ACM ICPC World Finals 2017

Solution sketches

Disclaimer *This is an unofficial analysis of some possible ways to solve the problems of the ACM ICPC World Finals 2017. 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

The major but actually not that big change this year was the addition of Python as an available language in the Finals. This is a major change in the sense that it has taken several years to put into action (because changing the World Finals rules is a very slow process), but not that big in the sense that it actually has a pretty small impact on the practicalities of the contest.

Congratulations to St. Petersburg ITMO, the 2017 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
Solved	35	8	105	31	127	123	18	0	127	1	15	27
Submissions	710	37	255	311	191	174	33	18	137	14	99	150

In total there were 617 Accepted submissions, 1279 Wrong Answer submissions, 164 Time Limit Exceeded submissions and 69 Run Time Errors, and 16 Compile Errors (though these do not give any penalty). The most popular language was (as usual) C++ by a wide margin: 1946, trailed by Java at 111, Python 3 at 8, and C at 2.

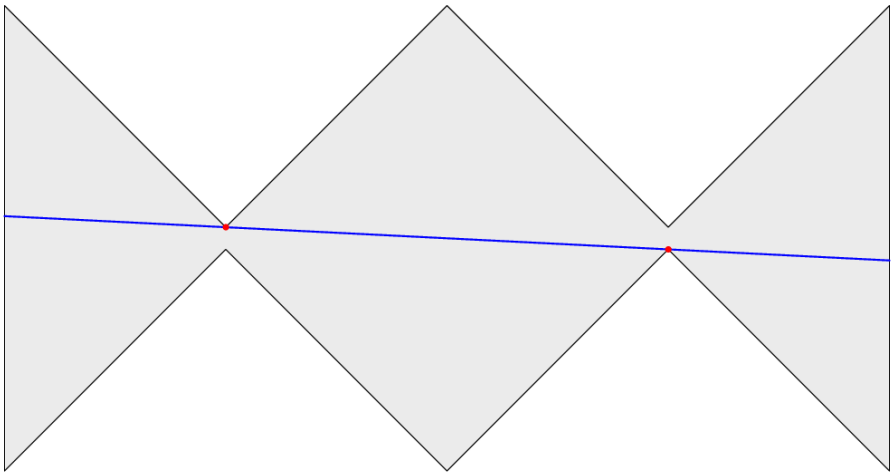
About Python: Ultimately, Python did not get a lot of submissions (but more than C did!), and only from a single team, meaning that this was the only language where the judges wrote more solutions than the teams did. Whether this is because it was a new feature this year (which is still in the process of trickling down to regionals), or whether Python will remain unpopular among ICPC World Finalists, remains to be seen. Teams had the option of submitting either in Python 2 using the pypy interpreter, or in Python 3 using the standard CPython interpreter. For each problem below, we'll list the extent to which the judges had written Python solutions (whether there were Python solutions in both Python 2 and 3, or only in Python 2 – pypy is a lot faster than CPython – or none at all).

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 – though some of us may have a tendency to overcompactify their code – 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! Previous years, the shortest team solutions have been roughly on par with (or shorter than) the shortest judge solutions for pretty much all problems. This year, there are a few where the shortest judge solutions are noticably shorter. In all of those, this difference seems to come from the shortest judge solution being in Python (which often results in fairly short code) but teams not solving it in Python.

Problem A: Airport Construction

Shortest judge solution: 1460 bytes. Shortest team solution (during contest): 2365 bytes.
Python solutions by the judges: only Pypy

The initial thought is the standard in geometry – there has to exist an optimal landing strip that passes through two vertices of the polygon. The proof is also relatively standard: you can elongate any landing strip until both ends hit the edge of the polygon and, keeping one end fixed, rotate the landing strip – it will increase length in one of the directions of rotation – until either reaching the end of the edge, or until blocked by a different vertex of the polygon. Notice that the longest edge doesn't necessarily have to be a diagonal, in fact, it can happen that both of the ends of the longest landing strip are in middles of polygon edges, as in the figure below.



Given the $n \leq 200$ limit, a cubic algorithm will run in time – so, the plan is to try all pairs of points A, B, and for each pair figure out first whether the interval AB is wholly contained in the polygon, and if so, find out how much can it be extended. In principle, this is easy – for the first question, find out whether the polygon boundary crosses AB, and find out the first place the polygon boundary crosses the half-lines extending AB in either direction.

In practice, the most difficult part is correctly classifying the cases where the polygon boundary touches the line AB, possibly runs along the line for some time, and then veers away – either back to the same side it came from (in which case it didn't cross the line), or to the other side (in which case it did cross the line). This requires some care to get right (although is made a bit easier by the fact the problem statement disallows subsequent collinear edges).

Problem B: Get a Clue!

Shortest judge solution: 1871 bytes. Shortest team solution (during contest): 1841 bytes.
Python solutions by the judges: both Pypy and CPython

This is a problem that can be solved by a brute-force search, but the implementation can be a bit messy, and depending on your exact approach, it may be important to have good constant factors.

The very simplest approach is just to generate all disjoint sets of cards for the four hands (well, one hand is already given so there's only one option for that) and then go through all the played rounds and check that they are consistent with assignment of cards to hands. However, this will probably not run in time unless it is fairly carefully implemented, so something a bit better is needed.

The next approach could be to first guess the murderer, weapon, and room, and then do the above-mentioned brute-force search for a partition of cards into hands, and break as soon as a valid solution is found. This turns out to run a bit faster and is definitely possible to make fast enough (because the worst case for the above is when pretty much all partitions of cards into hands yields a valid solution, and many of those partitions into hands will result in the same murderer/weapon/room).

A different algorithmic approach which essentially makes constant factor worries go away is to generate the possible hands separately: for player 2 we compute all possible subsets of 5 cards $S_1, S_2, \ldots$ that are compatible with all the played rounds, and similarly for player 3 all sets of 4 cards $T_1, T_2, \ldots$ and for player 4 all subsets of 5 cards $U_1, U_2, \ldots$. Now for a given guess of murderer/weapon/room, let X be the set of remaining cards after removing the three answer cards and the hand of player 1. We are then trying to find i, j, k such that $S_i \cup T_j \cup U_k = X$. This can easily be done in time $O(\#S \cdot \#T)$ – simply try all S_i's and T_j's and then check if $X \Delta S_i \Delta T_j$ is one of the U_k's (by keeping a dictionary of all U_k's). The subsets are most conveniently represented by integers, which makes lookups quick. This solution is fast enough that it can even be done in CPython.

Problem C: Mission Improbable

Shortest judge solution: 818 bytes. Shortest team solution (during contest): 1178 bytes.
Python solutions by the judges: both Pypy and CPython

If not for the top-down view, this problem would be really straightforward. Since solving easy problems is easier than solving harder problems, let's go over that first.

Consider the highest stack of crates in the input, and assume it has height H_1. It has to appear both in the front view and the side view at least once. Assume that it appears m_1 times in the front view, and n_1 times in the side view, and without loss of generality, assume $n_1 \leq m_1$. In that case, we will need at least m_1 stacks of height H_1, and we can achieve the correct side and front view by arranging m_1 stacks so that there is at least one in each of the m_1 columns and n_1 rows.

Then, proceed to the next height, H_2. Again, we have m_2 columns and n_2 rows that have to contain a stack with height H_2. The one thing that is different here is that it is possible for,

say, m_2 to be zero – in which case we will put the stacks of height H_2 in the column(s) already containing stacks of height H_1. Following this pattern, we will use up a total of

$$\sum H_i \cdot \max(m_i, n_i)$$

crates. After placing all the crates for all the heights, the side and front views are already correct, so (disregarding the existence of the top view) we can just leave the remaining spaces empty.

Now, the existence of the top view changes two things in that strategy. First, at the end, we might have to leave a single crate (instead of zero crates) in some of the remaining spaces, to prevent the top view from noticing the spaces are empty. This is easy – we just keep track of how many spaces we filled, and then add to the final answer the number of spaces seen in the top view minus the number of spaces already filled.

The more tricky part is that due to the top view seeing empty spaces in some spots, it might be impossible to put a stack of height H_i in each of the m_i columns and n_1 rows using just $\max(m_i, n_i)$ stacks. We want to have as many stacks as possible to cover both a row and a column, and then we can make the remaining columns and rows covered by just putting a stack of height H_i wherever it was in the original input. Notice that this is a bipartite matching problem – we have a set of rows and a set of columns, and we can connect a row to a column when the top view shows a non-empty stack. So, for each height H_i appearing in the input, we run bipartite matching to find out how many stacks can cover both a row and a column, and then replace $\max(m_i, n_i)$ with $m_i + n_i - Bipartite(i)$. Note that this formula works fine even if one of the sides is zero. Since the runtime of bipartite matching is super-linear, the worst-case for this problem is if all the columns and rows in the front and side views are of the same height. With $r, c \leq 100$, this will easily run in time.

Problem D: Money for Nothing

Shortest judge solution: 1552 bytes. Shortest team solution (during contest): 1178 bytes.
Python solutions by the judges: only Pypy

First, observe that this is at heart a geometry problem. We are given a set of lower-left and upper-right vertices, and we're asked for the area of the largest rectangle (with sides parallel to the axes) between some two chosen corners. One observation we make is that we can prune the input set. If there are two lower-left corners, (x_1, y_1) and (x_2, y_2), with $x_1 \leq x_2$ and $y_1 \leq y_2$, then we can remove (x_2, y_2) from the set. After this pruning, the set of lower-left corners forms a sequence $L_1, L_2, L_3, \ldots, L_n$, with $L_i = (x_i, y_i)$, and $x_i < x_{i+1}, y_i > y_{i+1}$. We can perform similar pruning on the upper-right corner set, getting the sequence $U_1, U_2, \ldots, U_m$, with $U_i = (p_i, q_i)$, and again $p_i < p_{i+1}$ and $q_i > q_{i+1}$.

We'll cover two solutions to the problem. The first one is less geometric. Define $u(i)$ to be the index of the optimum upper-right corner for L_i – that is, the rectangle $L_i, U_{u(i)}$ has area no smaller than any other L_i, U_j. In the case of ties, choose the rightmost one. We're claiming that $u(i)$ is non-decreasing. A geometric proof is to draw out the picture, and notice that for any $i < j$, $k < l$, the sum of areas of L_iU_k and L_jU_l is larger than the sum of areas L_iU_l and L_jU_k. You can also just write out the areas and see the inequality holds.

This allows a divide-and-conquer solution. Take the middle of all the L_is, and find $u(i)$ (through a linear scan). Then, for all $j < i$, we can consider only the upper right corners up to $u(i)$, and for all the $j > i$ we can consider only the upper corners starting from $u(i)$, and recurse into both branches. Since we're halving the set of Ls at each pass, we will have $\log n$

levels of branching, and in each of the levels each of the Us is getting considered only for one interval (with the exception of the boundaries, but these sum up to $O(n \log n)$ as well), so the runtime will be $O((m+n)\log n)$.

The other solution is geometric. For any two points U_i, U_j, let us consider the set of points L for which U_i gives a larger rectangle than U_j. The boundary of this set is a straight line, so the set of points L for which U_i is the best choice is an intersection of half-planes, i.e., a convex polygon which is possibly unbounded in some directions. The division of the plane into these polygons contains a total of $O(n)$ vertices, and so we can run a sweep-line algorithm, with the events being a point from L appears, or the set of regions changes. This will run in $O((m+n)\log(m))$ time.

Problem E: Need for Speed

Shortest judge solution: 321 bytes. Shortest team solution (during contest): 413 bytes.
Python solutions by the judges: both Pypy and CPython

This was one of the two easiest problems of the set. Given a guess for the value of c, we can compute the resulting distance that this would result in. If the guess of c was too high, the travelled distance will be too high (because in each segment travelled we're overestimating the speed), and if the guess was too low, the travelled distance will be too low. Thus we can simply binary search for the correct value of c. The potentially tricky part is what lower and upper bounds to use for the binary search. If in some segement the speedometer read v, the value of c needs to be at least $-v$. Thus, c needs to be at least $-\min v$. For the upper bound, a common mistake was to assume that c could never be larger than 10^6. This is almost true, but not quite – the maximum possible value is $10^6 + 1000$ (our true speed can be as large as 10^6, but the reported readings of the speedometer can be -1000).

Problem F: Posterize

Shortest judge solution: 771 bytes. Shortest team solution (during contest): 686 bytes.
Python solutions by the judges: only Pypy (but for no good reason, should be feasible with CPython as well)

This is a fairly straight-forward dynamic programming problem. Let $C(i,j)$ be the minimum squared error cost of posterizing pixels with intensities $r_1, \ldots, r_i$ using j colors. The quantity we are looking for is then $C(d,k)$.

We can formulate the following recurrence for C:

$$C(i,j) = \min_{0 \le i' < i} C(i', j-1) + F(i'+1, i),$$

where we define $F(a,b)$ to be the minimum cost of posterizing pixels with intensities $r_a, \ldots, r_b$ using a single color. Assuming for the moment that we have computed the function F, it is a standard exercise in dynamic programming to turn this recurrence into an algorithm for computing $C(i,j)$ in time $O(i^2 j)$.

Computing $F(a,b)$ can be done in a few different ways. Note that

$$F(a,b) = \min_{x \in \mathbb{Z}} \sum_{i=a}^{b} p_i (x - r_i)^2.$$

This is just a quadratic function in x, so the best x can be found by basic calculus (or some form of ternary search for those so inclined).

There are just d^2 different possible inputs to F and the answer for all these can be precomputed to be used when computing C. The bounds were actually small enough that it was even possible to get away with recomputing $F(a, b)$ every time it was needed, at least if the implementation had good constant factors.

Problem G: Replicate Replicate Rfplicbte

Shortest judge solution: 1637 bytes. Shortest team solution (during contest): 1440 bytes.
Python solutions by the judges: none

The judges had this pegged as a medium difficulty problem but the teams apparently felt otherwise. There are a few small observations to make, in order to understand how to solve the problem.

The first thing to realize is that when applying a step of the process, the bounding box of the pattern always grows by at least one step in every dimension, *even if there is an error*. Equivalently: without errors, the resulting bounding box after an evolution step will have at least 2 filled cells on every side. In other words, when going backwards, the bounding box will shrink by at least one step in each dimension for every step. This means that the total number of iterations to go backwards can be at most $\min((n+1)/2, (m+1)/2)$.

Suppose for a moment that no errors happen. Then we can simply reconstruct the previous pattern X from the current pattern Y row by row – if we're reconstructing cell (r, c), and have already reconstructed all rows $r' < r$, and all cells (r', c') with $r' = r$ and $c' < c$, then we can compute the previous value $X_{r,c}$ by $Y_{r-1,c-1} \oplus X_{r-2,c-2} \oplus X_{r-2,c-1} \oplus X_{r-2,c} \oplus X_{r-1,c-2} \oplus X_{r-1,c-1} \oplus X_{r-1,c} \oplus X_{r,c-2} \oplus X_{r,c-1}$ (where $\oplus$ is XOR a.k.a. addition mod 2). Proceeding in this way we can reconstruct the previous step.

OK, that's easy, but what if there are errors, how do we even detect that? Suppose that cell (r, c) has an error. By the observation above, this will cause the reconstruction of cell $X_{r+1,c+1}$ to get the wrong value. This will in turn cause $X_{r+1,c+2}$ to get the wrong value. However, then $X_{r+1,c+3}$ will actually get the correct value, because the two errors from $X_{r+1,c+1}$ and $X_{r+1,c+2}$ cancel out. Then similarly $X_{r+1,c+4}$ and $X_{r+1,c+5}$ will get the wrong value, and $X_{r+1,c+6}$ will get the right value, and it will continue cycling like that with two incorrect cells followed by one correct cell. That means that when we get to the end of the row, we can verify that the first two cells that should be outside the bounding box (by the observation above) become empty. If an error happened in row r, at least one of these two cells will get an error and become non-empty. When this happens, we can run the reconstruction again, but column by column instead of row by row, which allows to detect that an error happened in row c. We have then found the error, can undo it, and then run the reconstruction step again to get to the previous pattern.

The process ends when the pattern is either a single filled cell, or if the reconstruction process still finds errors after the first error is fixed. Going back one iteration using the above process takes $O(n \cdot m)$ time, so by the observation above on the maximum number of iterations, this results in an $O((n+m)^3)$ time algorithm.

Note that there are actually no choices to make in how to do the reconstruction, meaning that the answer is in fact uniquely determined (though figuring this out was part of solving the problem).

Problem H: Scenery

Shortest judge solution: 1753 bytes. Shortest team solution (during contest): N/A bytes.
Python solutions by the judges: none

OK, this is the difficult problem of the set...

This problem really tempts you to try some sort of a greedy solution. Let us see how that would go. Let us process time from the beginning. At any point in time, when we decide to take a photograph, we should – out of all the photographs we can currently take – choose the one for which the end time is the smallest (by a standard interchange argument).

The tricky part is to choose whether you should actually start taking a photograph. Consider an input with two photographs to take. The first one will be available in the range $[0, 5]$, the second in the range $[1, 3]$, and the time to take a single photograph is 2. Then, at time zero, we will fail if we start to take the first photograph. On the other hand, if the second range was $[2, 4]$, we would fail if we didn't start a photograph at zero. So, some sort of smarts are going to be needed.

We will describe two solutions. The first, which we will describe in some detail, comes from the paper "Scheduling Unit-time Tasks With Arbitrary Release Times and Deadlines" by Garey, Johnson, Simons, and Tarjan (SICOMP, 1981). In that paper it is shown that the problem can even be solved in $O(n \log n)$, but we felt that getting a quadratic time solution was hard enough, and that is what we describe here.

We are going to adapt the naive solution above. Notice that in order to make the photograph with the range $[1, 3]$ be even feasible, regardless of all the other photographs, we cannot start any photograph in the open range $(-1, 1)$ – if we do, it will overlap the range $[1, 3]$, and we will not be able to fit $[1, 3]$ in. We will try to generalize that observation.

Take any interval $[s, e]$, and consider all the photographs which become available no earlier than s, and become unavailable no later than e. Then, all these photographs have to fit into the interval $[s, e]$. We will now totally disregard their constraints (except that they have to be taken somewhere in $[s, e]$), and try to take them as late as possible. Note that since we disregard their constraints, they are all identical and we can just schedule them greedily. Let us look at the time C when we started taking the first (earliest) of those photographs. However we schedule these photographs in the interval, the first start time will not be later than C. If $C < s$, then we will simply not be able to take all the photographs. The interesting case is if $C - s < t$. Then, if any photograph gets started in the open interval $(C - t, s)$, we will be unable to take all the photos from the interval $[s, e]$, since we will not be able to take the first of them at C or before.

This means we can mark the interval $(C - t, s)$ as "forbidden", and no photograph can ever be started then. We can also take any forbidden intervals we found so far into account when doing the greedy scheduling from the back (which might help us create more forbidden intervals).

The full algorithm works as follows. We order all the availability times of all the photographs from latest to earliest. For each such time s, we iterate over all the end times e of the photographs, and for each interval $[s, e]$ we run the algorithm above – take all the photographs that have to be taken fully within $[s, e]$, ignore other constraints on them, assign times to them as late as possible (taking into account already created forbidden regions), and find the time C, which is the earliest possible start time of the first of those photographs. If $C < s$, return NO, and if $C - s < t$, produce a new forbidden region ending in s.

The above, implemented naively, is pretty slow. First, notice that while as written we can have a quadratic number of forbidden regions, it is trivial to collapse all the forbidden regions ending at any s into one (the one corresponding to the smallest C). Also, note that when we move from s to the previous s', we don't have to run the time assignment from scratch – we can just take the previous C value, and if the photograph starting at s' ends before some e, we just have to add one more photograph before the C (by default, at $C - t$, but possibly earlier if $C - t$ is in a forbidden region). This allows us to progress through this stage in $O(n^2)$ time.

Then, after this is done, we just run the standard "greedy" algorithm, taking the forbidden regions into account. The tricky part, of course, is to prove that this actually solves the problem. Obviously, if the greedy algorithm succeeds in taking the photographs, the answer is YES. Now, we will prove that if the greedy algorithm ends up taking a photograph after its end time, then we would have actually failed in the first phase.

Assume the greedy algorithm did take a photograph incorrectly. First, notice that if there are idle times (that is, times when no photograph is being taken by the greedy), we can assume they are all within forbidden regions. If there was an idle period without a forbidden region, let's say at time X, it means that we have had no photograph available to take. So, all the photographs that were available before X got assigned and ended before X, and so they do not affect the photographs that become available after X. So, we can just remove these photographs from the set altogether and solve the smaller problem.

Now, assume the greedy algorithm failed. Take the first photograph P_i, with a deadline of e_i, that got scheduled so it ends after e_i. If no photograph that got scheduled earlier has a deadline later than e_i, it means that the photographs up to P_i actually failed to fit into the range s_0, e_i – and so for that interval the first phase would've returned NO. If there is a photograph that has a deadline later than e_i, let P_j be the latest of those photographs. Consider all the photographs $P_{k_1}, \ldots, P_{k_l}$ that got scheduled between P_j and P_i, and let s be the earliest time at which any of those photographs (including P_j) became available. Consider the first phase for the interval s, e_i – it had to have failed (because the greedy solution failed to put these photographs into this interval).

Thus, in all the cases where the greedy algorithm fails, the first phase had to have failed for some interval as well – so, if the first phase finished successfully, we are guaranteed the greedy algorithm will return a correct answer.

An alternative approach, which we will not prove, is an approach where we modify a naive branching approach. Naively, we can process forward through time, and maintain a set of possible states, where a possible state is the time when the photograph currently being taken (if any) ends, and the set of photographs that are available to take, but not yet started. This set of states is potentially exponential in size. We branch out (from all the states where there is no photograph being taken) when a new photograph becomes available, and we branch out when a photograph taking ends (to either not start a new photograph, or to start the one with the earliest deadline from those in the state).

However, it is provable that we can actually have only one state where no photograph is being taken – when the algorithm ends up produces a new "nothing runs" state because some photograph just finished, one of the two states is strictly worse than the other. The proof, and the details of turning this into a quadratic algorithm, are left as an exercise to the reader.

Problem I: Secret Chamber at Mount Rushmore

Shortest judge solution: 405 bytes. Shortest team solution (during contest): 498 bytes.
Python solutions by the judges: both Pypy and CPython

This was one of the two easiest problems in the set. The set of translations forms a directed graph on the 26 letters of the alphabet. Given two query words $s_1 s_2 s_3 \ldots s_L$ and $t_1 t_2 t_3 \ldots t_L$ of equal length L (if the lengths are different, the answer is clearly `no` and we just answer that), we need to check whether for each $1 \leq i \leq L$, there is a path from s_i to t_i in the graph of letter translations.

One natural way of doing this is to precompute the transitive closure of the graph using the Floyd-Warshall algorithm, which allows you to check each pair (s_i, t_i) in constant time. But the bounds are quite small, so you can use pretty much any polynomial time algorithm for this (e.g. doing a DFS for every pair (s_i, t_i)).

Problem J: Son of Pipe Stream

Shortest judge solution: 1649 bytes. Shortest team solution (during contest): 5492 bytes.
Python solutions by the judges: only Pypy

Even though the problem does its best to avoid using the word "flow", it should be pretty clear that this is in fact a maximum flow problem, though it has a few twists to resolve. First, the viscosity parameter v is a red herring that is pretty much irrelevant to the problem and we will ignore it here.

A first small observation is that the desired flow will be *a* maximum flow from the water and Flubber sources to the destination. Let Z be that total maximum flow. Assume for the moment that it was possible to distribute this arbitrarily between Flubber and water, so that it was possible to route any amount F of Flubber between 0 and Z and $W = Z - F$ water. Then a bit of calculus (left as an exercise) shows that the maximum flow would pick $F = a \cdot Z$.

There are two different reasons why the assumption made above might not hold. One is a "trivial" one: the maximum amount of Flubber routable from Flubber source to destination might be less than $a \cdot Z$, or the amount of water routable might be less than $(1 - a) \cdot Z$. If this is the case, we should simply set the desired amount of Flubber F to value nearest to $a \cdot Z$ in the interval $[Z - W_{\max}, F_{\max}]$. Let us refer to the resulting potentially optimal values of F and W as F^* and $W^* = Z - F^*$.

The second reason is a bit more subtle – it is not clear whether it is the case that we can simultaneously achieve Flubber F^* and water W^*. Let $\vec{f_1}$ be a flow which routes $F_{\max}$ Flubber and $Z - F_{\max}$ water, and let $\vec{f_2}$ be a flow which routes $Z - W_{\max}$ Flubber and $W_{\max}$ water. Then for $\alpha \in [0, 1]$, $\alpha\vec{f_1} + (1 - \alpha)\vec{f_2}$ is a flow of fluids in which $\alpha F_{\max} + (1 - \alpha)(Z - W_{\max})$ of the fluid originates at the Flubber source, and we can set α appropriately to a constant so that this equals F^*. However, there is a snag with this: there might be pipes where $\vec{f_1}$ and $\vec{f_2}$ route fluids in opposite directions, so it is not clear that we can achieve this flow while satisfying the "water and Flubber must not go in opposite directions" constraint. Phrased a bit more abstractly, the "no opposing flows" constraint is not a convex constraint.

It turns out that this second reason is actually a non-reason – it is always possible to achieve Flubber F^* and water W^*. But we now have to construct the actual Flubber and water flows. One way of doing that is to take the mixed flow $\vec{f}^* := \alpha\vec{f}_1 + (1-\alpha)\vec{f}_2$ from above. This tells us how much fluid we would like to send (and in which direction) along each pipe, but it doesn't tell us how much of that fluid should be Flubber, and how much should be water. To figure that out, we make a new flow graph where we set the (directed) capacity of each edge to be the (directed) flow in $\vec{f}^*$ along that edge. We then compute the maximum flow from the Flubber source in the new graph. This gives the Flubber flow, and the unused capacity gives the water flow. An issue here is that the new graph has non-integer capacities, so one may have to be a bit careful when computing the maxflows.

Problem K: Tarot Sham Boast

Shortest judge solution: 473 bytes. Shortest team solution (during contest): 585 bytes.
Python solutions by the judges: both Pypy and CPython

This problem has a beautiful solution that is not impossible to get an intuition about, but hard to actually prove correct.

A naive way of computing the probability that a string X of length ℓ appears in a uniformly random string of length n is to use the principle of inclusion-exclusion:

$$p(X) = \sum_{I \subseteq [n-\ell+1]} (-1)^{|I|+1} \Pr[\text{there is an occurence of } X \text{ at all positions in } I]$$

For $I = \{i\}$ consisting of a single element, the probability in the sum is simple $3^{-\ell}$ – the probability that characters $i, i+1, \ldots, i+\ell-1$ match X. Thus the first-order contribution from index sets I of size 1 to $p(X)$ is simply $(n-\ell+1)/3^\ell$, regardless of what X looks like.

For the second-order terms $I = \{i, j\}$, things are a bit more interesting. If $j \geq i+\ell$ then the probability is simply $3^{-2\ell}$ since the two matches of X involve disjoint positions. But for $j < i+\ell$ however, the probability is 0 unless $j-i$ is an *overlap* of X, where we say that t is an overlap of X if the prefix of the first $\ell - t$ characters of X equals the suffix of the last $\ell - t$ characters of X. If t is an overlap of X, then the set $I = \{i, i+t\}$ contributes $-1/3^{2\ell-t}$ to the expression for $p(X)$ above.

This hints at the following intuition: strings X with more overlaps should have smaller values of $p(X)$. Among different overlap values t, higher values of t seems to result in smaller probabilities, because the subtracted terms $1/3^{2\ell-t}$ are larger. So a somewhat natural hypothesis is that if we write the overlaps of X in decreasing order, then strings with lexicographically smaller overlap sequences have higher likelihood.

However, this is just an intuition, and it is not at all clear what happens with higher-order terms – the degree-3 terms (having $|I| = 3$) give positive contributions to $p(X)$ and more overlaps will by a similar reasoning cause these to be larger. It turns out that the hypothesis above is correct, and that lexicographically smaller overlap sequences leads to larger probabilities. We only have a very long and non-intuitive proof of this, which we don't include here. But since several people have expressed an interest in it, it has been made available as a separate document here: `http://www.csc.kth.se/~austrin/icpc/tarotshamproof.pdf`.

As an example, consider the two strings $X =$ `RPSRPSRPS` and $Y =$ `RPRRPRRPR`. The overlaps of X are $Ov(X) = (6, 3, 0)$. The overlap sequence of Y is $Ov(Y) = (6, 3, 1, 0)$. This means that in general, X has a higher likelihood of appearing than Y (since $(6, 3, 0)$ is lexicographically smaller than $(6, 3, 1, 0)$).

However, there was an additional mistake that could be made. If $n \leq 2\ell - 2$, only overlaps $t \geq 2$ can come into play. This means that for such small values of n, the strings X and Y are actually equi-probable. In general, we need to ignore any overlap values that are smaller than $2\ell - n$ when constructing the overlap sequence.

Overlap sequences can be constructed in $O(\ell)$ time using KMP or hashing, leading to an $O(\ell s \log s)$ time algorithm.

Problem L: Visual Python++

Shortest judge solution: 1776 bytes. Shortest team solution (during contest): 1727 bytes.
Python solutions by the judges: only Pypy

This problem can be solved by a sweepline algorithm. Let us sweep from left to right, keeping a set of encountered but unmatched top left corners, ordered by r-coordinate. When we encounter a new corner:

- If it is an upper left corner, add it to the set of unmatched corners. If there was already a corner in the set with the same r-coordinate, we have encountered a syntax error – it will never be possible to create non-nesting matchings for these two top left corners.
- If it is a lower right corner with r-coordinate r_2, match it with the top left corner in our set with the largest r-coordinate r_1 that is $\leq r_2$. If there are no such top left corners in our set, we have encountered a syntax error.

Each event can be processed in $O(\log n)$ time so we have a time complexity of $O(n \log n)$.

However, we are not yet done. Even if this process finishes, the constructed rectangles may intersect. To check for this, we run essentially the same sweep again, but this time, since we know exactly how the rectangles look, we also check when adding and removing top left corners from our active set that adjacent pairs of rectangles in this set do not intersect.

Note that the solution, if it exists, is actually unique, but like with the Replicate problem, figuring this out is part of solving the problem.