ACM ICPC World Finals 2011

Solution sketches

Disclaimer *This is an unofficial analysis of some possible ways to solve the problems of the ACM ICPC World Finals 2011. Any error in this text is my error. Should you find such an error, I would be happy to hear about it at* `austrin@kth.se`.

Also, note that these sketches are just that—sketches. They are not intended to give a complete solution, but rather to outline some approach that can be used to solve the problem. If some of the terminology or algorithms mentioned below are not familiar to you, your favorite search engine should be able to help.

Finally, I want to stress that while I'm the one who has written this document, I do not take credit for the ideas behind these solutions—they come from many different people.

— Per Austrin

Mini-analysis

A common theme that recurred in many of this year's problems is guessing: guess a part of the answer, and then with this part fixed do some computations to see if the guess leads to a correct answer. (Of course, the details of exactly what to guess and how to figure out if the guess leads to an answer differs.) Problems where this applies are: A, B, D, I, and K.

Regarding the difficulties of the problem, my guesses were even more way off than usual. I had expected problem B to be the second easiest problem which turned out to be *way* off (never underestimate the deterring effect of a large mass of text...). Instead, the two easiest problems turned out to be C (which I did guess) and K (which I should have guessed but somehow did not). Almost all teams solved both of these. The next chunk of problems were E and J, both solved by slightly more than half of the teams. Continuing in the same pattern, problems A and H, were solved by roughly a third of the teams. The remaining problems were less popular, with at most 7 solutions for each. Problem D was the only one that was not solved.

Here are the stats of how many solutions there were[1]:

Problem	A	B	C	D	E	F	G	H	I	J	K
Submissions	138	36	192	12	223	33	107	195	10	396	297
Solved	35	7	98	0	63	2	7	39	2	60	94

As an additional curiosity fact: the judges were somewhat worried this year that some team would solve all 11 problems. We estimated roughly 4 hours of coding time to solve all problems. Of course, this does not take debugging time into account or the fact that it is almost impossible to utilize the computer at a 100% efficiency. Turns out the problemset was more than hard enough, as usual.

Congratuliations to Zhejiang University for winning their first ICPC World Champions title!

[1] Seven of the submissions to problem D were actually submissions to problem C.

Problem A: To Add Or To Multiply

First we note that the number of multiplications used in any solution must be small: there can be at most 29 of them (since if $m > 1$ using more multiplications inevitably gives us numbers larger than 10^9, and if $m = 1$ multiplications are useless). So we'll try all possible values g for the number of multiplication operations used by the solution.

Next, given the value of g we'll have to figure out how to do the additions. For $0 \leq i \leq g$, let us denote by h_i the number of additions performed between the i'th and $(i+1)$'th multiplication operations. So for example h_0 is the number of additions performed before the first multiplication and h_g is the number of additions after the last one. Then, the resulting program transforms input x to $\alpha(x) = x \cdot m^g + a \cdot \sum_{i=0}^{g} h_i \cdot m^{g-i}$. The constraint that the input range $[p, q]$ is mapped to $[r, s]$ is then equivalent with requiring that $\sum h_i m^i$ lies in the interval $[(r - pm^g)/a, (s - qm^g)/a]$. Let us denote this new interval by $[u, v]$. If $[u, v]$ contains no integers, it will be impossible to choose valid h_i's (for this choice of g). Otherwise, there is a solution.

To minimize the total number of operations, i.e., the sum of the h_i's (plus g, which is fixed at this point) we can choose the h_i's in a greedy fashion: starting from h_m working our way down to h_0, try to choose h_i to be as large as possible without having $\sum h_i m^i$ exceeding v, with the exception that once the sum goes beyond u we should stop.

What remains is to be able to compare solutions to pick the lexicographically smallest shortest one, we'll leave this to the reader.

Problem B: Affine Mess

There are four unknown parameters to determine: the matching of input points to output points, the rotation, the scaling, and the translation. There are at most 6 ways of matching the input points to the output points, and at most 80 different possibilities for the rotation, so we can simply try all of these.

Once the matching of points and rotation has been fixed, determining if there are valid scaling and translation parameters (and verifying their uniqueness) boils down to simple algebra.

Problem C: Ancient Messages

This problem may at first glance appear pretty difficult, but the key point is to note that all the different hieroglyphic shapes contain a different number of "holes". This makes the problem one of the easiest: find the connected regions of the picture and then, for every black region R, count how many different white regions are adjacent to R (making sure to pad the entire picture with a white frame to make the outer region one single region).

A different, more fancy solution, is to use the Euler characteristic to compute the genus of each black region.

Problem D: Chips Challenge

This problem has a very distinct max flow smell, and indeed, flows are involved in the solution. First, suppose we think of the solution board as the adjacency matrix of a directed graph, with a component in row i column j being a unit of flow from i to j. Then, the constraint that there are equally many components in row i and column i just corresponds to each vertex having the same in-flow as out-flow. In other words, the solution is a *circulation* in a flow graph. To be specific, this flow graph is the graph on N vertices where there is capacity 1 from i to j if row i column j of the input board is not blocked. Furthermore, if we put cost 1 on those capacity 1 edges, the cost of the circulation precisely corresponds to the number of widgets used, so it seems hopeful that maximum cost circulations (which are perhaps less known than their cousins the min cost max flows, but can also be found efficiently) should be involved in the solution somehow.

However, there are two obstacles. First, there are some widgets that need to be used in the solution. This can be handled by either applying a lower bound on the flow of the corresponding edges (which makes finding an initial feasible solution a bit more complicated), or, more simply, by putting a large negative cost on these edges, heavily penalizing any solution that does not use them.

Second, there is the requirement that no row or column can contain more than an A/B fraction of all widgets. If this constraint was an absolute constraint, saying that at most m widgets can be placed on any row or column, it could easily be handled by introducing vertex capacities m in the graph (using the standard trick of splitting each vertex into an in-vertex and an out-vertex). Now, to handle the relative constraint, we can guess the value of m, solve the problem with that absolute bound, and check that the total number of widgets becomes at least $m \cdot B/A$. There are at most $N + 1$ possible values of m so this only incurs an extra factor N in the running time (which in fact can be avoided by reusing the result for $m - 1$ when computing the maximum number of widgets for vertex capacities m).

Problem E: Coffee Central

To solve this problem, it is useful to mentally rotate the city by 45 degrees. The problem then (essentially) asks for the maximum array sum in a subarray. This is a well-known problem that has a simple $O(n^2)$ solution (per query), based on computing prefix sums. The details (hidden in the "essentially" above) end up being slightly messier: since the city grid is discrete you can't really rotate it 45 degrees and apply the maximum array sum solution. Rather, you should translate the maximum array sum solution to the rotated case.

Problem F: Machine Works

This is a quite tricky problem. We'll process the machines by order of availability date, and use a carefully built data structure to keep track of what our best options so far are. The data structure consists of a list of straight lines of the form $M = c + kD$ for some constants c and k, indicating that at day D we can have M dollars. We keep this list pruned so that only lines that are the best possible for some day are actually present, and sorted by slope. For a given

day D we can find the maximum amount of money for day D in logarithmic time (in C++ STL the most natural way of implementing this gives log-squared time but that's OK).

When we process a new machine (with availability day D_i), we compute the maximum amount of money available on day D_i. Then, from any day after D_i, a new line is possible by switching to machine i on day D_i. (This new linear function is only valid from day D_i and onwards, but as we're processing the machines in order by availability day that's ok.) We then need to add this new line to the list of available lines (and prune away lines that are no longer optimal at any point), which can also be done in logarithmic time.

Problem G: Magic Sticks

The first step is to figure out what to do in the case when all segments have to be used to form a single polygon. If one plays around with it a little bit, it is pretty natural to guess that in that case the best solution is to put all the vertices on a common circle. To figure out the exact placements of the sticks one then has to compute the radius of this circle. This can be done by a standard bisection search (i.e., real-valued binary search). Some care needs to be taken here, as there are two cases to consider (whether the center of the circle lies inside the polygon or not).

With the single-polygon solution as a toolkit, it is not very hard to solve the full problem, all that needs to be figured out is how to partition the sticks into groups. There is a straightforward $O(n^3)$ dynamic programming solution to this, but this is too slow. To make it fast enough, the key insight is the following: if we're not using all the segments to form a single polygon, we should discard the longest segement. This leads to an $O(n^2)$ solution.

Problem H: Mining Your Own Business

This problem can be solved by finding the 2-*vertex-connected components* of the graph (or in other words, by the decomposition of the graph based on its *articulation vertices*). The set of 2-vertex-connected components form a tree, and it is easy to prove that it is necessary and sufficient to place an escape shaft in each leaf of this tree. The total number of ways in which this can be done is just the product of the sizes of the 2-vertex-connected components corresponding to those leafs.

There is a tricky border case to take care of, which is the case when the entire graph is 2-vertex-connected is a special case; then 2 escape shafts are needed (1 escape shaft is never sufficient since the junction of that single escape can collapse) and they can be placed in $\binom{V}{2}$ ways, where V is the number of vertices.

Problem I: Mummy Madness

There are two observations to make in this problem. The first, minor observation, is that the answer, if finite, is at most B (where B is the bound on the coordinates). This means that we can do a binary search for the longest possible survival time, answering "infinity" if it exceeds B.

To check if one can survive for T time steps, the main observation is that it suffices to check if there is some point that is reachable in T steps that no mummy can reach in T steps. This amounts to checking whether the union of n squares of side length $2T + 1$ completely

cover another square of side length $2T + 1$. This can be done in $n \log n$ time using a standard sweepline approach.

Problem J: Pyramids

This problem is pretty much a standard knapsack problem. We can precompute a matrix `back[k][N]`, indicating which pyramid to use if we have N cubes and are trying to produce a solution using k pyramids (or an indication that no such solution exists). This may seem like a huge matrix since N can be 10^6, but one can easily check that, when there is a solution for $N \leq 10^6$, the smallest number of pyramids is at most 6, and the number of different pyramids is small. Then for every query N we find the smallest k such that `back[k][N]` is possible.

People who are more clever than me may also realize that the problem is also solvable brute force.

Problem K: Trash Removal

Despite being the only real computational geometry problem in an unusually geometry-free problem set, this problem is not too hard. Without loss of generality, one can assume that in an optimal solution there are two verties of the polygon touching one of the two sides of the trash chute, so we'll try all such pairs of vertices and figure out the resulting chute width (which is easy).