ACM ICPC World Finals 2015

Solution sketches

Disclaimer *This is an unofficial analysis of some possible ways to solve the problems of the ACM ICPC World Finals 2015. 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. If you find an error, please send an e-mail to* `austrin@kth.se` *about it.*

— Per Austrin and Jakub Onufry Wojtaszczyk

Summary

The judges felt this year's problem set was a bit on the easy side (at the same time guessing that this probably meant that it was about the right difficulty since we always underestimate difficulties).

Congratulations to St. Petersburg State University of IT, Mechanics and Optics, the 2015 ICPC World Champions for their awesome performance, the first team ever to solve all problems in a World Finals!

In terms of number of teams that ended up solving each problem, the numbers were:

Problem	A	B	C	D	E	F	G	H	I	J	K	L	M
Solved	128	6	77	116	20	93	1	20	92	59	2	68	15
Submissions	140	20	150	185	62	324	16	28	237	110	1	81	108

In total there were 697 Accepted submissions, 562 Wrong Answer submissions, 161 Time Limit Exceeded submissions and 52 Run Time Errors. The most popular language was C++ by a wide margin: 1415 submissions compared to 74 for Java. There was a single submission in C (which was accepted!)

A note about solution sizes: below the size of the smallest judge and team solutions for each problem is stated. It should be mentioned that these numbers are just there to give an indication of the order of magnitude. The judge solutions were not written to be minimal and it is trivial to make them shorter by removing spacing, renaming variables, and so on. And of course the same goes for the code written by the teams during the contest!

Problem A: Amalgamated Artichokes

Shortest judge solution: 332 bytes. Shortest team solution (during contest): 343 bytes.

This was the easiest problem in the set, and every team solved it in 47 minutes.. Simply go through the points in increasing order, keeping track of the maximum seen so far. When processing a new value, compare its difference to the maximum seen so far and save this difference if it is the largest seen so far.

Problem B: Asteroids

Shortest judge solution: 3480 bytes. Shortest team solution (during contest): 4026 bytes.

This problem is conceptually simple: simply run a ternary search for the answer, but the details makes it a bit difficult to implement this correctly.

The first detail to resolve is that we want to figure out if the polygons never intersect or touch. Since it can be the case that the two asteroids just touch in a single vertex for a single time step, we need to be a but careful here, and the safest course is to use exact, integer arithmetic, when checking if the two polygons ever touch.

The second detail to resolve is: for a given time t, what is the overlap area at this time? This boils down to computing the intersection of two convex polygons. In general, polygon intersection is pretty messy, but for convex polygons, one can use the following approach. The vertices of the intersection are: (i) the vertices of polygon 1 that are contained in polygon 2, (ii) the vertices of polygon 2 that are contained in polygon 1, and (iii) the intersection points of the edges of the two polygons. To compute the order in which these vertices come, we can sort them by angle around the center of mass of the intersection polygon (the center of mass is simply the average of the points). Now we have the intersection polygon and can compute its area.

The last (but minor) detail to resolve is that the intersection area can be constant and maximum for a time period, and we want to find the smallest time this happens. So if in the ternary search we look at times $t_1 < t_2$ and encounter two intersection areas that are the same (up to some small ϵ error from floating point computations) we need to make sure to throw out the times $> t_2$ from our search rather than the times $< t_1$.

Problem C: Advance Catering Map

Shortest judge solution: 1142 bytes. Shortest team solution (during contest): 1431 bytes.

This is a classic max-flow/bipartite matching application. The transport schedule is defined by deciding, for each location $2 \leq i \leq n+1$, which location $1 \leq j < i$ the catering team that serves request i comes from. In order for the schedule to be valid, it needs to satisfy two conditions:

- For the company location, node 1, at most k locations can select it as their predecessor location (reflecting that we only have k catering teams available).
- For each other location $2 \leq j \leq n+1$, at most one location i can choose j as their predecessor location.

In other words we have a bipartite graph with n vertices on the left side and $n+1$ vertices on the right side, with an edge from node i on the left side to node $j < i$ on the right side, with cost c_{ji} (the cost of transporting a catering set from location j to location i). An optimal transportation schedule is given by a minimum cost "matching" in this graph, where node 1 on the right side is allowed to be used up to k times (rather than at most once as in an actual matching).

Problem D: Cutting Cheese

Shortest judge solution: 983 bytes. Shortest team solution (during contest): 966 bytes.

This was one of the easiest problems in the set. To compute where to make the ith cut, simply binary search for the z coordinate where the volume up to coordinate z is an i/s fraction of the total volume of the cheese.

To compute the volume up to a given z coordinate, we subtract the volume of holes below this z coordinate. This involves calculating volumes of spherical caps, which is a relatively straightforward calculus problem; the volume of a spherical cap of height h of a sphere of radius 1 is

$$\int_{x=1-h}^{1} \pi(1-x^2)\mathrm{d}x = \pi\left(h - \tfrac{1}{3} + \tfrac{(1-h)^3}{3}\right).$$

(And for a spherical cap of height $h \cdot r$ of a sphere of radius r, the volume is scaled by a factor r^3.)

Problem E: Parallel Evolution

Shortest judge solution: 1756 bytes. Shortest team solution (during contest): 1576 bytes.

Fun fact: this problem was originally in the reverse direction, so the description of the solution below is "backwards" from the point of view of the problem, but this does not really change anything.

We have two copies of a string of characters (only three different characters appear, but we will not use this fact in our solutions). We want to subsequently remove characters from the copies, so that all of the strings given on the input appear in one of the copies during the process.

In graph-theoretic terms, we have a directed acyclic graph (the strings being the vertices, and an edge from string A to string B if B is a subsequence of A), and we would like to know if there are two paths, each starting at the base string, that together cover all vertices.[1]

Like most problems on DAGs, this has a relatively standard dynamic programming solution, in this case with running time $O(N^2)$. There is a catch however: a naive way of creating the graph takes time $\Omega(N^2S)$ (comparing all $\approx N^2/2$ pairs of strings to each other, each for $\Omega(S)$ time). So we have to be a bit more clever, and in fact, once one understands the problem well enough to construct the graph more quickly, it turns out that the problem admits a simple greedy solution.

First, note that a topological sort of the graph can be obtained by sorting the strings by decreasing length (which can be done in $O(NS + N \log N)$ time). Now we go through the strings in topological order. At each step, we consider pairs of decreasing sequences that both start with the base sequence and then use the first k elements of the topologically sorted sequence. We only care about the last string of each sequence (as they are the only ones we have to compare the next string to). We will show there are at most two pairs of strings we need to consider as "possible lasts". Furthermore, if there are two of them, they have a special structure: they share an element, and this shared element is smaller then the other two elements in the pairs. I.e., they are of the form (A, B) and (A, C), where $A \leq B$ and $A \leq C$.

[1] In combinatorial terms, we'd like to know if a given poset can be covered by two chains. By Dilworth's Theorem, this is equivalent to asking if the width of the poset is ≤ 2. But while this is something everyone should know, it is irrelevant for this particular problem.

Initially we have a single pair – the two base strings. In each step we add another string, and we want to append it to one of the sequences. We consider two cases.

Case 1: a single candidate If we currently have a single candidate pair of sequences, we have to append our new string to one of the elements of the pair. Depending on whether this can be done in 0, 1, or 2 different ways, we get 0, 1, or 2 different new candidate pairs (in the case of 0, answer is impossible, and in the case of 2, note that the two candidate pairs have the special structure described above).

Case 2: two candidate pairs We have two candidate pairs (A, B) and (A, C), and we try to add a new element D. If $D \leq A$, we can replace any string by D, nominally giving the three candidates (D, A), (D, B), and (D, C). However, the pair (D, A) is redundant – since $A \leq B$, any completion of (D, A) to a full solution is a valid completion of (D, B) as well.

If D cannot be appended to A, then we will have a sequence ending with A and a sequence ending with D — thus we have only one pair (D, A) of possible sequence ends. To check if this is indeed possible, we have to compare D to B and C, and if it can be appended to either of them, (D, A) becomes our candidate pair, while otherwise we return impossible.

This algorithm obviously makes a constant number of comparisons (2.5 comparisons per step, for we can make three comparisons only every second step), so it runs in a total time $O(NS)$.

Problem F: Keyboarding

Shortest judge solution: 1801 bytes. Shortest team solution (during contest): 1285 bytes.

This problem was one of the easiest (though sufficiently many of the judges goofed up in strange ways while solving this problem that we erronously thought it was harder than it was)

Consider the following directed graph $G = (V, E)$. The nodes V are all triples (i, j, k), where (i, j) is a position on the virtual keyboard $1 \leq i \leq r$, $1 \leq j \leq c$, and k is a position in the text $1 \leq k \leq n + 1$ (where n is the length of the text, in which we include the final '$\star$'). The edges from a node (i, j, k) are as follows:

1. For each of the four direction buttons, there is an edge to (i', j', k) where (i', j') is the position on the virtual keyboard reached from (i, j) by pressing that direction button.
2. If the character at position (i, j) on the keyboard equals the k'th character in the text, there is an edge to $(i, j, k + 1)$.

The number of keys required is now the minimum distance from $(1, 1, 1)$ to any node of the form $(i, j, n + 1)$. This can be computed with BFS – the number of nodes in the graph is $|V| = r \cdot c \cdot n \leq 50 \cdot 50 \cdot 10\,000 = 25\,000\,000$, and the number of edges is at most $5|V|$, so this is feasible.

Problem G: Pipe Stream

Shortest judge solution: 568 bytes. Shortest team solution (during contest): 1432 bytes.

Effectively, every knock on the pipe answers the question "did the Flubber already get to this point", which tells us "is the velocity of the Flubber greater or equal to the minimum speed needed to get to this point." So, the problem would be very easy if the pipe had infinite length. For an infinitely long pipe, we would simply run a binary search on the velocity, and return the answer in $\lceil \log_2 \left(\frac{v_2 - v_1}{t}\right) \rceil$ knocks. However, due to the finite length of the pipe, some velocity queries are impossible to make after some points in time. This may cause the whole binary search to be impossible to perform (as in sample input 3), or it may cause us to have to adapt the search, possibly using more queries (as in sample input 2).

Notice that the effect of the flubber moving out of the pipe is that some queries about the last parts of the velocity interval become illegal. This means that if the velocity is in some final part of the possible interval, and we have not determined what it is yet, we will not be able to determine it in the future. So, our knocking plan must be able to tell these velocities apart before the flubber flows out.

Let us describe how we will construct a good strategy of knocking at the pipe. We will always knock as soon as we can (that is, at s, $2s$, $3s$, etc. seconds) After k knocks, our strategy will have already adapted to determine the velocity (within a t error range) in some final range of the velocity interval, and we are still searching in the range $[v_1, v(k)]$. If not for the need to determine some answers fast, we would have, at this point, divided the range of velocities into 2^k intervals, and the strategy would tell us in which of these intervals the real answer is. However, some of these potential intervals might have been used up – so, the range of $[v_1, v(k)]$ is only divided into some smaller number $n(k) \leq 2^k$ intervals, and our strategy will tell us in which of these intervals the answer is. If $t \cdot n(k) \geq v(k) - v_1$, we can just let these intervals be an equidivision of the velocity range, and k is going to be the number of knocks our strategy uses.

The initial values are easy: $v(0) = v_2$ and $n(0) = 1$ – before knocking, we are searching the $[v_1, v_2]$ range, and the range is "divided" into a single interval. Let's see what happens when we add another knock. Before we do this, we need to determine what range of velocities becomes impossible to query (if any) — the higest velocity that's possible to query at time k is simply $v_f(k) = \frac{l}{s(k+1)}$. We need to determine the velocity of the Flubber for velicities above $v_f(k) + t$ in the first k steps. In other words, if $v(k) > v_f(k) + t$, then we need to "use up" $n_f(k) = \lceil \frac{v(k) - v_f(k) - t}{t} \rceil$ intervals out of our $n(k)$ intervals to make sure we cover the high velocities with sufficient accuracy before it is too late. For the remaining $n(k) - n_f(k)$ intervals, we do not decide yet what are they going to be, but whatever they are, we ask a new query to split them in two. Thus, $n(k+1) = 2(n(k) - n_f(k))$, and $v(k+1) = v(k) - n_f(k) \cdot t$.

This analysis allows us to solve the problem. We will iterate over k, keeping track of $v(k)$ and $n(k)$. Once $t \cdot n(k) \geq v(k) - v_1$, we can answer k. If $n(k)$ drops to zero or below at any point, we answer "impossible". becomes how big k can become? It's pretty difficult to get an accurate estimate, but after getting some intuition for the problem, it is relatively easy to believe that it's not going to be very high. The worst test case we found (subject to the input bounds in the problem) had an answer of 74. If you can beat this, please let us know!

Problem H: Qanat

Shortest judge solution: 650 bytes. Shortest team solution (during contest): 668 bytes.

The first thing to note is that the problem is scale-invariant: if we multiply the dimensions w and h by a factor f, the optimal positions to place the shafts will be scaled by a factor f and the total excavation cost will be scaled by a factor f^2. Thus let us normalize the size to $w = 1$ and $h < 1$.

Now suppose we have fixed the placement $0 < x < 1$ of the right-most vertical shaft. An optimal solution subject to this restriction looks as follows:

- The dirt in the mother well is transported directly upwards for a cost of $h^2/2$
- The dirt in the horizontal channel between points x and 1 is split up, some of it being taken right and then up the mother well, and some of it being take left and then up the shaft at position x. Given the position $x \leq x' \leq 1$ of the breakpoint – how much dirt to move to the left and how much to move to the right – the cost of this part is a quadratic function in x', so with a little pen and paper work one can figure out the best choice of x' and therefore the cost of this part.
- The remaining dirt is excavated according to an optimal solution with $n - 1$ shafts in a qanat of width x and height $h \cdot x$. By scale invariance this can easily be computed from an optimal solution with $n - 1$ shafts in a qanat of width 1 and height h, so by iteratively computing the solution for $1, 2, \ldots$ shafts, we can assume we know the cost of this part as well.

When one works out the details (with some more pen and paper work), one sees that the resulting cost is a quadratic function in x, so the choice of x that minimizes the overall cost can again be figured out.

Problem I: Ship Traffic

Shortest judge solution: 765 bytes. Shortest team solution (during contest): 815 bytes.

This problem is somewhat intimidating due to the long problem statement. However, once you get through the problem statement, it is actually pretty easy. Each ship determines an interval of time when the ferry cannot enter the lane, which translates to a time interval where the ship cannot start the crossing. You do have to take care to calculate everything correctly. Fortunately, the answer is continuous, so we don't have to worry about precision that much, and just use doubles.

Once we know all the forbidden intervals, the easiest way to determine the longest allowed interval is to sort all the forbidden interval beginnings and endings, going through them one by one, and keeping track of how many forbidden intervals are now open.

Problem J: Tile Cutting

Shortest judge solution: 1201 bytes. Shortest team solution (during contest): 1478 bytes.

This problem, despite the pictures, is not about geometry, but rather about number theory. Consider a particular way to get a tile according to the rules. We have a rectangular tile of size $x \times y$. Let the cut on the lower edge be a from the left and b from the right (so that $a + b = x$). To make a parallelogram, the cut on the upper edge has to be b away from the left edge. Similarly, if the cut on the left edge is c away from the bottom, and d away from the top (with $c + d = y$), then the cut on the right edge has to be d away from the bottom and c away from the top.

The area of the whole rectangle is $(a + b)(c + d)$, while the area of the triangles we cut away is $ac/2$, $bd/2$, $ac/2$ and $bd/2$, and the area of the tile is $ac + bd$. A set of a, b, c, d uniquely determines a way of cutting a tile. So, to get a way to cut a tile of area A, we need to represent A as the sum of two numbers, and then represent each of those two as a product of two numbers.

The number of ways to represent a number as a product of two numbers is simply the number of divisors of the number, which we'll denote $d(n)$. Calculating $d(n)$ for all n up to 500 000 is easy. So the number of ways to get a tile of size A is $N(A) := \sum_{x=1}^{A-1} d(x)d(A - x)$, and we need to calculate the maximum of this function over the input range.

There are two approaches that you can take here. The standard one, that most (all but one?) teams used, is to use the Fast Fourier Transform. This allows you to compute the convolution (a function like the one above) in $O(B \log B)$ time for all A up to B.

However, there is also another approach that you can take if you don't know the FFT algorithm (although you really should learn it, if you don't know it!). You can obviously calculate all the $N(A)$ values in quadratic time by just applying the formula directly. This is too slow to pass within the two-second time limit, but it's enough to do it locally on the contestants computer. This allows for a precomputation-based solution.

The limit for the source code size is 128 KB. The $N(A)$ values go up to some 170 million in the range we care about, so even storing them as decimal representations we can fit one value in 9 bytes. So you can easily fit 12 500 numbers into your source code. So, divide the whole range $[1, 500\,000]$ into intervals of size 40 each. Precompute and store in your code the maximum value in each of these intervals. Using these values, you can easily answer queries quickly, and the precomputation should easily run within a few minutes.

Problem K: Tours

Shortest judge solution: 1224 bytes. Shortest team solution (during contest): 1955 bytes.

Let us translate the problem into graph theoretic language. We have a simple graph G which contains at least one cycle. We want to find all numbers k, such that the edges of G can be colored with k colors, so that each simple cycle contains the same number of edges of every color. Obviously k will have to divide the length of every simple cycle in G. The author expected some of the contestants to believe this was a sufficient condition without sufficient analysis, and fail.

First consider any edge e in G which is a *bridge* (that is no simple cycle passes through e). Then the answer to the problem in G is the same as in G with e removed — we may give e

any color, and it will not occur in any simple cycle, so it will not change the correctness of the solution. Thus we may begin by removing all bridges from G, from now on we will assume G is bridgeless.

Now consider the following relation on edges: we say two edges e and f are *related*, removing both e and f from G increases the number of components. This is clearly an equivalence relation. It can be proved that k is a good solution if and only if the size of every equivalence class is divisible by k. (The "only if" part is easy to prove, and the "if" part a bit more work – this is left as an exercise.) Thus the problem is equivalent to finding the greatest common divisor of the sizes of all equivalence classes.

Note that the contestants need not prove the characterization above, just gain enough intuition about the problem to become sure it is true.

The sizes of the equivalence class containing an edge e can be computed by removing that edge and then checking how many bridges there are in the resulting graph (and adding 1 to the result). Since bridges can be found in linear time using DFS, this gives an $O((|V| + |E|)^2)$ solution.

Problem L: Weather Report

Shortest judge solution: 1206 bytes. Shortest team solution (during contest): 1054 bytes.

This problem asks for the cost of a Huffman coding of the 4^n strings of length n over the alphabet $\{S, C, R, F\}$, where the probability of a string with f_S S's, f_C C's, f_R R's, and f_F F's is $p_{\text{sunny}}^S \cdot p_{\text{cloudy}}^{f_C} \cdot p_{\text{rainy}}^R \cdot p_{\text{frogs}}^F$. Unfortunately, 4^n is so large that we can not simulate the Huffman coding procedure naively. However, due to the product structure of the distribution, there will be many nodes in the Huffman coding having the same probability, so we can group these together and just keep their count.

Initially, there are $\binom{n}{f_S, f_C, f_R, f_F} = \frac{n!}{f_S! f_C! f_R! f_F!}$ nodes with the probability $p_{\text{sunny}}^S \cdot p_{\text{cloudy}}^{f_C} \cdot p_{\text{rainy}}^R \cdot p_{\text{frogs}}^F$ (for each combination of non-negative values f_S, f_C, f_R, f_F summing up to n). Now we proceed with Huffman encoding: pick the smallest probability node, say this probability is p. If there are K nodes with this probability, they give rise to $\lfloor K/2 \rfloor$ nodes of probability $2p$, and if K is odd then one remaining node of probability p gets matched with one of the nodes with the second smallest probability value.

Problem M: Window Manager

Shortest judge solution: 3481 bytes. Shortest team solution (during contest): 3520 bytes.

This problem is a bit tedious, but the test data is small enough that one does not have to be particularly clever. Just keep a list of open windows around, and whenever you want to open or resize a window, or find the window containing a point, do a linear search.

The only part that might be a bit tricky is the MOVE operation. The way different windows affect each other is non-trivial, and the easiest thing to do is to go through the windows in reverse order (e.g., if we are moving towards the right, go through the windows by decreasing x coordinate of the left border) and compute how much the window could possibly move. Then we process the move request by actually moving the windows (either as far as requested or as far as possible).