Problem A. Amazing Trick

Problem author: Michael Mirzayanov; problem developer: Borys Minaiev

For permutations of size at most four, we can try all possible options.

For larger inputs, we can choose one permutation randomly, and compute the second permutation based on that. What is the probability that two permutations will not have fixed points? It is known that randomly chosen permutation doesn't have fixed points with probability roughly e^{-1}.

The second permutation is also chosen randomly, and also will not have fixed points with probability e^{-1}. Unfortunately, we can't just multiply two probabilities to get the final probability of finding the correct solution because they are not independent. But at least it gives some intuition why a random solution could work.

We leave formal proof of this algorithm as an exercise to the reader.

Problem B. BinCoin

Problem author: Michael Mirzayanov; problem developer: Nikolay Budin

Let's find the root of the tree. In order to do that, iterate over all employees, suppose we are looking at employee v. Consider a set of employees that are before v in one of the permutations. For the root, there are only two possible distinct sets. For any other node, there are at least four possible such sets. And since we have at least 50 permutations, that were generated independently at random, the probability that non-root employee would have no more than two such distinct sets among given permutations is no more than $\frac{1}{2^{50}}$.

In order to check sets equality fast, we can use hashes. We will always want to calculate the hash of the set of some subsegment of one of the given permutations. We can assign value $2^v \bmod M$ for employee v, and the hash will be just the sum of these values modulo M. These sums can be calculated using prefix sums.

Now, when we found the root, we can split the task into two independent parts for two children of the root. And solve them in the same way.

The probability that we failed at least once is no more than $1 - (1 - \frac{1}{2^{50}})^{500} \approx 4 \cdot 10^{-13}$ which is really small.

This solution can be implemented in $O(n^2)$ time.

Problem C. Cactus Meets Torus

Problem author and developer: Egor Kulikov

First let's look at topological structure of cycles. Let's divide all cycles into classes of equivalence, within each class any cycle can be transformed into any other cycle by continuous transformation (we look at each cycle as a line, ignoring graph vertices).

If there is 3 or more such classes it is easy to see that there is a point that belongs to all cycles, and we can cover that case using 1 class instead.

If there are 2 such classes, it can be proven that there is either one point that belongs to all cycles or one of classes consists of only one cycle, and each cycle from the other class intersect it. We can check if this is the case by, for example, counting for each vertex c_v — number of cycles that contain this vertex. Then if there is a cycle for which sum of c_v is number of cycles in graph + number of vertices on this cycle - 1, then answer is possible.

If there is just one such class, then we can order all cycles in such a way that there is a path between any two adjacent cycles in this order that only have edges that are not part of any cycle (it can consist of a single vertex though). There are many different way to check if such order exists, for example we can replace edges on each cycle with special vertex that is connected to all vertices on said cycle, then we just

need to find a simple path from which all special vertices are at a distance no more then 1. This can be done using depth first search.

If graph does not has structure described above then answer would be negative.

Problem D. Dominoes

Problem author: Vitalii Aksenov; problem developer: Pavel Mavrin

1 Biparitie graph

Let's paint over all the cells of the board in black and white in a checkerboard pattern. Now each domino must cover one white cell and one black cell. Let's build a bipartite graph with black cells as left part and white cells as right part. Now the domino covering corresponds to the perfect matching in this graph.

Now the problem is: given the biparite graph, find the number of pairs of vertices (u, v) such that if we remove them from the graph, there will be a perfect matching M_{uv} for all remaining vertices.

2 Two vertices from the same part

If we remove two vertices from the same part of the graph, it will be obviously impossible to build the perfect matching. If we have n_1 and n_2 vertices in left and right parts, then the number of ways to remove two vertices from the same part is $X = \frac{n_1(n_1-1)+n_2(n_2-1)}{2}$.

If $X \geq 10^6$, we can output 10^6 and finish.

3 Two vertices from the different parts

Now we need to count the number of pairs of vertices from different parts. We only need to do this if $X < 10^6$, and it means that the number of vertices is $n \leq 2000$. And since the degree of each vertex is at most 4, the number of edges is $m \leq 4000$.

If we try to build the perfect matching M_{uv} for each pair of removed cells (u, v) using Kuhn's algorithm, the total complexity will be $O(n^2 \cdot nm)$. That is too slow, so let's try improve it.

Let's build the perfect matching for initial graph M. When we remove vertices u and v, let's remove the affected edges from M, get the matching M'. At most two edges will be removed, so we only need to run one phase of Kuhn's algorithm to build M_{uv} from M' (or find out that it is impossible). Now the complexity is $O(n^2 \cdot m)$ which is better but still too slow. Let's improve it even more.

Imagine if we remove verices v and u. If the edge uv is in M, then M' is a perfect matching. If not, then we have two unsaturated vertices in M', let's say v' and u'. We can build the perfect matching M_{uv} iff there is an alternating path from v' to u'. We can precalculate for each pair of nodes (v', u') if there is an altertating path from v' to u' in $O(nm)$ simply by running DFS from each node v' of the left part. Now we can check for each pair (u, v) if it is good or not in $O(1)$. So the total complexity will be $O(nm + n^2)$.

Problem E. Easy Assembly

Problem author: Elena Kryuchkova; problem developer: Roman Elizarov

This is an easy problem. First, you need to index all blocks with distinct consecutive integers, e.g. from 0 to $m - 1$, where $m = \sum_1^n k_i$ — the total number of blocks. Then, you can compute the minimal number of required split operations s as the number of places in the initial towers where a block with index x is followed by a block with index y and $x + 1 \neq y$. These are the only split operations that are absolutely

necessary. Now, after s split operations, the number of towers will become $n+s$, so in order to combine them into a single tower, you need to perform $c=n+s-1$ combine operations.

Problem F. Football

Problem author: Oleg Hristenko; problem developer: Niyaz Nigmatullin

There are several cases:

1. There is one match. You have nothing to choose, just print the match result, and the number of draws depend on this result.
2. The total number of goals scored a and conceded b is less than the number of games. The maximum number of non-draw matches are $a+b$. Each of these matches is either 1:0 or 0:1. Other matches are 0:0.
3. $a+b \geq n$ and a or b is 0. Just print all matches as x:0 (or 0:x if $a=0$), where $x>0$.
4. Otherwise, make first two matches 1:0 and 0:1. Then depending on a and b fill other matches as 1:0 or 0:1. The remaining goals can be distributed among first two matches, such that the first one is x:0 and the second is 0:y, for some x and y.

Problem G. Game of Questions

Problem author and developer: Gennady Korotkevich

Let t be the bitmask of people still in the game, and let $f(t)$ be the probability that Genie will win the game if it reaches state t. The answer to the problem is $f(2^m-1)$.

Note that we don't need to store any information about what questions have already been asked besides the bitmask t. Any question that differentiates the participants in t has not been asked, and any question that doesn't differentiate them has either been asked or will be asked in the future, but even in the latter case, it will not change the set of people in the game.

Thus, the only thing we are interested in, given the bitmask t, is what the first asked question that differentiates the participants in t is. Any question that does so is equally likely to be first.

We can build an $O(2^m \cdot n)$ dynamic programming solution this way. Let a_i be the bitmask of people who can answer question i. Let $i_1, i_2, \ldots, i_k$ be all questions that differentiate the participants in t. If $k=0$, then $f(t)=1$ or $f(t)=0$ depending on whether Genie belongs to t. Otherwise, $f(t)=\frac{1}{k}\sum\limits_{i\in\{i_1,\ldots,i_k\}} f(t \wedge a_i)$.

It is also possible to get an $O(nm+4^m)$ solution: instead of iterating over the questions, we can iterate over the values of a_i if we count how many questions have each possible value of a_i beforehand.

To arrive at an $O(nm+3^m)$ solution, we can iterate over the values of $t \wedge a_i$ instead. (Such values are always submasks of t, and the total number of submasks of all $t=0,1,\ldots,2^m-1$ is well-known to be equal to 3^m.)

Essentially we need to be able to answer queries of the form "how many bitmasks among $a_1,\ldots,a_n$ satisfy the given mask with wildcards?" (a mask with wildcards consists of `0`, `1`, and `?`, where the question mark means "can be either `0` or `1`"). Let $g(w)$ be the answer for one such mask with wildcards, w.

The easiest way to answer these queries in $O(1)$ is to precompute the answers to all $O(3^m)$ such queries using $O(3^m)$ memory. For a mask without `?`'s, it's easy to find the answer. Otherwise, if there is a `?` in w, let's find any of them: $w=$`...?...`. Then, $g($`...?...`$)=g($`...0...`$)+g($`...1...`$)$.

To find any `?` in every mask w in $O(3^m)$, one can either generate the masks using recursion and save the position of the first `?`, or just find it naively which can also be shown to take $O(3^m)$ time.

Problem H. Hot and Cold

Problem author and developer: Mikhail Dvorkin

First, let's find out the "Closer" phrase. We propose a way to do it "almost always" in 4 questions.

Visit bottom-left and top-right corners in this manner: $(0, 0)$, $(1, 1)$, $(10^6, 10^6)$, and $(10^6 - 1, 10^6 - 1)$. Compare the phrase received after the second and the fourth point. If they are the same, they are both the "Closer" phrase. Otherwise, the treasure is in the bottom-left corner, or in its neighbor points, or in the top-right corner, or in its neighbor points. Just examine these 6 points one by one (actually, two are already examined), and solve the case.

As for the general case, we now have 60 questions left, a known "Closer" phrase, and a rectangle $10^6 \times 10^6$ with treasure candidates. Let's visit a point in the center of the candidates rectangle, then move one unit right, and then move one unit up. Ignore the first phrase. The second phrase tells you which horizontal half of the rectangle contains the treasure, and the third one tells the vertical half. Thus by 3 questions you divide the height and the width of the candidates rectangle by 2 (with rounding up).

This way, after 19 such triples of points, we will have 3 questions left, and a rectangle $[1, 2] \times [1, 2]$ with treasure candidates. If it is smaller than 2×2, just use 1 or 2 questions to examine all its points. Otherwise, ask about the two bottom points. If you haven't won yet, you have 1 question left, and you know which of the two top points contains the treasure.

Also, MIPT Yolki-palki team (Nagibin, Mustafin, Evteev) were able to find the "Closer" phrase in just 3 questions! Visit $(0, 0)$, again $(0, 0)$, and then $(1, 1)$. If you haven't won yet, and the phrases are different, then the later phrase is "Closer". And if they are the same, just visit $(0, 1)$ and $(1, 0)$.

Problem I. Interactive Factorial Guessing

Problem author: Oleg Hristenko; problem developer: Pavel Kunyavskiy

First of all, we need to compute all factorials. Unfortunately, using BigInteger from Kotlin/Java or Python is too slow, as it uses binary representation under the hood, and converting back to decimal is too slow. So one needs to implement their own decimal BigInteger. To avoid big memory usage, one can use base 10^9 instead of base 10.

After this, a lot of solutions are possible. The main idea is to choose a query that will make the biggest set of indistinguishable numbers as small as possible. The naive implementation works reasonably fast to check that all tests can be done within 10 queries but is a bit slow to solve the problem.

Any of the following optimizations (and probably a lot of others) make it fast enough:

- Precompute the first several queries for all answers.
- Check only the last 2000 positions, as it's enough to catch the non-zero part of all numbers
- Precompute the whole branch with zero answers and check positions nearby the first non-zero answer without optimizing anything for them. This works, as the non-zero part of the values is quite random, and the only problem we have is too many zeros.

Problem J. Jumbled Trees

Problem author: Maxim Akhmedov; problem developer: Artem Vasilyev

This problem was about studying the linear space over $\mathbb{Z}_p$, generated by spanning trees of the given graph. It is easier to think about this problem as making all counters zeros, starting from given target values, which is, of course, equivalent to the original problem.

Let's look at a block (a biconnected component along with all affected vertices) with n vertices. Let S be the sum of the values of all operations, and T be the sum of all target values of all edges inside this block.

Since every spanning tree contains exactly $n-1$ edges inside this block, $T = (n-1)S \pmod p$. In the particular case of a bridge, this congruence states that the bridge's target must be equal to S.

The congruences among all blocks either are true for any S, for exactly one S, or none of them, in which case, there is no solution. When there is a unique S, let's first build an arbitrary spanning tree with the value S. After that, the sum of (remaining) targets in every block is zero. It turns out that it is always possible to construct a solution under these conditions.

Consider two spanning trees, T_1 and T_2, that only differ by edges e_1 and e_2: $T_2 = T_1 \setminus e_1 \cup e_2$. Let's see what happens if we invoke two operations $(+x, T_1)$ and $(-x, T_2)$. Every edge's counter except for e_1 and e_2 doesn't change, the counter of e_1 increases by x, and the counter of e_2 decreases by x (notice that the total sum in this block is still zero). We'll call these two operations a *transfer* from e_1 to e_2. Our goal is to make all counters zero using these transfers.

Take any edge with a non-zero counter, we can take any cycle it is on, and set it to zero, transferring its value to another edge. If we succeed in doing that for every edge in the block, except for one, the last one automatically becomes zero (because the sum of all targets in every block is zero). There are many strategies that achieve it; here is one of them that we found the easiest to implement:

Run the DFS on the input graph, calculating the depth of each vertex. Consider all the *base cycles* formed by one back edge along with tree edges connecting the endpoints. Take the *deepest* edge with the non-zero value, and transfer its value to a higher edge (we chose an edge with the smallest value of $\text{depth}(u) + \text{depth}(v)$). This can be implemented as follows: during the DFS, keep track of the highest edge that an edge's value can be transferred to, and update these whenever you encounter a base cycle. Now you can process all edges in order of decreasing $\text{depth}(u) + \text{depth}(v)$ and transfer their value to a higher edge by building two spanning trees.

This solution uses at most 2 spanning trees per edge (and doesn't use a tree for the last edge in the block) and, possibly, one tree in the beginning; at most $2m-1$ spanning trees in total. The time complexity is $O(nm)$, which comes mostly comes out of output size.

Problem K. King's Puzzle

Problem author: Michael Mirzayanov; problem developer: Dmitry Yakutov

The problem is to build connected graph on n vertices without loops or parallel roads such that there are exactly k distinct degrees among all vertices.

If $n = k = 1$ then graph on 1 vertice without edge suits.

Note that if $k = n$ than vertices have all degrees in range $0, \ldots n-1$. If $n > 1$ than we have vertice of degree $n-1$, which is connected to all other vertices, and vertice of degree 0. That's impossible. So if $k = n > 1$, then the answer is "`NO`".

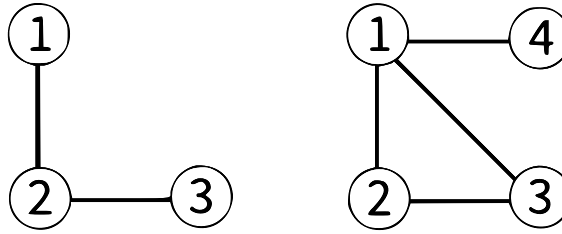
Otherwise the answer is "`YES`". If $k = 1$ then cycle of length n suits. And if $n = 2$ and $k = 1$ then complete graph on n vertices suits.

Let's build a suitable graph $G(n, k)$ for $n \geq 3$ and $k \geq 2$ step by step. We will build G in such way that there is exactly one vertice $A(G)$ of degree $n-1$. And there are some vertices of degree 1, let $B(G)$ be any of them.

If $k + 1 < n$ then let's build $G' = G(n-1, k)$. If we create new vertice u and connect it to $A(G')$ then resulting graph suits.

If $k + 1 = n$ then consider the following cases:

- For $n = 3$ and $n = 4$ graphs below suit.



- For $n \geq 5$ let $G' = G(n-2, k-2)$. Note that G' has vertices of all degrees in range $1, \ldots, n-3$. Let's add two new vertices u and v and connect u to all other vertices including v. In new graph vertices of G' have all degrees in range $2, \ldots, n-2$, and vertices u and v have degress $n-1$ and 1. So new graph suits.

There are some other ways to build graph step by step. Also there are some ways for build required graph using constructive approach.

Problem L. Lisa's Sequences

Problem author: Evgeny Kapun; problem developer: Roman Elizarov

Let's first write an algorithm that checks in one linear scan if a given sequence a_i is boring or not. For this, we will check all elements a_i for i from 1 to n and maintain the following state:

- $d \in \{\uparrow, \downarrow\}$ — indicating the increasing or decreasing direction, respectively, of the longest monotone subsequence ending at the current element.
- t — the integer length of that longest monotone subsequence.
- s — the integer length of the last *horizontal* subsequence (a subsequence that consists of equal elements); note that $s \leq t$.

Initially, (d, t, s) are initialized to $(\uparrow, 0, 0)$. In fact, the initial value of d does not matter. On each element a_i the state is updated based on the comparison of the current element $c = a_i$ with the previous element $p = a_{i-1}$. The first element a_1 is considered to be equal to the previous element, so we let $a_0 = a_1$.

- $c > p$: if $d = \downarrow$ then $t \leftarrow s$; always do $d \leftarrow \uparrow$, $t \leftarrow t + 1$, $s \leftarrow 1$.
- $c < p$: if $d = \uparrow$ then $t \leftarrow s$; always do $d \leftarrow \downarrow$, $t \leftarrow t + 1$, $s \leftarrow 1$.
- $c = p$: $t \leftarrow t + 1$, $s \leftarrow s + 1$.

If we ever encounter a state with $t = k$ where k is the boredom threshold, then the sequence is boring.

Now, let's turn this algorithm into a dynamic programming solution to the problem. We'll start with the following simple lemma, that is given here without a proof:

Lemma. *There exists a solution b_i to the problem in which all the changes from the original sequence are either to the value of* $100\,000$, *which we'll call* $+\infty$, *or to the value of* 0, *which we'll call* $-\infty$.

Just as before, the solution is doing a single pass of the sequence a_i, moving forward for i from 1 to n. In addition to d, t, and s, it maintains two more state variables:

- m — the total number of updated elements so far.
- $u \in \{\top, -, \bot\}$ — whether the previous element was updated to $+\infty$, left unchanged, or updated $-\infty$ respectively.

The solution maintains sets r_i of reachable states (m, u, d, t, s). Initially, the set r_0 consists of a single state $(0, -, \uparrow, 0, 0)$. For each i from 1 to n, the set of previously reachable states r_{i-1} is scanned and a new set of reachable states r_i is filled.

For each state $(m_0, u_0, d_0, t_0, s_0)$ from r_i there are three possible choices for update of the element a_i defined by u_1 ranging over the set of $\{\top, -, \bot\}$. We use the same state update rule as the checking algorithm as we have all the information on the current value c (defined by u_1 and a_i) and the previous

value p (defined by u_0 and a_{i-1}). Thus, we get a new state $(m_1, u_1, d_1, t_1, s_1)$, with $m_1 = m_0$ when $u_1 = -$ and $m_1 = m_0 + 1$ otherwise. The new states are added to the set r_i if $m_1 < k$. The answer to the problem is the minimal value of m in the last set of reachable states r_n.

When implemented in this straightforward way, the algorithm will not fit into the time limit, as the number of reachable states will grow as i increases. In order to turn it into $O(n)$ algorithm, we'll need to make some modifications, reducing the number of states we keep. There are several ways to reduce the set of states, one of them is explained below.

Instead of a full set of reachable state r_i, we'll keep a reduced set r'_i. In each r'_i we'll keep at most one state per pair of (u, d), so each r'_i associates pairs (u, d) to a triple (m, t, s) instead of being a set of all possible quintuples (m, u, d, t, s).

If there are multiple states with a given (u, d), we'll keep in r'_i the one with the minimal m, among those we'll keep the one with the minimal t, and among those the one with the minimal s. Since d takes only two possible values ($d \in \{\uparrow, \downarrow\}$) and u takes only three possible values ($u \in \{\top, -, \bot\}$), there are at most 6 states to keep in each r'_i. In fact, among those 6, $(\bot, \uparrow)$ and $(\top, \downarrow)$ can never appear, leaving only four possibilities: $(\top, \uparrow)$, $(-, \uparrow)$, $(-, \downarrow)$, $(\bot, \downarrow)$.

The initial set r'_0 associates $(-, \uparrow)$ with $(0, 0, 0)$, then we proceed with updates as in the full algorithm, for each (u, d) keeping the state with the minimal triple (m, t, s) in lexicographic order. Just as before, we'll look for the minimal reached m at the end.

The full correct proof of this algorithm is too technical and requires analysis of a lot of cases. We'll just point out here some facts about the way it works.

First of all, instead of keeping the minimal triple (m, t, s) we can instead keep the minimal triple (m, s, t) and the algorithm will work just as well. In fact, if we look at the states reached in sets r_i for each (u, d, m) and find the minimal reached t and the minimal reached s among them, there will be always a state with the both t and s minimums reached at the same time.

Let's introduce a definition. For each i and each (u, d) pair, we'll call the state that reaches the minimal value of m and, amongst them, the state that reaches the minimal values of t and s the *leading* state.

The second observation is that states in the reduced sets r'_i are not always the leading states from r_i, but most of them are. In particular, in depends on (u, d):

- For $(\top, \uparrow)$ and $(\bot, \downarrow)$ the states in r'_i are always the leading states from r_i. Moreover, it can be shown that their (m, t, s) is of the form $(m, t, 1)$. Also, it can be additionally shown that it is never optimal to change two elements in a row.
- Between $(-, \uparrow)$ and $(-, \downarrow)$, at least one of such states in r'_i is a leading state from r_i. Moreover, among the two, the state that has the best (m, t, s) triple is the leading state from r_i.

This way, among the four states in r'_i, the state with the minimal m corresponds to the state with the minimal m in the full non-reduced set of reachable states r_i and corresponds to the solution of the problem.