Разбор задачи «Draw Polygon Lines»

For a permutation p, let's define $a(p)$ as the number of acute angles among the angles $\angle A_{p_{i-1}} A_{p_i} A_{p_{i+1}}$.
task $= 1$
You need to find a permutation p such that $a(p) = n - 2$.
It is possible to iterate through all permutations in $O(n!)$.
For subtasks with random points, optimization approaches can be used (we will discuss later).
The complete solution for this case is as follows:

- Let's build the permutation, with the first point being A_1.
- At each iteration, we will select the point $A_{p_{i+1}}$ (from the remaining points) such that the length of the segment $A_{p_i} A_{p_{i+1}}$ is maximal, i.e., the farthest point.
- Notice that $\angle A_{p_i} A_{p_{i+1}} A_{p_{i+2}}$ is acute because $A_{p_i} A_{p_{i+1}} \geqslant A_{p_i} A_{p_{i+2}}$.
- We obtain a permutation such that $a(p) = n - 2$, which is the maximum.

Optimization Ideas
Optimization works well for subtasks with random points!
We need to minimize the functional $f(p) = -a(p)$ (if $task = 1$) or $f(p) = |a(p) - k|$ (if $task = 3, 4$).
For example, we can swap two random elements of the permutation and recalculate the functional in $O(1)$. Then, using local optimization or simulated annealing, we can find the answer.

To cover all values of k, we can use "almost discrete continuity". When locally changing the function, it changes by $\leqslant 6$. Therefore, we can minimize the function and then, conversely, maximize it. Thus, if we remember the answers for all obtained values, we will cover all necessary k values quickly.

$x_i < x_{i+1}, y_i < y_{i+1}$
The main idea about monotonicity:

- Suppose we have three points $A_1(x_1, y_1)$, $A_2(x_2, y_2)$, $A_3(x_3, y_3)$, such that either $x_1 < x_2 < x_3$ or $x_1 > x_2 > x_3$, and either $y_1 < y_2 < y_3$ or $y_1 > y_2 > y_3$.
- Then the angle $\angle A_1 A_2 A_3$ is always obtuse, and the angles $\angle A_2 A_1 A_3$ and $\angle A_2 A_3 A_1$ are always acute.

Then the solution works as follows:

- For maximization, we simply consider the permutation $1, n, 2, n-1, 3, n-2, \ldots$.
- To obtain a specific k, the following permutations work:

$$[1, k+2, 2, k+1, 3, k, \ldots], [k+3, k+4, \ldots, n]$$

or

$$[k+2, 1, k+1, 2, k, 3, \ldots], [k+3, k+4, \ldots, n]$$

Monotonic Subsequences

- Any permutation of length $n \leqslant 2x^2$ can be divided into $\leqslant 2x$ increasing or decreasing subsequences. Algorithm:
  - Find the longest increasing subsequence of length k in $O(n \log n)$.
  - If $k \leqslant 2x$, then we automatically obtain a division into k increasing subsequences (for example, using the binary search algorithm).
  - If $k > 2x$, then we remove this increasing subsequence and proceed to a permutation of length $\leqslant 2x^2 - 2x - 1 \leqslant (x-1)^2$.

- The total construction time is $O(n\sqrt{n}\log n)$.

task $= 2$

- We like monotonic subsequences — we can create many obtuse angles.
- We find a division of the set of points into $\leqslant \sqrt{2n} \leqslant 400$ monotonic subsequences.
- It is possible to connect the subsequences, and the number of acute angles will be $\leqslant 2\sqrt{2n}$.

task $= 3$

- In monotonic subsequences, we can obtain any number of acute angles.
- We want to connect the broken lines between subsequences. Problems will occur at the boundaries (unpredictable change in the number of acute angles).
- If we fix the first, second, last, and penultimate elements in each subsequence, then when connecting, the number of acute angles will be predictable.
- Now we can obtain any k, but with a certain parity (due to the fixing).
- If we try several random options, we will cover all parities.

task $= 4$

- We create similar examples.
- For all of them together, we can carefully calculate the hashes.
- The previously described solution immediately provides any $k \in [5\sqrt{2n}, n - 5\sqrt{2n}]$.
- It is possible to improve the constant to 2 carefully by analyzing the cases of connection.
- The overall complexity is $O(n\sqrt{n}\log n + qn)$.

Разбор задачи «Evidence Board»

General note: Let's rephrase the problem. Instead of "building" the tree from scratch, we will do the opposite and remove edges from the tree. After that, we will reverse our obtained answer. In the tutorial we will solve the problem in the format of "removing the tree". The list of stickers in the tree vertex will be called a stack, and the top of the stack will always be the sticker that should be removed first.

- **Subtask №1**

  This subtask is solved by full bruteforce. There are $(n-1)!$ possible orders of edge removal. Each order is trivially checked in $\mathcal{O}(n)$.

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

- **Subtask №2**

  Bamboo.

  Dynamic programming on the prefix. $dp[v][i]$ a boolean value: whether it is possible to remove the bamboo from the vertices from 1 to v, but if the stack for vertex v contains a single element $c_{v,i}$ ($i \in \{0, 1\}$).

  Dynamic programming base: $dp[1][0] = dp[1][1] = true$

  Recalculation of dynamic programming: we need to consider all four variants with which elements of the stacks the edge between v and $v-1$ will be removed, if this edge can be removed, we recalculate through $dp[v-1][i]$.

The answer exists if $dp[n][0] = true$, otherwise no.

To restore the answer, for each edge, we need to remember with which i_1, i_2 it was removed. First remove all edges of the form $i_1 = 0, i_2 = 0$, then $i_1 = 1, i_2 = 0$, $i_1 = 0, i_2 = 1$ and $i_1 = 1, i_2 = 1$.

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

- **Subtask №3**

  Star.

  Go through the leaves from 2 to n. Let's say we are currently considering leaf v. In the array c_1, find the minimum number $\geqslant w_v - c_{v,1}$. It is easy to understand that it is optimal to remove the edge leading to v with this number. After that, remove the found number from c_1. In the implementation, we maintain c_1 in std::multiset and use the lower_bound method.

  The required order of edge removal corresponds to the original order of numbers in c_1.

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

- **Subtask №4**

  Two stars, bases connected by an edge.

  Solve two stars similarly to group 4. After that, there will be one number left in both bases. If they allow removing the edge between the bases, the answer is Yes, otherwise No. The suitable order of edge removal: all edges of the first star before the base vertex, all edges of the second star before the base vertex, the edge between the bases, all edges of the first star after the base vertex, all edges of the second star after the base vertex.

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

- **Subtask №5**

  The values of the numbers in the vertices are increasing.

  If at some point an edge can be removed, then it can always be removed later. Therefore, at any time, it is permissible to remove any possible edge. The solution will be a greedy algorithm: we look for any edge that can be removed and remove it from the tree, repeating $n - 1$ times. If at some point there are no edges ready for removal, the answer is No.

  Asymptotics: $\mathcal{O}(n^2)$

- **Subtask №6**

  Note that for any test, the following fact is true: if we reverse all the stacks of the original test and call it a *new* test, then if there is no answer for the original test, there is no answer for the new one. And if the answer exists, then the answer to the *new* test will be the reversed answer for the original test. From this observation, the solution of group 6 directly follows from the solution of group 5: we need to reverse all the arrays, as well as the final answer.

  Asymptotics: $\mathcal{O}(n^2)$

- **Full solution**

  The full solution is based on one key idea: "For any way to assign a pair of stack elements to each edge of adjacent vertices (without repetitions), there exists an edge removal order such that each edge will be removed together with its assigned elements".

  Proof of the key idea: Consider any assignment. Each vertex points to one adjacent edge to which the top element of the stack is assigned, as "half ready for removal". Since there are n vertices and $n - 1$ edges, there will be an edge to which both adjacent vertices point. This edge can be removed. After removing it, the tree splits into two other trees, and the same reasoning applies to them.

  After the key idea, the problem split into two parts.

  1. Build any correct assignment of edges to stack elements

2. Restore the order of edge removal

To build the assignment, we use a greedy algorithm similar to the solution of group 3. We write down the order of vertices of any tree traversal. We iterate through the vertices in the reverse order of traversal (from the leaves). Let the currently iterated vertex be v. Then we assign the minimum suitable number ($\geqslant w_v - c_v$) from the parent vertex to the only remaining number in the stack of vertex v.

For the implementation, we store a `std::multiset` of numbers in each vertex. We use `lower_bound` and `erase` in it.

Restoration: for each edge, we maintain how ready it is for removal (0/1/2). The edges ready for removal are stored in the stack. Iteratively, we take any edge from the stack, remove it, and increase the "readiness for removal" of no more than 2 other edges. If any of them becomes ready for removal, we add it to the stack.

Final asymptotics: $\mathcal{O}(n \log n)$

Разбор задачи «More Gifts»

Subgroup 1. $n \leqslant 100, k \leqslant 10$.

We use dynamic programming. Let $dp[i]$ be the minimum number of subsegments in a prefix of length i, with $dp[1] = 1$.

Recalculation for state i: $dp[i] = \min(dp[i], dp[j] + 1)$, if the subsegment $[j + 1, i]$ contains no more than t different numbers.

If we recalculate this dynamic from the end, we can check the condition in $O(1)$, maintaining the number of different numbers in the subsegment. Thus, we get a solution for $O((nk)^2)$ or $O((nk)^2 \log(nk))$.

Subgroup 2. $t = 1$.

We divide the original array into blocks of consecutive equal numbers. Let the number of blocks be cnt. Then, the answer will be $cnt \cdot k$ if the first block does not match the last one, otherwise $(cnt - 1) \cdot k + 1$.

Proof of the Greedy Solution.

Consider some optimal partition, let it differ from the greedy one. Find the first position where there are differences. In this case, the length of the subsegment in the optimal partition is less than in the greedy one. Shift the right boundary of this subsegment to the position in our greedy solution. The partition will remain correct and the number of subsegments will not increase. By continuing this process, the optimal partition will turn into the greedy one, and the number of subsegments will not increase. It follows that the greedy partition is optimal.

Subgroup 3. $n \leqslant 1000, k \leqslant 1000$.

We construct a fully repeated array of size nk. Using a greedy algorithm, we obtain the minimum number of gifts. Solution for $O(nk)$ or $O(nk \log(nk))$.

Subgroup 4. $n \leqslant 1500, k \leqslant 10^6$.

Consider the following algorithm: from position i in the original array, we greedily collect gifts until the end of the current stack. Then we will collect from the next stack, as long as it does not increase the number of subsegments. For each position i in the original array, we will remember the position following the one where we stopped, as well as the number of subsegments obtained in the process, and call these values $go[i]$ and $cnt[i]$.

We calculate the arrays go and cnt for $O(n^2 \log n)$ straightforwardly. Now, to calculate the answer, we introduce a parameter cur, which will indicate the position in the array up to which we optimally divided the array. We iterate from 1 to k, adding $cnt[cur]$ to the answer on each iteration, and changing cur to $go[cur]$. The answer calculation takes $O(k)$, adding the pre-calculation, we get $O(n^2 \log n + k)$.

Subgroup 5. $k \leqslant 10^6$.

We will learn to find the arrays go and cnt in $O(n)$. To do this, we will use the two-pointer method to find the maximum length of a subsegment containing no more than t numbers for position i, which we will call len. Calculating the arrays go and cnt from the end of the original array, the recalculation for position i can be done as follows:

- If $i + len \geqslant n - 1$, then $go[i] = (i + len)\%n + 1$, where % is the modulo operation, and $cnt[i] = 1$.
- Otherwise, $go[i] = go[i + len + 1]$, where % is the modulo operation, and $cnt[i] = cnt[i + len + 1] + 1$.

Calculating the answer as in subgroup 4, we get a solution for $O(n + k)$ or $O(n \log n + k)$.

Complete Solution.

We optimize the answer calculation. Notice that if $k > n$, then when calculating the answer, the value of *cur* will take on some values several times. Find the first repetition. Then find the number of steps at which this repetition occurred, as well as the value added to the answer during this time. By dividing the remaining number of stacks by the number of steps, we find how many more repetitions will occur, and process the remaining at the end of the stack separately.

In total, the solution is for $O(n)$ or $O(n \log n)$.

Разбор задачи «Big Persimmon»

To begin with, let's notice two useful things:

- Maximizing the sum of taken elements is the same as maximizing the difference between what you take and what your opponent takes.
- As long as there is at least one piece left on the table, both people have eaten the same amount, so the difference between them is zero.

To start solving the problem in $O(n^2 \cdot \max_i(w_i))$, we will use dynamic programming:

Let's denote dp[l][r][dif] as the difference between the amount the first person takes and the amount the second person takes, if they were to play on the segment of elements [l, r], with the first person starting to eat after dif seconds from the second person (where dif can be negative).

The base case for the dynamic programming is: dp[i][i-1][dif] = 0, and the answer lies in dp[0][n-1][0].

The sign of dif determines who will make the decision about which piece to take first. Thus, the transitions are as follows:

For dif $\leqslant 0$: dp[l][r][dif] = max of

- $dp[l + 1][r][dif + w[l]] + w[l]$
- $dp[l][r - 1][dif + w[r]] + w[r]$

For dif > 0: dp[l][r][dif] = min of

- $dp[l + 1][r][dif - w[l]] - w[l]$
- $dp[l][r - 1][dif - w[r]] - w[r]$

In this dynamic programming, $|\text{dif}| \leqslant \max_i(w_i)$, so there are a total of $O(n^2 \cdot \max_i(w_i))$ states and two transitions from each.

To solve subgroups with the restriction $w_{i+1} \leqslant 2 \cdot w_i$, we will prove the following fact: in such constraints, for any achievable state (l, r, dif), it holds that $|\text{dif}| \leqslant w_{r+1}$ (exception: if $r = n - 1$, then $|\text{dif}| \leqslant w_{n-2}$).

To prove this, note that in any transition, dif changes towards zero, so its absolute value either decreases or becomes no more than w_i (where w_i is the last taken piece). Thus, in transitions [l, r] $\to$ [l+1, r], the relationship is preserved in any case, and in transitions [l, r] $\to$ [l, r-1], it is preserved because $w_{r+1} \leqslant 2 \cdot w_r$.

Therefore, for each l, there are only $O(\sum_{i=0}^{n-1} w_i) = O(W)$ achievable pairs r, dif, so there are a total of $O(n \cdot W)$ states and two transitions from each.

Now, to solve the full problem, we will change the transitions in the dynamic programming to preserve the relationship $|\text{dif}| \leqslant w_{r+1}$.

To do this, note that if one person took several pieces before passing the turn to the opponent, they could always do so by first taking the smallest pieces and then the largest ones. Therefore, we will keep

the transitions [l, r] → [l+1, r] as they preserve the invariant, and instead of transitions [l, r] → [l, r-1], we will add transitions [l, r, dif] → [l, r', dif'], meaning that the person takes the largest elements so that the turn passes to the opponent, or the game ends.

These transitions preserve the invariant $|\text{dif}| \leqslant w_{r+1}$, so in this dynamic programming, there are $O(n \cdot W)$ states and still two transitions from each.

Also note that the values of r' depend only on r and dif, but not on l, so they can be computed in $O(W \cdot \log(n))$.

The resulting solution works in $O(n \cdot W)$, which is sufficient to pass all tests.

Разбор задачи «Parallel Universes»

Let's call the graph that needs to be made connected the first one, and the graph whose connectivity cannot be violated the second one.

Suppose the second graph contains an edge (v, u) such that the vertices v and u are in the same connected component of the first graph. Then consider any vertex a from a different component of the first graph. Notice that one of the operations (v, a) and (u, a) does not violate the connectivity of the second graph. Thus, it is possible to perform one operation to merge the component of vertex a with the component of vertices v and u, which solves the problem.

Now, notice that if there exists a pair of vertices v and u such that there is no edge (v, u) in both graphs, then it is possible to perform an operation with this pair of vertices, after which the problem reduces to the previous case.

If such an edge is not found, then all connected components of the first graph are cliques, and the second graph contains all edges that are not in the first graph.

Now, let's select any connected component of the first graph, the size of which is at least 2 (if there is no such component, then perform an operation with any pair of vertices, after which such a connected component will appear). In this connected component, select arbitrary vertices a and b. In addition, from all other connected components of the first graph, select one vertex. Let's call them $v_1,\ v_2,\ \ldots,\ v_k$.

Notice that if the first graph contains at least three connected components or each connected component has a size of at least 2, then the following sequence of operations will work: $(a, v_1),\ (b, v_2),\ (b, v_3),\ \ldots,\ (b, v_k)$.

If this is not the case, then the first graph contains an isolated vertex c, while all other vertices form a clique. In this case, the second graph is a star. In this case, if $n = 3$, there is no answer, otherwise, any pair of vertices a and b different from c can be chosen, and two operations (a, b) and (a, c) can be performed.

This allows solving the problem in $\mathcal{O}(n + m_1 + m_2)$ or $\mathcal{O}\left((n + m_1 + m_2)\log(m_1 + m_2)\right)$ depending on the implementation.

Разбор задачи «Three Arrays»

Subgroup 1. $n \leqslant 15$

We will iterate through all possible operations, and choose the appropriate answer among them. The time complexity is $\mathcal{O}(2^n \cdot n)$.

Subgroups 3, 5. $D_i = 0$

The values of A_i, B_i can only change during the minimum operation.

We will iterate through the final value $A_n = c$. This can be any of the values $L_i \leqslant a_0$ or a_0 itself.

The value of B_n is determined as the minimum of b_0 and the values R_i such that $L_i < c$.

We obtain the solution in $\mathcal{O}(n^2)$ or $\mathcal{O}(n \log n)$ depending on the method of calculating the value of B_n.

Full solution.

Notice that the final values have the form $A_n = L_p + D_{p+1} + D_{p+2} + \ldots + D_n$ and $B_n = R_q + D_{q+1} + D_{q+2} + \ldots + D_n$ for some $0 \leqslant p, q \leqslant n$.

Replace L_i with $L_i + \sum_{j=i+1}^{n} D_j$, R_i with $R_i + \sum_{j=i+1}^{n} D_j$.

We reduce the problem to the case $D_i = 0$.

We obtain solutions of different complexities from $\mathcal{O}(n^3)$ to $\mathcal{O}(n \log n)$.

Разбор задачи «Burenka and Pether»

Subgroup 1.

$n, q \leqslant 100$. In this subgroup, there are small constraints on n, q, so it is possible to construct a graph of jumps and independently find the shortest path for each query using breadth-first search. $O(n^2 q)$

Structure of our graph

Note. Let's understand what our graph looks like, namely: let's show that from vertex u edges are drawn to all vertices t of some segment $[u, r_u]$, such that $a_t > a_u$.

Proof. To do this, let's remove the restriction $a_i < a_j$ for the jump from i to j (now the second end of the jump can be any, the main thing is that an interesting sequence of intermediate vertices is found). If with such jumps we can jump to some subsegment of vertices, then with the original, the only additional restriction will be $a_i < a_j$, which is what we need. Let there be a jump $u \to v$. Let's show that then there is a jump $u \to v - 1$. We find the first $t > v$. We show that from v we can jump to $t - 1$. In vertex v we get through some sequence of intermediate vertices. Let the last intermediate vertex be t. Then $v - t \leqslant d$, so either $t = v - 1$, or the sequence can be used to come to vertex $v - 1$.

Finding segments.

Now let's learn to find r_i for all vertices. We will iterate over a_i in descending order and maintain the subsegments of "activated"elements in the DSU. When a new element is "activated it may need to be combined with the components of the neighbors in the DSU. We will also maintain a set of ends of "long"subsegments of length $\geqslant d$. To find our right boundary based on this information, we need to find the end of the "long"subsegment closest to the right of $i + d$ (let it be e), and then set $r_i := \max(i + 1, e - sz_e + 1) + d$, where sz_e is the size of the subsegment with the end at e. This recalculation is correct, because all not yet activated elements are less than a_i, so we can jump over them without restrictions, and the only obstacle on our way will be the "long"subsegment of elements $\geqslant a_i$.

Subgroup 2.

$n \leqslant 1000$. Fix the second vertex of the query. We will iterate over the vertices in order of *decreasing* identifier. To update the distance from the current iteration vertex u to the fixed vertex v, we just need to find the minimum on some segment. This can be easily done using a segment tree. $O(n^2 \log n + q)$.

Subgroup 3.

$a_n = n, v_i = n$. It's time to make one of the main observations in the problem.

Claim. Let (u, v) be a query for which we want to find the answer, and $a_v = n$. Then from u we need to jump to the largest number on the segment $[u, r_u]$.

Proof. Let's assume that the maximum on the subsegment is x, and instead of it we jumped to y. After that, we made some number of jumps. Let's consider the first moment when we reached $q \geqslant x$: $y \to t_1 \to \ldots \to q$. If $q = x$, then indeed it would be more profitable to just jump to q. Otherwise, if $pos_y < pos_x$, then in q we can get there using some suffix of the path $y \to t_1 \to \ldots \to q$ (pos_x is the position of the vertex with value x in the array). If $pos_y > pos_x$, then there is a path $u \to g_1 \ldots \to g_k \to y$. We can take a suffix of this path, and then combine it with the path $y \to t_1 \to \ldots \to q$ to get the path $x \to g_i \to \ldots \to g_j \to y \to t_1 \to \ldots \to q$, that is, there is a jump from x to q. This means that it is at least as profitable to jump to q as to y.

This interesting fact allows us to build a tree with all paths being the shortest from u to v. The jumps can be easily found using a segment tree. The queries can be answered using binary lifts or dfs. $O((n + q) \log n)$

Subgroups 4, 5.

It is noted that all the above arguments are true even for $a_v \neq n$ with the only difference that in this case there are no vertices with identifiers greater than a_v — we definitely cannot reach them from a_v. This is formulated as a statement, but in fact we have already proved it.

Claim. Let (u, v) be a query. Then from u we need to jump to the largest number x on the segment $[u, r_u]$ such that $x \leqslant a_v$.

Therefore, in this subgroup, it is sufficient to build a single tree, binary lifts on it, and easily answer all queries $O((n + q) \log n)$.

Subgroups 6, 7.

We will continue to develop our idea: we know that there are not many jumps, so we can simply iterate over the queries in ascending order of v_i and maintain a segment tree on the maximum. Then,

when answering a query, we just need to find the maximum on some subsegment no more than ans times. $O(n \log n + q \cdot ans \cdot \log n)$.

Subgroup 8.

This subgroup is perhaps the first hint of the existence of a solution to the problem in $O((n+q)\texttt{polylog}(n,q))$. All this time we tried to jump to the largest number to reduce the distance. However, in this subgroup, the distance should not be the shortest.

Claim. Let (u, v) be a query and the distance is not important. Then it is possible to jump from u to the smallest number greater than a_u on the segment $[u, r_u]$ or to vertex v.

Proof. Let's assume that from u it is not possible to reach v in one jump, and there was a path $u \to t_1 \to \ldots \to v$. We will show that after jumping from u to the smallest number greater than a_u on $[u, r_u]$ (let it be vertex p), it is still possible to find such a path. We find the first $t > p$. We show that from p it is possible to reach $t-1$. In t we got from some sequence of intermediate vertices. Let the last intermediate vertex be t. Then $v - t \leqslant d$, so either $t = v - 1$, or the sequence can be used to come to vertex $v - 1$.

After changing all queries intersecting the cut in this way, we reduce the problem to two independent ones on the halves. It is not difficult to implement the processing of such queries in $O(q \log n)$ at each recursion level. The final asymptotic complexity of the entire solution will be $O((n+q)\log^2 n)$ due to the $\log n$ layers of recursion from "divide and conquer".

Subgroup 9.

Of all the previous subgroups, it becomes clear that the values of the numbers play a more important role than their indices. Let's consider a_i in descending order of values. We will divide them into blocks of size $\sqrt{n}$. To answer queries of the form (u, i), we need to jump in the tree using binary lifts, and each such jump is optimal if it leads to a number $\leqslant a_i$ as proven earlier. Therefore, when answering a query, we need to use binary lifts no more than $\sqrt{n}$ times (we will jump to the leftmost element from which there is an edge in the tree to a number greater than a_i, and then we will jump exactly 1 time using the segment tree. We need to do this no more than $\sqrt{n}$ times, because the number of different "bad"elements to which we can jump in binary lifts is no more than the number of elements in our block

Разбор задачи «Almost Certainly»

Subgroup 1.

For each prefix, let's iterate through pairs of numbers that will differ. After that, we will remove them from the arrays and count how many operations are needed to make the arrays equal. The number of operations is equal to the sum of elements in the first array minus the sum of elements in the second array. And they can only be equalized if for each i it holds that $a_i \geqslant b_i$. This solution works in $O(n^4)$.

Subgroup 2.

Let's improve the solution of the previous subgroup. We want to remove one element from both arrays so that the difference between them is as large as possible. Let's sort both prefixes. Notice that if we can remove a_i and b_j, then we can also remove a_{i-1} and b_j, as well as a_i and b_{j+1}. Then we don't need to iterate through $O(n^2)$ pairs for removal, but we can iterate through the elements of the first array, and find the element of the second array by moving a pointer. This solution works in $O(n^3)$.

Subgroup 3.

Let's further improve the solution of the previous subgroup. We need to check the condition $a_i \geqslant b_i$ faster. To do this, we need to notice that the b_i element is compared either with a_i or with a_{i+1}. Moreover, both prefixes will be divided into $O(1)$ segments, in which comparisons of one of two types need to be made. Let's precalculate with prefix sums whether the comparisons of each type are satisfied, after which the correctness check can be performed in $O(1)$. This solution works in $O(n^2 \log n)$.

Ideas for the complete solution.

Let's look at the two arrays as a set of segments $[b_i, a_i]$. Notice that it is never beneficial for the final answer to have $a_i < b_i$. Then let's see what the final answer will look like. Remove all indices for which $a_i = b_i$. Sort the remaining numbers in ascending order of a_i, then $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n$ and $b_1 < a_1, b_2 < a_2, \ldots b_n < a_n$. It is easy to understand that we need to remove a_n and b_1.

What conditions should the segments satisfy so that after removing a_n and b_1 everything is valid? $b_2 \leqslant a_1, b_3 \leqslant a_2, \ldots, b_n \leqslant a_{n-1}$. If we talk about this in terms of segments, this means that they all form

a connected component. Then the answer is the sum of the lengths of the segments minus the length in coordinates of the longest component.

Subgroup 4-5.

Using this, it can be understood that the answer in subgroup 4 is equal to the sum of the lengths of the segments minus the length of the longest segment. For the fifth group, it is necessary to maintain the current component, and when possible, expand it and remember the longest component before that.

Complete solution.

In the implementation of the complete solution, let's maintain the current segment components. When moving to a new prefix, we need to be able to merge these components if they intersect with the new segment. All this can be easily maintained in std::set. The time complexity of the solution is $O(n \log n)$.