European Junior Olympiad in Informatics 2022 Editorial

Anton Trygub, Roman Bilyi, Matvii Aslandukov, Ihor Barenblat

Special thanks to all the testers:

Kostiantyn Savchuk, Andrii Stolitnii, Kostiantyn Lutsenko, Vladislav Zavodnik, Tymofii Reizin, Alex Danilyuk, Yahor Dubovik, Ashley Khoo

Special thanks to Anton Tsypko for his colossal work put into the organization of EJOI

Problem A. Adjacent Pairs

Author: Jeroen Op de Beek
Developer: Jeroen Op de Beek
Editorialist: Jeroen Op de Beek

Subtask 1. Let's call an array **good** if all adjacent pairs of elements are different. At the beginning the array is good, and after each operation the array will stay good. So the ending array must have the form: $[c, d, c, d, \dots], c \neq d$. This means that there are $O(n^2)$ possible ending states. (For any ending state with c or d not in $\{1, 2, \dots, n\}$, it doesn't matter what the value is exactly, so there are only $O(n)$ interesting options for those cases).

For this subtask we can try all pairs.

So we fix some c and d. Let's call the final array $\mathbf{b} := [c, d, c, d, \dots]$.

Then we greedily make moves, trying to change elements in $\mathbf{a}$ to $\mathbf{b}$. It could happen that $\mathbf{a} \neq \mathbf{b}$ but no greedy move is possible:

$\mathbf{a} = [1, 2, 3, 1, 2]$

$\mathbf{b} = [1, 2, 1, 2, 1]$

The last three elements of the array cannot be changed greedily, because this would cause adjacent equal valued elements.

Such a blockage is always caused by some subarray that is of the form $[d, c, d, c, ...]$ which is the desired pattern, but with the wrong parities. Let's call subarrays which have values d and c at indices of the wrong parity and that can't be extended further **bad** subarrays. It turns out that changing the second element of any bad subarray to 10^9 is optimal (for the proof, see subtask 2).

So a simple algorithm will try to find a greedy move, if there's no greedy move, it finds any starting point of a bad subarray and changes the second element.

This can be implemented to run in $O(n)$ per iteration, and there are at most $O(n)$ iterations per pair of c and d, so this will run in $O(n^4)$, with a small constant.

Subtask 2. Instead of simulating the process of converting the array we can calculate the number of moves needed more directly. Firstly, for each c and d we find all bad subarrays, with a single for loop.

For a bad subarray of size k it's optimal to first do $\lfloor k/2 \rfloor$ extra moves, where we place the value 10^9 on positions $2, 4, 6, \dots$ of the subarray. After this, the whole subarray can be finished greedily. To show that this is the best we can do, let's consider all possible moves we can make on this subarray.

When a value in the subarray is replaced, we are always left with two bad subarrays of sizes l and r, such that $k = l + r + 1$. Notice that for a bad subarray of size 1, there's no problem. Because it is maximal, the elements around it cannot be equal to c and d, but this means the only element of the subarray can be greedily changed. By induction on the size of the subarray, and basecases 0 and 1 the lowerbound of $\lfloor k/2 \rfloor$ extra moves can be proven. The formula for the number of moves needed will be:

$$\#\text{moves} = n - (\#\text{ of i, such that } a_i = b_i) + \sum_{\text{bad subarrays}} \lfloor k_i/2 \rfloor$$

Now the time per pair is reduced to $O(n)$, and the total complexity is $O(n^3)$

Subtask 3. It's intuitively clear that values that appear more frequent are better candidates for c or d. We can show that instead of $O(n^2)$ pairs, we can only examine the $O(n)$ best pairs. For all pairs c and d, calculate

$$\#\text{greedy moves} = n - (\#\text{ of i, such that } a_i = b_i)$$

and sort them increasingly on this quantity. To calculate this value in $O(1)$ per pair, we can count the number of occurrences of x $(1 \leq x \leq n)$ at odd and even indices as a precomputation step.

Notice that the total number of bad subarrays over all pairs c and d is $O(n)$ (here we ignore bad subarrays of length 1, because they don't change the answer). This is because two bad subarrays cannot overlap by more than 1 element. So there are only $O(n)$ pairs of c and d that cannot immediately be finished by greedy moves. So by the pigeonhole principle, if we examine more pairs than this, we will always have at least one pair with no bad subarrays. After this point no other pairs in the increasing order of greedy moves can give a better answer, so we can break the loop early. With this observation we only need to check $O(n)$ pairs, and the time complexity becomes $O(n^2)$ or $O(n^2 \log(n))$, depending on the implementation.

Subtask 4. Instead of finding bad subarrays for each pair c and d independently, all bad subarrays can be found at once, with a single pass through array **a**, with some simple logic. We can use a map to store $(c, d) \rightarrow$ extra moves needed, and add $\lfloor$bad subarray size$/2\rfloor$ to the map appropriately.

Now for finding the best (c, d) pair, iterate through all possible c, the final value of all odd indices of the array.

For the value of d, we can loop through all d in decreasing order of $\text{occ}_{even}[d]$ (the number of occurrences of d in even positions). By reusing the same observation as subtask 3, we can break this loop over d as soon as (c, d) can be solved with only greedy moves (i.e. it isn't in the map). The total number of iterations of the inner for loop is bounded by (size of the map) $+ n = O(n)$.

This gives an $O(n \log(n))$ algorithm, because of the sorting of d and the map. By changing out the map to a hash map, and changing the sorting algorithm to counting sort, $O(n)$ is also possible. For deterministic linear time, with clever use of a global array that is reused between the iterations of c, the hashmap is no longer needed. These optimizations were not needed to obtain the full score.

Problem B. Where Is the Root?

Author: Ray Bai
Developer: Roman Bilyi
Editorialist: Anton Trygub

Subtask 1. You can just ask about each possible subset of nodes. As we will show later, for each two root candidates there is a query for which their answers would be different, so we can determine the root uniquely. It's enough to ask $2^n - 1$ queries.

Subtask 2. Let's ask a query for each pair of nodes. Clearly, if root is v, then the answer to each query (r, v) for $v \neq r$ is `YES`. If there is only one node that gives `YES` for each query, it must be the root. Suppose that some node $r_1 \neq r$ also gives `YES` for each query in which it's included. If r is not a lead, then there is a node $v \neq r_1, v \neq r$ such that $LCA(r_1, v) = r$, contradiction. So, r must be a leaf.

Suppose that there is another leaf r_1 that gives `YES` for all queries. Then consider any leaf v different from r, r_1 (in a graph in which at least one degree is at least 3, there are at least 3 leaves). Then, $LCA(r_1, v)$ can't be equal to r_1 and v, so the answer would be `NO`. Contradiction.

So, if there is only one such node, it's the root, otherwise it's the unique leaf among the nodes which gave `YES` for each query in which they were involved. It's enough to ask $\frac{n(n-1)}{2}$ queries.

Subtask 3. There are many different approaches that achieve various bounds. We will describe the solutions which work for $n = 500$ in 10 queries, and in 9 queries.

10 queries: Suppose that r is the root. First, let's ask a query about all leaves. Their LCA has to be r (if r is a leaf, it's clear; otherwise, there is a leaf in each of the subtrees of r). So, we can determine if r is a leaf with this query.

Now if r is **not a leaf**, let's ask queries for sets, which contain all leaves. We will get `YES` iff we have included r into the set. So, we can do binary search on the non-leaf nodes, discarding roughly half at each time. This way, we will need at most 9 queries.

Suppose now that r **is a leaf**. If we ask a question about some subset of the leaves of size ≥ 2, the answer will be `YES` if and only if r is in the chosen set. So, we can once again do binary search on the leaves, until we get at most 2 candidates r_1, r_2. At that point, we can ask a query for nodes r_2, r_3, where r_3 is any other leaf, to determine if r_2 is the root. This way, we will also need at most 9 queries.

Adding the initial query about all leaves, we are done in 10 queries.

9 queries: Our algorithm loses one query at the first step: it might be the case that the number of remaining candidates is $n-$ some small number (when the root is a leaf, and almost all nodes are leaves, for example). Let's try to solve this issue.

Let's arrange our nodes in the following fashion: first all leaves, then all non-leaves. What happens if we ask a query about the some prefix of this order of length at least 2? Let's consider two cases.

- Our prefix contains all the leaves, and, maybe, some other nodes. Then their LCA is r, and we will get `YES` if and only if r is in that prefix.
- Our prefix is some subset of the leaves. Then, if r is among them, we will get `YES`, otherwise their LCA would be some node different from any of them, so we will get `NO`. Once again, we will get `YES` if and only if r is in that prefix.

So, we can do binary search on this order of nodes! The only exception is when we end up with a prefix of length 2. Then, as in previous subtask, we should ask one of these leaves with one of other leaves.

This solves the problem in 9 queries.

Problem C. Bounded Spanning Tree

Authors: Ihor Barenblat, Matvii Aslandukov
Developer: Ihor Barenblat
Editorialist: Ihor Barenblat

Subtask 1. For each edge, its weight is known. There is need to check that edges can have different weights from the range $[1, m]$ and that edges with indices $1, 2, \ldots, n-1$ form a minimum spanning tree of the given graph. The last can be done using any suitable algorithm. Time complexity: $O(m \cdot log(m))$ or $O(m)$.

Subtask 2. Just try all possible assignments for the weights of the edges and for each one check compliance with the rules described in the statement. Time complexity: $O(m! \cdot m)$.

Subtask 3. Use dynamic programming on subsets of the edges. Lets denote $dp[S]$ as a boolean variable indicating that there is assignment of the weights $1, 2, \ldots, |S|$ to the edges from S, such that there is no rules were broken. Here by "breaking mst rule" we mean assigning to a *non-spanning-tree* edge such value that this edge will be taken into minimum spanning tree when considering it in the process of execution of Kruskal's algorithm. Time complexity: $O(2^m \cdot m)$.

Subtask 4. The same idea as in the `Subtask 5`, but with worse algorithm complexity. Also possible to solve using "matching in bipartite graph" approach. Time complexity: $O(m^3)$ or $O(m^2)$.

Subtask 5. We see that for this subtask there is no "mst rule". So we just need to find an assignment of the edge weights using integers from the range $[1, m]$. It is possible to do this using greedy approach: set 1 to the edge that have $l_i = 1$ with the minimum possible r_i. This is true, since any valid assignment can be changed without losing validity to be compatible with the mentioned greedy assignment. So the overall greedy algorithm is the following: for each i from 1 to m set i as edge weight for the edge with $l_j \leq i$ with the minimum possible r_j, between all edges with unset weight. Time complexity: $O(m \cdot log(m))$.

Subtask 6. Lets call the *non-spanning-tree* edge as "special". After the assigning a value for the special edge we need to be sure that there exist possibility of assigning for *spanning-tree* edges weights from the range $[1, m]$ (except the already assigned one). Lets find all suitable possible values that satisfy this condition, choose the greatest one from the range of the special edge and update r_j for the edges on the cycle to be sure that they are smaller that the value we assigned to the special edge. For finding all suitable possible values that satisfy mentioned condition we can use Hall's marriage theorem on the bipartite graph (first part consists of vertices corresponding to the edges and second part consists of vertices corresponding to possible edge weights): value k is suitable if and only if there is no L, R such that $L \leq k \leq R$ and $R - L + 1 \leq (\#\ i$ such that $1 \leq i \leq (n-1)$ and $L \leq l_i \leq r_i \leq R)$. Time complexity: $O(m \cdot log(m))$.

Subtask 7. Lets call pair of edges (e_{tree}, e_{non_tree}) `interesting` (here e_{tree} is a *spanning-tree* edge and e_{non_tree} is a *non-spanning-tree* edge) if e_{tree} lies on the simple path in the expected minimum spanning tree between vertices that connects e_{non_tree}. It is easy to see that for each interesting pair of edges we can do the following update: $e_{tree}.r = min(e_{tree}.r, e_{non\_tree}.r - 1)$, $e_{non\_tree}.l = max(e_{non\_tree}.l, e_{tree}.l + 1)$. With such modifications, any assignment found by the greedy algorithm from the `Subtask 5` will satisfy "mst rule". Time complexity: $O(m \cdot n)$.

Subtask 8. The same idea as in the `Subtask 7`, but with better algorithm complexity. You can use segment tree to do mentioned updates. Time complexity: $O(m \cdot log(n))$.

Subtask 9. The same idea as in the `Subtask 7`, but with better algorithm complexity. You can use heavy-light decomposition to do mentioned updates. Time complexity: $O(m \cdot log^2(n))$.

Subtask 10. The same idea as in the `Subtask 7`, but with better algorithm complexity. You can use binary lifting approach or non-trivial heavy-light decomposition approach to do mentioned updates. Time complexity: $O(m \cdot log(n))$.

Problem D. Game With Numbers

Author: Matvii Aslandukov
Developer: Matvii Aslandukov
Editorialist: Matvii Aslandukov

Subtask 1. In the first subtask $m = 1$ so you can find two sums s_1 and s_2: the sum of all elements that are divisible by b_1 and the sum of all elements that are not divisible by b_1. Then the answer is $\min(s_1, s_2)$ because the first player wants to minimize the sum of the remaining elements. Time complexity: $O(n)$.

Lemma 1. Before describing the solution for each individual subtask let's prove the following lemma: if both players can remove all the elements from the array a using only their own rounds, then the answer is 0. Indeed, in such a case both players can make the sum of the remaining elements in the array a equal to 0 regardless of the actions of the second player. The first player wants to minimize the sum, therefore the answer is not bigger than zero. At the same time, the second player wants to maximize the sum, therefore the answer is not less than zero. Thus the only possible answer is 0.

Subtask 2. In the second subtask two own rounds are enough for the first player to remove all the elements: he can remove all the numbers that are divisible by b_1 and are not divisible by b_3. At the same time, only one own round is enough for the second player to remove all the elements: he can make an opposite operation to the first player due to the condition $b_1 = b_2$. Thus by the lemma 1 the answer is 0 for $m > 2$. For $m = 2$ the answer is $\max(0, \min(s_1, s_2))$. Time complexity: $O(n)$.

Subtask 3. In the third subtask two own rounds are enough for each player to remove all the elements: the first player can remove all the numbers that are divisible by b_1 and are not divisible by b_3, and the second player can do the same using b_2 and b_4. Thus by the lemma 1 the answer is 0 for $m > 3$. For $m \leq 3$ you can either solve the problem recursively by considering each possible game scenario or consider up to 8 cases manually. Time complexity: $O(n)$.

Subtask 4. In the fourth subtask no additional observations are required. The entire game can be simulated recursively: you can represent the state of the game as a pair $(operations, pos)$, where $operations$ is an array with chosen operations and pos is a current round. Then at each state of the recursion you can try to make all two possible operations and select the best one. The total number of states is $O(2^m)$ and for each final state with $pos = m$ you can calculate the sum of the remaining elements in $O(nm)$. Therefore such a solution works in $O(2^m \cdot nm)$.

Subtask 5. In the fifth subtask you can modify recursion in the following way: instead of representing the state of the game as a pair $(operations, pos)$ you can represent it as a pair (a, pos), where a is an array with remaining elements and pos is a current round. Then at each state of the recursion you can try to make all two possible operations and select the best one. The total number of states is $O(2^m)$ but the total size of all arrays a across all states is only $O(nm)$ due to the fact that each initial element of the array a is contained in exactly m states. Therefore such a solution works in $O(2^m + nm)$. See the following code on `Python` for a better understanding.

```
def solve(a, pos):
    if pos == len(b):
        return sum(a)
    na = [[], []]
    for x in a:
        na[(x % b[pos]) == 0].append(x)
    return [min, max][pos % 2](solve(na[0], pos + 1), solve(na[1], pos + 1))

n, m = map(int, input().split())
a = list(map(int, input().split()))
b = list(map(int, input().split()))
print(solve(a, 0))
```

Subtask 6. In the sixth subtask you can notice that the majority of $O(2^m)$ states inside the recursion have an empty array a that allows to immediately return 0 as a result:

```
def solve(a, pos):
    if len(a) == 0:
        return 0
    ...
```

Such optimization gives time complexity $O(nm)$ which is enough for the sixth subtask.

Subtask 7. In the seventh subtask $a_i \geq 1$ that means that the final sum of the remaining elements is always non-negative. It simplifies the proof of the lemma 1 because now the goal of the first player is to remove all the elements from the array a. Also it can be proven that the answer is equal to 0 when $m \geq 19$ under the constraint $a_i \leq 10^9$. Such proof is left as an exercise for the reader (see bonus section). Such a fact means that the only difference from the fifth subtask is that we can just output 0 when $m > 20$.

Subtask 8. In the eighth subtask the second player can always remove all the elements by making the opposite second operation. It means that the answer is always non-negative. When the answer is positive the only strategy for the second player is to repeat the corresponding operations of the first player. It allows us to speed up the solution from the fifth subtask to $O(2^{m/2} + nm)$ which is ok for the $m \leq 40$. See the solution for the next subtask to understand what to do in case $m > 40$.

Subtask 9. For the full solution you can notice that $O(\log n)$ own rounds are enough for each player to remove all the elements. Indeed, at each round one of the two possible operations removes at least half of all remaining elements, therefore $\lceil \log_2 n \rceil$ rounds are always enough to remove all the elements. It means that the only difference from the sixth subtask is that we can just output 0 when $m > 100$ by the lemma 1.

Bonus. What is the maximum value of m where the answer is not equal to zero?

Problem E. Longest Unfriendly Subsequence

Author: Anton Trygub
Developer: Anton Trygub
Editorialist: Anton Trygub

Subtask 1. Any subsequence of such a sequence is nondecreasing. Unfriendly sequences, however, do not allow equality of two adjacent elements, so any unfriendly subsequence of this sequence has to be **strictly** increasing. This means, that each value will appear in such a subsequence at most once.

But then we can delete duplicates and take a subsequence containing precisely one occurrence of each element that appears in a. As all elements of this subsequence are distinct, it's unfriendly. So, the answer for this subtask is just the number of distinct elements in a. We can find it in $O(n)$.

Subtask 2. We can just consider all possible $2^n - 1$ nonempty subsequences of a, check each for unfriendliness in $O(n)$ time, and output the length of the longest unfriendly. This takes $O(2^n n)$ time. As $t \leq 10^5$, this easily fits in TL for $n \leq 8$.

Subtask 3. Clearly, for $n = 1$ answer is 1, and for $n \geq 2$ it's ≥ 2 (as any subsequence of length exactly 2 is unfriendly).

Let's use dynamic programming. Let $dp[i][j]$ for $1 \leq i < j \leq n$ denote the length of the longest unfriendly subsequence of a, in which the last element is a_j, and the second last is a_i. If $a_i = a_j$, $dp[i][j] = 0$. Otherwise, $dp[i][j] = max(2, max_{1 \leq k < i} dp[k][i] + 1))$ over k for which $a_k \neq a_i$ and $a_k \neq a_j$. We can calculate this dp table in $O(n^3)$ for a single test case, which is fast enough.

Subtask 4. Let's look at any unfriendly sequence $b_1, b_2, \ldots, b_m$ such that for all i $1 \leq b_i \leq 3$. Each 3 consecutive elements of b are distinct, therefore b_i, b_{i+1}, b_{i+2} are some permutation of $1, 2, 3$ for $1 \leq i \leq n-2$. Then, however, $b_{i+1}, b_{i+2}, b_{i+3}$ also are such a permutation. As b_i and b_{i+3} both differ from two distinct values (b_{i+1}, b_{i+2}), they must be equal. So, $b_i = b_{i+3}$ for each i; b has to be periodic with period 3.

Then, just try each possible start of the subsequence b p_1, p_2, p_3 — every permutation of $(1, 2, 3)$. For each of them, take elements $p_1, p_2, p_3, p_1, p_2, \ldots$ as soon as you see them. Output the largest answer over these 6 options.

Subtask 5. Let's go through our sequence a from left to right and keep the following dynamic programming table: let $dp[x][y]$ denote the length of the longest unfriendly subsequence of a up to this moment, whose last element is y, and second last element is x. Initially, we can set each value in this table to $-INF$ (where $INF = 10^9$, for example). Let's also keep track of what elements have already appeared in our sequence.

It turns out that it's easy to update this table: when we are at position i, we just need to update the values of $dp[x][a_i]$ for each $x \neq a_i$. If x hasn't appeared before, there is no subsequence ending with (x, a_i), otherwise, do $dp[x][a_i] = max(dp[x][a_i], 2)$. Then, we need to do $dp[x][a_i] = max(dp[x][a_i], dp[y][x]+1)$ over all $y \neq x, a_i$. Updating this table after seeing the next element takes $O(MAX^2)$, with overall complexity $O(MAX^2 n)$ per test case, which fits easily.

Subtask 6. Let's modify our algorithm from **Subtask 5** a little. Clearly, we can assume that elements are in the range $[1, n]$ (just map k-th smallest value to k, we don't care about the exact values of elements, we only care about which elements are equal to which). Now, again, let's keep $dp[x][y]$ for $x \neq y$: the length of the longest unfriendly subsequence of a up to this moment which ends with (x, y). The difficulty lies in updating $dp[x][a_i] = max(dp[x][a_i], dp[y][x] + 1)$ over all $y \neq x, a_i$: this can take $O(n^3)$, which for $n = 10000$ has no chance of passing.

But let's note that we don't actually need **all** the values $dp[y][x]$ to update this table. We need the largest value among the ones for which $y \neq a_i$. Then, for each y let's keep two values $x_1 \neq y, x_2 \neq y$, such that the values $dp[x_1][y], dp[x_2][y]$ are the largest among all $dp[x][y]$. Then, we would just have 2 (at most) candidates to check. After we do this for each y, we will recalculate the best choices for the previous element for a_i.

This way, processing new element takes $O(n)$, and the entire algorithm runs in $O(n^2)$ time, which passes easily.

Subtask 7. For this subtask, we will have to analyze the structure of the longest unfriendly subsequence a bit more.

Consider the longest unfriendly subsequence of a. Suppose that it contains a_i. What could be the previous element before a_i, if there is any? Clearly, if it's some value x, it's optimal to take the last occurrence of x before a_i.

What we did in previous subtasks was going through all possible candidates for x. However, as it turns out, we don't need that many. Among all last occurrences of elements before a_i, consider 5 rightmost (if there are at least 5). Suppose that we don't take any of those as our x. Then, I claim, we can extend our unfriendly subsequence by inserting one of these rightmost 5 last occurrences into it.

Indeed, two (or less, if there are less than two) elements to the left of a_i in this subsequence, a_i, and the element to the right, if there is any. They are the only prohibited values for the x (if we want to insert x right before a_i in this subsequence). Then one of those 5 last occurrences would not be prohibited, and the subsequence wouldn't be the longest possible.

So, for each a_i, we know the set of at most 5 possible candidates for the previous element in the longest unfriendly subsequence. Therefore, we can once again use dynamic programming of form $[cand][last]$, indicating the length of the longest possible unfriendly subsequence, ending in (a_{cand}, a_{last}). For each $last$, we have at most 5 cand. So, when processing new $last$, we need to do just $MAGIC^2$ checks (where $MAGIC = 5$).

We can keep this dp in maps, and keep the last occurrence of each element with a simple set. The total complexity is $O(n(5^2 + \log n))$.

Problem F. LCS of Permutations

Author: Anton Trygub
Developer: Anton Trygub
Editorialist: Anton Trygub

Subtask 1. For $a = b = 1, c = n$, there always is a solution. It's enough to take $p = (1, 2, \ldots, n)$ and $q = r = (n, n-1, \ldots, 1)$.

Subtask 2. Let's notice the simple property:

Let a, b be any sequences of length n consisting of integers from 1 to n, and p be any permutation of integers from 1 to n. Then, we have the following property:

- $LCS(a, b) = LCS((p_{a_1}, p_{a_2}, \ldots, p_{a_n}), (p_{b_1}, p_{b_2}, \ldots, p_{b_n}))$

Indeed, we don't care about which element is larger than which. We only care about which element is equal to which.

With that in mind, we might notice that if there is such a triple of permutations p, q, r, then there also is such a triple with $p = (1, 2, \ldots, n)$. This allows us to fit the brute force in time.

Let's try all candidates for permutations q, r (there are $(n!)^2$ such pairs), and for each pair of candidates, candidate pairwise LCSs (we can find the LCS of two permutations in $O(n^2)$, for example). For each triple (a, b, c) that we saw, memorize any triple of permutations (p, q, r) that produced it. This allows us to solve the subtask with preprocessing, taking $O((6!)6^2)$, which passes easily.

Subtask 3. Note that if $LCS(q, r) = n$, we must have $q = r$, so we also must have $a = b$. We are still considering $p = (1, 2, \ldots, n)$. Also, note that $LCS((1, 2, \ldots, n), q)$ is just equal to the length of the longest increasing subsequence of q.

Now, we need to check if there exists a permutation with $LIS(q) = a$ (from now on, by $LIS(q)$ I will denote the length of the longest increasing subsequence of q). It turns out that it exists for each $1 \le a \le n$. Indeed, it's enough to consider $q = (n, n-1, \ldots, a+2, a+1, 1, 2, \ldots, a)$.

Subtask 4. If $LCS(p, q) = 1$, then q must be the reverse of p. In our case, we would have $p = (1, 2, \ldots, n), q = (n, n-1, \ldots, 1)$. Note that $LCS((n, n-1, \ldots, 1), r) = LDS(r)$ for any permutation, where by $LDS(r)$ we denote the length of the longest **decreasing** subsequence of r.

So, we just need to determine if there exists a permutation of length n with $LIS = b$ and $LDS = c$. Here, the Erdős–Szekeres theorem might be useful. It states that in any sequence with length $(r-1)(s-1)$, there is an increasing subsequence of length r, or a decreasing subsequence of length s. We will use it in the following form: $LIS(p)LDS(p) \ge n$ (indeed, if $LIS(p)LDS(p) \le n-1$, then the subsequence of length n would have an increasing subsequence of length $LIS(p)+1$, or a decreasing subsequence of length $LDS(p)+1$.

So, we must have $bc \ge n$. Is this condition sufficient? Sadly, no. Consider $b = c = n$, for example. We would have to have $p = q = r$, but $p \ne q$ (for $n > 1$).

Then, we might notice the second condition: $LCS(p) + LDS(p) \le n + 1$ for any permutation p of length n. Indeed, any increasing subsequence may have at most 1 common element with any decreasing one, so at most one of n elements can contribute to both LCS and LDS, and others can contribute to at most one of them.

So, we get: $b + c \le n + 1, bc \ge n$. Are these conditions sufficient? Turns out, yes. Consider permutation $c, c-1, \ldots, 1, 2c, 2c-1, \ldots, c+1, 3c, \ldots, bc, bc-1, \ldots, bc-c+1$ of integers from 1 to bc. It's easy to see that its LIS is b, and its LDS is c. (For example, the argument for LDS: it clearly has an increasing subsequence $c, 2c, \ldots, bc$, but also is split into b decreasing blocks, none of which can contain more than one element from LIS).

Consider some subsequence of this permutation, which contains elements $c, 2c, \ldots, bc$, as well as elements $c, c-1, \ldots, 1$ (c in written twice, yes) — $n-1$ elements in total. Any such subsequence will have $LIS = b$ and $LDS = c$. As $b+c-1 \leq n \leq bc$, we can take some extra $n-(b+c-1)$ elements of this permutation, and obtain a sequence with $LIS = b$ and $LDS = c$ of length n. Then, we can just "compress"the sequence to the permutation by mapping different values to $1, 2, \ldots, n$ in the relative order.

Subtask 5. In some sense, this subtask was included so that participants would be able to check if their criteria were correct, basically guessing before getting to the actual construction. That's what we are going to do here, leaving the proof and the construction for the **subtask 6**.

Let's try to guess these criteria. We already know some criteria for the case $a = 1$. Maybe we can generalize them somehow?

First, let's try to get something similar to $LIS(p) + LDS(q) \leq n + 1$. Consider common subsequences of (p, r) and (q, r). If their lengths are b, c correspondingly, they must have at least $b + c - n$ elements in common. These common elements would form a common subsequence of p, q, so $b + c - n \leq a$, or $b + c \leq a + n$.

Now, let's try to get something similar to $LIS(p)LDS(p) \geq n$. I claim, that $LCS(p, q)LCS(q, r)LCS(p, r) \geq n$. Proof: suppose that $abc \leq n$. As $p = (1, 2, \ldots, n)$, we get that $LIS(q) = a \implies LDS(q) \geq \frac{n}{a} \geq bc + 1$. Consider some decreasing subsequence of q of length $bc + 1$. Let's look at how the elements of this subsequence are situated in r. No $b+1$ of these elements can form an increasing subsequence in r (as then we would have $LCS(p, r) \geq b+1$. No $c+1$ of these elements can form a decreasing subsequence in r (as then we would have $LCS(q, r) \geq c + 1$. But at least one of these has to hold, by the Erdős–Szekeres theorem! Contradiction.

So, we found two necessary conditions:

- $b + c \leq a + n$
- $abc \geq n$

It turns out that these conditions are actually sufficient. We will prove this in the next section.

Subtask 6. Let's prove that for such a, b, c, n there always exists such a triple of permutations. We will prove this by induction. We already know this is the case if $a = 1$, and it's clear for $n = 1$. Now, suppose that it's true for all tuples (a_1, b_1, c_1, n_1) with $a_1 \leq a, b_1 \leq b, c_1 \leq c, n_1 \leq n$.

Suppose that $b+c \leq a+n$ and $abc \geq n$. If $a > 1$ and $(a-1)(b-1)(c-1) \geq n-1$, then we know that there is such a triple of permutations for tuple $(a-1, b-1, c-1, n-1)$ as well. Consider these permutations p_1, q_1, r_1. Let's append n to each of them. Clearly, the LCS of each pair will increase by precisely 1, so we would get the desired outcome.

Now, let's provide a construction for the case $abc = n$. Let's take:

- $p = (1, 2, \ldots, abc)$
- $q = (abc - a + 1, abc - a + 2, \ldots, abc, abc - 2a + 1, abc - 2a + 2, \ldots, abc - a, \ldots, 1, 2, \ldots, a)$
  (bc increasing blocks, each of length a)
- $r = (ac, ac - 1, \ldots, 1, 2ac, 2ac - 1, \ldots, ac + 1, \ldots, abc, abc - 1, \ldots, abc - ac + 1)$
  (b decreasing blocks, each of length ac)

It's easy to see that $LIS(q) = a$ and $LIS(r) = b$, it only remains to prove that $LCS(q, r) = c$. Sequence $ac, (a-1)c, \ldots, 2c, c$ is a subsequence of both. Suppose that some sequence of length $c+1$ is a subsequence of both. If some two elements $x < y$ of it are from different blocks of length ac (here I mean blocks $[ac, ac-1, \ldots, 1], [2ac, 2ac-1, \ldots, ac+1], \ldots, [abc, abc-1, \ldots, abc-ac+1]$, then in q x goes after y, and in r before, which is impossible. So, they all must be from the same block, say, $[kac, kac-1, \ldots, (k-1)ac+1]$.

Then, this subsequence must be decreasing. However, the elements $[kac, kac-1, \ldots, (k-1)ac+1]$ in q go in order $([kac-a+1, kac-a+2, \ldots, kac], [kac-2a+1, kac-2a+2, \ldots, kac-a], \ldots, [(k-1)ac+1, (k-1)ac+2, \ldots, (k-1)ac+a])$ — that is, c increasing blocks, each of size a. So, it can't have decreasing subsequence of length $c+1$ (as some two elements would have to be in the same block). Contradiction.

How to use this construction for $n = abc$ to get construction for smaller n, the same way as we did in the case $a = 1$? Let's select elements $1, 2, \ldots, a$, $1, ac+1, 2ac+1, \ldots, (b-1)ac+1$, $ac-c+1, ac-2c+1, \ldots, 1$ (1 appears in all 3 of these sequences). We want to take some subset of size n of integers from 1 to abc, containing all the $a+b+c-2$ elements above, and remove from each of p, q, r all elements not contained in this subset (and later "compress"by mapping k-th largest number among selected to k). If we do this, we will get precisely $LCS(p,q) = a, LCS(p,r) = b, LCS(q,r) = c$). We can do this when $a+b+c-2 \leq n$.

If we haven't succeeded, we have the following conditions:

- $(a-1)+(b-1)+(c-1) \geq n$
- $(a-1)(b-1)(c-1) \leq n-2$

When is $xyz \leq x+y+z-2$ possible in general for positive integers $x \leq y \leq z$? If $x \geq 2$, we get $xyz \geq 4z > x+y+z > x+y+z-2$, so we must have $x = 1$, or $yz \leq y+z-1$. For $y \geq 2$, we get $yz \geq 2z \geq y+z > y+z-1$, so we must have $y = 1$. In this case, we get $a = b = 2$, and $c+1 \geq n, c-1 \leq n-2$, implying $c = n-1$. So, the only remaining case is $(2, 2, n-1, n)$ (when $n \geq 3$).

For this case, there is a simple construction:

- $p = (1, 2, \ldots, n)$
- $q = (n, n-1, \ldots, 5, 4, 3, 1, 2)$
- $r = (n, n-1, \ldots, 5, 4, 1, 3, 2)$

We have covered all cases, and provided an algorithm how to construct such permutations, so we are done.