Problem A. A Challenge For a Tourist

We are given a weighed graph and we have to answer queries of form "what is the minimum possible value w such that if we remove all edges heavier than w, there will exist a path from f to t with `xor` of the weights on it equal to h?".

The core idea is to calculate the `xor` of every cycle in a connectivity component containing two given vertices f and t. These values are important because if we consider two different paths between f and t and remove all their common edges, the remaining edges will form several cycles (simple cases analysis shows that after such action each vertex will have an even degree). If we have some set of edges V (first path) and then we take `xor` of $V \cup A_1 \setminus A_2$ (and V, A_1 and A_2 are non-intersecting), it will be equal to the `xor` of $V \cup A_1 \cup A_2$ due to the properties of `xor` function.

This means that if the hash of one particular path is h, any other path will have a hash that differs from h by applying a `xor` with some cycles in that component to it. The opposite is also true: for every subset A of cycles in the component, there exists a path with hash equal to $h \oplus \left(\bigoplus_{a \in A} \texttt{hash}(a) \right)$. This is true because if the cycle is in the same component, we can "leave" our path, go to that cycle, traverse it once and then return the same way: the middle part will be included twice and have a `xor` of 0.

Now let's address the easiest of the remaining steps. If we had fixed limits w for each query and already had a graph containing only edges with weight $\leq w$ with w from current query with the "basis" of cycles hashes in the component that contains f and t, how to determine if we can achieve the desired hash h?

If we know some path between f and t with hash h_0, we just need to get the value $h \oplus h_0$ using the values from the basis. To check if we can do that, first we'll normalize the basis so that no more than one value has 1 in the highest bit, no more than 1 of the remaining values has 1 in the next bit and so on. This can be done using binary Gaussian algorithm. Since there are only 30 bits to consider, no more than 30 relevant values will be left in the basis. Then, to check if we can get the value $h \oplus h_0$ we'll iterate through the values in the basis and if the corresponding bit is equal to 1 we will have to include the value in our decomposition.

Now to get rid of predefined w for each query, we will search them using binary search on the answer.

The only thing left to do is to describe how to build a basis for each query. One of the approaches is to take advantage of the "offline" nature of the problem, sort all queries in the increasing order by w and "build" the graph as w increases. To do so, we also sort all edges of the graph in the increasing order by their weight and add all necessary edges before processing the next query.

To calculate the basis in each component, we will store each component in the DSU (disjoint set union) maintaining the spanning tree and all extra edges. In the spanning tree we also calculate hashes (`xor` of edges) on the paths from each vertex to the root.

- When a new edge (a, b) between different components is added, we connect connect one spanning tree to another, re-rooting it by b and re-calculating hashes to the root, using "small-to-large" optimization which ensures that for each vertex no more than $\log n$ recalculations will be made.
- When a new edge (a, b) between vertices of the same component is added, a new cycle is created that we need to take into consideration (we don't need every cycle, just the basis from which every other cycle can be constructed) and that is a new edge with a path $a \to b$ in the spanning tree.

  We calculate hash on the path as $h^* = h_a \oplus h_b$ and then add new hash $h^* \oplus w(ab)$ to the basis, re-normalizing it.
- When the new query has to be processed, the hash h_0 is calculated as described in above and then we check that a desired hash can be obtained using our basis.

We should note that the binary searches for each query should be run in parallel. On each iteration of the algorithm we will determine the next middle value in the binary search for each query and then build a graph again from scratch. This solution uses binary search and spends extra $\log n$ time to recalculate hashes in "small-to-large" merging of the components thus working in $\mathcal{O}(n \log^2 n + (m + q) \log(n + q))$ time.

Alternatively, you could get rid of recalculations by noticing that when the tree is re-rooted and connected to another tree, all hashes on paths to the root are `xor`ed with the same value so all recalculation can be done by bulk operations on DSU's subtrees. This removes one $\log n$ from the first term in time complexity.

Problem B. Brackets

From the rules for constructing the expression, it follows that the final expression can have sequences of pluses with lengths of 1, 2, 3, and 5. Moreover, only for sequences of length 3 is it impossible to unambiguously determine to which variable the pre-increments and post-increments relate. i.e.

- sequence of length 1 is an operand;
- sequence of length 2 is either a pre- or post-increment depending on whether the variable is on the left or on the right;
- sequence of length 5 is one post-increment, one operand and one pre-increment.

Since the task allowed adding a number of brackets equal to the number of operands, it was sufficient to accumulate each operand with a greedy strategy and add brackets to it before putting it in the answer.

Problem C. Credit Cards

Firstly, we'll get rid of 1. No triangle with distinct sides can be created since for any $x \neq y$ the triangle inequality doesn't hold: $\min(x, y) + 1 \leq \max(x, y)$. Now we've left with sides from 2 to n. The obvious estimation of the answer is then $\left\lfloor \frac{n-1}{3} \right\rfloor$ and no more than that. Now we'll constructively provide a method to build exactly this many obtuse triangles using given sides length.

First approach is to split the segment $[2, n]$ into four equal parts (for now, assuming that $n-1$ is divisible by 4). In this case, let's look at the value x from the first quarter and pick values $x+k$, $x+2k$ and $x+3k$ as the sides for another triangle (here $k = \frac{n-1}{4}$). For $k \geq 2$, $x \leq k+1 < 2k$ and we can calculate that

$$(x+k)^2 + (x+2k)^2 < (x+3k)^2 \text{ because}$$

$$2x^2 + 6xk + 5k^2 < x^2 + 6xk + 9k^2 \text{ because}$$

$$x^2 < 4k^2$$

so the selected triangle is obtuse. This leaves us with a segment $[2, k+1]$ which we can process in the same way recursively.

For cases where $n-1$ is not divisible by 4 we'll just make the last three parts equal and the first one "almost equal" to them. With such parts the same reasoning will work but for $k \geq 4$. And for smaller k we will run the subset dynamic programming, for example in $\mathcal{O}(2^n \cdot n^3)$ time to split all the remaining sides into triangles.

An alternative approach is to split our segment into three equal parts. Since we are aiming for the answer $\left\lfloor \frac{n-1}{3} \right\rfloor$, if the segment can not be split into equal parts, we'll drop one or two "extra" sides with maximal length.

We're going to match the sides from the second and the third parts into pairs and then match them to the elements in the first part. If the parts have odd number of elements, we'll math the maximum element from the second part with the minimum element from the third part. Then, after they each have even number of elements remaining, we'll process the second and the third part independently and match the opposite elements in each of them (in the increasing order: first with last, second with second from the end, etc.).

The remaining step is to sort these pairs (a_i, b_i) in the order of increasing $|a_i - b_i|$ and match them with sides $2, 3, \ldots$ from the first part in this exact order. There are two or three pairs with a difference of 1, two pairs with a difference of 2, etc. So the sides from the first part grow faster and for each triple the triangle inequality will hold. And with some relatively rough estimates it can be proven that each of such triangles will be obtuse.

Problem D. Doors of the Wardrobe

According to the conditions, the doors must be open strictly in sequence. Let's trace the process in reverse order.

Consider the last door. We need to slide the handles so that all M items are visible. Additionally, the less we move the handles of this door, the less we will need to slide the previous doors so that the handles of this door are not blocked. So we will move the handles by the minimum possible distance. Let's project all the items in the direction of the door's movement and take two extreme points. Note that there is no profit in moving the handle in the wrong direction, so if the extreme point is on the other side, then you don't need to move the corresponding handle.

Now consider the penultimate door. It should be expanded so that all M objects are visible, as well as the entire segment over which the handles of the last door were moved. Again, we move the handles by the minimum possible distance, projecting everything onto the axis. The penultimate door needs to be expanded so that M objects and two handles' segments of the movement are visible. And so on.

Now we need to understand how to perform this procedure faster. Obviously, the set of points and segments is visible from behind the door when its convex hull is visible. Thus, we can build the convex hull of M objects, and then add two points from each door to this hull. Additionally, for each door, we need to quickly find the maximum point of the convex hull along the direction. So we need an algorithm for the incremental construction of the convex hull, as well as the search for extreme points on the hull.

This can be implemented in the following way: select a point strictly inside the hull and call it the center. We will store all the vertices of the hull in a search tree, ordered by the polar angle relative to the center. This tree will allow us to quickly build the hull when adding a new point: we just need to find the sector in which the point falls, and then delete the vertices around this sector as needed.

Additionally, we need a second search tree in which the vectors of the sides of the hull will lie, also ordered by the polar angle. This allows us to quickly find the maximum point in a given direction: to do this, simply rotate the direction by 90° and find the side vectors closest to it.

The resulting algorithm works in for $\mathcal{O}(K \log K)$ time, where $K = M + 2N$ is the total number of points.

On a technical note, the following can be observed.

- First, you need to define the center: for this, you need to find three objects that are not lying on the same straight line. To do this, you can take any object as the first, then take the one furthest from it as the second, and then take the third object, the furthest from the line connecting the first two.
- When sorting vectors by polar angle, there are problems with the "cut line" — the angle defined as both α and $\alpha + 2\pi$ simultaneously. It is desirable to move the cut to some random place so that the vectors are unlikely to hit it, rather than leaving the cut at $\alpha = \pi$ or $\alpha = 0$.
- Finally, the task requires high accuracy of the answer, so you should choose a fairly accurate tolerance ε for defining the border cases.

Problem E. Effective Management

In this task we had to calculate the most "profitable" products given their individual profit, production time and dependency graph.

It's guaranteed in the problem statement that the dependency graph is acyclic and thus can be topologically sorted. After we perform a topological sort on the dependency graph, this problem becomes an example of standard dynamic programming on a topological sort.

Let's remember that the *actual production time* is the maximum of *pure production times* of each of this product's dependencies. But if we look at the dependency graph, all non-direct dependencies' production times are already "included" in *actual production times* of its **direct** dependencies. Thus,

$apt[v] = \max\left(ppt[v], \max\limits_{v \to u} apt[u]\right)$ where *apt* and *ppt* stand for actual and pure production times, respectively.

This formula can be calculated for every product in $\mathcal{O}(n+m)$ time in the order of topological sort, after which we just need to select the maximum ratio of $\frac{profit[v]}{apt[v]}$. To avoid problems with precision, we will store both numerator and denominator of each ratio and instead of comparing $\frac{a}{b}$ with $\frac{c}{d}$ we will compare $a \cdot d$ with $c \cdot b$.

Problem F. Form Processing

We are given a string with even elements equal to '`?`' and odd elements equal to mathematical operands. The goal is to get this expression to evaluate to a maximal possible **integer** value by changing all '`?`' into digits.

Let's forget about the "integer" part and find the total maximum. For this we have to split this expression into terms (with '`+`' and '`-`' between them) and independently maximize each term with a '`+`' before it and minimize each term with a '`-`' before it. Since we can't use 0, to maximize a term we should place '`9`' after each multiplication and '`1`' before each division and vice versa for minimization.

We'll denote the maximal possible (not necessarily integer) value as X.

Now all that remains is to find the maximal possible **integer** value. Obviously, we can't get more than $\lfloor X \rfloor$. Let's try to get exactly this much. We can notice that every term that we tried to maximize if already integer: it contains only multiplications by 9 and divisions by 1.

And in the same time each term that we subtract looks like $\frac{1}{9^k}$ where k is the number of substrings "`/?`" in that term. Let's sort all subtracted terms by their k in increasing order. We then get values $\frac{1}{9^{k_1}}, \frac{1}{9^{k_2}}, \ldots, \frac{1}{9^{k_t}}$. If we process them in that order, we can maintain the fractional part of their sum in the numeral system with base 9. It will always be a number of form $0.\cdots$ and with each subsequent 9^{-k_i} we will add 1 to either its last non-zero digit or even later digit. This means that this sum can never "jump" from being less than 1 to being strictly more than 1. And if it ever becomes exactly 1, we just "forget" about it because we are only interested in its fractional part.

Now the problem arises only if we get to the end of the sequence of terms and the resulting sum is non-zero. We'll only look at the last segment of this algorithm where this sum starts with 0. Any value 9^{-k_i} can be increased to $c \cdot 9^{-a}$ with $a \le k_i$ and $1 \le c \le 9$ by changing $k_i - a$ characters '`9`' to '`1`' and changing one '`1`' to c in the corresponding term. Note that we limit ourselves to changing only one character '`1`'!

The algorithm to make the remaining terms sum to 1 is as follows: for each term we will first multiply it by 9 until it becomes $\ge 1 -$ `current_sum` and then multiply it by maximal possible $c \le 9$ such that it still is strictly less than $1 -$ `current_sum`. For the last remaining term it will always be possible to increase it in such a way so that our sum reaches exactly 1 for the reasons mentioned above. Each possible "increasing" action increases the current term's value by $\frac{1}{9^x}$ for some x and adding 9^{-x} to a base-9 number can not make it "jump" from being less than 1 to being greater than 1, so sooner or later we will reach such a value for the last term that it will make the sum equal to exactly 1 which was our goal.

Problem G. Go Play Wall-nut Bowling!

In the problem, we need many data structures; let's start with something simple.

Instead of moving all zombies to the left, we will move the field window to the right. For this, we will keep a global counter T — the current time. To scroll through time, we simply increase this counter, and when receiving requests with coordinates, we add the current time to the column number.

To find the next hit, we need to be able to search for the nearest zombie for each diagonal. To find the first hit, we need to be able to find the nearest zombie horizontally. Finally, to determine which zombies go for the brains, we need to maintain the order of the zombies by increasing the column. At the same time, zombies can appear and disappear at any moment.

These problems can be solved using four search trees ordered

- by the pair (R, C);
- by the pair (C, R);
- by the pair $(R + C, R - C)$;
- by the pair $(R - C, R + C)$,

where R is the row number and C is the column number. It makes sense to store all zombies in one array, and in the search trees to store their indices.

All that remains is to figure out what to do with the hits. Let's assume that each zombie has two halves: the upper and the lower. Moreover, if the nut bounces diagonally upwards after the hit, it hit the upper half, otherwise — the lower half.

Consider a graph in which the zombie halves are vertices, and transitions to the next hit are edges. Obviously, this graph will always be a set of independent chains. When a player launches a nut, they hit all the halves in one chain, starting from a certain one. Thus, if we store all halves of each chain in some structure that supports segment queries, we will be able to quickly process a multitude of hits.

Furthermore, zombies sometimes appear or disappear in a random field location, and this changes the chains as follows. Two chains that pass through a given zombie switch suffixes after this zombie. That is, in addition to segment queries, we will need to cut chains into parts and glue them into a different configuration. Obviously, we will need a Cartesian search tree with an implicit key: it allows cutting/linking, as well as performing operations on a segment.

A significant problem lies in the fact that each zombie has a common HP level for both of its halves. When we perform a hit on a chain, we can quickly update the HP on a segment in that chain; however, these same zombies are part of other chains, and finding them quickly won't be possible. The task would be simpler if each zombie half had its own HP level — then the hit wouldn't affect other chains.

We'll reduce the task to the version with separate HP for halves as follows. If a zombie initially has H health points then we'll assign each of its halves $\lceil \frac{H}{2} \rceil$ health points. As long as both halves are alive, the whole zombie also cannot die. When the HP level of one of the halves drops to zero, we can remove the zombie from the game, recalculate its total HP amount, and add it back, again dividing the remaining HP into halves. After each such update, the zombie's life level is reduced at least twice, so each zombie will be updated no more than $\log H$ times during the game.

The next question: how to quickly find the dead halves after hitting a chain? To simply know whether they exist or not, it is enough to store a statistic in the Cartesian tree: the minimum HP level in the segment. However, we also need to find their positions. For this, we'll add another statistic: the number of the first half whose HP level equals to the minimum HP in the segment. This statistic can be automatically recalculated along with the minimum, and if the minimum is zero, it gives us the position of the first dead half.

Finally, when a zombie half dies in a chain, we need to remove its other half from another chain. To do this, we need to know the position of the second half in its chain, as well as the root of that chain. Therefore, we need to store a pointer to the parent node in the Cartesian tree for each node: this allows us to climb the tree in $\mathcal{O}(\log Q)$ and find both the root and the position of the original node immediately.

Putting it all together, we need:

1. A single array of all zombies.
2. Four search trees for coordinates (`std::set<int>` will suffice).
3. A Cartesian tree with an implicit key for halves:
   - Store a pointer to the parent (to find the position of the half by pointer).
   - Statistics: minimum HP and the position of the first minimum (to find dead halves in the chain).
   - Deferred modification: addition on a segment (hit all zombies in the segment at once).

Also, for each half we need to be able to quickly find its opposite half.

When adding a zombie we find its diagonal neighbors, determine their chains and positions in them, cut the chains, and link them in reverse, inserting the halves of the added zombie.

During a hit, we find the first hit half, determine its chain, and number. Then, we cut the chain, subtract 1 from the life level of the entire suffix, and link the chain back together. Now, if the minimum health points in the chain is greater than zero, there's nothing more to do. Otherwise, we find the first dead half through the statistic, cut both halves of this zombie from two chains. Update the zombie's health points: if it's still alive, we glue the chains back as they were, but if it's dead, we glue them in reverse. Then we check the minimum HP level again, and if it's zero, we repeat the procedure.

Thus, adding a zombie takes $\mathcal{O}(\log Q)$ time, for a total of $\mathcal{O}(Q \log Q)$. The hit is more complex: here, $\mathcal{O}((K+1)\log Q)$ time is required where K is the number of dead and updated zombies. Since each zombie is updated at most $\mathcal{O}(\log H)$ times and there are a total of Q zombies, the total time spent on hits is $\mathcal{O}(Q \log Q \log H)$.

The total time is $\mathcal{O}(Q \log Q \log H)$, and the memory is $\mathcal{O}(Q)$.

Problem H. Hacking the Project

We can send requests of type "`s k`" and get $\leq k$ first in lexicographic order strings starting with `s` from a hidden set.

First of all, let's solve this problem with the limit on total k in queries $K_{lim} = 4000$ instead of 3800. We'll make the requests as follows:

1. "`a` 1";
2. "`a` 2";

   ...

i. "`a` 2^i";

until we get less strings in the reponse than corresponding $k = 2^i$. This will give us all of the strings starting with letter '`a`'. Then we'll repeat the same process for each letter of the alphabet, which will give us all the words that Lewis used.

Now we'll prove that we didn't exceed the limit of 4000 on k. Total k used for one letter is $1 + 2 + \ldots + 2^i < 2^{i+1}$ while the number of words starting with that letter is between 2^{i-1} and 2^i. Which means that if the number of words starting with letter c is n_c, we've spent no more than $4n_c$ total k making queries with that letter. Since the number of words is ≤ 1000, we didn't exceed out limit of $K_{lim} = 4000$.

However, the limit in the statement is slightly more strict. Let's optimize our solution. First of all, we can improve our estimation by 26, since for each letter $n_c \geq 2^{i-1}$ and we spend no more than $2^{i+1} - 1$ of our limit. This -1 in total for all letters gives us an estimate $4000 - 26$.

Then, there are several approaches to limiting the total k used in requests to a smaller value. One is to use different "steps" instead of powers of 2. One such list of steps that works is $[1, 3, 9, 22, 49, 101, 193, 345, 569, 839, 1000]$. Using these step values, we'll get the estimate of total k spent as $\leq 3.8n_c$.

Another approach is to optimize the initial solution. Instead of iterating through letters in the outer loop and steps in the inner loop, we'll swap them. When only one letter remains, we already know exactly how many words start with this letter and we'll make a request containing exactly that number. Another alternative is to ask for $\min(2^i, N - \texttt{known})$ on each step where `known` is the number of words starting with different letters that we already know. Both of these optimization combined or any of them with careful implementation and some other small optimizations were enough to pass all tests.

Problem I. Inspection (32 MB ML!)

The problem is solved using dynamic programming.

Let $R(i, a, b)$ be the minimum possible number of **un**eliminated pits under the conditions:

- consider only the first i kilometers of the road;
- repair exactly a sections of the first type;
- repair exactly b sections of the second type.

From state $R(i, a, b)$, there are transitions to:

- to $R(i + 1, a, b)$ with the addition of the number of pits at the i-th kilometer to the cost;
- to $R(i + L_1, a + 1, b)$ without changing the cost;
- to $R(i + L_2, a, b + 1)$ without changing the cost.

The dynamics result in $\mathcal{O}(NK^2)$ time and memory. In this problem, there is a very strict memory constraint, so you can't just save a table of size $N \times K \times K$.

It can be noticed that the target array is "layered" by parameter a (or by parameter b), so it would be possible to store values only for two adjacent layers by a. In this case, the memory consumption is reduced to $\mathcal{O}(NK)$, but this only finds the objective function of the optimal solution, not the solution itself. There are two ways to solve the problem with answer recovery.

The first method is to compress the data volume to represent the backward path. In this case, we will store the objective function only for two neighboring layers.

For each state, you need to store a single number from 1 to 3 for the reverse code — the option of the transition by which we got into this state in the optimal solution. You can pack this number into 2 bits, essentially packing 16 numbers into one 32-bit word. This approach allows you to reduce memory consumption to $\frac{N \cdot K^2}{4}$ bytes, which in the task constraints amounts to 25 megabytes.

The second way is to restore the answer using the "divide and conquer" method.

Suppose we want to know any intermediate state in the optimal solution with $a = \frac{K_1}{2}$ — the median value of the layer number. To achieve this, we will, in each state of dynamics, in addition to the objective function, also store the numbers i^* and b^* — parameters of that very "median" state through which we passed.

Now the problem can be solved as follows. First, we run all the dynamics completely, from the initial state $R(0, 0, 0)$ to the final $R(N, K_1, K_2)$. Now we take the median state and divide the process into two parts: before and after it. We recursively run the solution on the first half and the second half. In general, within the recursion, we run dynamics on the subregion $R(i_l \dots i_r, a_l \dots a_r, b_l \dots b_r)$ and find a new intermediate solution.

This allows us to solve the problem in $\mathcal{O}(NK^2)$ time and $\mathcal{O}(NK)$ memory, and the required memory does not exceed the memory limit.

Problem J. Jeopardized Betting

Let's try to informally formulate how out strategy will work. Since we want to have 0 money if our team loses and twice as much compared to what we had initially if it wins, it seems logical to keep the following invariant: "in each moment in time the amount of money we have is proportional to the probability that our team will win".

Let's analyze why this strategy seems like a good option. After the last match the probability of win is either 1 or 0, and we'll have either $2^{2N+1} \cdot 1$ or $2^{2N+1} \cdot 0$ money. And in the beginning we can say that the probability of our team to win is 0.5 and the amount of money we have is $2^{2N+1} \cdot 0.5 = 2^{2N}$.

Now we'll explain how to maintain such invariant. First of all, we'll calculate the probabilities of winning in every possible situation. This can be done using dynamic programming. Let the $wp[i][j]$ be the probability of our team to win if our team has already won i matches and lost j matches. The formula to recalculate this DP is as follows:

$$wp[i][j] = \frac{1}{2} \cdot (wp[i + 1][j] + wp[i][j + 1])$$

since with equal probability the next match will be either won or lost.

The good thing about this formula is that $|wp[i][j] - wp[i+1][j]| = |wp[i][j] - wp[i][j+1]|$ which means that if we bet X money proportionally to $|wp[i][j] - wp[i+1][j]|$ we will either double it or lose it. Which is exactly what we want since we want to have the amount of money proportional to either $wp[i+1][j]$ or $wp[i][j+1]$ after this match.

So, as we can see, the amount of money we need to bet equals to $C \cdot \frac{|wp[i][j]-wp[i+1][j]|}{wp[i][j]}$ where C is out current amount of money. The only remaining question is why this amount will be an integer on every step. But this can be proven as follows:

- $C = wp[i][j] \cdot 2^{2N+1}$ by induction (we keep our invariant that the amount of money is always proportional to the probability of winning).
- Thus, the amount we have to bet equals to

$$2^{2N+1} \cdot |wp[i][j] - wp[i+1][j]|,$$

  and this value is integer because the only prime number in the denominator of our probabilities is 2 (by formula above) and its power does not exceed $i + j + 1$ which is $\leq 2N$ because the game was not over yet.

Also it should be noted that it's better to calculate $2^{2N+1} \cdot wp[i][j]$ instead of exact $wp[i][j]$ since the floating point arithmetic is not accurate enough.

Problem K. KPI Calculation

This is a simple problem in which it's enough to implement what's described in the statement "as it is". For each line that is "visible" (i.e. that has at least one printable character) you should calculate the position of first visible character and then output the sum of these values.

It's easy to check whether there is a visible character in any given line and find the first position of such character simultaneously with one "`for`" with "`break`" command.

Problem L. Limited Shuffle Restoring

In the initial array each element was swapped with an element no more than 2 positions to the right. In the beginning $a[i] = i$ for each i, and the goal is to guess the final state of the array by comparing elements with each other.

If we had $2n$ queries, the problem would be easier. Let's retrace Bobo's actions from the last to the first. On the n-th swap nothing was changed, on the previous swap the last two elements may have changed and on the swap before it the last three elements could be affected. No other swap touched the last element so the maximum should be on one of the last three positions. Using two comparisons we can determine where the maximum is and then reduce the problem to the smaller one, once again searching for the maximum in the last three remaining positions.

What we need to do to not exceed this problem's limitations is to consider groups of **five** elements instead of three. There exists a binary decision tree that finds the complete order of 5 elements using 5 queries given that:

- among those 5 elements there are 3 "new" ones (on the left) and 2 "old" ones (on the right);
- the order of the "old" elements is known;
- the conditions of the problem are satisfied (each element was moved according to the algorithm in the statement).

We will not provide the exact structure of the tree but we will explain how to build it:

1. Take all permutations of length 5 that satisfy the conditions above.
2. For each node of the tree go through every possible query and select the one that splits the pool permutations into two groups with sizes as close to each other as possible.

3. Recursively build nodes for two outcomes of the selected comparison.

This recursive algorithm allows to prepare a decision tree for ordering two already ordered positions on the right with three new positions next to them on the left. The height of such a decision tree will be equal to 5. So the algorithm goes as follows: first, order two last elements in one query and then repeat

- order 5 rightmost known positions;
- remember three maximums;
- replace them with three next positions.

This will take $\lceil \frac{n}{3} \rceil$ iterations, each of them will take 5 queries. To fit into the problem's limitations in every case, you can use first 5 queries to order k last elements such that $k \geq 2$ and $n - k$ is divisible by 3 and then run this algorithm. In this case k will be no more than 4 and it's possible to order elements in ≤ 5 queries. And the rest will take no more than $\lfloor \frac{n}{3} \rfloor$ iterations.