2(r - l)$ the pair of blocks that may partially lie in $G(x)$. Then, in time proportional to the sum of the sizes of these blocks, we determine which block is partial and split it into a pair of blocks.

Thus, we have devised a fairly query-efficient way to split blocks into even smaller blocks. The proposed solution randomly chooses an element x for which $G(x)$ is unknown (elements from L and from R are chosen with equal probability), and then performs the described block-splitting procedure. Eventually all blocks are split into blocks of size 1, and victory is achieved. Of course, the sets $G(x)$ are not maintained directly; only the partition into blocks is computed.

This part can be implemented to run in expected $O(n \log n)$ time. For this, it is enough to work in time $O(|B_x| + |B_a|)$, where $|B_x|$ is the size of the block containing x, and $|B_a|$ is the total size of the blocks from the other half that turned out to be relevant for consideration.

Phase 3.

After the third phase, all values $p_2, \ldots, p_{n-1}$ are restored ($p_{\frac{n+1}{2}} = 1$); it remains only to place the elements from C (corner elements) into positions p_1 and p_n. This cannot always be done uniquely, but it is claimed that it is sufficient to consider the values $f(c, p_2)$ and $f(c, p_{n-1})$ ($c \in C$) and from them restore any suitable arrangement of the corner elements.

It is not easy, but more or less intuitively, one can show that the expected number of queries of such a solution does not exceed $O(n \log n)$.

Solution for 85+ points. (Information theory)

Suppose we are in the second phase of the algorithm. Essentially, we have already determined C — the corner elements (at distance $\frac{n-1}{2}$ from one), determined the positions of some elements, and split the remaining elements into blocks $B_1, \ldots, B_k$, where each block consists of some values for which it is known that they occupy some set of consecutive positions on which no restored permutation elements have yet been placed (some already found element may lie between two possible positions for one block), and these sets of positions do not intersect.

Then, given this information, there are

$$2^J = |B_1|! \cdot |B_2|! \cdot \ldots \cdot |B_k|!$$

possible permutations, and the amount of information J is therefore

$$J = \sum_{i=1}^{k} \log_2 |B_i|! = \sum_{i=1}^{k} \sum_{j=1}^{|B_i|} \log_2 j.$$

In many problems, a heuristic based on information theory is applicable — it is enough to choose a greedy strategy that at each step maximizes the ratio of the decrease in the amount of information to the number of spent queries, i.e. maximize the expected value of

$$\frac{-\Delta J}{\Delta q}$$

over all allowed steps, where ΔJ is the amount of information after the next step, and Δq is the number of queries spent on this step.

In our case, a step consists of choosing some element x with unknown position, and then finding $G(x)$ for it ($G(x) = \{y \mid f(x, y) = 1\}$), after which we learn the position of the element x and also split some block into two parts.

Let us compute the number of queries and the decrease in the amount of information that occurs if we determine the position of the element with number i:

- $-\Delta J = \log_2 |B(i)| - \log_2(X!) - \log_2(Y!) + \log_2((X + Y)!)$, where $|B(i)|$ is the size of the block containing element i, and X and Y are the sizes of the blocks into which the block is split after locating block i.
- $\Delta q \approx X + Y + \log_2 M(i)$, where X and Y are the same quantities as in the previous item, and $M(i)$ is the number of blocks that can be split by an element x' lying in the same block as x.

Accordingly, the expected value of the ratio of obtained information to spent actions $E[-\frac{\Delta J}{\Delta q}]$ for a particular block B_j is equal to the arithmetic mean of $\frac{-\Delta J}{\Delta q}$ over all possible indices $i \in B_j$ where elements from B_j may lie. And the greedy strategy will be to take a random value from the block in which this expectation is as large as possible.

It remains only to learn how to recompute all the described values. Surprisingly, all this can be done naively; the total amount of changes in ΔJ and Δq over all indices turns out not to be very large.

Solution for 100+ points. (Cartesian tree)

Suppose we have split all elements into 3 groups L, R, C (see the first phase of the 70-point algorithm). Then we can spend $n \log_2 n$ queries to build a Cartesian tree on the elements of L and on the elements of R. A Cartesian tree is a tree built for an array $[a_1, \ldots, a_n]$ as follows:

- Choose the minimum element by value, a_i; it becomes the root of the tree, and the left and right children of this vertex are the Cartesian trees built for the subarrays $a[1 \ldots, i-1]$ and $a[i+1, \ldots, n]$.

It is easy to see that if we choose two values $x, y \in L$, then $f(x, y)$ will be equal to the LCA of vertices x and y in the Cartesian tree for the subarray of the permutation where the elements from L are located.

To build the Cartesian tree for the set L, we can gradually add elements to it in increasing order. Suppose we add an element x. Then we can identify a path from the root in the already built tree that always goes into the larger subtree. If this path ends at a leaf y, then after querying $f(x, y)$ we understand the LCA of vertices x and y, that is, the moment when we need to leave this path. Once we leave the path, we enter a subtree that is twice smaller, so we can repeat the query; in $O(\log n)$ queries we can understand where the vertex x should be attached. The only nuance is that in the built tree it is unclear which children are left and which are right, but even without this information the built tree helps speed up the second phase of the algorithm.

Note that when some block is split, or when we learn the value of some element from a block and need to cut it out of the Cartesian tree, we can recompute everything without any problems. It is enough to split the original Cartesian tree into the minimum possible number of trees, and then connect the roots of the resulting trees into one tree using the same algorithm.

Now let us consider why the Cartesian tree can help at all. Suppose we have an element x and a set of blocks with built trees $B_1, \ldots, B_n$, and at the current moment we are splitting some block B using the element x. Denote $a(v) = 1$ ($v \in B$) if $f(x, v) = 1$, and $a(v) = 0$ otherwise. Then we have the following prunings:

- If v is the root, and l and r are some vertices from its different subtrees, then if $a(v) \neq a(l)$, we have $a(v) = a(r)$.
- If v is the root, and one of its subtrees contains some y with $a(y) \neq a(v)$, then we can separate from the tree the child of v whose subtree contains the vertex y, and uniquely determine which tree is to the left and which is to the right (that is, additionally split everything into blocks).
- It is advantageous to compute the values $a(v) = f(x, v)$ in decreasing order of subtree sizes of v.

This pair of prunings works well because it splits a block not into one block, but into at least $\Omega(\log(n))$ blocks in expectation, if we split blocks at random places. Moreover, this method gives some guarantees as well (handwaving follows, but still):

- We split the tree at a random place, so most often $f(v, x) = 1$ and $f(v, x) \neq 1$ will occur with roughly comparable probability. Therefore, the pruning from the first item will work very often, and we will make few queries on a balanced tree.
- If, on the other hand, we split the tree in a very uneven proportion, then we will make many queries, but many edges will also be cut, because we will descend quite deeply.

Problem Tutorial: “«Titanic»”

Problem author and developer: Alexander Zavarin

First, let us understand that since the hook can only be fired perpendicular to the rescue boat's movement, there is exactly 1 point on the line passing through points A and B. This means that we can compute the segment on that line during which the hook will be blocked if we catch a lifeboat at its initial position. To compute the point where we need to catch the lifeboat, we should draw a perpendicular to line AB; the foot of that perpendicular will be the required point. To compute the point where the hook becomes free again, we need to draw from the initial point a vector in the direction of the rescue boat's movement whose length is equal to the length of the perpendicular. Let us call, for a lifeboat, the starting point of this segment the — *capture point*, and the ending point the — *rescue point*.

Consider subgroup 1. In it, there is a restriction on the y-coordinate of points A and B: for both of them it is equal to 0. W.l.o.g., let the rescue boat sail from left to right (otherwise, we can simply mirror the coordinates with respect to the line $x = 0$). Then, if the i-th lifeboat is at point (x_i, y_i), its capture point has coordinates $(x_i, 0)$, and its rescue point is $(x_i + y_i, 0)$, since the distance between the capture point and the lifeboat was exactly y_i. We can discard all lifeboats whose capture point has an x-coordinate less than a_x or greater than b_x, because some of them we will not be able to catch with the hook, and the others we will not have time to pull in before the game ends. Now let us sort all lifeboats by the x-coordinate of their capture point in order to use dynamic programming. For each lifeboat i, let us compute the value dp_i — the maximum number of lifeboats among those whose number in the sorted order is less than i, while still being able to rescue the i-th lifeboat. Now we simply iterate once over the lifeboat number in sorted order; let the current one be i, and then we iterate over all $j < i$. For the j-th lifeboat, we check that the x-coordinate of its rescue point is not greater than the x-coordinate of the capture point of our i-th lifeboat, otherwise we will not be able to rescue it. We compute $dp_i = \max_{j<i}(dp_j) + 1$ over all suitable j. This works because for dp_i, among the rescued lifeboats with numbers $j < i$, there is a lifeboat with the maximum number j_{max}, and for it, if we remove the rescued i-th lifeboat, then $dp_{j_{max}} = dp_i - 1$ by the definition of dp_i. Then, since the answer also contains some lifeboat i_{max} with the largest number, the answer is $dp_{i_{max}}$, i.e. it can be computed as $\max_i(dp_i)$.

Now let us move to subgroup 2. Sorting the lifeboats so conveniently no longer works, but it can still be done if we now use as the sorting parameter the distance between point A and the capture point of a lifeboat. This can be computed using the formulas for constructing a perpendicular to a line and for computing distances between points, but the author's solution uses vector and dot products to calculate this distance. Let the lifeboat be at point P. Then the distance from A to the capture point can be computed as $|AP| \cdot \cos\alpha$, where α is the angle between AP and AB. This can be expressed via the dot product of vectors as $|AP| \cdot \cos\alpha = \frac{AB \cdot AP}{|AB|}$. The length of the perpendicular is computed similarly, but via the cross product: $|AP| \cdot \sin\alpha = \frac{|AB \times AP|}{|AB|}$. It is important not to forget to take the absolute value of the cross product: in the first case, the absolute value was not used so that lifeboats outside the rescue boat's reachable area would have a negative distance from point A for convenience. Now, using these two values, we can compute the distance from point A to the rescue point of the lifeboat as their sum. Note that these distances have a common denominator — the length of segment AB. So let us simply multiply all values by this number; this will not change the order of the lifeboats or their sorting, but it will allow us to switch to integer computations. Now, having for each lifeboat two values — the distance from point A to the capture point and to the rescue point — and also using the dynamic programming idea from subgroup 1, we can solve this one as well.

Let us move to subgroup 3, and now think about how to improve the computation of dp_i. For this, we use a *scanline* idea. We will have 2 types of events: "we arrived at the capture point of some lifeboat" and "we arrived at the rescue point of some lifeboat". We sort all events by their distance from point A, and then process them in order while storing the answer — the maximum number of lifeboats that we can rescue after processing the first i events; let us denote it by $score$. If we encounter an event of the 1-st type for the j-th lifeboat, then by definition we compute $dp_j = score + 1$. If we encounter an event of the 2-nd type for the j-th lifeboat, this means that if we had been rescuing this lifeboat, then the hook would now become free, and therefore we can use dp_j to update $score$: $score = \max(score, dp_j)$. It is important that if several events are located at the same point, then events of the 2-nd type must be processed first, and

only then events of the 1-st type. After processing all events, *score* will be our answer.

Subgroups 4 and 5 use the combined ideas of subgroups 2 and 3.

Problem Tutorial: "Playing Go"

Problem author: Alexey Mikhnenko, developer: Igor Markelov

First, let us look at the optimal answer. Obviously, the area of the answer is at least the area of the convex hull of the original set of points. In the case when the area is larger, it is claimed that the white point we moved is shifted along the normal to one of the $n \cdot (n-1)/2$ segments connecting the original points.

This observation allows us to write an $O(n^4 \cdot \log(n))$ solution — we iterate over the direction and the point, shift it, and recompute the optimal answer as the area of the convex hull of the resulting set.

Note that (of course, the proof of this amazing observation is left as a most interesting exercise for the curious reader) for each oriented direction it is sufficient to consider at most 5 points (the farthest in this direction). Thus, the previous solution turns into $O(n^3 \cdot \log(n))$ if for each direction we first choose at most 5 candidates and then build the convex hull for each of them.

Now let us learn how, for a fixed set of k points, to answer queries of the form «the area of the convex hull of the set if one more point is added» in $O(\log(k))$ time with $O(k \cdot \log(k))$ preprocessing. Build the convex hull on the k points, and compute prefix sums of the areas along the convex hull from the first vertex to the i-th one for all vertices. To answer a query, first use the standard algorithm in $O(\log(k))$ to check whether the point lies inside the convex hull. If it does — the answer is the area of the original hull. If it does not — find the tangents from the point to the convex hull; the answer area is equal to the area of the triangle formed by the tangent points and the query point, plus the area of the original convex hull without the part of the polygon cut off by the diagonal between the tangent points. The tangent search can be implemented in $O(\log(k))$, and the required areas can be computed in $O(1)$ using the precomputed prefix sums.

Apply this primitive to the previous solution: precompute the convex hulls of the original set with each of the white points removed, then the solution will work in $O(n^3)$, since the heaviest part is considering $O(n)$ candidate points for $O(n^2)$ directions.

A careful reader of the statement, having reached this point in their reasoning, may have appreciated the generous gesture from the jury — to remove from consideration test cases with three or more points on one line.

Let us optimize this part using the standard rotating sweep-line technique over directions. For $O(n^2)$ directions, we need to maintain a list of $O(n)$ points sorted along the direction. The total number of events will be $O(n^2)$, so now the solution will run in $O(n^2 \cdot \log(n))$.

Problem Tutorial: "Moving"

Problem author: Alexander Babin, developer: Ivan Piskarev

$O(S! \cdot S)$

Let us iterate over all possible ways to perform the relocation and choose the best one.

$O(S^3 \cdot \log m)$

We will search for the answer using binary search. We want to check whether it is possible to organize the relocation so that the maximum distance is at most k. Let us build a bipartite graph where the vertices

in the left part correspond to places before the relocation, and the vertices in the right part correspond to places after the relocation. There is an edge between two vertices if the distance between their houses does not exceed k. It is easy to see that a relocation with distance at most k is possible if and only if the graph contains a perfect matching.

A maximum matching in a bipartite graph can be found with Kuhn's algorithm in $O(nm)$, where n is the number of vertices and m is the number of edges.

$O(2^A \cdot AB)$

Instead of explicitly finding a maximum matching, it is enough just to check Hall's lemma condition. Namely, to check that for every subset X of vertices in the left part, at least $|X|$ vertices in the right part are directly connected to it. Since such a set of vertices in the right part depends only on the set of cells containing the vertices of the set X, it is enough to consider only 2^A variants of such a set.

$O(m)$ for $n = 1$

Let us sort the places before the relocation and the places after the relocation by increasing j-coordinate. Notice that the relocation method where the resident from the k-th place before the relocation moves to the k-th place after the relocation is optimal. The maximum distance for such a relocation method can be found with a two-pointers pass.

$O(m \cdot \log m)$ for $n \leq 2$

Let us do binary search on the answer. We want to check whether it is possible to organize the relocation so that the maximum distance is at most k. To do this, we will check Hall's lemma condition.

Let X be a subset of houses.

- Let $W_1(X)$ and $W_2(X)$ denote the total number of residents in X before and after the relocation, respectively.
- Let $N(X)$ denote the set of houses at distance at most k from X.

Then Hall's lemma condition is equivalent to the inequality $W_2(N(X)) - W_1(X) \geq 0$ holding for all X. To check this, let us find the minimum value of $f(X) = W_2(N(X)) - W_1(X)$ and compare it with zero.

To find the minimum value, we use dynamic programming: let `dp[i][mask]` be the minimum value of $f(X)$ if

- X consists only of houses up to the i-th column inclusive
- X contains at least one house in the i-th column
- `mask` is the intersection of X with the i-th column

Notice that if X contains at least one house in the i-th column, then whether a house strictly to the right of the i-th column belongs to $N(X)$ does not depend on whether houses strictly to the left of the i-th column belong to X, and vice versa. Therefore, for the transition it is enough to iterate over the nearest house to the right that will be added to X.

Notice that it is enough to transition only to columns that are 1 and $2k$ houses to the right. Indeed, when transitioning by at most $2k - 1$ columns, including all skipped columns into X will not make the answer

worse, and therefore such a transition will be no worse than several transitions one column to the right. And when transitioning by more than $2k$ columns, if the value of `dp` is at least 0, it is not beneficial to use it, while if the value is negative, then a counterexample to Hall's lemma condition has already been found.

$O(m)$ for $n \leq 2$

Binary search can be removed. Notice that after merging the first and second rows, the answer either does not change or decreases by one (since the distance between any two places either does not change or decreases by one). Therefore, it is enough to solve the same problem for n decreased by one and then perform a check for a single value of k.

Full solution $O(mn^2 \cdot 3^{2n})$

Suppose we have $\mathbb{N}$ paints of different colors, numbered $0, 1, 2, ...$

Let us introduce several definitions:

- For a set of houses X, define the coloring of houses $C(X)$ in which each house is painted with the color whose number equals the distance to the nearest house from X.
- For a coloring of houses C, define the sets of houses $X(C), Y(C)$ consisting of houses of color 0 and houses of colors $0, 1, 2, ..., k$ respectively.
- Define the cost of a coloring C as $f(C) = W_2(Y(C)) - W_1(X(C))$.
- Call a coloring good if the difference between the color numbers of any two neighboring houses does not exceed 1.

Notice that

- $f(X) = f(C(X))$
- $C(X)$ is a good coloring
- If C is a good coloring, then $f(C) \geq f(X(C))$

In the previous solution, we reduced the problem to computing $\min\limits_{X} f(X)$.

Notice that the three properties above imply that $\min\limits_{X} f(X) = \min\limits_{C \text{ good}} f(C)$.

Therefore, it is enough to learn how to compute $\min\limits_{C \text{ good}} f(C)$.

To do this, let us redefine `dp[i][mask]` as the minimum value of $f(C)$ over all good colorings C such that

- In W_1, W_2, only residents of houses up to the i-th column inclusive are counted
- `mask` encodes the color numbers of the houses in the i-th column

Transition in $O(m^2 n \cdot 3^{2n})$

Let us first try to simply compute `dp[i+1]` from `dp[i]`.

Notice that it makes no sense to consider colorings without color 0, so the maximum color does not exceed $n + m = O(m)$. Notice that if the color of the first house in a column is fixed, then the remaining ones can be colored in at most 3^{n-1} ways so that the coloring remains good.

Therefore, the number of possible values of `mask` is at most $O(m \cdot 3^n)$.

Notice that from `dp[i][mask]` we can transition to at most 3^n states in `dp[i+1]`.

One transition from `dp[i][mask1]` to `dp[i+1][mask2]` can obviously be done in $O(n)$.

In total, we have $O(m^2 \cdot 3^n)$ `dp` states, and from each of them we make at most 3^n transitions, each taking $O(n)$. Therefore, the running time of such a solution is $O(m^2 \cdot n \cdot 3^{2n})$.

Notice that it is enough to consider only colorings using colors $0, 1, ..., \mathrm{k}, \mathrm{k}+1$. This allows solving the problem in $O(\mathrm{ans} \cdot mn \cdot 3^{2n})$.

Transition in $O(mn^2 \cdot 3^{2n})$

We will explicitly store only those states where color 0 is present in `mask`. Notice that the number of such values of `mask` is exactly 3^{n-1} (we choose $n-1$ differences between neighboring cells from $\{-1, 0, +1\}$ and add a constant so that the minimum becomes zero).

Notice that it is enough to transition only by $1, 2, ..., 2n$ or $2k - 2(n-1), ..., 2k-1, 2k$ columns to the right. Suppose we transitioned from column i_1 to column i_2. If $i_2 - i_1 < 2k - 2(n-1)$, then by adding columns from $i_1 + n$ to $i_2 - n$ to the set X, we will not increase the cost, because $Y(C)$ will remain the same. If, on the other hand, $i_2 - i_1 > 2k$, then if the value of `dp` is negative (taking into account residents in columns to the right of i_1), we can already stop, and if it is at least zero, then it is not beneficial to use it in the transition.

Thus, we have $m \cdot 3^{n-1}$ states, and from each state we make at most $4n \cdot 3^{n-1}$ transitions, each of which is done in $O(n)$.