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 $\log n$