ACM ICPC World Finals 2016

Solution sketches

Disclaimer *This is an unofficial analysis of some possible ways to solve the problems of the ACM ICPC World Finals 2016. 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 and Jakub Onufry Wojtaszczyk

Summary

This year provided some exciting behind-the-scenes action that we fortunately managed to keep away from the teams during the contest. Long story short, at 10 in the evening the night before the contest, we realized that one of the problems was irreparably broken – four different judges had come up with precisely the same natural but very subtly incorrect algorithm, we had no correct solutions, and the problem suddenly looked *a lot* harder. Fortunately, we had preparations for such eventualities, and we had another problem ready (J – Spin Doctor) that was a somewhat suitable swap-in for the broken problem. Several hours later, the judges had become very skilled at unstapling problem sets, replacing pages, and re-stapling them. This last minute change made the problem set harder than we had originally intended, and it would probably have been sufficient to stick with 12 problems and not swap in the new problem.

We'll see if we can figure out how to solve the broken problem until next year, maybe you will some day see it in some form...

Congratulations to St. Petersburg State University, the 2016 ICPC World Champions!
In terms of number of teams that ended up solving each problem, the numbers were:

Problem	A	B	C	D	E	F	G	H	I	J	K	L	M
Solved	36	50	128	51	92	12	96	0	1	2	66	108	8
Submissions	123	192	160	87	357	32	416	1	22	43	203	402	26

In total there were 650 Accepted submissions, 1210 Wrong Answer submissions, 158 Time Limit Exceeded submissions and 46 Run Time Errors. The most popular language was C++ by a wide margin: 1904 submissions compared to 161 for Java. There were exactly zero submissions in C, but on the other hand all of them were accepted.

A note about solution sizes: below the size of the smallest judge and team solutions for each problem is stated. It should be mentioned that these numbers are just there to give an indication of the order of magnitude. The judge solutions were not written to be minimal and it is trivial to make them 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!

Problem A: Balanced Diet

Shortest judge solution: 643 bytes. Shortest team solution (during contest): 741 bytes.

The solution to this problem will simulate Danny's optimal behavior for as long as possible, to find the answer. The first issue to deal with is the "forever" answer — for how long do we need to simulate before we can safely claim that Danny will be able to eat sweets forever.

Let A denote the sum of all a_i. Notice that if at whatever point Danny has n sweets, where n is positive and divisible by A, then the numbers $nf_i - 1$ and $nf_i + 1$ are integers. Thus, there is only one possible choice of the number of sweets of any given type, for all the types, and so $s_i = nf_i$. If we extend the sequence cyclically (that is, the $n + 1^{st}$ sweet is equal to the 1st, the $n + 2^{nd}$ is equal to the second, and so on), then it will always be balanced — because in the inequalities, all three expressions increase by nf_i.

So, we need to simulate just to the nearest multiple of A. Since A is at most 10^5, this means we need to be able to simulate a single step (single choice of a sweet) in logarithmic time.

This, in its heart, is a scheduling problem. For any sweet type i, there is some minimum total number of sweets we need to have to be able to buy the k^{th} sweet of that type (let's call that number $L(i,k)$), and some maximum total number of sweets, where if we did not buy the k^{th} sweet of type i and we already have this number of sweets, then our diet is already unbalanced (let's call that number $U(i,k)$). The solution (as usually with scheduling problems) is to always take the element which has the earliest deadline — so, in this case, the sweet with the lowest $U(i,k)$ value. The proof is a relatively standard exchange argument, which we leave to the reader.

Implementation-wise, we can hold two priority queues — the queue of sweets we can't yet buy, ordered by $L(i,k)$, and the queue of sweets we can buy, ordered by $U(i,k)$. Notice that we know up front which sweets we will need to buy to get up to A, so we can begin by inserting them all into the appropriate queues. At each day, begin by moving sweets that we can now buy from the first queue to the second, and then pick the first sweet from the second queue and buy it. If it turns out that is was already expired when we buy it, the previous day was the last we could have had a balanced diet, otherwise we proceed.

One simplification that we can apply to this is that the first queue is in fact not required: if we assume all the sweets are available immediately, the solution may start constructing unbalanced sequences, but it will not be able to extend the sequence to a larger number of days, so the answer remains the same. The proof is rather standard, but a bit technical, so we will skip it.

Problem B: Branch Assignment

Shortest judge solution: 1310 bytes. Shortest team solution (during contest): 1359 bytes.

This problem has two rather independent components. The first component is to compute the shortest paths between the headquarters and every branch (in both directions). This part is completely standard and can be solved using Dijkstra's algorithm.

The second part is to find an optimal partitioning of branches into subgroups. First let us formulate the problem more concretely. For branch i, let a_i denote the distance from headquarters to branch i, and b_i the distance from branch i to headquarters. Then, if we assign a set

$G \subseteq \{1, \ldots, b\}$ of branches to the same sub-projects, the total distance travelled to deliver the messages within this group equals $(|G| - 1) \sum_{i \in G} a_i + b_i$ (because each branch needs to send messages to and receive messages from the $|G| - 1$ other branches in the group). Since this only depends on the sum $a_i + b_i$, let us write $c_i = a_i + b_i$ for brevity.

In other words, our goal is to partition $\{1, \ldots, b\}$ into s subsets $G_1, \ldots, G_s$ such that $\sum_{j=1}^{s} (|S_j| - 1) \sum_{i \in S_j} c_i$ is minimized. This can be solved using dynamic programming.

Another way of phrasing the objective we are minimizing is $\sum_{i=1}^{b} c_i \cdot (g(i) - 1)$, where $g(i)$ denotes the size of the group to which we allocate branch i. From this, we can observe that if $c_i < c_j$ for some branches i and j, then we should have $g(j) \leq g(i)$ (otherwise, we can swap groups for i and j and get a better assignment). This also implies that we can assume without loss of generality that the smallest group contains the t_1 branches with largest c_i values (for some t_1), the second smallest group contains the t_2 branches with the remaining largest c_i values, and so on.

This gives a natural dynamic programming algorithm where we define $D(i, j)$ to be the minimum distance of partitioning the i largest branches into j different groups. A straight forward dynamic programming algorithm using these ideas might use the recursive identity

$$D(i, j) = \min_{1 \leq k \leq i} D(i - k, j - 1) + (k - 1) \sum_{i - k < \ell \leq i} c_\ell \tag{1}$$

representing that we can try all possibilities for the size k of the last group. However, this leads to an $\Omega(b^2 s)$ running time which is too slow. The final observation to make this fast enough is almost immediate given our observations and setup so far (but depending on exactly how one arrives at and formulates the $b^2 s$ time solution, the last step may be hard to see): since we may assume that the group using the largest elements is the smallest groups, we may bound k in (1) above by i/j instead of i. This little change changes the running time of the resulting dynamic program to $O(\sum_{i=1}^{b} \sum_{j=1}^{s} i/j) = O(b^2 \log s)$.

Problem C: Ceiling Function

Shortest judge solution: 829 bytes. Shortest team solution (during contest): 550 bytes.

This was clearly the easiest problem in the problemset. What we were asked to do was to build an unbalanced binary tree out of each sequence of numbers we were given, and then group the trees by shape (and report the number of groups).

Forming the trees is simply simulation of the process. To compare the shape of the two trees, the easiest thing is to do it recursively: if both trees are empty, they are of equal shape, if only one is empty, they are not of equal shape, and if they are both non-empty, they are equal if both the left and right subtrees are equal.

For the grouping, the limits are low enough we can afford to do it quadratically — for each tree, check whether it is the first of this shape in the sequence, and if yes, increment the answer by one.

Problem D: Clock Breaking

Shortest judge solution: 2027 bytes. Shortest team solution (during contest): 2155 bytes.

First, notice that if a segment is on at one time, and off at another, it has to be working (or the answer will be "impossible") — it can be neither burnt in nor burnt out. On the other hand, if a segment shows the same value in all the displays in the input, it is possible that it is burnt (in or out, depending on the value), and it gives us no information as to what time the clock is really showing.

So, the solution will first identify the working segments. There are $24 \cdot 60 = 1440$ different minutes the first display could possibly be showing. So, we can try them all, and check whether they match all the displays we are given in all the places where we know we have working segments. This way we get a list of possible starting times for the displays. If this list is empty, the answer will be "impossible".

Otherwise, we already know the segments which are definitely working. For all the others, we need to find out if they are definitely burnt in / out, or is it possible that they are working correctly, but the same value happens to be displayed throughout the whole time range. For this, we need to — for each segment that always displays the same value — iterate over all the possible start times, and check if for at least one of them all the displays would indeed legitimately show that value. We have at most 1440 possible times, 100 displays and 28 segments, so there isn't a real risk of running into problems with the time limit.

This problem needs care in implementation — you need to hard-code the shapes of all the digits, the positions of all the segments, take into account that clocks do not display leading zeroes for hours, but they do for minutes, and handle the wrapping of time around midnight. Which is why this problem did not attract too many solutions during the contest.

Problem E: Forever Young

Shortest judge solution: 826 bytes. Shortest team solution (during contest): 836 bytes.

The most naive algorithm would be to simply try all possible bases b (after a moment's thought one sees that b can be at most $y - 1$) and see which ones yield only decimal digits. However, the numbers are too big for this to work.

The key observation is that *either* b must be small, or y written in base b must be small. In particular, if we let d denote the number of digits of y written in the optimal base b, we have that $b^{d-1} \leq y$. With $y \leq 10^{18}$, this implies that either $b < 10^5$, or $d < 5$. Thus we can check all values of b up to 100 000, and all 4-digit numbers x as the value of y in base b, and see what yields the best solution (this requires a routine for computing b given y in base 10 and y in base b, which is easiest done by binary search over b though one has to be careful about overflows).

In general, for n-bit numbers, the time complexity of this algorithm is $2^{O(\sqrt{n})}$. There is also a polynomial time algorithm for this problem (we leave this as an exercise).

Problem F: Longest Rivers

Shortest judge solution: 1220 bytes. Shortest team solution (during contest): 1888 bytes.

This was conceptually one of the hardest problems in the problemset, although the implementation was actually relatively simple.

Let's begin by focusing on a single river. So, we pick a river and want to push it as high up the ranklist as possible. Obviously, we always choose this river's name at any confluence, all the way to the sea. After this, our river has some length L. Now, the problem is to choose other river names at confluences, so that as few rivers as possible end up longer than L.

Let's call a river *long* if it's longer than L, and *short* if it's not. We aim to minimize the number of long rivers (which is equivalent to minimizing the places where a short river becomes long). Let's look at any confluence not involving our chosen river.

- If at least one of the rivers entering the confluence is long, then we can safely choose it to continue (and terminate the other river).
- If both rivers are short, we can always choose the shorter of the two.

This gives us an $O(n^2)$ solution to the problem — for each river, calculate its length, and then apply the algorithm above to determine the length of all other rivers.

Now, let's work on making this work for all rivers at one go.

First, observe that the value L for any given river is simply the distance from the source of this river to the sea. This can be calculated by a standard tree recursion going from the sea in $O(n)$ time for all the rivers.

Now, for each river R, we aim to answer the question "If we always choose R as the river at the confluences on its path to the sea, how many rivers of length larger than L do we have to form?". This question is difficult to answer for all the rivers at the same time. However, it is actually equivalent to the question "How many rivers of length larger than L do we have to form?" with no additional constraints. To see this, take the optimal naming choice to get as few as possible rivers longer than L described above. Take any confluence where we did not choose the name R. Notice that if we choose the name R at this confluence instead, the number of long rivers does not grow — since no matter how many times we choose R, it is not going to become longer than L, and the lengths of other rivers could only have decreased. Thus, what we have to answer is "how many rivers longer than L do we have to form?" for each value L. We will answer all such questions at one go, ordered by increasing L.

At each time we will keep the current value L. Additionally, for each confluence point and each river source, we will remember it to be in one of the three states:

- At least one of the rivers entering this confluence is long (river sources cannot be in this state). This confluence is irrelevant from the point of view of rivers *starting* to be long.
- Both rivers entering this confluence are short, or it's a river source, but the resulting river will become long before it reaches the next confluence point.
- Both rivers entering this confluence are short, or it's a river source, and the resulting river is still short when it reaches the next confluence point.

In the second and third cases, we choose the shorter of the two rivers entering the confluence to continue.

We begin with $L = 0$. All river sources are in the second state, all the confluences are in the first state, and we have N long rivers in the system.

When L grows, the only thing that changes is that rivers that were long with the previous value of L can now become short, possibly changing the state of some confluence point or river source from the second to the third; this can in turn change the state of some confluence from the first to the second case. This, in turn, can lead us to reconsider the choice of the river we continue in the confluence point (and possibly trickle down to changing the state of a confluence downstream, and so on).

So, we will do the following. We will keep the states as described above. Additionally, for each point in the second state, we will identify the value of L at which it will flip over to the third state, and keep a priority queue of those points. We repeatedly pick the lowest L value, flip the state of the point, and propagate changes downstream. Note that a single vertex will change state from the first to the second only once, and from the second to the third only once — so the total runtime will be $O(n \log n)$, where the log is for priority queue retrieval and insertion.

Problem G: Oil

Shortest judge solution: 1153 bytes. Shortest team solution (during contest): 977 bytes.

This was the "simple geometry problem" of the set. First notice that if all the wells are on one horizontal line, the well will hit only one of them, and it's obviously best to hit the largest one. Otherwise, in a reasonably standard geometric reasoning, we can notice that its always possible to move the well so that it touches at least two of oil layer endpoints (first by shifting to, say, the left, and then by rotating).

So, a naive solution will be $O(n^3)$ — for each pair of points in the input that are not on a horizontal line, try drilling a well through these two points, and check which oil layers are hit by the well. This, however, will be too slow.

To speed it up, we will use a rotating sweep line. Pick any point P through which we will drill a well (we will iterate over all choices of P), and then sort all the other points not on the same horizontal line by the angle of the line through P and the other point. Then, we will rotate the well going through P by iterating over the other points, in slope order. If we encounter the first point of an oil layer, we add the value of this layer to the current result, if we encounter the second point, we subtract the value of this layer from the current result. This algorithm runs in $O(n^2 \log n)$ time, which is fast enough.

When implementing, one needs to take care when sorting by slope (that's always tricky), and — as usually in sweep-line arguments — to deal correctly with tie-breaking (we need to first add layers, and then remove them). In particular, this means we need to store the slopes as pairs of integers, and not as floating points.

Problem H: Polygonal Puzzle

Shortest judge solution: 5282 bytes. Shortest team solution (during contest): N/A.
This geometry problem was quite painful, even compared to other painful geometry problems, and there was a lot of discussion about whether to include it or whether it was simply too hard. The algorithm is conceptually not too hard to figure out, but the implementation is a bit of a nightmare.

Let us call the two polygons P and Q. Suppose we are given an edge from P and an edge from Q that will overlap in the optimal placement (such edges always exist, unless the answer is 0, in which case we are done anyway). Let us rotate P and Q so that these two edges become horizontal with $y = 0$ and facing in opposite directions. Consider shifting (the rotated) P along the x-axis. The optimal placement is obtained by (at least) one such shift of P. Furthermore we may assume that in the optimal placement, either a vertex of P hits a (non-horizontal) edge of Q, a vertex of Q hits a (non-horizontal) edge of P, or a vertex of P coincides with a vertex of Q.

This means that there are only $O(n^2)$ interesting shifts of P. For each such shift, we can in $O(n^2)$ time check if the resulting polygon placements intersect, and if not, what the boundary overlap is. This polygon-polygon intersection test is already quite messy to code, but is probably something several teams have in their code library. However, this will not be fast enough: we assumed that we are given an edge from P and an edge from Q that will overlap, but we are not, which means that we have to try all n^2 possibilities for this as well, resulting in an $\Omega(n^6)$ time complexity overall.

Adding several heuristics to the $\Omega(n^6)$ was actually enough to make it run in time, but there is also a genuinely faster algorithm. The $O(n^4)$ time complexity for finding the best one-dimensional shift can be improved to $O(n^2 \log n)$ by a sweep-line style algorithm. As we shift P from $-\infty$ to $+\infty$, various interesting events happen:

1. A vertex of P enters or leaves the interior of Q
2. A vertex of Q enters or leaves the interior of P
3. An edge of P starts or stops crossing an edge from Q
4. Parallel edges of P and Q become overlapping

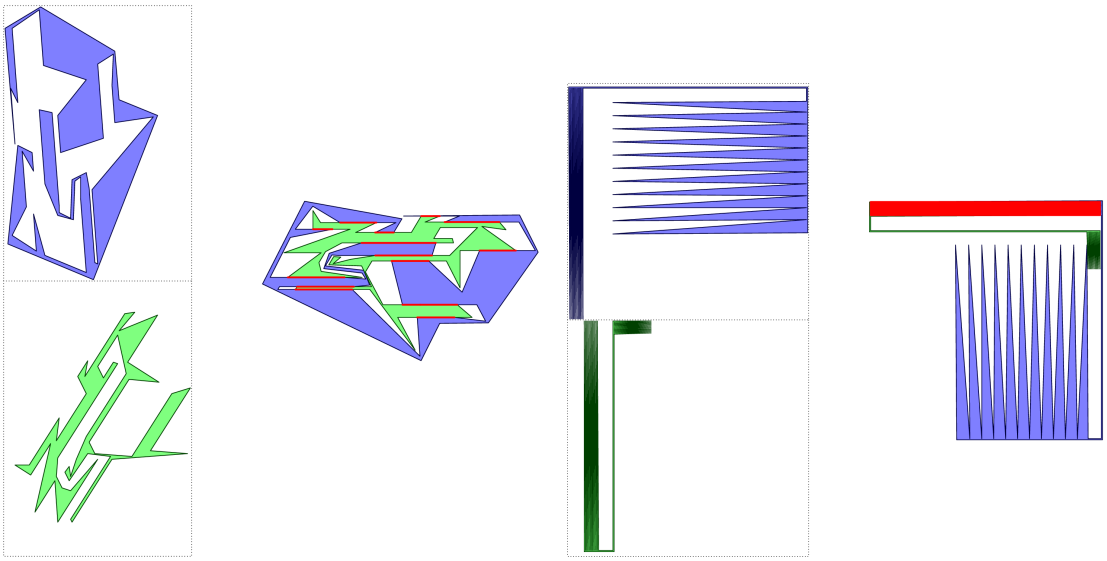
There are in total only $O(n^2)$ such events, which we can find in the same running time. We then sort the events by the x-shift at which they happen, and process them in order. At each event, we may have some obstructions causing the current shift to be invalid (because two edges cross, or a vertex is inside the other polygon, etc), or there may be no obstructions, and then we check the current overlap.

Note that when two parallel horizontal edges of P and Q overlap, their contribution to the total overlap behaves like a piecewise linear function that starts at 0, then goes up, then stays constant for a while (unless the two segments were the same length), the goes down to 0 again, and we have to keep track of the sum of these piecewise linear functions while processing the events.

Several implementation details are being swept under the rug in the above description (e.g. getting things right when a vertex of P travels along a horizontal segment of Q and vice versa), but this is the general principle. Overall one ends up with $O(n^4 \log n)$ time (where the bottleneck is sorting the $O(n^2)$ events for each of the n^2 choices of overlapping edges).

One approach to make the problem more manageable is to first triangulate the polygons. This reduces the intersection tests to triangle-triangle intersection, which is much easier.

Here are some cute test case pictures:



Problem I: Road Times

Shortest judge solution: 3775 bytes. Shortest team solution (during contest): 5056 bytes.

Similarly to problem B, this problem had two independent components, the first of which is a straightforward shortest paths exercise, and the second of which is the core of the problem. In this problem, the second part can be solved using linear programming.

For each edge e in the graph, let us denote by x_e the time it takes to traverse edge e. Initially we know that $d_e \leq x_e \leq 2d_e$ where d_e is the length of the edge. For a source/destination pair (s,t), let us denote by $R(s,t)$ the set of edges on the shortest route from s to t. When we are told that a delivery from s_i to t_i took a_i hours, this can be formulated as the equation $\sum_{e \in R(s_i,t_i)} x_e = a_i$. Given this information, finding the minimum possible time it could take to go from s to t is then given by the optimum of the following linear program:

$$\begin{aligned} \min & \sum_{e \in R(s,t)} x_e & \\ \text{subject to} & \sum_{e \in R(s_i,t_i)} x_e = a_i & \forall\, 1 \leq i \leq r \\ & d_e \leq x_e \leq 2d_e & \forall\, e \end{aligned}$$

Similarly the maximum possible time is obtained by replacing min by max in the linear program.

Thus we have $2q$ linear programs to solve, each having $r + 2e$ constraints and e variables. The $2q$ different programs are in fact all over the same polytope, only the objective function differs, which means that if one uses the Simplex algorithm (which is the most natural choice) one only has to run the first phase (finding a feasible solution) once instead of for each of the

$2q$ programs. (But this optimization was not needed to get accepted – the time limit was very lenient.)

As asides, we are very curious to learn the answers to the following questions (which we haven't been able to answer ourselves – if you figure out the answers, please let us know!):

- Is there a combinatorial algorithm for this problem?
- Does the Simplex algorithm run in polynomial time for the type of linear programs induced by this problem, or can it run in exponential time (possibly by allowing exponentially large distances)?
- Is it possible to construct test cases in which a naive Simplex implementation that does not implement any anti-cycling rule goes into an infinite loop?

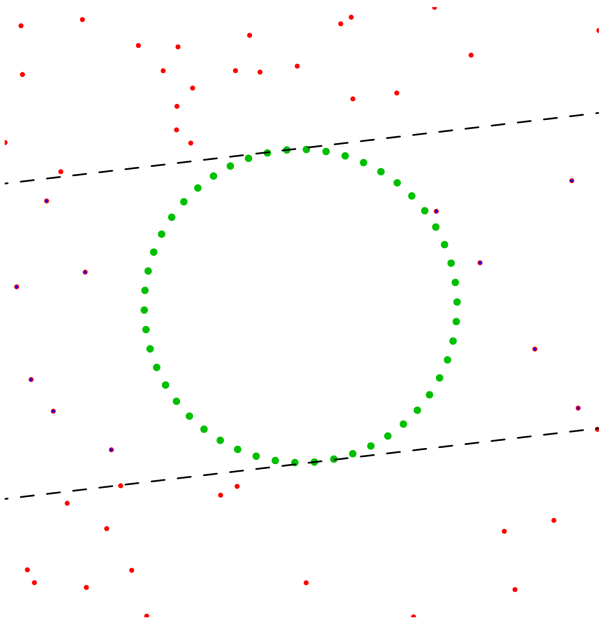
Problem J: Spin Doctor

Shortest judge solution: 3195 bytes. Shortest team solution (during contest): 5540 bytes.

This problem is a geometry problem in disguise. For each person i, let us view (a_i, b_i) as the Cartesian coordinates of a 2-dimensional point. We then have two (multi-)sets of points P_1 (the set of people with $c_i = 1$) and P_0 (the set of people with $c_i = 0$).

For a choice of the parameters S and T, the set of points (a_i, b_i) with $S \cdot a_i + T \cdot b_i = v$ for some v forms a line in the plane, with (S, T) as the normal vector. This means that the points that fall in between the first and the last point from P_1 are those which are contained in a stripe between the two tangents of the convex hull of P_1 with direction $(-T, S)$ (including points exactly on the tangents).

Here is an illustration of a small case, where the green points represent P_1 (which happen to form a circle in this test case), the dashed black lines are the two tangents with direction $(-T, S)$ in the optimal solution, the red points are the points of P_0, and the red points with a blue dot in them are the points in P_0 that fall inside the stripe.



In order to find the best possible (S, T), we have to try rotating the tangents and keep track of how many points from P_0 fall between them. The only time a count changes is when one of the two tangents hits a point from P_0. Since we only care about the convex hull of P_1, we can find the tangents from any point q to the hull of P_1 in $O(\log n)$ time using binary search – this is needed since there can be a lot of points. Once we have all the tangents, we can apply the sweep-line methodology to try all relevant rotations in $O(n \log n)$ time.

There was one very tricky special case, which is the case when $|P_1| = 1$. In this case the answer is always 1, even if there are points from P_0 that coincide with the point in P_1. Since the judges are such nice and friendly people, this special case was included in the sample data. In addition to this, usual geometry caveats with collinear and duplicate points apply.

Problem K: String Theory

Shortest judge solution: 798 bytes. Shortest team solution (during contest): 731 bytes.

This problem can be solved in a few different ways, ranging from dynamic programming to writing a regular expression describing k-quotations, but let us describe an inductive definition/characterization of sequences of k-quotations that directly leads to a simple greedy algorithm for checking if a given input is a k-quotation.

For $k = 1$, a string is a sequence of 1-quotations if it starts and ends with a quote character, and contains an even number of quote characters.

For $k > 1$, a string is a sequence of k-quotations if it is also a k-quotation. To see this, take an arbitrary sequence of k-quotations. This sequence must start with k quote characters then $k-1$ quote characters (possibly after some spaces) then $k-2$ quotes, and so on. The analogous pattern in reverse must hold at the end. In between there must be an even number of quote characters (since in total there must be an even number of quotes), which, by the $k = 1$ case, is a valid sequence of 1-quotations, meaning that the entire string is a valid k-quotation.

In other words, the only situation in which a k-quotation is not the same as a sequence of k-quotations is when $k = 1$. This directly leads to a recursive greedy solution that tries all k (you should try all k such that $k(k-1) \geq \#\{\text{quote characters}\}$).

Problem L: Swap Space

Shortest judge solution: 561 bytes. Shortest team solution (during contest): 660 bytes.

This is a simple, but deceivingly tricky problem that most teams managed to solve, but most of them needed more than one attempt to get it right. Most teams correctly guessed a greedy approach will be required, but figuring out the *right* greedy approach at the first attempt turned out far from obvious.

First, notice that obviously we should reformat the drives for which we gain space due to reformatting prior to any drives for which we lose space due to reformatting. A standard exchange argument if we have two such drives adjacent proves that (if we swap them, we have more free space available both when looking at the first drive formatting, and when looking at the second one).

Now, we need to order the drives for which we gain space, and the drives for which we lose space. For a drive for which we gain space, its best to start with the drives that are the smallest before reformatting (since they require the least space to start — again, an exchange

argument proves this). For drives for which we lose space, you can notice that the problem is symmetrical — if you wanted to reformat the drives back to the old filesystem, you could reverse the order of reformattings and do it in the same space. So, the drives for which we lose space should be ordered by space after reformatting, in decreasing order. Again, an exchange argument shows this to be correct.

Once we know the correct ordering, we can just simulate, adding the extra space as needed, and the total amount of extra space added will be the final answer. Some teams instead performed a binary search on the answer, which is also fine, although unnecessary.

Problem M: What Really Happened on Mars?

Shortest judge solution: 1814 bytes. Shortest team solution (during contest): 2254 bytes.

This problem had a long, scary and convoluted statement hiding a simple implementation. In fact, it gave you exactly the algorithm you need to implement!

We need to simulate the processor's decisions. The limits are low enough that we can actually afford to do this step-by-step, although we could optimize by fast-forwarding if a string of compute instructions are getting executed and no new tasks start.

Let us look at the individual steps to implement. Identifying running tasks is simple. The tricky part is determining what is blocked, and checking the current priorities of tasks. There are two approaches to that. We assume we keep the information on which task holds which resource locked.

The first one, which is harder to analyse but easier to code, is to repeatedly try to iterate on the rule for blocking and priority inheritance, until no new actions to perform are found. So, in each execution step, we would perform a loop until nothing changes. In the loop, for each pair of tasks, we check if the lower-current-priority one blocks the higher-current-priority one, and if yes, increase the first task's priority to be equal to the second task's. It's hard to figure out why that would actually reach a fixed point eventually, but the problem statement tells us there exists a unique solution, so one might hope it will.

The second approach is to go from highest-priority tasks first. If the highest base priority task T isn't blocked, it will be granted the processor — no task can have a priority higher than the highest base priority, and we are guaranteed there will be no ties in the priority resolution. If T *is* blocked, then we identify all the tasks that block T, and raise their current priority to the highest base priority. Note that T will never become unblocked (the first condition does not depend on priorities, and the priority of the highest-priority task will never change), so these priority changes are permanent. So, we can iteratively analyse each of these tasks, until we finally get to a task that is not blocked — and that task will get the processor (note that we're relying on the guarantee there will be no ties). This obviously runs in $O(t^2)$ time for each determination without any optimization, so it will be fast enough.

Once we know which task to run, we execute one instruction, possibly increment the clock according to rules in step 3, and go back to the beginning.

We consider the fate of this task — which only got solved by Shanghai Jiao Tong University a quarter of an hour before the scoreboard freeze — a reminder to always try to read and understand the statement of all problems early in the contest!