The contest is based on problems from the 2025 ICPC Belarus Qualification and the 2025 ICPC Belarus Regional Contest. The problems were designed, prepared, and tested by (in no particular order):

- alumni of the Belarusian State University:
  Yahor Dubovik, Dzianis Kim, Yafim Klimasheuski, Andrey Lukashevich,
  Ivan Lukyanov, Ilya Malochka, Aleksey Tolstikov;
- alumni of the Immanuel Kant Baltic Federal University:
  Aleksandr Balobanov, Giordano Chernozhukov.

We would like to thank Oleg "Snark" Hristenko for help with setting up the run-twice problems.

Problem A. Art of Multiplication

Idea:	Andrey Lukashevich, Ivan Lukyanov
Development:	Ivan Lukyanov
Tutorial:	Ivan Lukyanov

For simplicity, let us assume that $r = 10$. For other bases, the formulae are very similar.

The number z can be expressed as $\frac{a(10^n-1)}{10-1} \times \frac{b(10^m-1)}{10-1}$ by the formula for the sum of a geometric series. This is equal to $\frac{ab(10^{n+m}-10^n-10^m+1)}{9^2}$. The expression in parentheses, though having a very long decimal representation, consists almost entirely of 0s and 9s. (This can be visually verified by writing out the expression for various values of n and m.) After multiplying by ab, the resulting expression still contains only $\mathcal{O}(1)$ digits that are not 0 or 9. Example:

$$a = 4,\ n = 6, \quad b = 7,\ m = 11: \qquad \texttt{numerator} = 2799997199972000028.$$

We can then carefully construct the explicit but compressed representation of the numerator (as pairs "a digit; its quantity") and divide it by 9^2 using the basic algorithm of long division by a small number. However, when encountering long sequences of identical digits, we don't have to do division straightforwardly. Notice that remainders quickly enter cycles, and these cycles can be skipped in constant time if you memorize the initial carry value and accumulate the sum of digits in the period.

Be sure to process corner cases carefully when n and m are quite small, or when $|n - m| \le 3$.

There are solutions that do not do division or construct the decimal representation explicitly, but they pretty much have similar reasoning. Additionally, empirical solutions based on pattern detection also exist.

Problem B. Building a Reactor

Idea:	Andrey Lukashevich
Development:	Andrey Lukashevich
Tutorial:	Andrey Lukashevich

We will do dynamic programming on a broken profile: $dp[m][n][mask]$ = minimum number of cooling elements where:

- $m - 1$ layers are fully placed
- n is the position of the profile break (the number of cells on the current layer, for convenience $dp[m-1][n][*] == dp[m][0][*]$)
- $mask$ is a ternary profile of n trits (0 is uncovered fuel rod, 1 is a covered fuel rod, 2 is a cooling element).

There are 3 different transitions:

- '0': we insert the fuel rod. It is allowed only if the left cell is a covered fuel rod or a cooling element; the new value in the mask is covered only if the top or left is cooling.
- '1': insert the cooling element (+1 to the price); in the mask, you need to remember to update the value from the above cell $\max(1, up)$.
- '#': only if there is a wall; the difference from 0 is that we always put 1 in the mask.

The dynamics are forward; the initial values are infinity; the transition makes relaxation of dp values.

The starting state is a mask of $11\ldots11_3$. The final state is any mask without zero elements. Now you can just process from left to right when calculating the answer, and to restore the answer in the reverse order from the best final state.

When calculating dp, information can be saved to restore the response in the form of an action and a mask from where they came from; however:

```
n = 12, m = 30
short dp[m][n][3**n] = 364 megabytes
```

Another int won't fit to restore the response; however, if you really want to, you can squeeze it through the same bit fields, for example:

```
struct DpState {
    uint32_t cost : 10 = 500;
    uint32_t action : 2;
    uint32_t prev_mask : 20;
};
```

Alternatively, you can go through the reverse transitions and recalculate where you came from. However, these transitions are more complicated than the forward transitions.

Therefore, it is possible to invert the calculation, and now $dp[m][n][mask]$ is the minimum number of cooling elements that can be added to this arrangement in order to fully assemble the reactor. The transitions remain essentially the same; swap the left and right indices. It can be calculated either with a recursive function with memoization or with processing DP in reverse order. The starting and ending states will be reversed. Now, to restore the answer, it is enough to start with the mask $11\ldots11_3$ and, going from left to right, choose the best state from 3 transitions.

Problem C. Cell Flip Game

Idea:	Andrey Lukashevich
Development:	Andrey Lukashevich
Tutorial:	Ivan Lukyanov

For each binary matrix, we can obtain the *canonical* form that has all zeros in each row except the last one. To do this, we must eliminate rows from top to bottom: to eliminate another row i ($i < n$), for each cell j that has $a_{ij} = 1$, make a move into cells $(i + 1, j)$. We will call two binary matrices *equivalent* if they have the same canonical form.

Note two things: 1) to transform a matrix into its canonical form, only moves in rows between 2 and n are performed; 2) each move in rows between 2 and n flips a cell in a row above. Making moves in the topmost row is unique in this regard because they don't participate in nullifying the top $n-1$ rows. Also, only they are capable of changing the canonical form of the matrix!

This leads us to the following idea. Let us convert the matrix into its canonical form and denote L as its last row. Also, denote B_j as the zero matrix with only one move made into cell $(1, j)$. Let K_j be the last row of the canonical form of B_j.

Statement. If $K_{j_1} \oplus K_{j_2} \oplus \ldots \oplus K_{j_t} = L$ for some subset $\{j_1, j_2, \ldots, j_t\} \subseteq \{1, 2, \ldots, m\}$, then by making moves into cells $(1, j_1), (1, j_2), \ldots, (1, j_t)$ we can obtain a matrix equivalent to the zero matrix.

To find a necessary subset of indices $j_1, j_2, \ldots, j_t$, we can solve a system of linear algebraic equations $(K_1^\top \| K_2^\top \| \ldots \| K_m^\top) \cdot (b_1, b_2, \ldots, b_m)^\top = L$ w.r.t. the XOR operation in $GF(2)$. Here $b_j = 1$ if and only if cell $(1, j)$ has to be flipped. To do this, Gaussian elimination can be used. Take into consideration that this system can have no solutions (in this case, the answer is `NO`), and it can also have multiple solutions, so cases of matrix degeneracy should be handled carefully.

Time complexity is $\mathcal{O}(nm^2 + m^3)$ or $\mathcal{O}((nm^2 + m^3)/w)$ if canonical form calculation and Gaussian elimination are implemented using bitsets (w stands for the length of machine word size in bits, for example, $w = 64$), though it was not necessary for this problem. Memory complexity is $\mathcal{O}(nm + m^2)$.

Problem D. Detecting the Missing Ship

Idea:	Giordano Chernozhukov
Development:	Giordano Chernozhukov
Tutorial:	Ivan Lukyanov

Let's say that the position of the ship is (i, j) if its topmost and leftmost cell has coordinates (i, j).

If we hit cell (r, c), then we expose vertical ships on positions $(r - k + 1 \ldots r, c)$ and horizontal ships on positions $(r, c - k + 1 \ldots c)$. This way, we split a rectangular region of possible ship positions into 4 (possibly empty) rectangles. Also, note that each of these rectangles will expand by one because the ship can move by one cell.

This gives us a realization that if we hit somewhere far from the corner, then the region reduction will be smaller than that if we hit into the corner. In particular, hitting cell (k, k) exposes all ships in the first k rows and columns, and we are only left with rows and columns of $k, k+1, \ldots, n$ on the next move. Thus, in one move, we eliminate $k - 1$ rows and columns. This cannot be achieved by hitting into inner cells because the increase of the set of possible rows and columns of ship positions will be greater than one, since there are two nonempty parts from each side.

This means that hitting cells (k, k), $(2k - 1, 2k - 1)$, $(3k - 2, 3k - 2)$ and so on is an optimal strategy.

Problem E. Encountering a Friend

Idea:	Yahor Dubovik
Development:	Andrey Lukashevich
Tutorial:	Yafim Klimasheuski

If $n = 1$, then all the matches happen between the friends, and the answer is k.

If $n > 1$, then, after the first match they have, they are separated and can only play against each other every second match. So, if before their first match they have to play t games, the answer is $\lfloor \frac{k-t+1}{2} \rfloor$.

How do we find t?

If the difference $|a - b|$ is even, then Alice and Bob can move towards each other, closing the distance by 2 each game, and $t = \frac{|a-b|}{2}$.

If the difference $|a - b|$ is odd, one of the players has to approach either the 1-st or n-th table and play a game there without moving beyond the bound, so that the difference can become even. Let's name $x = \min(a, b)$, $y = \max(a, b)$.

Starting at y, approaching n, playing a game and then approaching x is equivalent to starting at $(n+(n+1-y))$ and moving towards x. Starting at x, approaching 1, playing a game and then approaching y is equivalent to starting at $(1-x)$ and moving towards y. So, $t = \frac{1}{2}\min((n+(n+1-y))-x, y-(1-x))$.

Note that if $t \geq k$, the answer is 0.

Problem F. Forgot to Refuel

Idea: Yahor Dubovik
Development: Ivan Lukyanov
Tutorial: Ivan Lukyanov

Let's calculate the sum of maximum distances between fuel stations over all possible placements and then divide the sum by the number of such placements: that will be the desired expected value. Without loss of generality, assume that the 1-st fuel station is located at the 1-st intersection.

Next, denote x_i as the distance between fuel stations i and $i+1$ if $i < n$, or between fuel stations n and 1 if $i = n$. Now let's look at the problem from a bit different perspective: we would like to count the sum of $\max\{x_1, x_2, \ldots, x_n\}$ over all integer solutions of the equation:

$$x_1 + x_2 + \ldots + x_n = \ell, \quad x_i \geq 1. \tag{1}$$

Direct counting sum of maxima looks hard. So let's try another way of summation. Denote $f(d)$ as the number of solutions of (1) for which $x_i \leq d$ for all $i = \overline{1, n}$:

$$x_1 + x_2 + \ldots + x_n = \ell, \quad 1 \leq x_i \leq d. \tag{2}$$

This is a well-known combinatorial problem that can be solved using the inclusion-exclusion principle. It's pretty hard to count solutions of (2) directly as well. The main idea is that instead of solving (2), we can solve (1) and then subtract all solutions in which at least one of $x_i > d$. This way, we will count only solutions of (1) that are solutions of (2) as well. And in order to do that correctly and consider each solution exactly once, we apply the inclusion-exclusion formula by iterating over the number k of "violated" x_i. This way, we get the following formula:

$$f(d) = \sum_{k=0}^{n} (-1)^k \binom{n}{k} \cdot \binom{\ell - k \cdot d - 1}{n - 1}. \tag{3}$$

The second binomial coefficient in the sum counts solutions of (1), provided that some chosen k of n variables are greater than d: we basically subtract d from each of them and obtain the original equation (1) with reduced sum $\ell - k \cdot d$. And then, we choose $n-1$ "dividers" between x_i among $\ell - k \cdot d$ cells: each divider i guarantees 1 for x_i, and the last cell is taken out to guarantee $x_n \geq 1$.

Formula (3) has $(n+1)$ addends, but we can notice that once $k > \frac{\ell - n}{d}$, the second binomial coefficient will always be zero. So we can calculate this sum only for $0 \leq k \leq \frac{\ell - n}{d}$.

Back to the original problem: we are finally able to write down the sum over all placements:

$$S = 1 \cdot f(1) + 2 \cdot (f(2) - f(1)) + 3 \cdot (f(3) - f(2)) + \ldots + (\ell - n + 1) \cdot (f(\ell - n + 1) - f(\ell - n)). \tag{4}$$

It is easy to see that $(f(i) - f(i-1))$ expresses the number of solutions of (1) with maximum *exactly* equal to i. Of course, the formula can be simplified a bit further. To obtain the final answer to the problem, we multiply S by ℓ (the first fuel station can be actually located at any intersection), then divide it by n (because the fuel stations on the circle are indistinguishable), and finally divide by $\binom{\ell}{n}$ (to get the expected value instead of just the sum).

Factorials and inverse factorials up to ℓ can be precomputed in $\mathcal{O}(\ell)$; we are now able to compute binomial coefficients in constant time. Next, each $f(d)$ is computed in $\mathcal{O}(\frac{\ell - n}{d})$. In total, we get time complexity $\mathcal{O}(\ell + (\ell - n)(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{\ell - n + 1})) = \mathcal{O}(\ell + (\ell - n)\log(\ell - n))$. Memory complexity is $\mathcal{O}(\ell)$.

Problem G. Guaranteed Medal

Idea: Aleksey Tolstikov
Development: Dzianis Kim
Tutorial: Dzianis Kim

Implement a structure that corresponds to the current result of a specific team for a specific problem. It should include the number of solved problems, the current penalty time, the current number of incorrect attempts until the first correct one (if it even exists), and the index of the first correct submission (if any). Then, you can recalculate it if another submission for this team and problem arrives.

You can actually add up those structures for the same team and different problems to find the overall result of the team, adding up all parameters except for the solving submission index, for which it's better to take the maximum of indices to account for the decisive submission.

Then, you can simulate the whole contest as follows. Maintain the results of a specific team in an `std::map` with team names as keys. The results should include p structures – one for each problem – and one more structure for an overall result. When a submission for a team arrives, we need to recalculate two structures: the one for the problem the submission is related to, and the overall one can be recalculated too by possibly adding more solved problems and penalty time, and possibly increasing the index of the last (of the first ones per problem) solution.

Simulate the contest once, find the final results for every team. To find when the medalist teams have guaranteed their medals, find which team has the 13th result, save that result, and simulate the contest again. Consider the current submission. If the submitting team hasn't exceeded the final result of the 13th team before the submission, but has exceeded after, output that team and the time of the submission.

The total time complexity is $\mathcal{O}(s \log s)$. Many solutions are possible, including the one with an additional $\mathcal{O}(p)$ factor where for every submission we naively recalculate the team's overall result, summing up everything over all problems. Other optimizations are possible too.

Problem H. Helpful Bits

Idea:	Dzianis Kim
Development:	Dzianis Kim
Tutorial:	Dzianis Kim

If not for the absent weights, the problem could be solved in a quite standard way using Dijkstra's algorithm with a data structure (e.g. `std::set`), resulting in time complexity $\mathcal{O}(\sum mq \log n)$ per test case.

We are only allowed to encode each of m weights with 12 bits on average, at the cost of some approximation of the final answer being allowed. Let's try to approximate the weights somehow.

Let $a = 10^{-3}$, and let's denote for a path P its "true" length (that is, the one expressed in initial weights of the graph) as $f(P)$, and its modified length (expressed in the approximated weights) as $g(P)$.

Let's try multiplying every weight w by a certain (unknown yet) number from the segment $[1-a, 1+a]$, and check if that would be a good enough solution. We can say that the length of every path in the graph will also be multiplied by some value from that segment, that is, $f(P) \cdot (1-a) \le g(P) \le f(P) \cdot (1+a)$. That is also true for the true shortest path A.

Now, if we run any shortest path algorithm using the modified weights and conclude that the path B has the smallest length (that is, $g(B) \le g(P)$ for any path P), that would imply

$$g(B) \le g(A) \le f(A) \cdot (1+a)$$

and

$$g(B) \ge f(B) \cdot (1-a) \ge f(A) \cdot (1-a),$$

which means we actually satisfied the approximation condition!

If we can select a set of at most $2^{12} = 4096$ (possibly floating point) numbers such that for any possible integer weight w between 1 and 10^6 there's a number from the set that lies on the segment

$[w \cdot (1-a), w \cdot (1+a)]$, we can just find a corresponding number for each weight, encode it by its index in the set, and solve the problem.

For that we can, for example, form segments of integers $[L, R]$ that satisfy the condition $R/L \le (1+a)/(1-a)$, and pick any suitable number inside it (for nontrivial segments, any number on $[L \cdot (1+a), R \cdot (1-a)]$) to replace any weight from that segment. Make sure to use a slightly smaller number than a. Form segments greedily, starting from $[1, 1]$, and setting the left border of the next segment equal to the right border of the previous segment plus one. That will result in 4089 segments, which is enough.

Problem I. Ivan Tsarevich in the Magical Grove

Idea:	Yafim Klimasheuski
Development:	Yafim Klimasheuski
Tutorial:	Yafim Klimasheuski

Suppose we have a graph with n vertices, where each vertex has an unknown number $s_i \in \{0, 1\}$ (either lies or tells the truth). Then each answer is an edge with a number $v_{i,j}$ in it, such that $s_i \oplus v_{i,j} \oplus 1 = s_j$, and each valid configuration of s_i where $\sum s_i = k$ represents a valid configuration of the trees.

Note that fixing a value s_i of any node in a connected component fixes all other node values in it. So, by breaking the graph into connected components, we obtain a set of pairs (x_i, y_i). Each pair indicates that in the component, either x_i vertices have the value 1 or y_i vertices do. For each such pair, consider $d_i = |x_i - y_i|$.

We obtain the knapsack problem with the weights d_i and the capacity of $k - \sum_i min(x_i, y_i)$, where for each element we need to determine whether the knapsack has a solution when taking or not taking this element.

If there are solutions in both cases where the value d_i is included or not included in the knapsack, then no conclusions can be drawn about this component. If, however, the knapsack is unsolvable in one of the two cases, then all elements of the component are definitively fixed.

To check this quickly, we can do the following:

First, we can notice that there would be no more than $m < \sqrt{2n}$ unique values of d_i among the knapsack elements.

Store a bitset of length $k+1$ for each knapsack state, representing whether each value can be achieved. For each of the pairs (number d_i is present in knapsack c_i times) we can update the knapsack in linear time (for small c_i with repeated shift and bitwise OR, for big c_i with linear DP).

If we store some intermediate states (after including 1/2, 3/4, 7/8, etc. of items) in a stack, we can add each knapsack element as the last one in $\mathcal{O}(m \log m)$ linear passes and roll back just by popping a bitset. This idea somewhat resembles the binary lifting trick.

So, overall, we can solve this problem in $\mathcal{O}(n\sqrt{n} \log n)$, which is reasonably fast due to the majority of operations being bitwise. Some other approaches are possible based on the knapsack problem.

There are two cases in which the input data can be contradictory: if one of the graph components has a cycle with an odd number of 0 edges (in which the component can never be satisfied), or if the knapsack has no valid solutions (in which case the sum restriction can never be satisfied).

Problem J. Jolly Wheel

Idea:	Yafim Klimasheuski
Development:	Yafim Klimasheuski
Tutorial:	Yafim Klimasheuski

First, multiply k and all a_i by 2, in order to also represent non-integer points.

Each roll represents the addition of touches over a certain segment, so they can be represented as a combination of adding $\lfloor \frac{a_i}{k} \rfloor$ full rotations, as well as adding 1 on no more than 2 segments.

Next, using a sweep line algorithm, we determine the sums at all points, which allows us to find the answer for each test case in $\mathcal{O}(n \log n)$.

Also, the problem can be solved by additions in a segment tree, either an explicit (after compressing coordinates of points where the wheel changes direction) or implicit one (without needing to compress coordinates). Time complexity is the same.

Problem K. Kitchen

Idea:	Yafim Klimasheuski
Development:	Yafim Klimasheuski
Tutorial:	Yafim Klimasheuski

Cockroaches' strategies can change only when a bait is eaten by someone. Thus said, do a direct simulation of m iterations in $\mathcal{O}(nm)$ each.

For each cockroach, determine the next bait in a linear-time check. Store a priority queue with events "Cockroach i reaches and eats bait j."

When a cockroach reaches the bait, re-check that it wasn't already eaten by another cockroach and that it's still alive. If it can eat the bait, remove the bait, update the cockroach's status, directly update the next bait for all cockroaches and add their new actions into the queue.

Problem L. Layered Caesar Salad

Idea:	Ivan Lukyanov
Development:	Ivan Lukyanov
Tutorial:	Yafim Klimasheuski

Note that $n \le 25 < 26$. Therefore, there is always at least one letter such that none of the given words start with it.

If not all the given words start with the same letter, we can encode them all to start with the same letter, and the restoration would be trivial.

The remaining case is when all the n words start with the same letter. In this case, just use a shift of 1 for all the words. To distinguish between two letters with n words starting with them in the second run, recall that only one of them has words starting with a previous letter in the alphabet.

Problem M. May We Answer Your Questions Right?

Idea:	Yahor Dubovik
Development:	Yahor Dubovik, Dzianis Kim
Tutorial:	Yahor Dubovik

To answer queries of both types, we will implement a segment tree, where each node stores Vasya's sum over its corresponding interval. The merge operation would then be:

$$\text{val}_{\text{new}} = 2^{\text{size}_{\text{left}}} \cdot \text{val}_{\text{right}} + \text{val}_{\text{left}}.$$

Storing these values directly is infeasible, as intermediate results may grow exponentially large. Instead, we represent each value in the form:

$$\text{val}_x = 2^{\text{size}_x} \cdot A + B.$$

That is, we store a pair (A, B), where B is kept as small as possible, effectively representing the "remainder" after factoring out the largest possible power of two consistent with the segment size. Clearly, $|A| \le M$,

so we store A precisely. For B, we clamp it to the range $[-2M, 2M]$, where M is the maximum value of an element: if $B > 2M$, we set $B = 2M$; if $B < -2M$, we set $B = -2M$.

Under this representation, the merge becomes as follows:

$$A_{\text{new}} = A_R, \quad B_{\text{new}} = B_L + 2^{\text{size}_{\text{left}}} \cdot (B_R + A_L).$$

Note that B_{new} may again exceed our clamping bounds. We must therefore adjust it appropriately and possibly re-distribute between A_{new} and B_{new} to maintain the canonical form.

Crucially, our clamping does not significantly distort the result. If B_R is extremely large or small, then B_{new} will remain so as well, and no adjustment to A_{new} is needed.

Otherwise, consider $B_R + A_L$:

- If $B_R + A_L = 0$, then $B_{\text{new}} = B_L$, and we are done.
- Else, if $2^{\text{size}_{\text{left}}}$ is sufficiently large (i.e., exceeds the range of representable values), we can safely set $B_{\text{new}} = \pm\infty$ (i.e., clamp to $\pm 2M$).
- Otherwise, compute B_{new} exactly, then adjust A_{new} and B_{new} by shifting powers of two to bring B_{new} back into $[-2M, 2M]$. Be sure to use wide enough integer types when clamping numbers.

Time complexity is $\mathcal{O}(n + q \log n)$ with some noticeable constant multiplier, memory complexity is $\mathcal{O}(n)$.