ACM ICPC World Finals 2019

Solution sketches

Disclaimer *This is an unofficial analysis of some possible ways to solve the problems of the ACM ICPC World Finals 2019. They are not intended to give a complete solution, but rather to outline some approach that can be used to solve the problem. If some of the terminology or algorithms mentioned below are not familiar to you, your favorite search engine should be able to help. If you find an error, please send an e-mail to* `austrin@kth.se` *about it.*

— Per Austrin, Bruce Merry, and Jakub Onufry Wojtaszczyk

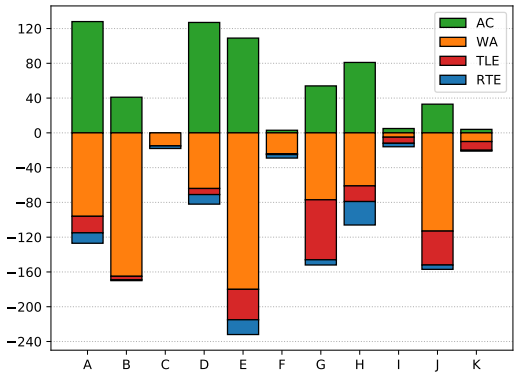
Summary

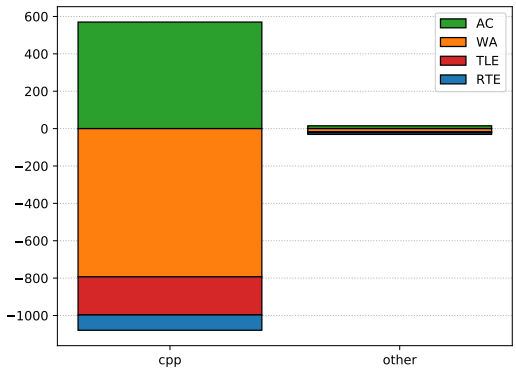
The contest started out with the University of Warsaw making an amazing start, at one point being in the lead with 5 problems solved while the team in second place still only had 2 problems solved. But after solving seven problems in the first two hours, they slowed down, and ultimately ended up with 8 solved in 4th place.

The last hour of the contest was very exciting in the judge's room (and for careful viewers of ICPC Live as well), with MIT and Moscow State battling for the first place: 3 minutes into the last hour, MIT took the lead by solving K in the first attempt, but a few minutes later Moscow State solved both I and K in quick succession, putting them ahead by one problem. However their penalty time was pretty high due to several incorrect attempts on K, so MIT had a window of 30 minutes to solve another problem and take the lead. With just 30 seconds remaining of that window, they submitted a correct submission on I and regained first place, with the same number of problems solved and a single penalty minute less than Moscow. However, their lead was short-lived: just a minute later, 21 minutes before the end of the contest, Moscow submitted a correct solution on F, reaching 10 solved problems, which earned them the victory.

Congratulations to Moscow State University
for successfully defending their title as the ICPC World Champions!

As general statistics, here are two graphs showing the number of submissions made for each problem and for each programming language, respectively. The positive y axis has the number of accepted solutions made, and the negative y axis has the number of rejected solutions made.





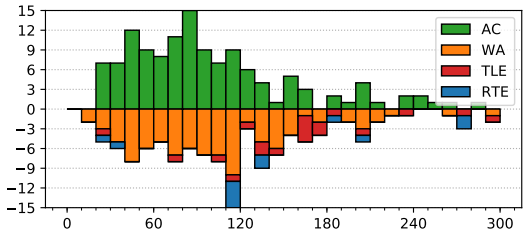
As can be seen, the most popular language was (as usual) C++, this year by an even wider margin than usual: more than 95% of all submissions were made in C++.

A note about solution sizes: below the size of the smallest judge and team solutions for each problem are given. These numbers are just there to give an indication of the order of magnitude. The judge solutions were not written to be minimal (though some of us may overcompactify our code) and can trivially be made shorter by removing spacing, renaming variables, and so on. And of course the same goes for the code written by the teams during the contest!

Explanation of activity graphs: below, for each problem a team activity graph is shown. This graph shows the number of submissions of different types made over the course of the contest: the x axis is the time in minutes, the positive y axis has the number of accepted solutions made, and the negative y axis has the number of rejected solutions made.

Problem A: Azulejos

Solved by 128 teams.
First solved after 21 minutes.
Shortest team solution: 1458 bytes.
Shortest judge solution: 1783 bytes.



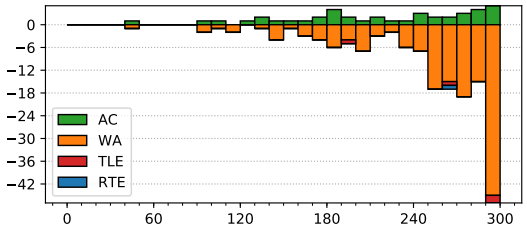
This was one of the easiest problems, having a pretty natural greedy solution, but it still required a bit of datastructure work.

Suppose that there are a front-row tiles tied for cheapest, and b in the back row, and assume without loss of generality that $a \leq b$. We need to decide which a of these b back-row tiles should be matched with the front row. We can do this greedily, always taking the shortest possible back-row tile for each front-row tile, which ensures that the left-overs are as tall as possible. Having done this, we can now ignore the left-most a positions and their tiles, and solve the problem again with the remaining tiles.

Finding the shortest available tile for each spot requires some form of query structure such as a balanced binary tree, which results in an $O(n \log n)$ running time. C++, Java and Kotlin all provide built-in classes for this, but solving this problem in Python requires a bit more manual labor.

Problem B: Beautiful Bridges

Solved by 41 teams.
First solved after 44 minutes.
Shortest team solution: 988 bytes.
Shortest judge solution: 590 bytes.

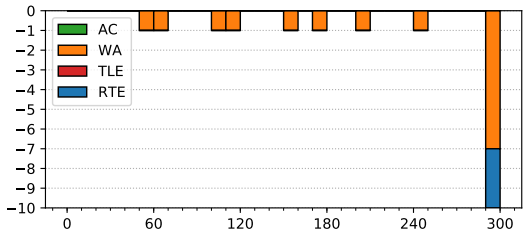


This problem was not as hard as the teams made it out to be, but the presence of a geometry component made it look a bit scary. There is a simple but too slow $O(n^3)$-time solution using quite standard dynamic programming: find the cost to cover the first i points of the ground profile (with a pillar on the i^{th} point), by considering all positions $j < i$ for the second-last pillar. One must then check that ground profile between j and i does not intersect the arch.

We improve this with some precomputation. Suppose there is a pillar at point i, and consider expanding the arch to the right of it. The left half of the arch will require increasingly more space, until it cannot grow further without intersecting the ground. Similarly, we can determine an upper bound for the size of the arch to the left of i for the right half of that arch not to hit the ground. By combining this information, we can determine in $O(1)$ whether any given arch is valid, allowing the dynamic programming to be performed in $O(n^2)$.

Problem C: Checks Post Facto

Solved by 0 teams.
Shortest judge solution: 3955 bytes.



While the judges figured this for the third hardest problem, it turned out to be the hardest, and only unsolved, problem. The problem is mostly a tricky implementation problem. We can start constructing a solution iteratively: start with an empty board, and play through the moves until an inconsistency is found; then go back and add/modify a piece until everything works out, according to the following conditions:

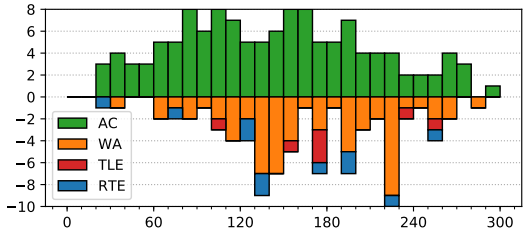
- If there is no piece where a piece is meant to move from, add a man.
- If a man moves backwards without having been promoted, replace it (at the start) with a king.
- If a piece captures but there is nothing to jump over, add a man there (of appropriate color).

These conditions used the following key observation: there is never any benefit to placing a king when a man would suffice, since a king might only force more choices later.

After the above has converged, there is still the problem of forced captures. If at some point a player does not take a forced capture, there must be another piece blocking it. It might as well be a man, but the color is unknown. At this point one can just try both options (recursively), and stop once one has found a solution that doesn't require more blocking pieces and does not contain a contradiction. While there are 32 spaces on the board, in practice this completes very quickly as it is difficult to set up situations where large numbers of choices need to be made correctly.

Problem D: Circular DNA

Solved by 127 teams.
First solved after 27 minutes.
Shortest team solution: 876 bytes.
Shortest judge solution: 793 bytes.

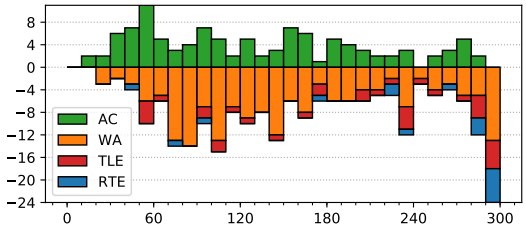


Start by considering only one gene type i. If the number of $\mathtt{s}_i$ and $\mathtt{e}_i$ differs, then gene type i clearly cannot form properly a nested sequences, so we will ignore those and only focus on the i with the same number of $\mathtt{s}_i$ and $\mathtt{e}_i$.

Now consider reading clockwise from the cut point, incrementing a counter at every $\mathtt{s}_i$ and decrementing it at every $\mathtt{e}_i$. The sequence will be properly nested if and only if the lowest value of the counter occurs at the cut point (possibly tied). We can start with an arbitrary cut point and a counter for each gene type initialized to zero, and determine the minimum counter value for each gene type. This will tell us the score for that initial cut point. We can now incrementally move the cut point around the circle: each time it moves, one gene type might become properly nested or cease to be properly nested, depending on the new counter value for the type of the marker we just moved past.

Problem E: Dead-End Detector

Solved by 109 teams.
First solved after 14 minutes.
Shortest team solution: 1245 bytes.
Shortest judge solution: 1048 bytes.

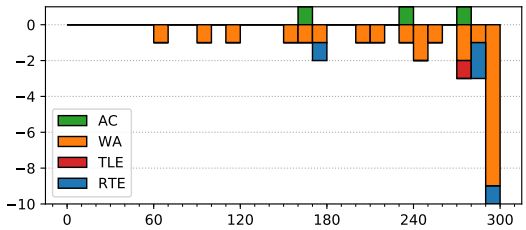


Each connected component of the graph is independent, so we will consider only a single component. If it is a tree, then every single edge is a dead end, but for any street from u to v where u is not a leaf, there is another dead end street from some w to u, making the dead-end sign from u to v redundant. So the only signs to place are those from u to v where u is a leaf.

If a component is not a tree, not all streets are dead ends. In general, a street from u to v is a dead end if, removing the edge between u and v in the graph, this splits the graph into two connected components and the component containing v is a tree. We could identify these edges by finding biconnected components/bridges in the graph, but this is overkill. The component will consist of some "core" of non-deadends, along with trees hanging off the core. We can identify these trees by repeatedly removing leaves until none are left. Any street from a non-removed vertex to a removed vertex gets a dead-end marker.

Problem F: Directing Rainfall

Solved by 3 teams.
First solved after 168 minutes.
Shortest team solution: 3309 bytes.
Shortest judge solution: 3384 bytes.



The solution to the problem has two essentially independent components: a geometric one, where we translate the problem to a non-geometric data-structure problem, and a data structure one, where we actually solve the problem.

The first, geometric part, consists of sorting the tarps in such a way that when water drops from one tarp I to another tarp J, tarp J appears before tarp I. Equivalently, we want a linear extension (a.k.a. topological ordering) of the following partial order (a.k.a. directed acyclic graph): $I < J$ if there exists some x such that (x, y_I) is on I, (x, y_J) is on J, and $y_I < y_J$. This sorting can be done by a somewhat difficult, but standard, sweep-line algorithm. At any given x, we will maintain a set of the tarps that intersect the vertical line at x, ordered by the y-coordinate of the intersection. This set will also contain two artificial tarps, one for the earth, and one for the sky. Also, with each tarp I in the set we will maintain an ordered list of tarps that have already finished, and need to be before I, but after the predecessor of I in our set; this list will be in correct order. Thus, when we finish, we will be left only with the earth and the sky, and all the tarps will be in the correct order on the list attached to the sky. This can be maintained in amortized logarithmic time when moving to the next interesting x.

Having sorted the tarps so that we know the order in which water flows between the tarps, the problem becomes a non-geometric data structure problem that happens on a one-dimensional line. We will have directed intervals (pieces of tarp directed towards the lower end of the tarp) arriving one by one (from lower to higher), and an integer value $V(x)$ at each point x of the line. The effect of an interval $[a, b]$ is to replace the value $V(x)$ on each point $x \in [a, b]$ by $V'(x) = \min(V(b), \min_{z \in [x,b]} 1 + V(z))$ if it is directed towards b, or by $V'(x) = \min(V(a), \min_{z \in [a,x]} 1 + V(z))$ if it is directed towards a. We want a data structure

that allows us to provide initial values for V, update V when intervals arrive, and query for the minimum of V on an interval.

When applying this data structure to the tarps, we get the solution to the problem: we set the initial values $V(x)$ to be 0 within our field $[\ell, r]$, and ∞ outside. When adding the intervals one by one in the order produced by the geometric sorting phase, this maintains the invariant that $V(x)$ equals the minimum cost of getting water from the point (x, ∞) to our field on the subinstance consisting only of the pieces of tarp added so far. Thus, after processing all the pieces of tarp, V will represent the cost of getting rain falling from the sky (and hitting the first thing on its path) to our field, and so the final output will be the minimum of V on the interval $[\ell, r]$.

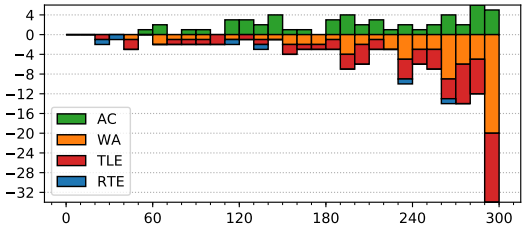
To implement the desired data structure, we use the following facts about the function V:

- Its values are all integers.
- It is piecewise constant (that is, the real numbers can be divided into a finite set of intervals, and V is constant on each interval).
- It is bitonic. That is, it is non-increasing up to some point, and then non-decreasing.

This means that to represent and update V, we can use a simple data structure that contains all the points where V changes value and by how much the value changes. Storing this in two sets (one for the non-increasing part, one for the non-decreasing) will allow amortized logarithmic-time updates. At the end, we can perform a linear scan to obtain the final value. (The observation that the function is bitonic is not strictly necessary, but it simplifies the implementation.)

Problem G: First of Her Name

Solved by 54 teams.
First solved after 55 minutes.
Shortest team solution: 845 bytes.
Shortest judge solution: 1300 bytes.

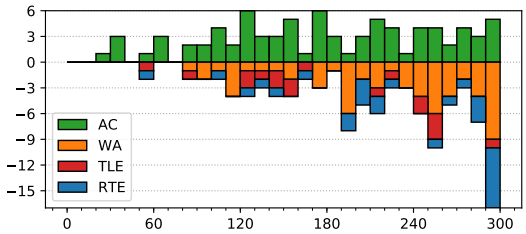


There are at least two possible ways to solve this problem. The first, which was taken by most of the judges who solved the problem, is to sort the Royal Ladies. While a naïve sort could take $O(n^2 \log n)$ time, one can use the same doubling approach normally used for building suffix arrays to perform the sort in $O(n \log n)$ time (or $O(n \log^2 n)$ time if one uses a standard built-in sort instead of implementing counting sort). After this, binary search for each query in the sorted list; if the total length of the queries is L, then this part requires $O(L \log n)$ time.

In the other approach, we start by reversing all strings, so that each Lady's name is formed by appending a letter to her mother's name (rather than prepending), and the queries are for suffixes. The queries are all placed into a trie, with suffix links as in the Aho-Corasick algorithm. Now for each Lady in turn one can walk the trie to find the longest suffix that matches a node in the trie. There are a few more details to work out regarding query strings that are suffixes of other query strings, but overall the algorithm requires $O(n)$ for a fixed-size alphabet.

Problem H: Hobson's Trains

Solved by 81 teams.
First solved after 28 minutes.
Shortest team solution: 1165 bytes.
Shortest judge solution: 1814 bytes.



The problem can be solved in linear time. The structure of this type of graph is well-known: a collection of rings with trees hanging off the vertices. Call a node a "ring node" if it is part of a ring, otherwise a "tree node". We will solve the problem separately for ring nodes and tree nodes (identifying the rings in linear time is also a standard problem).

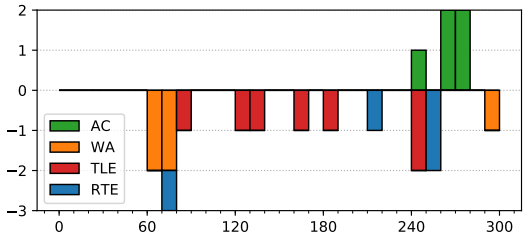
First we solve the problem for tree nodes. Since tree nodes can only be reached from their subtree, we can solve this independently for each tree rooted at a ring node (the answer we get for the root will be wrong since it ignores the ring, but we fix that later). Firstly, we can identify the k^{th} ancestor of every node: during a recursive walk of the tree, keep a stack of ancestors, and look back k entries when visiting a node to find its k^{th} ancestor (it might not exist if the node is too shallow). From these pointers one can trivially compute f_i, the number of k^{th}-level descendants of i, that is, those that can reach i in exactly k legs. We actually want g_i, the number of descendants at most k legs away. But we can compute this recursively by summing g_i over the children, subtracting the sum of f_i over the children (which will be $k+1$ legs away), and adding 1 for the node itself.

This ignores journeys from a node in one tree to a ring node other than the root. From each node, the reachable ring nodes form a contiguous arc of the ring. Thus, if we encode the number of such journeys to each ring node as a set of adjacent differences, we can update this structure in $O(1)$ per source node.

The problem can also be solved using mergable heaps. For each vertex i, the program maintains a heap of nodes which can reach vertex i in k or less steps. The heap is sorted by distance from i to allow easy removal after incrementing the distances of vertices in the heap. By sweeping from the leaves of the "tree nodes" one can update the parent node by incrementing all values in the heap by 1 and then merging its heap with its parent's heap. The merge can be done in $O(\log n)$ with a mergable heap (e.g., a leftist tree or a binomial heap) or in amortized $O(\log^2 n)$ by merging the smaller heap into the larger one. After solving the trees, one can do the same calculation for the cycle. However, this approach overcounts nodes that traverse the entire cycle once before reaching vertex i. This can be resolved by spinning around the cycle a second time to calculate the overcounted nodes and removing them from the total. The resulting runtime is $O(n \log n)$ for mergable heaps and $O(n \log^2 n)$ for a binary heap. Both solutions have a comparable empirical runtime to the linear time solution.

Problem I: Karel the Robot

Solved by 5 teams.
First solved after 245 minutes.
Shortest team solution: 3792 bytes.
Shortest judge solution: 3052 bytes.



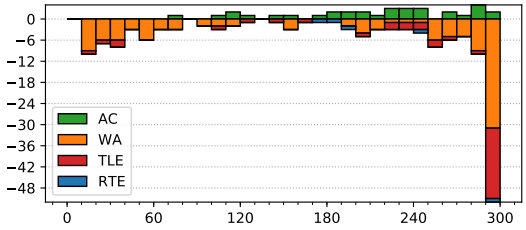
This was probably the problem that the judges underestimated the most. Like the also underestimated problem C, it is mostly an implementation challenge, but not a very painful one. There are two parts to the problem, both quite standard. The first part is to parse the input

programs and to be able to simulate it in a naïve step-by-step way. Since a program may run for a very large number of steps without going into an infinite loop, this will be very slow. So the second part is to speed up the simulation, which we can do with dynamic programming. There are only $4rc \leq 6400$ possible configurations (position in the grid and heading) that the robot can be in. And there is only some number $s \leq 36 \cdot 100$ possible positions in the code that the execution of the program can be in. Thus there are in total at most $4rcs \lessapprox 2 \cdot 10^7$ possible states that the simulation can be in, and by memoizing the result of each state we get an $O(rcs)$-time solution. The time limits were generous enough that the memoization could also be done with some dictionary data structure incurring an extra log factor overhead.

The running time can be improved noticeably by only considering the positions in the code that are branch points (that is, `i` and `u` commands) and function entry points when memoizing. Between such points the simulation progresses linearly, and while the number of steps taken does not improve when not memoizing them, not caching every single step is much more cache-friendly.

Problem J: Miniature Golf

Solved by 33 teams.
First solved after 72 minutes.
Shortest team solution: 1073 bytes.
Shortest judge solution: 1195 bytes.



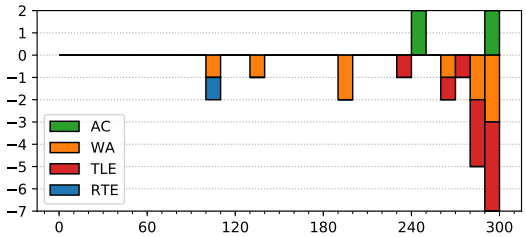
The players' best ranks are determined one player at a time. Let i be the player whose best possible rank we wish to determine. For each other player $j \neq i$, we find the intervals of ℓ for which player i strictly beats player j. This can be done by sweeping ℓ from 0 to infinity, stopping each time it equals a score of either of the players. Between these points, both players' scores vary linearly, so it is straightforward to identify the critical values of ℓ at which player i starts or stops beating j.

Once these intervals have been found for all $j \neq i$, one needs to identify the maximum number of overlapping intervals, which can be done by a second sweep counting the overlap depth.

There can be at most $O(p^2h)$ values of ℓ at which two players exchange ranks, or $O(ph)$ values of ℓ at which a single player changes rank. For each player these $O(ph)$ values need to be sorted for the second sweep, leading to a total of $O(p^2h \log(ph))$ time for the algorithm described above. In fact, this can be slightly reduced to $O(p^2h \log p + ph \log h)$ by noting that these events start as p already-sorted lists, but this was not needed to pass.

Problem K: Traffic Blights

Solved by 4 teams.
First solved after 243 minutes.
Shortest team solution: 1971 bytes.
Shortest judge solution: 961 bytes.



This was one of the hardest problems in the set, and it has a very nice solution based on an approach that could ostensibly be called "divide and conquer" or "meet in the middle".

First, it is easy to see that the system of traffic lights is periodic, with some period $P =$

$\operatorname{lcm}(p_1, \ldots, p_n)$, where $p_i = r_i + g_i$ is the period of the i^{th} traffic light. So there is a naïve solution to the problem in time $\Omega(nP)$ by checking for each time $t \in \{0, \ldots, P-1\}$ what would happen to a car arriving at time t. But P can be as large as $\operatorname{lcm}(1, \ldots, 100) \approx 10^{40}$ so this has no chance of running in time.

Next, if all pairs of periods p_i and p_j were relatively prime then the problem would be easier, because now the probabilities that a car passes light i and light j are independent (this follows from the Chinese Remainder Theorem), so in order to compute the probability that a car passes some set of lights we just have to multiply the probabilities of passing the individual lights, and we could solve the problem in $O(n)$ time. Extending this slightly, if some pairs of lights have the same period $p_i = p_j$ rather than being relatively prime, then we can also solve the problem rather easily, say in $O(np)$ time (where $p \leq 100$ is the maximum period length of any individual light) by, for each period, keeping track of which times modulo that period would result in having stopped at a red light. As yet another small extension of this idea, if the periods of some pairs of lights are divisible by each other, the same solution applies (e.g. if one light is green at times 2–3 modulo 5, and another is green at times 7–11 modulo 15, we can think of the first one as instead being a light which is green at times 2–3, 7–8, and 12–13 modulo 15 so that both have the same period).

These small observations let us deal with some situations but in general the periods will have some shared common factors that will mess things up (e.g. if $p_1 = 64$ and $p_2 = 92$ then $\gcd(p_1, p_2) = 4$ and the periods are clearly not divisible by each other).

We now get to the key idea of the solution. Let us pick some number X and for each time $t \in \{0, \ldots, X-1\}$ compute the answer for cars arriving at time t modulo X, that is, times t, $t+X$, $t+2X$, etc. When restricted to these times, a traffic light with period p_i will still behave in a periodic way, but with period $p_i / \gcd(X, p_i)$. These "reduced periods" are in general smaller than the original periods, and the hope is that maybe they are sufficiently reduced that the small common factors are eliminated and the "all periods relatively prime or divisible by each others" solution from above can be applied. So how large do we need to make X? A sufficient condition for all the reduced periods being relatively prime or divisible by each other is that they are all prime powers, so we can choose the smallest X with the property that $p / \gcd(p, X)$ is a prime power for all $1 \leq p \leq 100$. It turns out that the smallest such X is quite small, namely $X = 2^3 \cdot 3^2 \cdot 5 \cdot 7 = 2520$ (which can be figured out by some pen and paper work, or in an engineering way by writing a small program to try all X).

All in all, this leads to an $O(X \cdot n \cdot p)$ time algorithm, which for $X \leq 2520$, $n \leq 500$, and $p \leq 100$ is reasonably fast. The judges considered increasing the limit of p up to 200, because while growing exponentially in p, the value of X at that point is still only 27720 (you need to add in a factor 11 from the previous bound). But we decided that this problem was hard enough as it is, and that it was probably more approachable with the bound of 100.