Problem A. Anti-Tetris

Problem author: Georgiy Korneev; problem developer: Ilya Zban

The key observation in this problem is solving it backwards. It is much easier to determine if a tile could be placed last on the field, remove it, and greedily solve a lesser problem. If multiple tiles could be placed last on the field, the order of their placement doesn't matter, as removing one tile from the field can't prevent another tile from being able to be placed last among other tiles.

So, the solution is a greedy implementation of this process. On each iteration we can iterate over all remaining tiles, and remove current tile if it can be moved to the top of the field by moving it left, right and up.

Problem B. Building Forest Trails

Problem author: Jakub Onufry Wojtaszczyk; problem developer: Pavel Kunyavskiy

Let's set up a find-union structure on our villages. That allows us to answer queries. The tricky question is to know which groups to join when a road is built.

First, notice that the exact set of roads doesn't matter. We need to know the sets of connected villages. For example, if villages 1, 3, 7, and 9 are connected, it does not matter if the roads are 1-7 and 3-9, or 1-3, 1-7, 1-9, or 1-3, 3-7, 7-9 — if any road intersects any of these sets of roads, it intersects every one (a road will intersect this set if it's an A-B road, and there's a C in the connected set such that $A \leq C \leq B$). So, let's choose a canonical representation, which will connect each vertex to the next one in a set (i.e 1-3, 3-7, and 7-9 in the example above).

Now, we'll store for each vertex how many roads (after replacing all groups with their canonical representation) you would have to cross to get there from a (fictional) village lying between N and 1. For example, if we have one group of 1-3, 3-7, 7-9, and also a road 4-6, then to get to 5, you have to cross 2 roads (3-7 and 4-6), to get to 2, 4, 6 or 8 you cross one road, and everywhere else you don't cross a road. Let's call this number of crossed roads a *level* of a vertex.

Additionally, for each component, we'll keep the leftmost and rightmost village (that is, the smallest and the largest number) that belongs to that component.

We have several cases which we need to join:

- If both ends are in the same component (which can be checked with find-union), nothing changes.
- If the two endpoints are on different levels, then our road will obviously cross some other road. Let's say A has a larger level than B. Then the innermost road (the road closes to A) will need to be crossed (or touched, if B is one of the endpoints of that road). To find that road, we need to have some data structure to answer the query "find me the closest point to the right of A that has a lower value than this one". Then we need to join A with that village, recalculate levels, and repeat this procedure.
- Otherwise, we are joining two components on the same depth. Let's say A is to the left of B (so, R(A) $<$ L(B)). There can be at most one component between them, which is on a smaller level. We can find the next vertex after A, which has a smaller level, by the same query, as above. If it's to the right after B, there is nothing separating A and B, and they can be just joined. Otherwise, there is a single component separating A and B, and both of them should be joined with it. At this point, we need to recalculate levels, and we are done.

Now, we have to perform a "join two components" operation, while updating our data structures. First, we need to join the two components in the find-union. That's easy. The leftmost village of the union is the "more to the left" village of the two, and the rightmost is the "more to the right" one. Levels are the harder part. There are two cases. Joining components are either not intersecting as segments, or one is

nested to another. In the first case, canonical representation for the new component is just a merge of two old ones, with one new edge. In the second case, we need to remove one edge and add two instead. Anyway, there are $O(1)$ segments where the level changes by +1 or -1.

So we need a data structure, which supports adding ± 1 to subsegment, and finding the next value after some position, which is at most the given value. This can be done by segment tree storing subtree minimum. Adding ± 1 is a classical exercise for delayed updated technique. The second one could be done recursively by prioritizing left branches and pruning branches with too big a minimum value or branches with no values after the given one. It can be made to work in $O(\log n)$ time.

Problem C. Cactus Lady and her Cing

Problem author: Vitaly Aksenov, Anton Paramonov; problem developer: Niyaz Nigmatullin

We will tell about the main idea of the solution. Many technical details are not included in this text.

The first fact: in any embedding there exists a path in a given graph that starts in the minimum y coordinate and goes only up, left or right, and reaches the maximum y coordinate of the embedding.

If we know this path, then we can try to place the vertices of this path one by one, going up or left/right, also placing other vertices that are not on the path.

Let's solve for a tree. For every vertex on a path, there exists no more than one other vertex adjacent to it that is not on the path. So placing one vertex requires you to place its adjacent non-path vertex along with its subtree. To check the path one can do two dynamic programming functions: $up(i)$ and $side(i)$. $up(i)$ — try to place all vertices on a path up to v_i: $v_1, v_2, \ldots, v_i$, such that the non-path adjacent vertex of v_i is placed exactly above v_i (that means it's subtree should form a path, and it's placed in a line above v_i), and the value of $up(i)$ — is the minimum possible occupied y coordinate in another column. $side(i)$ — the same, but the non-path adjacent vertex of v_i is either doesn't exist or placed to the left/right of the v_i.

To get a linear solution for a tree, we use this idea and make a subtree dynamic programming, recalculating the same values for a subtree rooted at some vertices. And a standard trick to make this linear.

To make cactus, we can shrink each cycle into a vertex, making everything a tree. Do something to handle how cycles can be placed, and another way to calculate the same dynamic programming function.

Problem D. Dragon Curve

Problem author and developer: Evgeny Kapun

Let's consider two sets of dragon curves (each set contains four curves). We are going to call them order n curves and order $n+1$ curves, although actually both of them are of infinite order. Order $n+1$ curves are the curves described in the problem statement, and order n curves are the same curves rotated by 45° counterclockwise and scaled by a factor of $\sqrt{2}$.

Notice that order n curves can be transformed into order $n+1$ curves by replacing each segment with a pair of segments (each shorter by a factor of $\sqrt{2}$) connected at a right angle. The directions of the newly added turns alternate: the first turn on each curve is left, the next is right, and so on.

Let's consider 2×2 square with even coordinates (that is, with the corners at $(2x, 2y)$ and $(2x+2, 2y+2)$ for some integer x, y). There are two types of such squares: "vertical" squares, where the segments are connected as $(2x, 2y) - (2x+1, 2y+1) - (2x, 2y+2)$ and $(2x+2, 2y) - (2x+1, 2y+1) - (2x+2, 2y+2)$, and "horizontal" squares, where the segments are connected as $(2x, 2y) - (2x+1, 2y+1) - (2x+2, 2y)$ and $(2x, 2y+2) - (2x+1, 2y+1) - (2x+2, 2y+2)$.

Looking at the illustration in the statement, one can easily notice that the squares with $x+y$ even are vertical, and those with $x+y$ odd are horizontal. This can be proven by induction: this can be verified directly for the squares adjacent to the origin, and if it is true for some square containing order $n+1$ segments produced from some order n segment, it will be also true for the squares containing the segments

produced from any adjacent order n segments. As all the order n segments are connected to the origin, this means that the statement is valid for the entire plane.

Now, given the coordinates, it is possible to determine the type of the 2×2 square the segment is in, and map it to the corresponding segment on the (rotated and scaled) order n curve. The illustration given in the problem statement helps very much with this step. The exact transformation is given by this Python code:

```
if ((x ^ y) >> 1) & 1:
    # The square is horizontal.
    x, y = (x >> 1) + (y + 1 >> 1), (y + 1 >> 1) - (x >> 1) - 1
else:
    # The square is vertical.
    x, y = (x + 1 >> 1) + (y >> 1), (y >> 1) - (x + 1 >> 1)
```

The next problem is to track the position within the segment. After the transformation given above is performed, each segment corresponds to two original segments. After the transformation is performed multiple times, each segment corresponds to a number of original segments. Unfortunately, there is no simple rule to determine the direction of the segments (specifically, which end is closer to the origin on the curve). So, a position on a segment can be defined using the following rule: measure the position starting from the end that is "more even". The "evenness" of the point (x, y) can be defined as follows: if $x + y$ is odd, the evenness is 0, otherwise, if both x and y are odd, the evenness is 1, otherwise, both x and y must be even, and the evenness of (x, y) is the evenness of $(x/2, y/2)$ plus 2.

It can be shown that the ends of any segment on an order $n + 1$ curve have different evenness. Also, the transformation from order $n+1$ to order n (rotation by 45° and scaling) reduces the evenness by 1 (points with evenness 0 transform to non-integer points). When transforming from order $n + 1$ to order n, the position on the segment either doesn't change, or gets subtracted from the length of the segment minus one (where the length of the segment is defined as the number of original segments that map to it).

After a transformation is applied repeatedly, the initial segment will eventually map to one of the segments adjacent to the origin. Subsequent transformations will not change it. At this point, the answer can be easily determined. Here is a complete Python implementation (which gets TL, but an equivalent implementation in C++ or Java works very quickly):

```
def dragon_curve(x, y):
    bit, pos = 1, 0
    # Loop until (x, y) is adjacent to the origin.
    while not (-1 <= x <= 0 and -1 <= y <= 0):
        # Increase the length of the segment.
        bit <<= 1
        if (x ^ y ^ (x ^ y) >> 1) & 1:
            # Need to flip the position on the segment.
            pos = bit - 1 - pos
        # This code converts the coordinates.
        if ((x ^ y) >> 1) & 1:
            x, y = (x >> 1) + (y + 1 >> 1), (y + 1 >> 1) - (x >> 1) - 1
        else:
            x, y = (x + 1 >> 1) + (y >> 1), (y >> 1) - (x + 1 >> 1)
    # The index of the curve is determined by the current point.
    # The position on the curve is the position on the current segment.
    # Add 1 because the output is 1-based.
    return (x & 1 ^ y & 3) + 1, pos + 1
```

Problem E. Easy Scheduling

Problem author and developer: Vitaly Aksenov

Consider k such that $2^k < p \le 2^{k+1}$. It can be seen that for the first k rounds there are less than p tasks on the level and the processes perform them level by level. After the first k rounds it can be shown that there exist p ready tasks almost always except, probably, for the last moment of time. Thus, the answer is $k + \lceil \frac{n-(2^{k+1}-1)}{p} \rceil$.

Problem F. Framing Pictures

Problem author: Bruce Merry; problem developers: Bruce Merry, Roman Elizarov

There are two parts to the problem:

1. A geometry part, to partition rotation angles into contiguous ranges within which the bounding (left-most, right-most, top-most and bottom-most) vertices are the same (and identify those vertices).
2. An algebraic part, to determine the contribution of each of those contiguous ranges to the expectation.

The geometric part can be addressed with a slight modification of the standard "rotating calipers" algorithm for finding the longest diagonal. Start by choosing an arbitrary edge and orienting it horizontally. Then incrementally rotate the polygon, pausing whenever an edge incident to a bounding vertex becomes horizontal or vertical and hence the neighbor becomes the new bounding vertex.

For the algebraic part, let the diagonal between the top and bottom bounding vertices have length V and the diagonal between the left and right bounding vertices have length H. Let the range of orientations be parametrized by θ ($\theta_0 \le \theta \le \theta_1$), and the diagonals are oriented such that the bounding box has a shape $V\cos(\alpha+\theta) \times H\cos(\beta+\theta)$. Then the area is

$$VH\cos(\alpha+\theta)\cos(\beta+\theta) = \frac{VH}{2}\left(\cos(\alpha+\beta+2\theta) - \cos(\alpha-\beta)\right),$$

and integrating over θ gives

$$\begin{aligned}&\int_{\theta=\theta_0}^{\theta_1} \frac{VH}{2}\Big(\cos(\alpha+\beta+2\theta) - \cos(\alpha-\beta)\Big)d\theta \\ =&\frac{VH}{4}\Big(\sin(\alpha+\beta+2\theta_1) - \sin(\alpha+\beta+2\theta_0) + 2(\theta_1-\theta_0)\cos(\alpha-\beta)\Big).\end{aligned}$$

This solution requires $O(n)$ time. It is possible to produce $O(n \log n)$-time solutions (for example, by first sorting all the events that occur in the solution above). Such solutions are not significantly simpler than the $O(n)$ solution and are accepted, too.

Problem G. Game of Chance

Problem author and developer: Vasiliy Mokin

The tournament bracket forms a binary tree. For each node we will recursively evaluate the probability of each of the player in the subtree to conquer this node.

Let node v have sons l and r. Assume a player A with strength x has probability p of conquering node l and the subtree of node r contains players with strengths $b_1, \ldots, b_m$ which have probabilities $q_1, \ldots q_m$ of conquering node r respectively. Then the probability of A conquering node v equals $pxf(x)$, where

$$f(x) = \sum_{i=1}^{m} \frac{q_i}{x + b_i}.$$

That means we need to evaluate f with a decent precision for all players in the subtree of node l. Let's expand f into a Taylor series at a point x_0.

$$\begin{aligned} f(x_0 + \Delta x) &= \sum_{i=1}^{m} \frac{q_i}{x_0 + \Delta x + b_i} = \sum_{i=1}^{m} \frac{q_i}{x_0 + b_i} \cdot \frac{1}{1 + \frac{\Delta x}{x_0 + b_i}} = \sum_{i=1}^{m} \frac{q_i}{x_0 + b_i} \sum_{k=0}^{\infty} \left(\frac{-\Delta x}{x_0 + b_i} \right)^k \\ &= \sum_{i=1}^{m} \frac{q_i}{x_0 + b_i} \sum_{k=0}^{\infty} \left(\frac{x_0}{x_0 + b_i} \right)^k \left(\frac{-\Delta x}{x_0} \right)^k = \sum_{k=0}^{\infty} c_k \left(\frac{-\Delta x}{x_0} \right)^k, \end{aligned} \tag{1}$$

where

$$c_0 \geq c_1 \geq \ldots \geq c_k = \sum_{i=1}^{m} \frac{q_i}{x_0 + b_i} \left(\frac{x_0}{x_0 + b_i} \right)^k$$

Notice that if $|\Delta x| \leq \frac{x_0}{2}$, then it is sufficient to take first 50 elements to achieve relative error of no more than $\text{eps} = 10^{-15}$. Now if we choose points $1, 3, 9, \ldots 3^i$ in place of x_0, then the segments $[x_0 - \frac{x_0}{2}, x_0 + \frac{x_0}{2}]$ will cover the whole ray $[1, \infty)$. That means we can calculate f of all of our points using no more than $\log \max a$ points. Solution works in $O(n \cdot \log n \cdot \log \text{eps} \cdot \log \max a)$.

Now we'll check that the obtained precision is sufficient. Let's denote by e_d the maximal possible relative error of the evaluated probability of a player conquering a node which has a subtree of depth d. It's easy to see that $(1 + e_{d+1}) \leq (1 + e_d)^2(1 + \text{eps})$. Thus

$$e_d \leq (1 + \text{eps})^{2^d - 1} - 1 = O(n \cdot \text{eps})$$

Problem H. Higher Order Functions

Problem author and developer: Roman Elizarov

The easiest way to solve this problem is using a recursive descent parser algorithm. The grammar given in the problem statement has only one non-terminal, so the implementation will consist of a single (and simple) recursive function that returns the order of the parsed type.

Let us keep the string to be parsed in the variable s and the index of the currently unparsed character in i. Let's make the following observations in order to write a recursive function that parses a type.

- A valid type always starts with '`(`' character, so we'll just skip it at the start.
- After skipping an opening '`(`' we can look at the current character to see if we are at the unit type.
- We can implement right-to-left associativity of the function type constructor with a '`while`' loop. Every time we loop, we have parsed the T_1 of the type constructor and adjust the current order correspondingly.
- The valid type may end only at the end of the string s or at '`)`' character.

This gives us the following code:

```
fun next(): Int {
    var ans = 0 // the type order to return
    while (true) {
        require(s[i++] == '(') // skip '('
        val cur =
```

```
            if (s[i] == ')') {
                0.also { i++ } // unit type
            } else {
                // recurse and skip the closing ')'
                next().also { require(s[i++] == ')') }
            }
        if (i >= n || s[i] == ')') {
            // cur was the result type of the function
            ans = maxOf(ans, cur)
            break
        }
        // If type is not over, it has to continue with '->'
        require(s[i++] == '-')
        require(s[i++] == '>')
        // cur was the parameter of the function
        ans = maxOf(ans, cur + 1)
    }
    return ans
}
```

Problem I. Interactive Rays

Problem author: Oleg Hristenko; problem developer: Roman Elizarov

Let's make queries at a specific angle α ($0 \le \alpha < 2\pi$) by rounding $(10^6 cos(\alpha), 10^6 sin(\alpha))$ coordinates to the closest integer values and denote the answer to the corresponding query as $f(\alpha)$. This function $f(\alpha)$ has a minimum value of 0 when the ray intersects the circle and the range of α values where this minimum is reached spans at most half the range of α. Also, $f(\alpha)$ has a maximum value that is equal to the distance from the origin $(0, 0)$ to the circle. The $f(\alpha)$ function reaches its maximum value for half the range of α.

Start with making four queries at $\alpha = 0$, $\pi/2$, π, and $3\pi/2$ — at the right angle to each other. Denote the maximum value of those four queries as f_{max} — this is the maximum value of $f(\alpha)$ and the corresponding α as α_{max}. Denote the minimum value of those four queries as f_{min} and the corresponding α as α_{min}.

The value of f_{min} is definitely less than f_{max} but is not necessarily zero. We can show that the α_{min} (at which the minimum value f_{min} is reached) is at least $\pi/8$ away from the range when $f(\alpha)$ reaches its maximum value f_{max}.

Our goal is to find three different values α_1, α_2, and α_3 so that $0 < f(\alpha_i) < f_{max}$. The results of the queries $f(\alpha_i)$ can be then used for a purely geometrical computation of the answer — (x_c, y_c) and r_c.

The general idea is to narrow down the initial range $\alpha_{min} \ldots \alpha_{max}$ to find those three α_i values.

Let's describe the geometrical part, first. The result of each query gives us a line tangent to a circle. With two queries we get two tangent lines and thus we can compute a locus of points (x_c, y_c) as a line that is bisecting the angle between the two tangents. The result of the third query gives us the ability to narrow down the specific (x_c, y_c) and r_c.

In order for this computation to have good numerical stability to produce the answer with the required precision, we need to ensure that the query angles α_i (that we will be using for the geometry part of the solution) are far enough from each other. There are many ways to achieve that.

The simplest one is to bisect the range of angles $\alpha_{min} \ldots \alpha_{max}$ with the binary-search algorithm aiming for the value of $f(\alpha)$ equal to $(f_{max} + f_{min})/2$. While doing this search, record all α values that produce $0 < f(\alpha) < f_{max}$ until we have three of a kind to run the geometry part of the solution.

Problem J. Just Kingdom

Problem author: Jakub Onufry Wojtaszczyk; problem developers: Onufry Wojtaszczyk, Pavel Kunyavskiy

Let's start with a specific $O(n^2 \log n)$ solution, which can be optimized later.

We will find answers for vertices one by one, in the order they are filled. At each moment, we will calculate two items for each vertex. First, the total amount of money distributed to all children which are already done d_v. Second is undone children set S_v ordered by the amount of money they need to receive from the beginning of the process to make any of their descendants done (e.g in std::set or java TreeSet, or heap or something else like that). Note, that because we search for vertices in order they are filled, these values are bigger than answers for all already filled vertices.

How this can be initially calculated? The first one is initially zero, and it's easy. For the second one, we can calculate the first event, and put it to the parent's set. An easy case if a vertex has no unfilled children. Then the event is a vertex itself, and it requires $d_v + m_v$ money to happen. Otherwise, let the first event in children (i.e. first element of this vertex's set) is vertex u and it requires x money to happen. Then the first fill event in that subtree will happen in vertex u. How much money would be required for that? As the answer for u is bigger than for all already filled vertices, they will be filled totally by d_v money, and then each of the other needs to receive x money. So we need $d_v + x \cdot |S_v|$ money received to vertex v totally. Note, this wouldn't overflow because of getting minimum on each step.

Processing one event in $O(height)$ time is quite straightforward. We need to go down into subtree where we got our first event, and when it's done, recalculate d_v, if this subtree is now empty, and add a new event to the set if it's not. This can be done in the same way it's done above.

How this solution can be optimized? The main fact is that an answer for a vertex is at least the product of unfilled children number for all their parents. So, there can be only $O(\log Ans)$ vertices with at least two unfilled children for the next vertex to process. And the answer is not too big, because if we add the sum of all needs then all vertices would be satisfied. So, the only remaining step is "compress" vertices, having only unfilled child to a single edge, and go down of them in once.

The total complexity is $O(n \log n \log sum)$.

Problem K. Kingdom of Islands

Problem author and developer: Danil Sagunov

We note two approaches to this problem. It is easy to note that each of k pairs either deletes an edge from the traditional conflict graph or adds an edge into it. We refer to such pairs as "deleted edges" or "added edges" in the actual conflict graph. Let k_d and k_a denote the number of such edges respectively. The problem question is to find the maximum-sized clique in this graph.

Easy-to-implement approach running in $\mathcal{O}(2^k + n)$.

If the actual conflict graph was equal to traditional, any maximum clique has size p and is formed by taking exactly one arbitrary vertex (jarl) from each part (island).

We first handle the deleted edges and then the added edges.

Each deleted edge uv adds a restriction to this: either u or v is not in the maximum clique. Since there are two options for each edge deletion, in 2^{k_d} we can guess what endpoints of deleted edges are not in the maximum clique. We mark these vertices as deleted from the graph.

We now only handle added edges that are incident to vertices that are not deleted. Each added edge can increase the size of the maximum clique if both its endpoints are taken into it. Then in 2^{k_a} we can guess what added edges are in the maximum clique. When these edges are fixed, then for each part (that has fixed added edges) we should check that the added edges inside this part form a clique. If they do not form a clique, then this fixed guess for added edges is incorrect so we skip it.

When both sets of options are fixed, the size of the maximum clique can be found as the sum of clique

sizes inside parts that contain fixed added edges plus the number of parts that do not contain any added edge but have at least one non-deleted vertex.

Note that all the checks and finding numbers above can be implemented using on-the-go updates working in $\mathcal{O}(1)$: for each part, we need to store

- the number of deleted vertices in this part
- the number of fixed added edges in this part
- the number of vertices that are covered by the fixed added edges

Using the last two values, we can check for the part whether the fixed added edges form a clique inside it in $\mathcal{O}(1)$. We also store

- the number of parts that are completely deleted
- the number of parts that do not contain any fixed added edge
- the number of parts where the fixed added edges do not form a clique
- the number of vertices covered by fixed added edges overall

These values allow to check that the guess is consistent and to find the size of the maximum clique for this guess. To handle these values correctly, one just needs a recursive algorithm that first handles deleted edges and then added edges. For each of them it guesses one of two options, going one step deeper in recursion and updating the values listed above in $\mathcal{O}(1)$.

Reduction approach running in time $\mathcal{O}(C(2k) + k^3 + n)$.

Here $C(t)$ is the running time bound for your favourite clique-finding algorithm in a t-vertex graph sufficient for the given constraints. This can be, for instance, the algorithm listing all maximal cliques in $\mathcal{O}(3^{t/3})$. Applying such algorithm results in $\mathcal{O}(2.09^k + n)$ total running time.

This approach based on graph transformations a.k.a. reduction rules. They allow to simplify the graph while preserving the maximality of the answer. They are provided here without a proof.

Reduction Rule 1. For each part that is not touched by any deleted or added edges, delete vertices of these part from the graph but add an arbitrary vertex from this part to the answer.

This rule is applied in $\mathcal{O}(n)$. We have only $2k$ parts in the remaining graph.

Reduction Rule 2. For each part that contains at least two untouched vertices, delete all untouched vertices from this part except one.

This rule is applied in $\mathcal{O}(k)$. We have only $4k$ vertices in the remaining graph (at most $2k$ vertices are touched, plus at most one untouched vertex per part). We want to improve this down to $2k$.

The following rule is very powerful and can be implemented in $\mathcal{O}(k^3)$. $N(v)$ here denotes the set of neighbours of v in the remaining graph.

Reduction Rule 3. If there are two vertices u, v that are not connected by an edge and $N(u) \subseteq N(v)$, delete u from the graph.

Note that after this rule is applied, parts consisting only of untouched vertices can appear. They are further eliminated by Reduction Rule 1. Note that in the remaining graph each part containing an untouched vertex is touched by at least one added edge and at least one deleted edge. Moreover, a part cannot contain simultaneously a vertex touched only by deleted edges and a vertex touched only by added edges. Hence, per each untouched vertex, there should be an added edge with one endpoint covered by a deleted edge.

Let us estimate the number of remaining vertices. Let $k_{a,2}$ be the number of added edges that do not share any endpoint with any deleted edge. Let $k_{a,1} = k - k_{a,2}$ be the number of other added edges. Then the number of touched vertices is at most $2k_d + 2k_{a,2} + k_{a,1}$.

Since the number of untouched vertices is at most $k_{a,1}$, we have that the total number of vertices in the remaining graph is at most $2k_d + 2k_{a,2} + k_{a,1} + k_{a,1} = 2k_d + 2k_a = 2k$.

It is left to apply your favourite algorithm for finding maximum clique to the remaining graph and append its result to the answer.

Problem L. Labyrinth

Problem author: Evgeny Kapun; problem developer: Egor Kulikov

Let's solve this problem from the end. At each moment of time all our vertices will be divided into several sets, for each set we will know the total width gained from candies in them and maximal starting width, such that we can eat all candies in those vertices if we already ate all candies in all other vertices and ended up in any of vertices from the set. Initially, all sets consist of a single vertex and the maximal starting width is infinite.

Let's process edges in descending order by width. If both ends of current edge are in the same set, we just ignore it. Otherwise suppose total width gain from all candies is $total$. For set i total width gain is tw_i and the maximal starting answer is ans_i, and width of current edge is $width$. Then suppose for current edge it ends are in sets i and j. Then we create a new set k, which is union of sets i and j, set $ans_k = max(min(ans_i, width - (total - w_i)), min(ans_j, width - (total - w_j)))$ and remove sets i and j.

Proof of the correctness of this algorithm is left as an exercise to the reader.

Problem M. The Mind

Problem author and developer: Borys Minaiev

It could be proven that we can always use only pure strategies. It means for a specific hand we can always choose only one turn, and make a move on this specific turn with 100% probability. For some hands with a very big smallest card it is better to not play them at all.

We can also only consider the smallest card from the hand and not use another four numbers at all. Theoretically, it is not always ideal, but for practical applications, it should be enough.

It also could be proven that for two hands A and B with the smallest cards $A_1 \le B_1$ optimal turns to play should satisfy condition $T_A \le T_B$.

So we can describe a solution as tuple $(x_1, x_2, x_3, x_4, x_5)$, where optimal behavior for hand A is:

- If $1 \le A_1 < x_1$ play on turn 1
- If $x_1 \le A_1 < x_2$ play on turn 2
- If $x_2 \le A_1 < x_3$ play on turn 3
- If $x_3 \le A_1 < x_4$ play on turn 4
- If $x_4 \le A_1 < x_5$ play on turn 5
- If $x_5 \le A_1$ do not play at all

The only question left is how to find x_i. We can calculate the probability that a randomly generated hand has minimal card S in it for all possible S. Then for a specific tuple x_i we can calculate average probability and compare it with the required 85%.

We can find optimal x_i using dynamic programming. Another possible way is just to split numbers into six groups with an almost equal sum of probabilities.