Problem A. Adrenaline Rush

Problem author and developer: Dmitry Gozman

Consider two cars numbered x and y, $x \le y$. Suppose these cars finish at positions i and j, respectively, such that $c_i = x$ and $c_j = y$. There are two possible scenarios:

- $i \ge j$ - The cars finished in the reverse order. For this to occur, car y must overtake car x exactly once, but car x must not overtake car y.
- $i \le j$ - The cars finished in the same order. In this case, either car y overtook car x and then car x overtook car y back, or there were no overtakes between them. To maximize the answer, we assume that two overtakes occurred in this case.

Based on the above, the total number of overtakes does not exceed $n(n-1) - \text{inv}(c)$. Here $\text{inv}(c)$ is the number of pairs (i, j) such that $i \le j$ and $c_i \ge c_j$, called the number of inversions in the permutation c.

To achieve the maximum possible number of overtakes, we can use the following strategy. Consider the cars in reverse order of their finishing positions, i.e., $c_n, \ldots, c_1$. Let each car overtake all the cars in front of it one by one, and then allow all the cars that finished earlier to overtake it.

Applying this algorithm to the first example:

- The cars start in the order of $1, 2, 3$.
- Car 1 finishes last. Since it starts first, there is no other car it can overtake. However, both car 2 and car 3 should overtake it. The relative order at this point is $2, 3, 1$, with a total of 2 overtakes so far.
- Car 3 finishes second. It can overtake car 2 in front of it, after which car 2 overtakes it back. This results in the same relative order $2, 3, 1$, with a total of 4 overtakes.
- Car 2 finishes first, but there is no car it can overtake at this point.
- In total, this gives a sequence of 4 overtakes.

Problem B. BitBitJump

Problem author and developer: Georgiy Korneev

There are a lot of different solutions. One of them contains two blocks of 17 instructions each. The first block starts from word 0, and the second one from word 64.

The i-th instruction of the block, counting from zero, depends on the $(i-1)$-th bit of the x, where bit -1 is considered to be zero.

Let $\text{IO}_{\text{word}} = 2^{12} - 1$ and $\text{IO}_{\text{bit}} = \text{IO}_{\text{word}} \cdot 16$ — address of the IO word and its first bit. For each instruction let $@_{\text{word}}$ and $@_{\text{bit}}$ to be addresses of its first word and bit.

For $i = 0..15$ if the $(i-1)$-th bit of the input is zero, then in the i-th instruction of the first is `bbj` $\text{IO}_{\text{bit}}+i, @_{\text{bit}}+32+6, @_{\text{word}}+3$, and instruction of the second block is "`return false`": `bbj` $0, \text{IO}_{\text{bit}}, \text{IO}_{\text{word}}$. If the $(i-1)$-th bit of the input is one, then instructions in the blocks are swapped.

If the 15-th bit of x is zero, then the last instruction of the first block is "`return true`": `bbj` $4, \text{IO}_{\text{bit}}, \text{IO}_{\text{word}}$, and in the second block is "`return false`". Otherwise, these instructions also should be swapped.

Problem C. Cactus without Bridges

Problem author and developer: Levon Muradyan

Let's define the following condition from the problem **"the length of each odd cycle is greater than or equal to the number of odd cycles"** as ***** condition.

Let's also define two types of cactuses:

- *Type 1* — The number of cycles of odd length is even;
- *Type 2* — The number of cycles of odd length is odd.

Let's prove that when the cactus is of *Type 2*, then it's impossible to give labeling. Let's suppose, to the contrary, we can give the labeling satisfying the conditions of the problem. Since the degree of each vertex of the graph is even and for each vertex, the labels should make an interval of integers, we can state that for each vertex v, the number of odd labels of the edges incident to the vertex v is equal to the number of even labels of the edges incident to the vertex v. Let's calculate the number of edges with odd labels by Handshaking lemma and we will get $\frac{E(G)}{2}$, which states that the number of edges in the graph should be even.

We will prove that for cactuses of *Type 1*, it's always possible to give labeling when ***** condition is available. Let's prove the following two facts:

- If the cactus is of *Type 1* and ***** condition is available, then we can label the edges of the graph satisfying the problem conditions;
- If the cactus is of *Type 2* and ***** condition is available, then we can label the edges of the graph satisfying problem conditions except for one arbitrary vertex u so that there will exist an integer x such that the labels of edges incident to the vertex u, x and $x + 2$ will form an interval of integers. More formally for the vertex u we have two missed integers for a full interval, and the difference between them is equal to 2.

First, let's note that ***** condition is satisfied for any cactus subgraph of the given cactus. We can prove one of these two facts, the other one will be proved in the same way. Let's prove the first one.

We will give proof using an induction method on the number of vertices of the graph. Let's take an arbitrary cycle from the given cactus, and let's consider that the cycle has the length k and the vertices of the cycle are u_1, u_2, ..., u_k. If we remove from the graph the edges of this cycle, we will have k components: G_1, G_2, ..., G_k, where G_i will contain the vertex u_i for each $1 \le i \le k$.

Now let's define the following **F** function for the vertices of the cycle in the following way:

- $|F(v_i)| = 1$, if G_i is of *Type 1* for $1 \le i \le k$;
- $|F(v_i)| = 2$, if G_i is of *Type 2* for $1 \le i \le k$;

where $|x|$ means an absolute value of x.

Let's define by c_1 — the number of i such that G_i is of *Type 1*, while by c_2 — the number of i such that G_i is of *Type 2* for $1 \le i \le k$.

If we label the edges of the cycle such that the difference between the labels of two consecutive edges will be equal to $F(w)$, where w is the common vertex of those two edges, then everything will be fine (we can shift the label of subgraphs in such a way that all conditions will be satisfied). Let's note that it's possible to label the edges of this cycle if the following condition $\sum_{i=1}^{k} F(v_i) = 0$ is true.

Now let's consider two cases:

Case 1: k is even. Here both c_1 and c_2 should be even, so we can take the values of F in the following way:

- The number of v_i such that $F(v_i) = 1$ is equal to $\frac{c_1}{2}$ for $1 \le i \le k$;
- The number of v_i such that $F(v_i) = -1$ is equal to $\frac{c_1}{2}$ for $1 \le i \le k$;
- The number of v_i such that $F(v_i) = 2$ is equal to $\frac{c_2}{2}$ for $1 \le i \le k$;
- The number of v_i such that $F(v_i) = -2$ is equal to $\frac{c_2}{2}$ for $1 \le i \le k$.

Case 2: k is odd. Here c_2 should be odd, while c_1 should be even. Since we have * condition, then you can note that $c_1 \geq 2$, so we can take the values of F in the following way:

- The number of v_i such that $F(v_i) = 1$ is equal to $\frac{c_1-2}{2}$ for $1 \leq i \leq k$;
- The number of v_i such that $F(v_i) = -1$ is equal to $\frac{c_1+2}{2}$ for $1 \leq i \leq k$;
- The number of v_i such that $F(v_i) = 2$ is equal to $\frac{c_2+1}{2}$ for $1 \leq i \leq k$;
- The number of v_i such that $F(v_i) = -2$ is equal to $\frac{c_2-1}{2}$ for $1 \leq i \leq k$.

Problem D. DAG Serialization

Problem author: Vitaly Aksenov; problem developer: Dmitry Petrov

Consider the lifetime of our register. Notice that it has no more than three phases:

1. The initial "`false`" phase
2. The intermediate "`true`" phase
3. The final "`false`" phase

Additionally, there are no more than 4 types of operations:

1. "`set true`"
2. "`unset true`"
3. "`set false`"
4. "`unset false`"

The operation of the 1st type "`set true`" transitions between the 1st and the 2nd register phases. The operation of the 2nd type "`unset true`" transitions between the 2nd and the 3rd register phases.

All the operations of the 3rd type are between the 1st and the 2nd. We can add logical arcs from the operations of type 1 to all the operations of type 3, and from all the operations of type 3 to the operation of type 2. Also, add a logical arc from the 1st to the 2nd operation.

Operations of the 4th type may occur in either the 1st or 3rd phase. Their exact placement can only be determined by the DAG.

We can finally build a graph G containing n vertices (all the operations), m given DAG arcs, and the logical arcs mentioned. Then, simply find a topological sort in G, starting from the operation of type 1, ensuring there are no loops, and return it as the answer.

Problem E. Expression Correction

Problem author and developer: Roman Elizarov

This is a straightforward problem. The limits allow for exhaustively checking all possible ways to move a digit, then parse the equality and check if it is a correct one. However, there are many corner cases that should be taken care of, including the following ones:

- Make sure that numbers with leading zeros are not allowed (WA 3).
- Make sure that 11-digit numbers are not allowed (WA 5).
- Make sure that you only move digits, not other characters (WA 6).

- Make sure that you allow zero "`0`" as a valid number (WA 7).
- Make sure that "empty" number is not allowed (WA 23).
- Make sure that "`00`" is not considered a valid number (WA 24).
- Make sure that expressions don't start with '`+`' or '`-`' (WA 50).

Problem F. Fix Flooded Floor

Problem author: Vitaly Aksenov; problem developer: Zakhar Iakovlev

We'll call a cell *empty* if it's damaged and not yet covered with a 1×2 piece, and *full* otherwise.

Let's process the columns greedily from left to right and try to fill the parquet with 1×2 pieces. We'll always maintain the invariant that all cells in the processed columns are full.

For the current column:

- If both cells are full, we don't need to do anything.
- If one cell is empty and the other one is full, the only thing we can do is put a new 1×2 piece horizontally, with its left half in the empty cell. Here, we need to make sure that the cell on the right of the empty cell is empty as well. If that's true, we put a new piece, so both empty cells are now full. Otherwise, the answer is "`None`".
- If both cells are empty, we have two options here. First, we could always put a vertical piece in this column. Second, only if both cells in the next column are empty, we could put two horizontal pieces covering these two columns. However, if we do the latter, we might as well use two vertical pieces for these two columns instead. Thus, if this happens, the answer can never be "`Unique`". We can just place a vertical piece and proceed, and if we manage to finish the tiling, the answer is "`Multiple`", otherwise it's "`None`".

To conclude, if the above process fails at any point, the answer is "`None`". If we ever encounter a situation where we could put either two vertical pieces or two horizontal pieces, the answer is "`Multiple`". Otherwise, the answer is "`Unique`".

Time complexity of this solution is $O(n)$.

Problem G. Geometric Balance

Problem author: Georgiy Korneev; problem developer: Egor Kulikov

- **Representing Directions and Movements:** We consider movement along four principal directions (North–South, East–West, Northeast–Southwest, and Northwest–Southeast). Each step can be viewed as a combination of these four directions. We return to a point if, when summed over all movements, there is no net displacement in any of these four directions.
- **Merging Consecutive Segments:** We start with a sequence of line segments. If two consecutive segments share a common endpoint and point in the same direction, we merge them into a single, longer segment. We repeat this process until no further merging is possible, resulting in a simpler representation of the picture.
- **Checking for Identical Pictures Under Rotation:** We want to see if the given picture can be matched to itself after certain rotations. We will test three possible rotation angles:
  - 45 degrees

- 90 degrees
- 180 degrees

If none of these rotations produce the original configuration, we conclude that only a full 360-degree rotation brings the picture back to itself.

- **Comparing the Sets of Segments After Rotation:** For each rotation we test:
  1. Rotate all segments by the chosen angle.
  2. Translate the rotated figure so that the lexicographically smallest endpoint matches the smallest endpoint of the original configuration.
  3. Compare the resulting set of segments to the original. If they match, we have found a rotation that maps the picture onto itself.

If no tested rotation (45°, 90°, 180°) results in a match, the answer is effectively 360°.

Problem H. Hunting Hoglins in Hogwarts

Problem author and developer: Tikhon Evteev

The main idea is to not do the binary search. If you try, you will probably end up with about 400 caught hoglins, which is not enough.

Instead we will maintain the set of disjoint segments $s_1, \ldots, s_k$ of approximately equal length. i.e. $||s_i| - |s_j|| \le 1$. After each iteration of the algorithm, it will hold that the hoglin is in one of those segments, and this segment is also the segment that the hoglin considers accessible. We will also keep track of the probabilities that the hoglin is in each of these segment, and denote them as p_i. We will reorder the segments so that $p_1 \ge p_2 \ge \cdots \ge p_k$.

From this point on we will not keep track of the fact that the segments are of approximately equal length, and write as if they were of exactly equal length. We will also not mention, that if at any point of the interaction we catch the hoglin, we should immediately stop and restart the process.

Initially the hoglin is in a segment $s_1 = [1, n]$ with the probability $p_1 = 1$.

For the iteration of the algorithm we will go through the segments in order from 1 to k and block the midpoint of the corresponding segment. If the interactor returns 1 after we block the midpoint of the j-th segment, than the hoglin was in one of the segments $s_1, \ldots s_j$ and now is in one of the $2 \cdot j$ new segments, each of which is approximately two times smaller than the segments in the previous iteration. If we don't get a 1 after we block the last midpoint, we will just wait for it by sending 0, and let $j = k$.

Let's for now denote these segments as $s_1^1, s_1^2, s_2^1, s_2^2, \ldots, s_j^1, s_j^2$, and compute the probabilities $p_1^1, p_1^2, p_2^1, p_2^2, \ldots, p_j^1, p_j^2$ with which the hoglin is in each of these segments.

The conditional probability that we will get a 1 after blocking the j-th midpoint, if the hoglin initially was in the i-th segment is $P(R_j|W_i) = \frac{1}{2^{j-i+1}}$, thus the probability that it is now in either of the segments s_i^1 or s_i^2 by Bayes' theorem is $p_i^1 + p_i^2 = P(W_i) = \frac{p_i \cdot P(R_i|W_i)}{P(R_i)} = \frac{p_i \cdot P(R_i|W_i)}{\sum_{l=1}^{j} p_l \cdot P(R_l|W_l)}$.

So we can recompute the probabilities p_i for the next step and continue the algorithm.

Now to prove the asymptotics we can prove by induction that the sequence $p_1, \ldots p_k$ looks like $p, \ldots p, \frac{p}{2}, \ldots \frac{p}{2}, \frac{p}{4}, \ldots \frac{p}{4}, \ldots$. More over, each sequence of the form $\frac{p}{2^l}, \ldots \frac{p}{2^l}$ is at most 4 elements long.

To show that we will use the fact that $p_i^1 = p_i^2 \propto p_i \cdot P(R_i|W_i) = \frac{p_i}{2^{j-i}}$. Not strictly: the sequence of probabilities is multiplied by the sequence 2^i, and then each one is duplicated and the sequense is normalized. Thus the sequence on the equal probabilities is no longer than 4, and the probabilities are increasing exponentially.

Knowing that, we do not need to maintain the probabilities p_i to sort the segments in the right order, because we can just reverse the order of segments after each iteration and it will maintain the right order.

Also knowing the exponentially decreasing nature of p_i we can see that the expected value of the number of queries for each iteration is at most $1 \cdot p + 2 \cdot p + 3 \cdot p + 4 \cdot p + 5 \cdot \frac{p}{2} + 6 \cdot \frac{p}{2} + \ldots$, where $p = \frac{1}{8}$, which is $\frac{5}{2} + 4 = 6\frac{1}{2}$. Since there are at most $\log_2 n$ iteration, the solution works in at most $6\frac{1}{2} \cdot \log_2 n$ queries per hoglin on average.

The problem demands no more then about $4.2 \cdot \log_2 n$ queries per hoglin, but one can see how our analysis was only a rough upper bound on the number of queries. In reality, the solution works in about $3.9 \cdot \log_2 n$ queries per hoglin, which leaves more then enough room for variance.

Problem I. Incompetent Delivery Guy

Problem author: Tikhon Evteev; problem developer: Mikhail Ivanov

Let us call a vertex a *k-vertex* if an orc of incompetence k, starting from this vertex, will definitely reach Orthanc, but an orc of incompetence $k+1$ is not guaranteed to do that. (Or -1-vertex if an orc with any level of incompetence will not be able to reach Orthanc; or ∞-vertex if an orc with any finite level of incompetence will eventually reach Orthanc.) This unique number k can be called the *type* of this vertex, and the problem essentially consists of finding the type of the entrance vertex. We will present an algorithm of finding the types of all vertices.

It can be derived from the statement that -1-vertices are exactly the vertices from which there does not exist an oriented path to Orthanc. In particular, an orc succeeds with their order if and only if they never visit -1-vertices. Can we then characterize the 0-vertices? If a vertex (which is not a -1-vertex) has an arc from it to a -1-vertex, it is definitely a 0-vertex, but it is not a necessary condition. The necessary and sufficient condition is that on each shortest path from a vertex (which is not a -1-vertex) to Orthanc there is at least one vertex from which there is an arc to a -1-vertex.

Formally, let us define an *immediate k-vertex* as a k-vertex from which there is an arc to a $k-1$-vertex. Then, we can find all immediate k-vertices as the vertices from which there is an arc to at least one $k-1$-vertex, but which themselves are not ℓ-vertices for $\ell \le k-1$. And k-vertices can be determined as the vertices from which any shortest path to Orthanc lies through at least one immediate k-vertex. Indeed, if a vertex is a k-vertex, then, no matter which shortest path Saruman chooses for an orc of incompetence $k+1$ in this vertex, at some moment they can hang around to a $k-1$ vertex, which means that they necessarily visit an immediate k-vertex. However, if an orc of incompetence at most k appears in this vertex, after the first hanging around they will appear in a vertex of type at least $k-1$. The converse (that if our method reported a vertex as a k-vertex then it is actually a k-vertex) can be proven similarly. Note that everything aforementioned does not apply to Orthanc, which is neither a k-vertex nor an immediate k-vertex for any finite k, it is always a ∞-vertex.

Hence, the solution is as follows. Firstly, we find $\overrightarrow{\text{dist}}(u, O)$ for all vertices u using Dijkstra's algorithm ($\mathcal{O}(n + m \log n)$ is fast enough). In particular, we determine all -1-vertices. Then, for each $k = 0, 1, 2, \ldots$ we do the k^{th} stage of the algorithm. During this stage, we firstly find all immediate k-vertices simply by iterating over all ingoing edges in $k-1$-vertices. Then we can find all k-vertices using an algorithm similar to retroanalysis. This part works in $\mathcal{O}(m)$ time. The overall complexity is $\mathcal{O}(n + m \log n)$.

Problem J. Judicious Watching

Problem author: Tikhon Evteev; problem developer: Pavel Kunyavskiy

Let's consider we have chosen which homework tasks should be done before time t_i, and which after. What is the optimal strategy of task scheduling in this case? Ideally, we should have as much time as we can to watch the series and have it as a one segment of time. To do this, we can do all tasks, which should be finished before at the first moment we can do them, and all tasks need to be done later at the last moment we can. Then we would have free between finishing tasks before, and either this last moment or the moment of the call.

To do this, we need to compute the last moment each task can be started, so all later tasks still can be finished in time. If we know these values, to answer the query, we can find all tasks which can be started after the moment of the call, and postpone them to that time, while doing all others in the beginning. This can be done in $O(\log n)$ time per query with $O(n)$ precomputation with binary search and prefix sums to find the total length of tasks to be done in advance.

To find this last moment, we need a greedy approach, going from right to left through our list of tasks. The last task can be started to finish it right at its deadline. Each previous task must be started to finish either at its deadline or at the last time the next task needs to start, whatever is smaller. This can be also computed in $O(n)$ time after sorting by deadlines.

So overall complexity will be $O(n \log n)$ for preprocessing, and $O(\log n)$ for each query.

Problem K. Knowns and Unknowns

Problem author: Tikhon Evteev; problem developer: Niyaz Nigmatullin

Get some professor's list, it consists of permutation elements and wildcards. Define a set of blocks consisting of:

1. a known element of the list;
2. a non-extendable set of consecutive d unknown elements.

For each second type of blocks there is a set of elements such that any of its subset of size d can be there, call the elements of this set as candidates for this block. This set consists of some elements from original permutation that are between the element on the left, and the element on the right of this block. For each first type of blocks the only candidate is the one that is known at that position. Note that each element is a candidate to at most one of the blocks.

Let's build the blocks for both professors.

Then we can build a flow network similar to bipartite matching network, the left part is for the blocks of the first professor, and the right — of the second one. From the source to each block of the first professor we have edges with capacity "the size of the block". The same is for the blocks of the second professor and sink. For each integer from 1 to n, if it is a candidate to a block of first professor, and to a block of second professor, then add an edge with capacity 1 from the first block to the second.

If the maximum flow in the network is less than k, then the data is inconsistent.

Otherwise, for each edge we need to check "is there a maximum flow with this edge in it?", and "is there a maximum flow without this edge in it?". Find a maximum flow, if an edge is in the maximum flow, then check if there is a cycle in a residual network with the reverse of the edge. If an edge is not in the maximum flow, then check if there is a cycle in a residual network with the edge. This is done by finding a path between the end of the edge and its beginning in the residual network, the path wouldn't contain the reverse edge, since it is already saturated. The solution works in $O(n^2)$.

Problem L. Legacy Screensaver

Problem author: Georgiy Korneev; problem developer: Gennady Korotkevich

The rectangles overlap if and only if their individual projections on the x-axis and the y-axis both overlap. Thus, we can solve the problem separately for the axes, and then merge the results in some way.

Consider the x-axis, and imagine that instead of two rectangles, we just have two segments of lengths w_1 and w_2 moving inside the $[0; W]$ range. Each segment is moving at a speed of -1 or 1 and reflecting off the ends of the range.

It can be seen that the first segment returns to the same state (position and speed) after exactly $2(W - w_1)$ time units. Similarly, the period of movement of the second segment is $2(W - w_2)$.

If we look at the rectangles as a pair, the period of their common movement is $t_x = \text{lcm}(2(W - w_1), 2(W - w_2))$, where $\text{lcm}(p, q)$ stands for the least common multiple of p and q. That is, at any time τ, the segments overlap if and only if they also overlap at times $\tau + t_x, \tau + 2t_x, \ldots$.

We can write down a bit string a of size t_x, where a_τ is 1 if the segments overlap at time τ, and 0 otherwise. Similarly, we can write down a bit string b of size t_y, containing the same information for the projections of the rectangles on the y-axis.

Now, for any time τ, we can see that the rectangles overlap if and only if $a_{\tau \bmod t_x} = b_{\tau \bmod t_y} = 1$.

Consider two bits a_i and b_j ($0 \le i < t_x$; $0 \le j < t_y$). Does there exist time τ such that a_i will be matched against b_j — that is, such that $\tau \bmod t_x = i$ and $\tau \bmod t_y = j$? It is well-known that a solution τ to this system of modular equations exists if and only if $i \equiv j \pmod g$, where $g = \gcd(t_x, t_y)$ is the greatest common divisor of t_x and t_y.

Finally, it can be shown that all pairs (a_i, b_j) which satisfy $i \equiv j \pmod g$ will be matched against each other with equal frequencies. For each $r = 0, 1, \ldots, g - 1$, we can find how many indices i exist such that $a_i = 1$ and $i \bmod g = r$; let this number be c_r. Similarly, we can find how many indices j exist such that $b_j = 1$ and $j \bmod g = r$; let this number be d_r. Now, the numerator of the answer fraction is $\sum_{r=0}^{g-1} c_r \cdot d_r$, and the denominator is $\text{lcm}(t_x, t_y)$. Don't forget to reduce the fraction.

Time complexity of this solution is $O((W + H)^2)$.

Problem M. Managing Cluster

Problem author: Tikhon Evteev; problem developer: Ivan Safonov

Despite this problem is formulated as a maximization problem, this is a constructive problem.

Let us find a maximum matching in this tree. It can be done with dp or a greedy algorithm in linear time. Let us define a set of edges in this matching as $M = \{(s_i, f_i)\}_{i=1}^{k}$.

Obviously, the answer does not exceed $|M|$. Let us construct swaps, such that the answer is exactly $|M|$ and equal replicas are placed in edges from M.

Let us construct a graph on n services, where we will connect services a_{s_i}, a_{f_i} with an edge. Each service will have a degree of at most 2, so this graph is a union of paths and cycles.

Let us consider some cycle. Suppose it consists of c edges $(p_1, p_2), (p_3, p_4), \ldots, (p_{2c-1}, p_{2c})$, where p_i are machines and $a_{p_{2i}} = a_{p_{2i+1}}$. Let us make $c - 1$ swaps between machines: $p_1 \leftrightarrow p_{2c-1}, p_2 \leftrightarrow p_{2c-2}, \ldots, p_{c-1} \leftrightarrow p_{c+1}$. As a result, all edges in the cycle will have replicas of equal services.

To solve paths, let us do the same operation as with cycles. We can do that because we can just consider that the start and the end of the path are connected.

With the constructed swaps, we will have the optimal answer. Time complexity $O(n)$.