

ICPC 2009 World Finals

Petr Mitrichev

Problem A

First, we loop over $8!$ possible orderings for the landing of the planes. Then, we do binary search on the answer. Having fixed the answer, we simply land each plane in the chosen order at the earliest possible time.

Problem B

This problem requires careful checking of all boundary cases, and doesn't seem to require any particular insight. One just verifies the output of the original scheme on all given inputs, as well as the output of all "wrong" schemes that can be obtained from the given one by introducing one mistake (since there's not more than 19 gates, there's not more than 57 such schemes), and then checks if the output can be uniquely determined. But there surely is some trickiness to implementing that.

Problem C

This is a very well-known problem if the ant moves along the surface of the cube; the octahedron doesn't change much. We try all possible ways to lay out the octahedron's sides on the plane (first place the first side, then place any of its neighbors next to it, and so on). The number of possible layouts should be on the order of hundreds or thousands. And in at least one layout, the shortest path will be just a straight segment - so we need to check those segments in all layouts, throw away those that go outside the layout, and then find the shortest one of the remaining. This solution should work as long as no shortest path passes through the same side twice; I don't have a

proof for that, but intuitively it seems true. The implementation will require a LOT of accuracy, though.

Problem D

One thing that comes to mind is to draw all possible layouts on paper, and figure out the formulas for the answer in each of them. But that's very error-prone, so I hope there's a more concise solution.

Problem E

Take any vertex in the graph. Let's say it has a "nice left" if any path from the start to that vertex has the same cost. Let's say it has a "nice right" if any path from the end to that vertex has the same cost. This can be determined by DFSen. If there's a vertex without both nice left and nice right, then there's no solution: at least one of the roads to the right of it will have to be modified, and at least one of the roads to the left of it will have to be modified - so there will be a path with two modified roads. Otherwise, the vertices split into 3 sets: A - vertices with nice left but no nice right, B - vertices with nice left and nice right, and C - vertices with nice right but no nice left. If both A and C are empty, the constraints are already satisfied and there's no need to change anything. Otherwise, A will always have the start vertex, C will always have the end vertex. Let's assume for now that B is empty. The above argument about no path with two modified roads tells us that the only way to achieve what's required is to modify the roads that lead from A to C. More specifically: take all pairs of vertices a from A and c from C, where there exists an edge from a to c . The cost of that edge plus the cost of the path from start to a and from c to end is a lower bound for the answer. Take the highest of those lower bounds, say H , and change the cost of each edge between a from A and c from C to $H - \text{cost}(\text{start}, a) - \text{cost}(c, \text{end})$. This will quite obviously be the minimal possible answer, and every path will have the same cost of H since it will pass along exactly one edge from A to C. Now, how to deal with B? It turns out that we can simply put all vertices from B to C, and the above solution will still work - since it always leaves at least one route from start to end unchanged, it can't make a non-optimal answer; and it will always produce a correct answer since the only thing it

requires is for all vertices from A have a “nice left“ and all vertices from C have a “nice right“. Does that make sense? :)

Problem F

First, let’s learn how to calculate the length of one fence required to enclose the given set of points. To find that fence, we find the convex hull of that set of points, and then “extend” it to the outside by the given margin. The result may have a complex structure of straight and circular segments, but its total length is easily computed: the total length of straight segments is equal to the perimeter of the convex hull, and the total length of circular segments is equal to one circle, $2\pi \cdot \text{margin}$ (because you have to “rotate” by 2π to complete the cycle - when you’re moving in straight line, your direction doesn’t change). The rest of the problem is standard: we do a DP that calculates the minimum possible cost for each subset of points, by choosing a subset of that subset to be contained by one fence, and taking the previously-computed DP value for the rest of the points.

Problem G

I think this problem requires one just to implement the game’s rules carefully. The number of possible states in the game is reasonably small: the position is characterized by the number of cards still lying in the row, the cards held by both players, if any, and the layout of the “top line” of the house of cards - which will always include not more than 8 cards. This gives us a rough estimate of $C(26,10) \cdot 26$ states, which is roughly 100 million - but in reality much less of the states will be reachable, since the top line only contains 8 cards in one case, and so on.

Problem H

If we introduce variables x_1, x_2, \dots for the decisions on the bills, then satisfying all ministers can be formulated via statements of form “if x_5 is false, then x_7 is true” (since if at least one decision contradicts minister’s choices, then all his other choices must be respected). That gives us an instance of 2-CNF satisfiability problem, and its solution is well-known. Checking which

variables have the same value in any solution is also quite routinely added to that well-known solution (for example, we fix that variable and run the satisfiability checking again).

Problem I

Another implementation problem here. One should understand the problem statement, and then simulate all resizes going from the outermost window inside.

Problem J

We can try solving this with a binary search + DP. We binary search over the answer, then do a DP on subtrees. For each subtree, consider all possible roundings that don't contradict the chosen answer (that is, the paths completely inside the subtree don't change by more than that answer), and from each of those roundings, we're interested in the lowest and highest differences for paths from the root of that subtree to its vertices (these two values are the only ones that affect paths that pass through the root of that subtree). So the DP can be: for a subtree T and a lower bound L on the difference on the paths from its root, what is the lowest possible upper bound U on that difference? This should give us only at most of $100 \cdot 30 \cdot 100$ states, and I think the processing of all states can be done in at most $100 \cdot 30 \cdot 100$ time (since there're only 100 edges, and "updating" along each edge will take one scan along a $30 \cdot 100$ array that has the U 's for each L), thus even a binary search outside still leaves the running time reasonable. I know this is very unclear, but I think the only way to figure out the details is to actually start coding the solution, and I'm too lazy for that :) A shot at clarifying: for a subtree, there're 3 parameters: 1) the maximum modulus of delta for paths inside that subtree 2) the maximum delta for paths from the root of that subtree into the subtree 3) the minimum delta for paths from the root of that subtree into the subtree Without the binary search, we'd be forced to add the 1st parameter to the DP, and that would make it too slow. That's why we use binary search outside.

Problem K

Consider the chain of transformation in the optimal answer. Consider the change(s) that affect the leftmost character of the string. Between those changes, the transformation is split into independent parts. That gives us the possibility of the following DP: consider the set of all suffixes of length L of all given strings (S, T , and all transformation strings). There're at most 202 of those for each L . Now, let's calculate the best way to transform each of those strings to each other, given that we know such answers for smaller values of L . First, we account for the transformations that don't involve the whole L characters - we can just take the value from the $L-1$ step. And then, there are transformations with the whole L characters - they are given by rules of length exactly L . And then, we need to combine those two kinds with a shortest-paths algorithm, like the Floyd-Warshall. That gives us 202^3 running time for each L , so about 160 million very simple operation for each testcase - that should be enough.