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

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