# Problem A. Another Problem about Queries on a Tree

First of all, note that instead of solving the problem for arbitrary $d_i$, we can reduce the problem to a tree with $d_i = 1$. To do this, let's reconstruct the tree $t$ so that in the new tree $t'$, the parent of vertex $v$ will be the vertex $u$ at a distance of $d_v$ upwards. In this case, all queries for moving pieces up the tree $t$ correspond to similar movements of piece in the tree $t'$, but now at a unit height upwards. The tree $t'$ can be constructed by traversing the tree using `dfs` while maintaining the vertices on the path to the root in a stack, or simply by counting binary lifts.

Instead of trying to quickly perform operations from the queries, let's try to count some useful characteristic using dynamic programming. We will find for each vertex $v$ the index of the first query after which there will be no pieces left in the subtree of vertex $v$. We will try to recalculate this characteristic for an arbitrary vertex $v$ through the values of its children. Note that for there to be no pieces in the subtree of $v$, each piece from the subtree of $v$ must first reach vertex $v$, and then be moved upwards by some query. Let's look at the moment when a piece last moves to vertex $v$. This moment $e$ is defined as $\max(dp_{u_1}, dp_{u_2}, \ldots, dp_{u_k})$, where $u_i$ is the number of the $i$-th child of vertex $v$ in the tree $t'$. Now it is not difficult to calculate $dp_v$. To do this, it is enough to find the number of the first query that will move a piece from $v$ to its parent, which occurs after the moment $e$ (in other words, the query that acts on the vertices characterized by the number recorded in vertex $v$). This can be done using the built-in function `std::lower_bound` applied to the array of queries.

In total, we obtain a solution that works in $\mathcal{O}(q + n \log q)$.

# Problem B. Partitioning into Triples

It will be convenient to precompute the array $cnt$ where $cnt_i$ $(1 \le i \le m)$ is the number of occurrences of the number $i$ in the array $a$.

Note that the problem can be solved using dynamic programming. We will define the dynamic programming state $dp_{i,\, left\_prev,\, left\_curr}$, which will store the number of ways to partition the numbers from 1 to $i$ into triples, if there are no numbers from 1 to $i-2$ left, $left\_prev$ numbers equal to $i-1$ left, and $left\_curr$ numbers equal $i$ left, and all triples $(i-1, i-1, i-1)$ have already been used. With this state, the base case will be $dp_{1,\,0,\,cnt_1} = 1$. The transitions of the dynamic programming will look as follows:

- For triples of identical numbers $i$, the transition will be $dp_{i,\, left\_prev,\, left\_curr} += dp_{i,\, left\_prev,\, left\_curr+3}$
- For triples of consecutive numbers, we will only be interested in the option $(i-1, i, i+1)$, since we want to use all remaining numbers $i-1$ and we can do this in only one way — by using them for consecutive triples (by the invariant of the dynamic programming). The transition for such triples will look like $dp_{i+1,\, left\_curr-left\_prev,\, cnt_{i+1}-left\_prev} += dp_{i,\, left\_prev,\, left\_curr}$ — we have used exactly $left\_prev$ numbers for triples of the form $(i-1, i, i+1)$.

To correctly recount, the first type of triples should be considered before calculating the next layer for the second type of triples, as we want to maintain the invariant that all triples of the form $(i, i, i)$ have already been used when recounting for layer $i+1$. The answer to the problem will lie in $dp_{m,0,0}$. Note that $left\_prev,\ left\_curr \le n$ for each $i : 1 \le i \le m$, which means that the solution works in $\mathcal{O}(m \cdot n^2)$, which is not enough to pass the time limit. The memory used by the solution is the same, which is not enough to pass the memory limit. Memory can be optimized by noting that for recounting the dynamic programming, we only need to keep the layers $i$ and $i+1$, and all unnecessary layers can be reused, thus instead of $\mathcal{O}(m \cdot n^2)$ memory, we will get $\mathcal{O}(n^2)$.

To solve the problem, it is necessary to slightly optimize the described dynamic programming. For this, we need to iterate over $left\_prev$ and $left\_curr$ within the bounds from 0 to their corresponding $cnt$. Now the solution for all $m$ will work in total for $\sum_{i=1}^{m} cnt_{i-1} \cdot cnt_i$. It is easy to see that such a sum can be bounded

above by $n^2$, knowing that $cnt_1+cnt_2+\ldots cnt_m = n$: $n^2 = (cnt_1+cnt_2+\ldots cnt_m)\cdot(cnt_1+cnt_2+\ldots cnt_m)$ $= cnt_1 \cdot (cnt_1 + cnt_2 + \ldots cnt_m) + cnt_2 \cdot (cnt_1 + cnt_2 + \ldots cnt_m) + \ldots + cnt_n \cdot (cnt_1 + cnt_2 + \ldots cnt_m)$, which is obviously not less than $\sum_{i=1}^{m} cnt_{i-1} \cdot cnt_i$, since $cnt_i \geq 0$. We obtain the final time complexity of $\mathcal{O}(m + n^2)$.

# Problem C. Klyukalo

If you change the $i$-th part, you can reduce the deviation by $\frac{1}{s_i}$. Therefore, it is better to start changing the parts which weigh is the least in the standard. Then, you can change the parts in non-decreasing order. If the current deviation $K_c - \frac{|a_i - s_i|}{s_i} > K$, then the part can be brought to the standard directly, and then $K_c$ can be reduced. However, if $K_c - \frac{|a_i - s_i|}{s_i} \leq K$, you need to solve the system of inequalities: $K_c - \frac{x}{s_i} \leq K$ and $K_c - \frac{x-1}{s_i} > K$. The solution to this system is $x = \left\lceil \frac{K_c - K}{s_i} \right\rceil$.

To avoid precision issues, all calculations should be performed using rational numbers. Alternatively, since the denominators of all fractions do not exceed 10, you can multiply all fractions by 2 520 (the least common multiple of the integers from 1 to 10) and solve the problem using integers.

The solution was obtained in $\mathcal{O}(n \log n)$.

# Problem D. Another Problem about the Game with Stones

Fix the pile $i$ for which we want to calculate the answer. Then iterate over the number of stones $x \in [l_i, r_i]$ that we want to take from this pile and determine if it is possible to do so while obtaining a total of $s$ stones. To do this, we need to find out if it is possible to collect a total of $s - x$ stones using the remaining piles.

Note that if we have two piles such that from the first pile we can take any number of stones in the range $[l_1, r_1]$, and from the second pile — any number of stones in the range $[l_2, r_2]$, then using both piles, we can obtain any number of stones in the range $[l_1 + l_2, r_1 + r_2]$.

Using this observation, we can easily compute the range of stone quantities that can be obtained using all piles except the $i$-th one, and then check if the number $s - x$ lies within this range. To do it we can calculate the range of stone quantities for each prefix and each suffix of piles.

We obtained a solution in $\mathcal{O}(nm)$ where $m = \sum(r_i - l_i)$.

The described solution can be optimized. Instead of iterating over the number $x$, we will determine in $\mathcal{O}(1)$ how many suitable numbers $x$ exist in the range $[l_i, r_i]$. We know that $x \in [l_i, r_i]$. We also know that using the remaining piles, we can collect any number of stones $y \in [L, R]$. We want to calculate the number of suitable $x$ such that there exists a number $y$ such that $x + y = s$. In other words, $x = s - y$. To do this, we need to intersect the ranges $[l_i, r_i]$ and $[s - R, s - L]$. The number of integer points in the intersection of these ranges equals the number of ways to choose the number $x$.

Asymptotics: $\mathcal{O}(n)$.

# Problem E. Lada Malina

The key part of a solution is to understand what are the locations that may be accessed from the origin in $T$ seconds. First observation is that we should investigate it only in case when $T = 1$ because $T$ is simply a scale factor. Let's denote this set for $T = 1$ as $P$.

We can find that the set $P = w_1v_1 + \ldots + w_kv_k \mid |w_i| \leq 1$ is always a central-symmetric polygon with the center in the origin. Actually, this polygon is a Minkowski sum of $k$ segments $[-v_i, v_i]$. Minkowski sum of sets $A_1, A_2, \ldots, A_k$ is by definition the following set: $A_1 + A_2 + \ldots + A_k = \{a_1 + a_2 + \ldots + a_k \mid a_1 \in A_1, a_2 \in A_2, \ldots, a_k \in A_k\}$. It can be built in $O(k \log k)$ time, although in this problem $k$ is very small, so one may use any inefficient approach that comes into his head.

After we found out a form of $P$, it's possible to solve the problem in $\mathcal{O}(qnk)$ by checking if each possible factory location belongs into a query polygon in $\mathcal{O}(k)$ time.

To solve the problem we need to use a trapezoidal polygon area calculation algorithm applied to our problem. Calculate the sum of points in each of $2k$ trapezoid areas below each of the sides of a polygon, and then take them with appropriate signs to achieve a result. Such trapezoid area can be seen as a set of points satisfying the inequalities $l \le x \le r$ and $y \le kx + b$. We have to answer queries like: "find the sum of all factories inside the trapezoid". Under transformation $x' = x, y' = y - kx$, this area becomes a rectangle, leading us to an $\mathcal{O}((q+n)k \log n)$ time solution using sweeping line and some data structure (for example, segment tree).

# Problem F. Choosing a Capital

In this problem the idea of binary search is applicable. Suppose we want to find the minimum *accessibility* by adding no more than $k$ edges. Then we can perform a binary search on the answer, find the minimum number of added edges, and compare it with $k$. We also note that it is optimal to direct edges only from the capital.

Consider the following greedy idea. Suppose we want to find the minimum number of edges so that the answer does not exceed $x$. Consider the farthest vertex $v$ from $s$. If the distance to it does not exceed $x$, then it does for all other vertices as well, and therefore no edges need to be added. Otherwise, consider the $(x-1)$-th ancestor of vertex $v$ in the tree (in other words, go up from vertex $v$ $x-1$ edges). We call this vertex $u$. Then we must direct at least one edge to the subtree of vertex $u$.

We will prove that there exists an optimal answer in which we directed an edge exactly to vertex $u$. Consider vertex $w$ from the subtree of $u$ to which we directed an edge. Then let's remove the edge $s \to w$ and add the edge $s \to u$. This does not increase the number of added edges, and all distances will still remain $\le x$, since the depth of the subtree of vertex $u$ does not exceed $x-1$, and for vertices outside the subtree, we have not broken anything.

Thus, the following greedy algorithm works: while possible, we find the farthest vertex $v$, direct an edge to its $(x-1)$-th ancestor $u$, and continue the algorithm. Note that after adding the edge $v \to u$, we can forget about the vertices in the subtree of $u$, so at each iteration, the graph remains a tree. We also note that if we have already directed $k+1$ edges, we can terminate the algorithm, so for a fixed vertex $s$, the naively implemented greedy algorithm works in $\mathcal{O}(nk)$.

Let's consider the complete solution to the problem. We will hang the tree from vertex number 1 and discuss the solution for the first vertex as the capital.

We need to be able to do the following: quickly find the deepest vertex, go up from it $x-1$ edges, and mark all vertices in the subtree as removed.

Consider the Euler tour of the tree (first the vertex is listed, then its subtrees are recursively listed). In such a tour, the subtree of any vertex forms a subsegment of the tour. Let's maintain the current distances to the vertices in the tree as the maximum on the segment of the Euler tour. Then, to find the deepest vertex, we need to take the maximum over the entire array of the tour. To mark the subtree as removed, we subtract a large number from all values on the segment (for example, it is sufficient to subtract $n$).

To go up $x-1$ edges, we will precompute binary lifts on the tree, so we can go up any distance in $\mathcal{O}(\log n)$.

Thus, if we perform the described process $k$ times, we will get a check for binary search in $\mathcal{O}(k \log n)$ for one capital after preprocessing in $\mathcal{O}(n \log n)$.

Now let's discuss how to generalize this solution for all capitals. We will calculate the answers for capitals in the order of depth-first search of the tree. When we descend from a vertex to its child, we need to recalculate the array of distances to all vertices. It changes as follows: for vertices in the subtree of the child, 1 is subtracted, and for all other vertices, 1 is added. Therefore, we can maintain the current segment tree with distances to vertices and recalculate it when moving to a child using segment additions.

A few details remain to be discussed. In each vertex, we perform binary search on the answer, during which we change the current array of distances with segment additions. After we find the minimum number

of edges (or realize that it exceeds $k$), let's roll back all the additions made using subtractions on the segment.

We also note that if for the original capital we needed to go up $x-1$ edges, then for the subsequent capitals $s$, we need to find the $(x-1)$-th vertex on the path from the current (deepest) vertex $v$ to the current capital. To do this, let's find $\text{LCA}(v, s)$ (the deepest common ancestor) and calculate the lengths of the paths from $v$ to LCA and from $s$ to LCA. Then we reduce the query to going up on one of these two vertical paths. The search for LCA can be implemented using any standard algorithm on the top of binary lifts.

Now we note that if for the original capital we subtracted $n$ on the segment of the subtree, then now we need to subtract on the subtree $u$, as if the current capital $s$ were the root. There are two cases. If vertex $u$ is not an ancestor of $s$, then the subtree when hanging from $s$ remains the same as when the original capital was used. If $u$ is an ancestor of $s$, then let's consider vertex $w$ on the path from $s$ to $u$, whose parent is $u$. To find this vertex, we need to go up from $s$ by a distance one less than the length of the path between $s$ and $u$. Now the subtree $u$ when hanging from $s$ consists of all vertices except the original subtree of vertex $w$, so to add to it, we can add on the suffix and prefix of the tour.

In the end, we get a solution in $\mathcal{O}(nk \log^2 n)$: binary search in each vertex, and checking in $\mathcal{O}(k \log n)$.

To fully solve the problem, we need to eliminate the binary search in each vertex. For this, we note the following fact: if vertices $v$ and $u$ are connected by an edge of the tree, then the answers for these vertices as capitals differ by no more than 1. This is true because when changing the capital from $v$ to $u$, only one edge changes direction (the edge between $v$ and $u$), and if we could previously reach from vertex $v$ to vertex $w$ in $x$ edges, then now we can reach it in $x+1$ edges by initially going through the edge $u \to v$.

Thus, the complete solution looks like this: in the original vertex, we will perform binary search between 1 and $n-1$, and then when transitioning in the depth-first search from vertex $v$ to vertex $u$, we will perform binary search between $\text{ans}[v]-1$ and $\text{ans}[v]+1$, and it will work in $\mathcal{O}(1)$ checks.

We obtain the final solution to the problem in $\mathcal{O}(nk \log n)$.

# Problem G. House Search

To begin with, we will describe the solution that uses $k = \lceil \log_2 n \rceil$. For each house, we say it "completes" a segment in the segment tree if it is the last index in that segment to be assigned a value. In each round, Anna picks $v =$ the lowest depth of any segment that $i$ completes. For Bertil, consider the values written on the doors assuming Anna picks the last value according to their strategy. In this case, Bertil can just find 1 and return its index. However, if there is no 1 in the array, then there are two cases:

1. The 1 was replaced by a 2. Then there are exactly two houses with 2s written on them, and we know that one of them must be Anna's, since there should only be one 2 if the strategy was followed for all indices. This is because the index that was picked later not only completes the segment on layer 2, but also necessarily completes the segment on layer 1 since the other half is already completed.
2. The 1 was replaced by a value greater than 2. Then, locate the half that contains the only 2 in the array. We know Anna's house must not be in this half, since it was completed first. Thus, we may recursively solve the problem on the other half, completing the solution.

Note that since 1 is never actually picked by Anna, we can subtract 1 from all values Anna picks to ensure we have exactly $k = \lceil \log_2 n \rceil$.

For illustrative purposes, consider an example where $n = 8$, Emma's house has index 2, and Emma and Anna visit the houses in the following order: $[0, 5, 4, 1, 6, 3, 7]$. Then, after Emma writes a number $x$ on her own house, the array $a$ will be $[3, 2, x, 3, 2, 3, 3, 1]$. We run through Bertil's strategy in the 3 possible scenarios:

1. $x = 1$: since there are duplicate 1s, it is clear that one of these must be for Emma's house. Guess 2, 7 and return.

2. $x = 2$: since there is a unique 1, Emma's house must be in the half of the array that does not contain 1. Now we get to $a[0, 3] = [3, 2, 2, 3]$. Repeat our logic recursively. Since there are duplicate 2s, it is clear that one of these must be for Emma's house. Guess 1, 2 and return.

3. $x = 3$: since there is a unique 1, Anna's house must be in the half that does not contain 1. Now we get to $a[0, 3] = [3, 2, 3, 3]$. Since there is a unique 2, Anna's house must be in the half of the array that does not contain 2. Now we get to $a[2, 3] = [3, 3]$. Since there are duplicate 3s, it is clear that one of these must be for Anna's house. Guess 2, 3 and return.

Now we are ready to describe the solution that uses $k = \mathcal{O}(\log \log n)$.

To solve the problem, we consider the following two-phase approach: In the first phase Anna just writes the number $k$ on the first $n - \Theta(\log(n))$ houses which are visited. In the second phase (which we describe later), we promise that Anna will not write the number $k$ on the remaining $\Theta(\log(n))$ houses. Now we need to handle two cases:

- If Emma does not write the number $k$ on her house, then we know it is one of the $\Theta(\log(n))$ houses not with a $k$.
- If Emma writes the number $k$, then we need to figure out which of the $n - \Theta(\log(n))$ houses with $k$ written on it belongs to Emma.

In the former case, we have essentially reduced the problem to an instance with $n' = \Theta(\log(n))$, but now with numbers $1, \ldots, k - 1$.

For the latter case, we make the following observation: If we known the sum $s$ of indices, modulo $n$, of the $\Theta(\log n)$ houses not part of the first phase, then we can figure out Emma's house in case she writes $k$. Indeed, we can identify that we are in the second case by counting the number of houses with $k$ written on them. Then, we know that Emma's house index plus all indices of houses without a $k$, must equal $s$. Hence we can solve for Emma's house index.

The strategy for the second phase is now the following: we want to essentially solve an instance where $n' = \Theta(\log n)$ while simultaneously encoding the sum $s$ (which is known at the start of the second phase). One has to be a bit careful on how to encode the sum $s$, in the remaining $n' = \Theta(\log n)$ houses, since one of these houses we do not have control over and Emma can write $k$ on it instead. One way of doing it is to write $s$ (modulo $n$) in binary and then duplicate each bit, to make sure that we can recover the sum $s$ even after one bit (the one corresponding to Emma's house) is dropped. Solving the instance on $n' = \Theta(\log(n))$ in our strategy can be done using the solution with $k = \lceil \log_2 n' \rceil$ which was described above.

Here are some hints for the correct implementation:

- Use $k = 7$ and $n - 32$ numbers for the first phase (so $n' = 32$).
- In the second phase we run described solution with $n' = 32$, which will use numbers $1, \ldots, 5$.
- Instead of encoding the sum $s$ modulo $n$, we use the fact that we have two guesses and encode it modulo $\frac{n}{2}$. This requires 16 bits of information, since $2^16 = 65\,536 > \frac{n}{2}$.
- Note that Anna will write exactly 16 5s in the second phase. Hence we can change some of these 5s to 6s to encode the sum $s$.

# Problem H. Records and Anti-records

We will use dynamic programming to solve this problem. Let $dp_{i,a,b}$ be the maximum answer if we split the prefix of length $i$ such that the last record in $q$ is equal to $a$, and the last anti-record in $r$ is equal to $b$.

Then, there are the following transitions from $dp_{i,a,b}$:

- $dp_{i+1,a,b} \leftarrow dp_{i,a,b}$, if $p_{i+1} < a$, take it in $q$
- $dp_{i+1,a,b} \leftarrow dp_{i,a,b}$, if $p_{i+1} > b$, take it in $r$
- $dp_{i+1,p_{i+1},b} \leftarrow dp_{i,a,b} + 1$, if $p_{i+1} > a$, take it in $q$
- $dp_{i+1,a,p_{i+1}} \leftarrow dp_{i,a,b} + 1$, if $p_{i+1} < b$, take it in $r$

In total, there are $\mathcal{O}(n^3)$ states, with $\mathcal{O}(1)$ transitions from each. Thus, the solution works in $\mathcal{O}(n^3)$ for one test case.

We will divide the states into two types:

*1.* State $dp_{i,a,b}$, where $a > b$. Notice that for any $x$, either $a > x$ or $b < x$. Thus, by taking the element $x$ into the corresponding subsequence, we can not change the record and anti-record. Therefore, from the remaining suffix, we can discard any element without changing the answer. Then we will choose some subsequence $p'$ in the suffix of length $n - i$ as records greater than $a$, and subsequence $q'$ as anti-records less than $b$, discarding elements that do not belong to $p'$ or $q'$. To maximize the answer, we need to choose as $p'$ the LIS (longest increasing subsequence) and as $q'$ the LDS (longest decreasing subsequence) in the suffix of length $n - i$.

*2.* State $dp_{i,a,b}$, where $a < b$. Notice that in this state, we have split the prefix of length $i$ into $p$ and $q$ such that the maximum element in $p$ is equal to $a$, and the minimum element in $q$ is equal to $b$. Thus, all elements $\leq a$ are in $p$, and all elements $\geq b$ are in $q$. There are at most $i + 1$ such partitions of the prefix of length $i$. In total, we find that the number of states of type *2* is $\mathcal{O}(n^2)$.

We will count for each $i, x$: $LIS(i, x)$ — the LIS in the suffix of length $n - i$, starting with an element $> x$. Similarly, we will count $LDS(i, x)$ — the LDS starting with an element $< x$. This part works in $\mathcal{O}(n^2 \log n)$ or $\mathcal{O}(n^2)$ depending on the implementation.

Next, we will maintain the dynamics only for states of type *2* and update the answer when transitioning to state type *1* in $\mathcal{O}(1)$.

Finally, notice that the dynamics can be recalculated more efficiently using a segment tree. Specifically, we can count the LDS and LIS using a segment tree by increasing the length of the suffix. However, since in the dynamics we need to increase the length of the prefix, we can roll back the LDS and LIS. This part works in $\mathcal{O}(n \log n)$.

Next, to maintain all states of type *2* — $(i, a, b)$, we can keep them in one of its elements, for example in $a$ in the segment tree. When considering the element $p_i = x$:

1. Exactly 1 state of type *2* $(i, a, b)$ is removed such that $a < x < b$, and two new states of type *2* — $(i + 1, a, x)$ and $(i + 1, x, b)$ are added.
2. From all states of type *2* $(i, a, b)$ such that $x < a < b$, we can transition to state type *1* — $(i+1, a, x)$. In this case, the answer will be $dp_{i,a,b} + 1 + LIS(i + 1, a) + LDS(i + 1, x)$.
3. Similarly for $a < b < x$.

Thus, if we maintain in one segment tree the element $a$ the value $dp_{i,a,b} + LIS(i + 1, a)$, we can update the answer for the second case in $\mathcal{O}(\log n)$. The same applies to case 3.

In total, we get a solution in $\mathcal{O}(n \log n)$ for one test case.

# Problem I. Traversals of a Binary Tree

Let's consider the simplified task when the tree is a bamboo and queries on the path to the root are prefix queries.

We will use the method of SQRT decomposition. We will divide the nodes into $\lceil \frac{n}{B} \rceil$ blocks: from 1 to $B$, from $B + 1$ to $2B$, and so on. Inside each block, we will sort the nodes in increasing order of depth. We will maintain the number of nodes on the prefix of the block that have the number 1.

Consider a query to assign the number $x$ on the segment $[l, r]$. This segment covers $\mathcal{O}(\frac{n}{B})$ blocks entirely and $O(1)$ blocks partially. If a block is covered entirely, the prefix sums on it are trivially computed (since now all numbers on the nodes are equal to $x$), and we can process this case in $\mathcal{O}(1)$. If a block is covered partially, we will recount it completely. Thus, we have processed the operation in $\mathcal{O}(B + \frac{n}{B})$.

For the second type query, we need to find the number of nodes on the path to the root that have the number 1. Notice that this path includes the prefix of nodes from each block. Thus, if we know the sizes of these prefixes, we can get the answer to the query in $\mathcal{O}(\frac{n}{B})$. We can obtain the sizes using binary search, but there is a more efficient solution.

Since we know all the queries in advance, we can sort them and precompute the corresponding sizes of the prefixes for each block using two pointers in $\mathcal{O}(q \log q + (q + B)B)$. This method requires $\mathcal{O}(qB)$ memory to store the precomputation. However, we can eliminate this if we traverse the queries separately for each block.

In total, we obtain a time complexity of $\mathcal{O}((n + q)\sqrt{n})$ with $\mathcal{O}(n + q)$ memory when $B \approx \sqrt{n}$.

Now we will generalize the described solution to an arbitrary binary tree.

We will divide the nodes into blocks of size $B$. For each block, we will build a compressed tree, that is, a tree on the nodes of the block such that $p$ is the parent of $i$ if $p$ is the deepest ancestor of $i$ that is contained in the block. (For blocks that do not contain the root of the original tree, it may be useful to introduce a dummy root.) Instead of prefix sums, we will maintain sums on the path to the root for each node.

It is easy to see that queries are processed similarly to the previous solution. For precomputing queries for a fixed block, instead of two pointers, we can perform a depth-first traversal while maintaining the deepest node from the block that is an ancestor of the current node.

We will find for each node $i$ the node $L_i$ — the deepest ancestor of $i$ such that the node lies in its left subtree. This can be done in $\mathcal{O}(n)$ using depth-first search. Notice that if node $i$ lies in the left subtree of its ancestor $p$, then $p = L(L(\dots L(i)\dots))$.

If we connect node $i$ and $L_i$ with an edge, we obtain a set of trees or a forest $L$. From the previous property, it follows that the ancestors of node $i$ in whose left subtrees it lies are the ancestors of $i$ in $L$. Thus, processing queries on $L$ can be done similarly to the solution described above. This solution also works in $O((n + q)\sqrt{n})$.

# Problem J. Sport is sport

To begin with, notice that Mike will run $\text{lcm}(n, k)$ trees and will make $\frac{\text{lcm}(n,k)}{k} = \frac{n}{\gcd(n,k)}$ runs (each run lasts until the boy stops to delete photos in the phone memory), which does not exceed $n$.

Also, note the following: the boy will take one video starting from all trees whose indices are multiples of $g = \gcd(n, k)$ (with zero-based indexing). From other trees, the boy will never start shooting. Therefore, we will reorder the segments so that the first corresponds to the half-interval $[0, k)$, the second $[g, k + g)$, and so on, the last $[n - g, (n + k - g) \mod n)$ — on this segment the boy finishes his report.

Now let's look at the problem in a new way: fix a segment and consider that the contribution to the answer is given only by the first tree of each type. It is clear that this approach is equivalent to the approach with removing duplicate elements, which was used to solve the second subgroup. We will take a non-trivial step: we will stop modeling the boy's behavior and calculate what total (across all segments) contribution each tree will give to the answer.

Let's fix a tree, say its index is $i$. We need to count the number of segments on which it will contribute to the answer, that is, the number of segments on which this tree will be the first tree of its type. To do this, let's look at the previous tree of the same type on the circle, let its index be $j$. The tree at position $i$ will contribute to the answer for the segment if and only if the segment includes the tree at position $i$ and does not include the tree at position $j$.

First, consider the case $j < i$. Since all segments start at positions that are multiples of $g = \gcd(n, k)$, we

just need to count the number of integers in the half-interval $(j, i]$ that are multiples of $g$. Their count is equal to $\left\lfloor \frac{i}{g} \right\rfloor - \left\lfloor \frac{j}{g} \right\rfloor$. This is true because the number of such integers in the half-interval from zero to $a$ equals $\left\lfloor \frac{a}{g} \right\rfloor$.

If $j > i$, then their count is equal to $\left\lfloor \frac{i+n}{g} \right\rfloor - \left\lfloor \frac{j}{g} \right\rfloor$ — we can imagine that we did not reset the index counter when moving from $j$ to $i$ through the zero index.

Thus, it is sufficient to consider each type of tree separately and count for each tree of that type its total contribution to the answer, based on the position of the previous tree of that type.

Time complexity: $\mathcal{O}(n)$.

# Problem K. Conference

To solve this problem, an important subtask will be determining the size of the maximum joint set for an arbitrary set of segments. A greedy algorithm can be used to solve this subtask. First, all events should be sorted in increasing order of $r_i$. After that, the segments should be iterated in the order of sorting, and the next segment should be included in the answer only if it does not overlap with the previous one. It is not difficult to show that this greedy algorithm always finds a correct answer to the subtask.

To solve the subtask in $\mathcal{O}(n^2)$, we will use the following idea: find a set $s_1$ where the size of the maximum joint set $\leq \frac{ans}{2}$ and a set $s_2$ where this size $\geq \frac{ans}{2}$, where $ans$ is the size of the maximum joint set in the original set of segments.

We will gradually transform the set from $s_1$ to $s_2$.

To do this, we will take any element $t_1$ from $s_1$ that is not in $s_2$ and replace it with an element $t_2$ from $s_2$ that is not in $s_1$. Note that removing $t_1$ from $s_1$ can only decrease the answer, and after inserting element $t_2$ into $s_1$, the answer can only increase. It is easy to see that when transitioning from $s_1$ to $s_1 - t_1 + t_2$, the answer will change by at most 1 (in either direction). Thus, at some point during the transformations, we will find a set where the answer will be exactly $\frac{ans}{2}$.

The implementation of this algorithm without optimizations gives us an asymptotic complexity of $\mathcal{O}(n^2 \log n)$, as we will run the greedy algorithm for each of the $n$ sets. To achieve a clean $\mathcal{O}(n^2)$ estimate, it is sufficient in the greedy algorithm not to run the sorting each time, but to precompute the order of sorting the segments.

For the complete solution, we will do the following: find the first $\frac{ans}{2}$ segments using the greedy algorithm by sorting by the right endpoint and passing from left to right, as well as the first $\frac{ans}{2}$ segments by sorting by the left endpoint and passing from right to left. We will call these sets $L$ and $R$: $|L| = |R| = \frac{ans}{2}$.

Then all other segments can be divided into two groups: those that intersect with some segment from $L$, and all the others. We will call these groups $L'$ and $R'$. Note that $|L| + |L'| + |R| + |R'| = n$. Thus, either $|L| + |L'|$ or $|R| + |R'|$ will be at least $\frac{n}{2}$. That is, we can take the larger of the sets $L \cup L'$ and $R \cup R'$, and the answer in it will be the sought one, but the set itself may exceed the required size of $\frac{n}{2}$. In that case, we simply remove some segments from it that are not in the answer to achieve the required size.

# Problem L. Apocalypse

Note that the infection spreads an equal distance from the original polygon, and its area on day $i$ is equal to $S_i = 2^i \cdot S_0$. Let the distance the infection has spread on day $i$ be $d_i$. Then $S_i = S_0 + P_0 \cdot d_i + \pi d_i^2$, where $P_0$ is the perimeter of the original polygon. This is a quadratic equation, and by finding solutions for it, we can determine $d_i$ for each day. It is easy to see that given the constraints on the input data, $d_{70} > 10^{10}$, which means that any settlement will be infected within the first 70 days. Therefore, it is sufficient to quickly calculate the distance $D_j$ from each settlement to the original polygon and find the first day $i$ when $d_i \geq D_j$. To quickly calculate the distance from a point to a convex polygon, you can follow several steps:

1. Check if the point is inside the polygon in $\mathcal{O}(\log N)$; if it is inside, the distance is 0, otherwise

proceed to the next steps;

2. Construct tangents from the point to the polygon in $\mathcal{O}(\log N)$, thus finding two vertices of the polygon with indices $l$ and $r$;

3. The polygon is split into two parts by the found vertices. Since the polygon is convex, in one of the parts, the distances to the point will represent a convex function, for which ternary search can be used to find the minimum value. In fact, it is sufficient to find the nearest vertex and then choose the minimum distance to the segments adjacent to this vertex. This step of the algorithm also works in $\mathcal{O}(\log N)$.

Thus, the algorithm for finding the appropriate day for each settlement works in $\mathcal{O}(\log N)$, and the entire solution will work in $\mathcal{O}(Q \cdot \log N)$.

# Problem M. Maximizing Profit

Consider the following solution that uses dynamic programming. The state of the dynamic programming will be a mask encoding a subset of the digits of the original number. For each such mask $m$, we will store in $\texttt{dp}[m]$ the minimum possible penalty required to obtain a prefix consisting of the digits corresponding to the mask. We will iterate over which digit from the set will be the last in the prefix, then we know $\texttt{dp}[m_{prev}]$ — the penalty for obtaining the prefix without this digit, and the number of swaps needed to move the last digit of the set to its place. This can be calculated in $\mathcal{O}(n)$ by finding the position of the digit in the original number and the number of digits from the set $m$ that were moved from positions to the right of it to the left.

If we also remember the digit that is beneficial to place last in the dynamic programming, we can reconstruct the optimal number at the end — it is enough to "collect" it from the end, each time moving to the mask without the last digit. This solution works in $\mathcal{O}(2^n \cdot n)$ if we precompute the number of digits moved to the prefix from each suffix (using suffix sums).

Now let's note the following fact: the penalty for moving a larger digit to a higher place than 9 is always less than the increase in the result. Indeed, to swap the digits in places $a$ and $b$, it requires swaps costing $(2a - 2b - 1) \cdot y$, while the benefit will be equal to $d \cdot (10^a - 10^b)$, where $d$ is the difference between the digits in these places. For $y \le 10^8$, the total penalty will not exceed $2 \cdot 10^5 \cdot 10^8 = 2 \cdot 10^{13}$, while the second expression is at least $9d \cdot 10^{a-1}$, and for $d > 0$ and $a > 13$, it is strictly greater than $2 \cdot 10^{13}$.

However, for places from 10 to 13, it is true that $2a - 2b - 1 < 30$, and thus the estimate for the penalty for maximizing these places can actually be refined: $(2a-2b-1) \cdot y < 3 \cdot 10^9 < 9d \cdot 10^{a-1}$. So, for $y \le 10^{16}$, it can be similarly proven that all places starting from place number $16 + 5 = 21$ will be occupied by the maximum possible digits, and for the remaining places, it is sufficient to apply the described dynamic programming solution.