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).