ACM ICPC World Finals 2013

Solution sketches

Disclaimer *This is an unofficial analysis of some possible ways to solve the problems of the ACM ICPC World Finals 2013. The writeups for problems B and I are written by Jakub "Onufry" Wojtaszczyk. Should an error be found, we will blame each other for the cause, but 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.

— Per Austrin

Summary

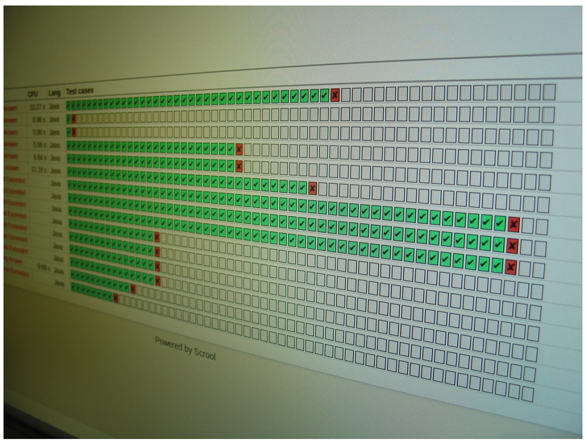
My guesstimated order for the problems to be solved was AFJDCHBKIEG The actual order in which the problems were solved was FDAJCHEIKB, with G left unsolved. In terms of number of teams that ended up solving each problem, the numbers were:

Problem	A	B	C	D	E	F	G	H	I	J	K
Solved	76	2	51	62	5	107	0	66	12	46	3
Submissions	219	34	169	380	50	273	35	160	37	273	9

In total there were 430 Accepted submissions, 815 Wrong Answer submissions, 315 Time Limit Exceeded submissions and 79 Run Time Errors. The most popular language was C++ by a wide margin: 1347 submissions compared to 323 for Java. There were no C submissions.

Congratulations to St. Petersburg State University of IT, Mechanics and Optics, the 2013 ICPC World Champions for their awesome performance! They were very very close to being the first team ever to solve all the problems at a World Finals. This is how close they were: the best of their many attempts to solve the last problem (Map Tiles) were only stopped due to a few extra cases that we added shortly before the contest (on those cases, their code used more than 5x the time limit). Maybe the following pictures give an indication of the excitement in the judge's room during the last few minutes of the contest.

This looks promising...



So close!

In case you are wondering what those last three cases where, they are depicted on the last page of this document (if I remember correctly the submissions failed all these three).

Subnormal behavior. The heroes of the day in the judging room were the University of Tokyo with their work on problem B (Hey, Better Bettor). Their initial solution was fast enough (but a close call) on the hardest cases but then timed out (with a wide margin) on a seemingly simple test case. This was very confusing to us, because the code was very simple, and looked like it should run exactly the same instructions with exactly the same memory access patterns, regardless of the input case. This caused *a lot* of head-scratching and headache and occupied the better half of the judges and the Kattis group for quite a while. Fortunately the Tokyo team rescued us from our worries by figuring out how to fix their code and got the first accepted solution to the problem. The issue turned out to be the following: in their solution they were precomputing all the powers $1, \frac{p}{1-p}, \left(\frac{p}{1-p}\right)^2, \left(\frac{p}{1-p}\right)^3, \ldots$ up to some number. After a while, the numbers became very small but non-zero, and for some values of p they reached long sequences of subnormal floating-point numbers. Unfortunately, calculations on subnormal floating-point numbers can be *really* slow. Adding an if statement that zeroed out the number if it was less than 10^{-100}, their solution became 10 times faster. A good lesson to learn!

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: Self-Assembly

Shortest judge solution: 774 bytes. Shortest team solution (during contest): 678 bytes.

This was one of the simplest problems, though I underestimated how intimidating it looks. The main idea is to realize that because both rotations and reflections of pieces are allowed, the geometry of the problem is completely irrelevant. That is, it suffices to check if there is an infinite chain of pieces $P_1, P_2, \ldots$ such that P_i can be connected to P_{i+1}. If such a chain exists then using rotations and reflections it can always be laid out in such a way that only the adjacent pieces touch each other (e.g. you can make it so that you always go upwards or to the right). If such a chain doesn't exist then clearly the answer is "bounded".

This reduces the problem to checking for cycles in a graph that consists of the $n \leq 40\,000$ pieces. This is however still too much, but there is one small additional trick: only the connections matter, so you can consider the graph consisting only of $2 \cdot 26$ nodes $A+, A-, \ldots, Z+, Z-$.

Problem B: Hey, Better Bettor

Writeup by Onufry

Shortest judge solution: 538 bytes. Shortest team solution (during contest): 1635 bytes.

This was one of my favorite problems in this contest — non-trivial, but with surprisingly little code once you got it, and additionally having "implicit" limits (in the sense that the precision with which x and p are the constraints, but the time complexity isn't really easy to express in terms of the big-O notation).

The first key to solving this problem is noticing that your next move should not depend on the history, but just the amount of money you currently have, as history gives you no extra information. Thus, the strategy can be described simply by defining two numbers — how much do you have to lose to quit, and how much do you have to win to quit. Let's denote the first number L and the second W. This means that at the end of the game you will always have either won W or lost $(1-x)L$ dollars, and the only question left is what is the probability of the first event (depending on the choice of W and L).

Denote the chance of winning in the end if at the moment you have D dollars by $P(D)$. Obviously, $P(W) = 1$ and $P(L) = 0$ (well, except for the very special case of $W = L = 0$, which means we don't play at all, and the expected win is obviously zero; we ignore this case from now on). For any D between L and W we have $P(D) = p \cdot P(D+1) + (1-p) \cdot P(D-1)$. This can be easily transformed to $P(D+1) = \frac{1}{p}P(D) - \frac{1-p}{p}P(D-1)$ — a recursive sequence definition. Solving such recursive equations is a well-known problem, and in this case we get

$$P(D) = \alpha + \beta \left(\frac{1-p}{p}\right)^D.$$

We now have to choose α and β to fit the boundary conditions of $P(L) = 0$ and $P(W) = 1$. This is just a system of two linear equations, and after solving them we get:

$$\beta = \frac{1}{r^W - r^L}; \qquad \alpha = \frac{-r^L}{r^W - r^L},$$

where $r = \frac{1-p}{p}$. We are interested in $P(0)$, which is $\alpha + \beta = \frac{1-r^L}{r^W - r^L}$.

So, for a given L and W we are able to determine the probability of winning, and thus — the expected value of the gain. Thus, we can check all reasonable values of L and W and choose the best pair. The last question remaining is "so what are the reasonable values of L and W, anyway"? The programmer's approach to this problem is to simply take the worst case (it both seems obvious and is true that the larger p and the larger x, the larger this range is going to be, so the worst case is $p = 0.4999$, $x = 0.9999$), incrementally increase the range, and check at what point does it stop increasing the expected value. The range we get this way is $L = 20528$ and $W = 2498$ (which means it's OK to just check all the possibilities). Formally, one also needs to prove that the expected value will not go up after a period of decreasing — an argument involving the convexity of the expected value in this problem which we'll leave as an exercise.

Problem C: Surely You Congest

Shortest judge solution: 1712 bytes. Shortest team solution (during contest): 1870 bytes.

First, we need to figure out which streets can possibly be used (and in which direction), i.e., those which are part of some shortest path to the downtown node. This is standard and can be done by running Dijkstra's algorithm from the downtown node. Writing $d(u)$ for the distance from the downtown for intersection u, an edge (u, v, t) can be used from u to t if and only if $d(u) = d(v) + t$.

Checking this for all edges gives a directed acyclic graph consisting of all edges that can be used by the commuters. Next, we observe that two commuters that are at two nodes that have a different distance to downtown can never interfere with each other. This means that we can group the commuter based on their distance to downtown, and process each group separately.

Processing such a group of commuters is again a fairly standard task, namely finding edge-disjoint paths. Add a dummy node s and connect it to all the starting nodes of the commuters in the group (with multiplicities, so you allow parallel edges). Then, the maximum number of commuters that can go simultaneously within this group equals the maximum number of edge-disjoint paths from s to downtown, which equals the max-flow from s to downtown if you put unit capacities on all the (directed) edges.

Problem D: Factors

Shortest judge solution: 1241 bytes. Shortest team solution (during contest): 841 bytes.

This problem is not very hard but requires a small leap of faith (or good number-theoretic intuition).

Given a number k with prime factorization $p_1^{e_1} p_2^{e_2} \ldots p_t^{e_t}$, the first observation is that the number of arrangements $f(k)$ of the prime factors is the multinomial number $\binom{e_1+\ldots+e_t}{e_1,e_2,\ldots,e_t} = \frac{(e_1+\ldots+e_t)!}{e_1!e_2!\ldots e_t!}$. Given the numbers $e_1, \ldots, e_t$, this is easily computed (though some care has to be taken to avoid overflow).

Note that permuting the exponents e_i or changing the values of the primes p_i does not change the value of $f(k)$. Since we are looking for the smallest k with the given value of $f(k)$, this implies that we may without loss of generality restrict attention to numbers of the form $e_1 \geq e_2 \geq \ldots \geq e_t$ and that p_i is the i'th prime (i.e., $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, and so on).

In other words, we can try to generate all numbers k that are of the form $k = 2^{e_1}3^{e_2}5^{e_3}\ldots$ satisfying $e_1 \geq e_2 \geq e_3 \geq \ldots$ and $1 < k < 2^{63}$. It turns out that there are exactly 43606 such numbers, so this can be done quickly. For each such number we compute $f(k)$ and check if it is less than 2^{63}, and then construct a lookup table which for each possible value of $f(k)$ (it turns out there are 19274 of them) gives the smallest preimage k.

Apart from overflows, there is only one corner case, namely $n = 1$, for which the answer is $k = 2$ (since the problem requires $k > 1$). Fortunately, this case was covered as the first sample data. Unfortunately, many teams didn't seem to notice this, and submitted solutions that failed on the sample data...

Problem E: Harvard

Shortest judge solution: 2926 bytes. Shortest team solution (during contest): 2843 bytes.

This was a pretty hard problem, not because its algorithmically deep but just because it's a bit messy. The main idea is to try all possible assignments of variables to banks, with the following optimizations.

1. We can always assume that bank 0 is full, because referencing variables allocated to bank 0 is always free.
2. Once bank-0 is allocated, consider the sub-program obtained by removing all the references to bank-0 variables. For each pair of remaining variables i and j, let C_{ij} be the number of times a reference to variable i is followed by a reference to variable j when executing the program (which can be counted pretty easily). Then given a bank assignment to the remaining variables, the total number of BSR instructions that need to be executed

is simply the sum over C_{ij} for all i, j that are allocated to different banks. This gives a way to very quickly evaluate the quality of a bank assignment, without going through the entire program.

3. Banks 1 and up are symmetric, so one should remove this symmetry by e.g. only generating assignments where the smallest variable in bank 1 is smaller than the smallest one in bank 2 which is smaller than the smallest one in bank 3, and so on.

4. If some bank only contains r variables, than all other banks should contain at least $s - r + 1$ variables, because otherwise two banks could be merged which always decreases the cost of the bank allocation.

With optimizations 1, 3, and 4 and at most 13 variables, it turns out that there are in the worst case around 3 million bank assignments to try. With optimization 2, each such bank assignment can be quickly evaluated.

Problem F: Low Power

Shortest judge solution: 541 bytes. Shortest team solution (during contest): 517 bytes.

This problem is also pretty easy. We describe a way to determine if a given target difference d is possible to achieve; finding the minimum is then a simple matter of binary search.

Sort the inputs so that $p_1 \leq p_2 \leq \dots p_{2nk}$. First observe that we may without loss of generality assume that in an optimal solution, the smallest outputs in each pair of chips will be of the form p_i, p_{i+1} (so that the power output difference is $p_{i+1} - p_i$).

Let us say that the *first* battery of a machine is the one with smallest power in the machine. Note that if the first battery of a machine is p_i then by observation above we can assume that the power difference of that machine is $p_{i+1} - p_i$.

Now consider the machines sorted in increasing order of power of their first battery. Clearly, the first machine has power difference $p_2 - p_1$. For the second machine, the first battery can be any one of $p_3, \dots, p_{2k+1}$. We then greedily choose the first $i^* \geq 3$ such that $p_{i^*+1} - p_{i^*} \leq d$, and use this for the second battery (if no such $i^* \leq 2k + 1$ exists, there is no solution). Then we look at the third machine. The first battery of this can be either of $p_{i^*+2}, \dots, p_{4k+1}$, and we again greedily choose the first one resulting in a power output smaller than d. We continue this process until all batteries have been assigned.

Problem G: Map Tiles

Shortest judge solution: 3798 bytes. Shortest team solution (during contest): N/A.

This was probably the hardest problem in the set (at least it was the one I struggled the most with). The basic idea, which is completely standard, is to simply try "all" possible grid placements and see which one gives the best result.

Let us write $m \leq 10$ for the maximum number of grid tiles in any row/column of the grid (well, in some cases 11 tiles might be needed but never mind).

To execute this idea, we first have to find a reasonably small set of candidate grid placements. Consider an optimal grid placement. By shifting it to the left as far as possible, we may assume that one of the following two cases occur.

Case 1: a vertex of the polygon lies on a vertical line L of the grid. In this case, consider shifting the polygon upwards, keeping the vertex on the vertical line. By a similar argument, one of the following two happens:

Case 1a: a vertex of the polygon lies on a horizontal line of the grid. There are only n^2 grid placements of this type: specifying which vertex lies on a vertical line and which vertex lies on a horizontal line completely specifies the grid.

Case 1b: a vertex of the grid lies on an edge e of the polygon (and the edge is not vertical). Suppose this grid vertex is t steps horizontally away from L. Then the grid vertex can be found by moving L horizontally a distance of $t \cdot x_s$ and intersecting it with the polygon edge. Trying all possible values of the polygon vertex that lies on L (n choices), the polygon edge e (n choices), and the offset t (m choices), this gives a total of at most n^2m candidate placements.

Case 2: a vertex of the grid lies on an edge of the polygon. Consider shifting the grid along the edge of the polygon (so that the vertex of the grid remains on that edge) as far as possible. Again, there are two possible outcomes:

Case 2a: a vertex of the polygon lies on a horizontal or vertical line of the grid. This case is analogous to case 1b (but possibly with a horizontal instead of a vertical line).

Case 2b: another vertex of the grid lies on some edge of the polygon, and this second polygon edge is not parallel to the first one. This case is somewhat similar to cases 1b and 2a, but now we need to guess both a vertical and a horizontal offset and then do intersection of the two polygon edges. This gives a total of at most n^2m^2 candidate placements. The three test cases illustrated on the last page of this document were designed to trigger as many candidates of this form as possible.

This gives a set of $O(n^2m^2)$ candidate grid placements. It turns out that many of these candidates are the same, and one should take care to remove duplicates. I don't have a good estimate for how much this saves, but on the judge data it tends to be around a factor 3-5, which might be sorely needed depending on how one solves the second part of the problem, described next. In the worst cases we had found, there are always less than 50 000 candidate grid placements after removing duplicates.

Given a candidate grid placement, we then need to figure out how many grid tiles it uses. The simplest way to do it would be to simply check for each grid tile whether it intersects the polygon. This is however too slow (time $\Omega(m^2n)$ with pretty lousy constants), so something faster is needed. I went for a pretty messy flood-fill variant. The basic idea is to first trace through the polygon and mark the tiles that it passes through (in time $O(mn)$ with pretty reasonable constants) and then do flood-fill to discover the rest of the tiles. Unfortunately this simple idea needs a bit of refinement to work correctly since one has to deal with polygon edges that run along the grid lines. This can be dealt with by also including the grid lines and grid vertices in the graph of tiles that we are flood-filling and with some careful coding one gets a correct (well, at least it passes the judge data) but somewhat sluggish solution.

A different (and faster!) approach is to do something reminiscent of a point-in-polygon test. For each row of tiles, find all x-coordinates where the polygon enters/exits the top of the row and all x-coordinates where it enters/exits the bottom of the row. Those tiles where the polygon either goes into the interior of the tile or has entered/exit the top/bottom an odd number of times will be used.

Problem H: Матрёшка

Shortest judge solution: 1396 bytes. Shortest team solution (during contest): 1531 bytes.

Let us write $x_0, \ldots, x_{n-1}$ for the sizes, $S(i,j)$ for the minimum number of operations needed to assemble the dolls in positions $i, i+1, \ldots, j-1$ to a single group (not necessarily consisting of consecutive dolls), and $M(i)$ for the minimum number of operations to assemble the dolls in positions $i, i+1, \ldots, n-1$ into complete sets (consisting of consecutive dolls). Finally let us say that an interval (i,j) is ok if $x_i, \ldots, x_{j-1}$ is a permutation of the integers from 1 to $(j-i)$.

What we seek is then $M(0)$. We can write the following recurrence for $M(i)$, with base case $M(n) = 0$.

$$M(i) = \min_{\substack{j \in \{i+1, \ldots n\} \\ (i,j) \text{ is ok}}} S(i,j) + M(j).$$

Thus with access to $S(i,j)$, computing the $M(\cdot)$ function can be done in $O(n^2)$ time using dynamic programming.

Computing the $S(\cdot,\cdot)$ values is another dynamic programming exercise. The optimal way of combining the dolls in the interval has as last operation the combination of a group consisting of the dolls from i to $k-1$ with a group consisting of the dolls from k to $j-1$ for some k between $i+1$ and $j-1$ (inclusive), so we can write the following recursion when $j \geq i+2$ (the base case when $j < i+2$ is left as an exercise):

$$S(i,j) = \min_{i<k<j} S(i,k) + S(k,j) + C(i,k,j),$$

where $C(i,k,j)$ is the cost of combining a group $[x_i, \ldots, x_{k-1}]$ with the group $[x_k, \ldots, x_{j-1}]$. The cost $C(i,k,j)$ can be computed as follows: if the group that contains the smallest dolls contains the t smallest dolls among $x_i, \ldots, x_{j-1}$, then $C(i,k,j) = j - i - t$ (because we have to open all but those t smallest dolls in order to combine the two groups).

If implemented naively, this leads to an $O(n^4)$ solution which is too slow, but with a little care the recursion for $S(\cdot,\cdot)$ can be implemented to give an $O(n^3)$ solution.

Problem I: Pirate Chest

Writeup by Onufry

Shortest judge solution: 1443 bytes. Shortest team solution (during contest): 1224 bytes.

This proved to be one of the tougher problems in the competition. The most naive solution to this problem would take $\Omega(n^6)$ time. With very little effort, this can be improved to a $\Theta(n^4)$ solution, but this is still too slow. Fortunately, yet another factor of n can be shaved off in the running time.

It is instructive to first consider a 1-dimensional variant of the problem. Given an array $x_0, \ldots, x_{n-1}$, and a subsequence $[j,k)$ of it (that is, $x_j, \ldots, x_{k-1}$), we are interested in the values $V(k,j) := (k-j) \cdot \min_{j \leq i < k} x_i$. In particular, we are interested in sequences that maximize this value.

Consider a subsequence $[k, j)$, assume that x_i is the minimal value of the elements of this subsequence. If $V(k, j)$ is a candidate to be the largest value, this means $x_{k-1} < x_i$ and $x_j < x_i$ — otherwise we could extend the interval to get a higher V value. So now, for each i, we will calculate the longest subsequence that has its minimum at x_i.

To do this, we will do a single run through the x_i sequence and a use stack. We will go through the sequence, and for each element x_i we will first pop all elements that are larger or equal to it from the stack, and then push x_i onto the stack. This way, the elements on the stack will always be in increasing order (since we never add an element that's smaller or equal to the one before it). When we pop an element x_i from the stack, we can actually calculate the longest segment with the minimum at x_i. Since we're popping x_i, the element we're inserting (call it x_k) is necessarily smaller than x_i; and since we didn't pop it so far, x_k is the first element smaller than x_i. Similarly, the element directly preceding x_i in the stack (call it x_j) has to be the last element smaller than x_i — if there was something between them, it wouldn't have been popped. Thus, when we pop x_i from the stack, we can add $x_i \cdot (k - j - 1)$ as a candidate for the largest $V(k, j)$ value. Thus, in $O(n)$ time, we can get all candidate values for the largest $V(k, j)$.

Now to solving the full problem. For a fixed column c, and each $1 \leq r, t \leq m$, we can calculate $min_{r \leq s < t} x_{c,s}$ in $O(m^2)$ time, in a number of ways (like DP, incrementally increasing interval length). Let's do it for all columns, in $O(m^2 n)$.

Now, fix the vertical dimension of the chest d, and fix the top row z in which the chest begins. Then we are interested in the highest value of $d \cdot (k - j) \cdot \min_{z \leq s < z+d} \min_{j \leq i < k} x_{i,s}$. We have already precalculated what the lowest value of $x_{i,s}$ is for any fixed i over $z \leq s < z + d$, let's call it $m(i)$ (it depends on z and d as well, but we're treating these as fixed). So, we're interested in $d \cdot (k - j) \cdot \min_{z \leq s < z+d} m(i)$ — but this is exactly the one-dimensional problem we know how to solve in linear time! Thus, we solve it for all z and d values, and obtain an $O(mn \max(m, n))$ algorithm for the whole problem.

Problem J: Pollution Solution

Shortest judge solution: 1913 bytes. Shortest team solution (during contest): 1755 bytes.

Recall that one possible way of computing the areas of a closed polygon $p_1, \ldots, p_n = p_1$ is to sum up the signed areas of the triangles $(0, 0)$, p_i, p_{i+1} for every line segment p_i, p_{i+1}. Understanding this, one can of compute the area of pollution by simply taking all those n triangles, intersecting them with the semicircle, and again summing up their areas.

In other words, we just need to figure out how to compute the (signed) area of a triangle $(0, 0)$, p_i, p_{i+1}, intersected with a circle, with center at the origin. To do this, we find the intersections between the line segment (p_i, p_{i+1}) with the circle, and then do a little case analysis: the intersection of the triangle with the circle consists of some smaller triangles and some circle sectors (or possibly just the original triangle or one big circle sectors). My solution uses a case analysis based on whether there are 0, 1, or 2 intersection points with the semicircle – drawing some pictures make it pretty clear what is going on.

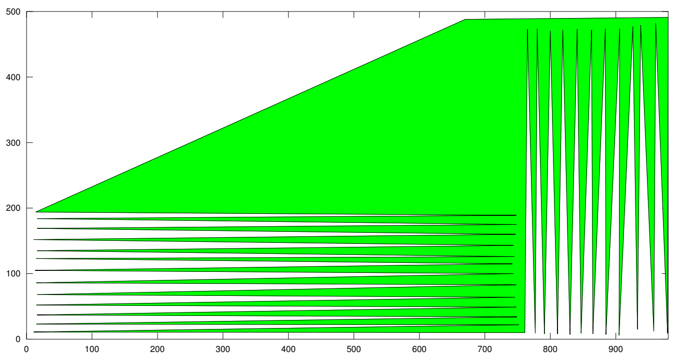
Problem K: Up a Tree

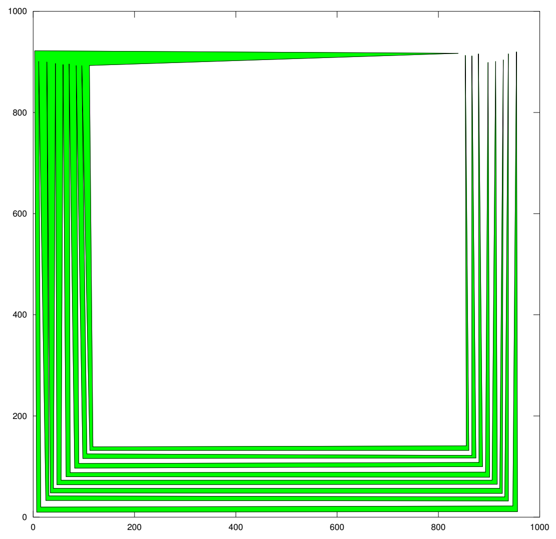
Shortest judge solution: 2165 bytes. Shortest team solution (during contest): 4180 bytes.

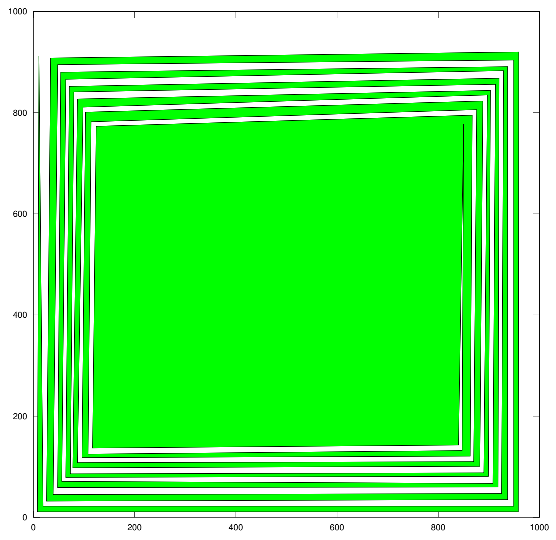
This problem is conceptually easy but surprisingly tedious to code if you are not careful. There are only $\binom{6}{2,2,2} = 90$ possible ways to put the pre/in/post calls, so we simply go through them all and check each one.

Checking if an assignment of calls is possible and finding the smallest tree can be done using dynamic programming. A state consists of three substrings of the inputs of equal length, each tagged as being the output of `prePrint`, `inPrint`, or `postPrint` (so a naive estimate for the number of states would be $3^3 n^4$ though the actual number is much smaller). To find the smallest tree that could yield these three substrings as observed outputs, we guess the size of the left subtree. Such a guess is either contradictory, or splits each of the strings into two substrings representing the two subtrees. For instance, if the string "ABCDEFGH" was printed by `prePrint` and we guess that the left subtree has 3 nodes, then the root node is "A", the observed output for the left subtree is "BCD" (and it is the observed output of the routine that we are currently trying as the first recursive call in the `prePrint` routine) and the observed output for the right subtree is "EFGH" (and it is the observed output of the routine that we are currently trying as the second recursive call in the `prePrint` routine). If in addition we had the string "BCDFAEGH" as the output of the `inPrint`, we would have a contradiction, since in that case if the left subtree had size 3 the root node would have to be "F".

The last three test cases for Map Tiles


$x_s = 100, y_s = 50.$


$x_s = 97, y_s = 93.$


$x_s = 97, y_s = 93.$