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.