ACM ICPC World Finals 2018

Solution sketches

Disclaimer *This is an unofficial analysis of some possible ways to solve the problems of the ACM ICPC World Finals 2018. 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 problem set this year was a bit harder than in the last few years. This may be a case of the judges having found their way back to a normal difficulty, after having somewhat overdone the easy end of the problem set spectrum for a few years following the way too hard problem set we gave in 2014.

Congratulations to Moscow State University, the 2018 ICPC World Champions!
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	106	135	0	5	10	126	7	46	47	0	124
Submissions	262	247	4	15	91	462	75	278	229	1	254

The most popular language was (as usual) C++ by a wide margin: 1813 submissions, trailed by Java at 101, and only 4 submissions for the other languages (C/Kotlin/Python 2/Python 3).

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 (though some of us may have a tendency to overcompactify our code) 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: Catch the Plane

Shortest judge solution: 1480 bytes. Shortest team solution (during contest): 811 bytes.

The judges (and Per and Onufry in particular) were divided on how difficult this problem is. The contestants decided that it is one of the easier problems in the set.

To solve it, we are going to go backwards in time, starting from when we have to be at the airport. We will store and update an array that for each station S holds the probability of getting to the airport in time, if we start on S at the time we are currently considering. We initialize this array to 0 in all stations but the airport, and to 1 at the airport, which represents the probabilities at time k. Let's figure out how to update this.

We will sweep by decreasing time, storing, as events, the arrivals and departures of buses. When a bus from station A arrives at station B, the probabilities do not change (the fact of a bus

arriving does not change the probabilities). However, we have to remember the current probability of reaching the airport from B – we will use that probability to update the probability at A when the bus departs. When a bus departs from A, we have to update the probability of getting to the airport from A. If the current probability at A is p, and the probability at the arrival time at B was q, then if $p \geq q$, there's no point in getting on the bus; while if $p < q$, we can update the probability at A to $rq + (1 - r)p$, where r is the probability the AB bus leaves.

The last thing to deal with is the possibly multiple buses depart from one station at the same time. In this case, we cannot update the probability immediately after processing a bus, we have to process all the buses, see which one gives the best update, and choose that one.

Problem B: Comma Sprinkler

Shortest judge solution: 704 bytes. Shortest team solution (during contest): 926 bytes.

This was the easiest problem in this problem set, but it still was not entirely trivial. The simple approach of just applying Dr. Sprinkler's rules until no changes are made is too slow.

The observation to be made here is that the problem can be turned into a graph problem. Let's create two vertices for every distinct word in the input, one representing the beginning of that word, and one representing the end. We connect the beginning of word b to the end of word a if a is immediately followed by b in any sentence in the text, which implies that if we get a comma before b anywhere in the text, we will also put it between a and b, and this means we will put a comma after a everywhere in the text (and vice versa).

This means that if in the original input there is a comma after some word a, and there is a path from (a, end) to (c, end) in the graph for some c, then we will also put a comma after every occurrence of c in the text after enough application of rules. Thus, the problem is about finding connected components in this graph. This is doable with an application of any graph traversal algorithm.

Thus, we begin by going over the text, and constructing the graph. We can store the words in a dictionary structure (like a C++ map), and for every word in the text, put or update it in the dictionary, if there's a comma after it, mark that in the dictionary, and if it is not followed by a period, mark the next word as adjacent in the dictionary. In a single pass we get enough information to construct the graph. Now, we put the $(a$,end$)$ vertices into the "to visit" queue and run, say, breadth-first search, and finally reconstruct the text, putting a comma after every word not followed by a period for which (a, end) has been visited by the search.

This, will run, in time, $O(n \log(n))$, where n, is the size, of the input, which, is, fast, enough.

Problem C: Conquer the World

Shortest judge solution: 2510 bytes. Shortest team solution (during contest): N/A bytes.

This was one of the two very hard problems in the set. What is not hard to see is that the problem is simply asking for a minimum cost maximum flow between a set of sources and a set of sinks in a flow graph. The flow graph in question has a special structure: most importantly, it is a tree (and in addition, all edge capacities are infinite – we can move as many armies as we like across an edge – only the source and sink vertices have limited capacities). However the tree can have around 250 000 vertices/edges, so running a general min cost max flow algorithm

on this would time out rather badly. So to solve the problem we have to (unsurprisingly) exploit the structure of the graph.

Let us without loss of generality make the following two assumptions:

1. There is no demand/supply on internal vertices of the input, or at the root in case that is also a leaf (for every internal node we can always connect it with an edge of cost 0 to a newly created child leaf which has the demand/supply).

2. The tree is binary (for every internal node with degree $d \geq 3$, we can make a chain of $d-1$ internal nodes connected by edges of cost 0).

(Applying the transformations may blow up the size of the tree by a factor 4, so from an implementation point of view, one might want to unravel what the transformations mean in the context of the solution below and not actually do the transformations on the tree – the solution works without these assumptions an we have made them only to simplify the presentation.)

Root the tree arbitrarily. A natural idea is to try to solve this by some form of dynamic programming on the tree, moving upwards from the leaves until we reach the root where we get the answer. But even so, it is not at all clear what state to keep in the dynamic program, or how to go up the tree efficiently.

Let us first describe the state to use, and how to use it, without worrying too much about efficiency, and then we will later figure out how to actually make it efficient. For a rooted tree T (which you can think of for now as the subtree of the input below some vertex v), define a function $c_T : \mathbb{Z} \to \mathbb{Z}$, where $c_T(a)$ is the answer to the question "What is the minimum cost of movements within the tree T, assuming that exactly a new armies have been added to the root of T?" If we have this function computed for T the entire input tree, the answer to the problem is then simply $c_T(0)$. Note that c_T is defined also for negative a, which is interpreted as an additional demand of a armies having been introduced at the root. Let Δ_T denote the net demand within T (sum of y_i values minus sum of x_i values). Then $c_T(a)$ is a well-defined non-increasing function for $a \geq \Delta_T$.

We now use this setup to make a solution that is too slow, running in time $\Theta(nX)$ where X is the sum of all x_i values in the graph. The idea is as mentioned above to build the functions c_T for subtrees of the input starting at the leaves and moving upwards. For T being a single leaf node of the input, the function c_T is trivial: $c_T(a) = 0$ for all a (there can never be any costs from movement within a tree of a single node). In order to progress up the tree, we need to be able to do the following two operations:

1. *Extend* a tree T rooted at some vertex v upwards by adding the parent w of v from the input, and joining v and w so that w becomes the new root. Call this new tree T'. Then,

$$\Delta_{T'} = \Delta_T \qquad\qquad c_T(a) = \min(c_T(a-1), c_{T'}(a) + |a| \operatorname{cost}(v,w)).$$

   The min here is because when adding a new armies to the root we have two options: either we ignore at least one army (and get cost $c_T(a-1)$) or we transport all of them to v (and get cost $c_{T'}(a-1) + |a| \operatorname{cost}(v,w)$). For negative a one can see that the min is always achieved by the second option (which is good because in this setting we are moving some armies out of the tree, and do not have the option of ignoring any of them).

2. *Join* the the two subtrees of a vertex v. Suppose T_L and T_R are the left and right subtrees of v after doing the extend operation above (so that they are both rooted at v), and let T denote the entire subtree rooted at v. Then it is not hard to see that

$$\Delta_T = \Delta_{T_L} + \Delta_{T_R} \qquad\qquad c_T(a) = \min_{\Delta_{T_L} \leq a' \leq a - \Delta_{T_R}} c_{T_L}(a') + c_{T_R}(a - a').$$

To get the slow $\Omega(nX)$ solution, we can simply represent the functions c_T as arrays of function values and implement the equations written above.

In order to speed this up, the perhaps key observation is that the functions c_T are not only non-increasing, they are also *convex* (proof omitted, it can be derived from the identies above, or more abstractly from properties of min cost max flows). I.e., there is a diminishing returns type of property – the savings in cost attained by moving in more armies decreases as we move in more armies. For this reason, instead of working with c_T directly, it turns out to be more convenient to work with the function $f_T : \mathbb{Z}_{\geq 0} \to \mathbb{Z}$ defined by $f_T(a) = c_T(\Delta_T + a) - c_T(\Delta_T + a + 1)$ (i.e., $f_T(0)$ says how much the cost decreases when going from Δ_T to $\Delta_T + 1$ armies moving out of T, and so on). So we will keep this function, and the value $b_T := c_T(\Delta_T)$, the base cost for the tree T if we only keep the bare minimum of soldiers inside it and move everyone else out of it. The fact that c_T is non-increasing and convex then translates into the function f_T being non-negative and non-increasing.

How does this change of perspective affect the identites for c_T above? The join operation becomes particularly nice in this new perspective: it follows from the convexity property that if we take all the f_{T_L} and f_{T_R} values and merge them into a big list sorted in non-increasing order, then that resulting list are the values of f_T (abstractly, this is because the operation we are doing is taking the Minkowski sum of the two functions c_{T_L} and c_{T_R}, and the Minkowski sum of convex sets behaves nicely). This suggest the following way of representing the functions f_T: keep the values as a sorted set, and when joining f_{T_L} with f_{T_R}, add the values from the smaller of the two sets to the larger of the two (and be careful not to do anything which takes linear time in the larger of the two sets). By a standard argument, this results in a total running time of $O(n + X \log^2 X)$ for doing all the joins.

For the extend operation, the identity for c_T above now becomes

$$f_{T'}(a) = \max(0, f_T(a) - \operatorname{sgn}(\Delta_T + a) \operatorname{cost}(v, w)$$

(where we define $\operatorname{sgn}(a) = +1$ if $a \geq 0$ and -1 if $a < 0$). Computing this by updating each individual f_T-value would be too costly. Ignoring for a moment the cap at 0 and focusing only on the second argument, we see that the transformation performed on f_T is fairly simple: it gets increased by $\operatorname{cost}(v, w)$ for all $a < -\Delta_T$ (which are none, if $\Delta_T \geq 0$, i.e., if the subtree does not have more surplus than demand), and decreased by the same amount for all $a \geq -\Delta_T$. We can arrange for this by augmenting the set structure suggested above with a split point and two global shifts – everything below the split point has been shifted by the first shift, and everything above the split point has been shifted by the second split. When we do the extend operation, there may already be an existing split point and shifts, we then have to move the split point, which can be done in time which is $O(d \log d)$ where d is the distance between the two split points. This value might be as large as X, but in total over all extend operations done in the tree, the sum of these distances can never be more than $2X$, so the total time we get from these moves of the split points is $O(n + X \log X)$. Coming back to the ignored $\max(0, ...)$ part from above, the easiest way to deal with that is to simply remove any negative values from the end of the set after having performed the operation above.

We also need to revisit the join operation and see that this can still be done with the augmented data structure with shifts. Implementationwise, instead of keeping a single ordered set with a split point, it is probably easier (at least all our implementations seemed to feel so) to keep two separate sets corresponding to the part of values below and the part of values above the split point. Another implementation detail is that using complete ordered sets is not necessary, one can get away with two simple max-heaps.

Problem D: Gem Island

Shortest judge solution: 1278 bytes. Shortest team solution (during contest): 1050 bytes.

Let us number the islanders $1, 2, \ldots, n$ in some fixed order. Let $(g_1, g_2, \ldots, g_n)$ be any integers summing up to $d + n$, with $g_i \geq 1$ – a potential distribution of gems after d nights. It turns out the probability the gems are distributed that way after d nights is the same for any distribution.

Let us prove this by induction on d. For $d = 0$, there is only one distribution $(1, 1, \ldots, 1)$. For a higher d, there are n potential ways to get to $(g_1, g_2, \ldots, g_n)$ – from any distribution where exactly one islander has exactly one gem less. The probability that we get from $(g_1, g_2, \ldots, g_k - 1, \ldots, g_n)$ to $(g_1, g_2, \ldots, g_n)$ is $(g_k - 1)/(n + d - 1)$ – one of the $g_k - 1$ has to be the one to split. Since the probability for each distribution for $d - 1$ is some fixed p by the inductive hypothesis, the total probability of getting to $(g_1, g_2, \ldots, g_n)$ is

$$\sum_k p \frac{g_k - 1}{n + d - 1} = \frac{p}{n + d - 1} \sum_k (g_k - 1).$$

And the sum of all $g_k - 1$ is d – the number of gems over the first one that all the islanders have (technically, we also need to exclude the distributions where $g_k = 1$ from the sum, since $(g_1, g_2, \ldots, 0, \ldots, g_n)$ is not a valid distribution, but the probability of that distribution is multiplied by $g_k - 1 = 0$ in the sum anyway, so it does not matter).

With that observation, we are prepared to calculate the expected value we are being asked for. Since all the distributions are equally probable, we can instead calculate the sum of the number of gems the r richest islanders have, over all possible distributions, and then divide by the number of all possible distributions. The number of ways to assign d gems to n people is $\binom{n+d-1}{d}$, and we can precalculate all the binomial coefficients in quadratic time and memory, so let's just look at the sum.

Let us denote the sum over all legal distributions of $n + d$ gems to n people of the number of gems the r richest people have by $S(n, d)$. We can write a recursion for $S(n, d)$. If $n \leq r$, $S(n, d) = n + d$.

For larger n, there is a contribution coming from the first gems of every islander – that is obviously $r\binom{n+d-1}{d}$. On top of that, we have some islanders that have only one gem – their contribution is finished – and some that have more gems. These islanders that have more gems, after removing their first gem, still have a valid distribution of the remaining d gems. So, if we fix g – the number of islanders that have exactly one gem – the remaining contribution is $\binom{n}{g} S(n - g, d - n + g)$. This gives us a recursion formula for $S(n, d)$:

$$S(n, d) = \binom{n + d - 1}{d} r + \sum_{g=0}^{n} \binom{n}{g} S(n - g, d - n + g).$$

This recursion formula translates to a dynamic program for calculating $S(n, d)$ in time $O(n^2 d)$, which is fast enough. Even less efficient solutions (with an extra log factor in there) had a chance to pass if efficiently implemented.

This problem can be solved even faster. For instance, a quadratic-time solution, communicated to us by Petr Mitrichev, goes roughly as follows. Let $V(k, y)$ be the number of distributions for which exactly k islanders have at least y gems. We have

$$S(n, d) = \sum_{k,y \geq 1} \min(k, r) V(k, y).$$

If we denote by $W(k, y)$ the number of distributions where at least k islanders have at least y gems, then $V(k, y) = W(k, y) - W(k + 1, y)$. And $W(k, y)$ can be calculated from the inclusion-exclusion principle as

$$W(k, y) = \sum_{\ell \geq k} (-1)^{\ell - k} \binom{\ell - 1}{k - 1} \binom{n}{\ell} \binom{n + d - \ell(y - 1) - 1}{d - \ell(y - 1)}.$$

The proofs of these equalities, as well as of the fact that the resulting algorithm is quadratic, are left as an exercise to the reader.

Problem E: Getting a Jump on Crime

Shortest judge solution: 1945 bytes. Shortest team solution (during contest): 2738 bytes.

This was one of the last problems solved, and the judges were a bit surprised by how few teams solved it. While the problem requires a bit of work, the hard part of that work is pen and paper calculations without the computer, which is a nice type of work for a three person team with one computer.

The problem has two parts – figuring out which jumps are possible, and then computing the shortest jump sequence from the starting position. The second part is easily solved with a breadth first search so let us focus on the first part. Given a horizontal jump distance d, and a height difference Δh between the takeoff and landing positions, we need to figure out how to split the total speed v into horizontal speed v_d and vertical speed v_h such that when making a jump with these initial horizontal and vertical speeds, we end up at a height difference of exactly Δh after having travelled d meters horizontally. From the equations in the problem statement we see that the time t it takes us to travel d meters horizontally is d/v_d, and that after this time our vertical position (relative to where we started) will be $v_h t - \frac{gt^2}{2}$ (follows by integrating the identity for the velocity from 0 to t). Plugging in $t = d/v_d$ and $v_h = \sqrt{v^2 - v_d^2}$, we thus need to solve the equation

$$\Delta h = \frac{d\sqrt{v^2 - v_d^2}}{v_d} - \frac{g^2 d^2}{2v_d^2}$$

for v_d. This may look like pretty nasty, but it can easily be transformed into a quadratic equation in v_d^2: move the $\frac{g^2 d^2}{2v_d^2}$ term to the left side, multiply both sides by v_d^2, and then square both sides (this may potentially introduce spurious solutions since it throws away sign differences). The expression for the solution gets a bit messy (a tip to make things a bit cleaner is to parameterize $v_d^2 = xv^2$ and $v_h^2 = (1 - x)v^2$ for an unknown $0 \leq x \leq 1$ and then solve for x instead) but can be found using only pen, paper, and patience; we do not want to spoil the joy of doing the math so leave the details of this to you. Solving this gives (up to) two solutions, one or both of which may be spurious due to the squaring trick above. We observe that if there are two real solutions, it is always better to take the one with higher value of v_h, because that corresponds to making a higher jump. It also turns out that the solution with higher value of v_h is never spurious so we can in fact take it without even checking if it is spurious.

With the calculus out of the way: suppose we now want to check whether we can make a jump from the building at position (x_1, y_1), with height h_1 to the one at position (x_2, y_2), with height h_2. This corresponds to making a jump with height difference $\Delta h = h_2 - h_1$ and horizontal length $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, so we use what we figured out above to determine

the jump parameters v_d and v_h (or we figure out that such a jump is impossible). Now that we have the jump parameters we know the exact parabola of the jump trajectory, we also need to check all buildings that lie between (x_1, y_1) and (x_2, y_2) to make sure that the jump trajectory goes above the building. To do this, it is sufficient to check the first and last times where we are over the building in question (because the parabola is concave). Finding the buildings to check can be done in $O(d_x + d_y)$ time but it was OK (at least if the checks were done reasonably efficiently) to do it in $O(d_x \cdot d_y)$ time by simply iterating over all buildings and checking which ones the trajectory passes by. Overall, this leads to an $O(d_x^2 d_y^2 (d_x + d_y))$ or $O(d_x^3 d_y^3)$ running time for building the graph (because we have to do this for all $O(d_x^2 d_y^2)$ pairs of buildings).

Problem F: Go with the Flow

Shortest judge solution: 872 bytes. Shortest team solution (during contest): 869 bytes.

This was one of the easiest problems in the set. The most straightforward approach is to try all line lengths from the shortest possible and upwards, until there is only one line. To try a given line length, we essentially just simulate – place word by word, and keep track of where in the previous line there was a river and how long it was.

A priori, the running time of this sounds dangerous – there are potentially $\Theta(nL)$ line lengths to try (where $L = 80$ is the max word length) and trying a line length naively takes $\Theta(nL)$ time for a total of $\Theta(n^2L^2)$ time which sounds pretty dangerous for $n = 2500$, $L = 80$. As far as we know it was not possible to get this to pass, but there are several easy optimizations that can be made, and doing any one of them was sufficient to pass:

1. Instead of going all the way up until we just have one line, we can stop when the longest river found is at least as long as the current number of lines.
2. Instead of checking a line length in $\Omega(nL)$ time we can do it in $O(n)$ time.
3. Instead of checking every possible line length, we can check "what's the next line length that will cause the words to be wrapped differently?" and jump directly to that one.

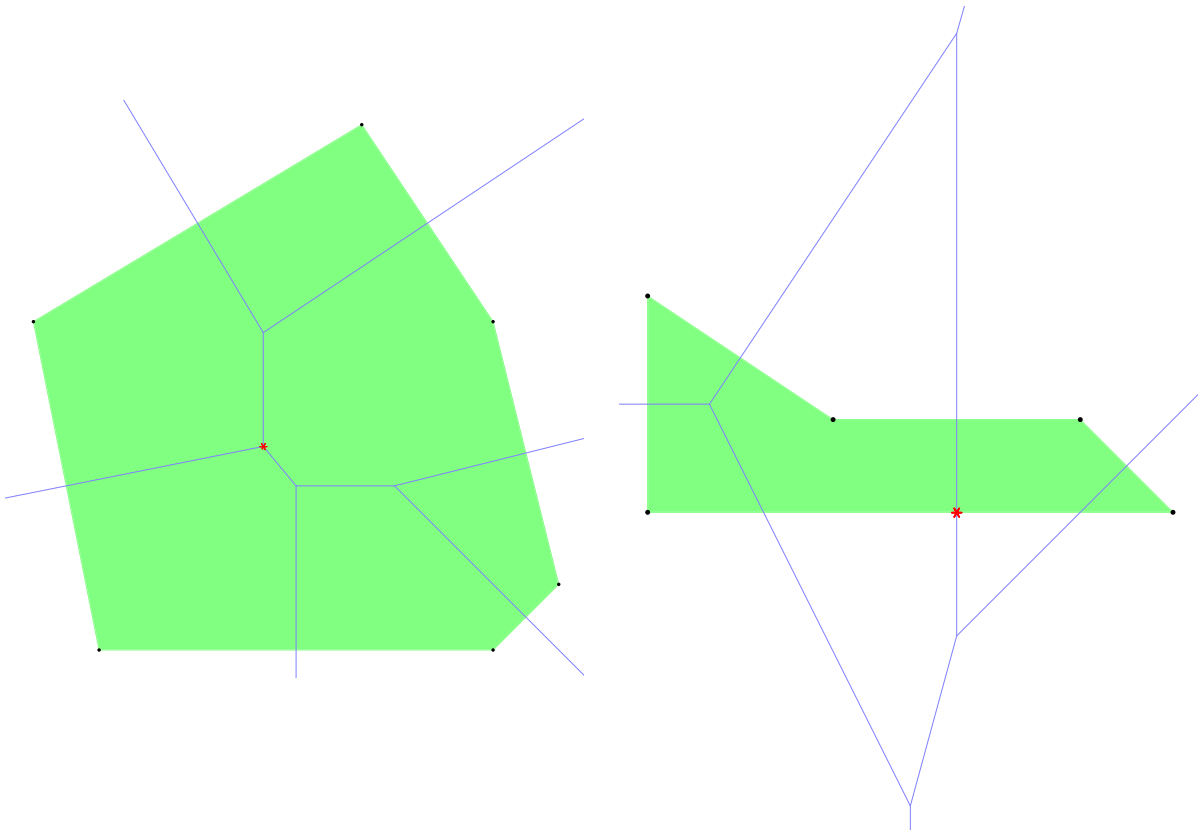
Optimizations 1 and 3 are very hard to analyze exactly how well they perform, and in general it is very hard to craft test data for this problem. To do well against optimization 1, it is useful to have cases with 2500 words where the longest river is as short as possible. The best such case we have (which was found only during the night before the contest!) has a longest river of 3 (but its average word length is not very high, which would be needed to really make optimization 1 work poorly). However, we don't even know whether it is possible to make such a case with maximum river length 1 or not (we suspect that this is impossible with word lengths bounded by 80, and if anyone finds a proof we would be very interested in seeing it)!

Problem G: Panda Preserve

Shortest judge solution: 4669 bytes. Shortest team solution (during contest): 4372 bytes.

This was the only computational geometry problem in the set. We are given a polygon, and asked to find the point inside (or on the boundary of) the polygon that is furthest away from any vertex of the polygon (or rather, we are only asked to find this furthest distance, but we

don't know of any way of finding that without also finding a furthest point). The key conceptual observation to make is the following: the furthest point will lie either (i) on a vertex of the Voronoi diagram of the set of polygon vertices, or (ii) on the intersection of a Voronoi edge with a polygon edge. Here are two examples (the second one being the first sample input, and the first one being one of the secret test cases) illustrating these two possibilities (the blue lines show the Voronoi diagram and the little red asterisk marks the furthest point):



With this easy theory sorted out, it is time to buckle up for the less easy implementation. There are two parts: first we need to compute the Voronoi diagram, and then we have to find the furthest point.

For the first part, there are well-known $O(n \log n)$ algorithms for Voronoi diagram, and if you have one of those in your team notebook then you can just go with that, but these algorithms are quite difficult to implement correctly and if you do not happen to have such an implementation available, the input bounds ($n \leq 2000$) do allow for something slower and easier. For instance, one can take e.g. Fortune's $O(n \log n)$ algorithm and simplify some steps while keeping it $O(n^2)$, or one can take a more direct approach and compute the Voronoi diagram in $O(n^2 \log n)$ time (but in this case one has to be a bit careful with the constant factors in the implementation) by taking each polygon vertex and computing the cell around it in $O(n \log n)$ time using a relatively simple circular sweep algorithm similar to the Graham scan algorithm for convex hulls. We make sure to store in our Voronoi diagram which polygon vertices define each Voronoi vertex/edge, so that we can evaluate the distance for each candidate point in $O(1)$ time.

The second part is easier (again owing to the fact that the polygon is relatively small), assuming one has access to basic geometric primitives such as point in polygon and line segment intersection (and if one does not, one should probably have stayed away from this problem in the first place). We simply go over every vertex of the Voronoi diagram, and check if it is in-

side the polygon (in which case it is a candidate furthest point). Similarly for every edge of the Voronoi diagram we compute all intersections with the polygon edges (and these are also candidate furthest points). This then takes $O(n^2)$ time, because there are $O(n)$ Voronoi vertices and edges, and both point in polygon tests and "compute intersection with every polygon edge" also take $O(n)$ time.

Fun fact: this problem was originally proposed with much larger polygons, requiring $O(n \log n)$ time, but we felt that this was just too hard, even for the World Finals. In this version, computing the Voronoi diagram in $O(n \log n)$ time is actually *not* the hardest part – the second part of finding the furthest point in the Voronoi diagram is even harder (it can be done using a complicated sweepline algorithm, but one has to be very careful, because the number of intersections between polygon edges and Voronoi edges can be quadratically large, so even enumerating all of them would be too costly).

Another approach than the one described above is to do a binary search on the answer and then try to determine efficiently whether a given circle radius covers the polygon or not. That task can be done in $O(n^2 \log n)$ with a sweepline approach. We also saw at least one team solving the problem with a direct sweepline approach without binary search or explicitly computing the Voronoi diagram, but our impression was that this solution was in some sense implicitly computing the Voronoi diagram and incorporated the second part into it.

Problem H: Single Cut of Failure

Shortest judge solution: 1484 bytes. Shortest team solution (during contest): 1519 bytes.

At first glance this looks like a very hard geometry problem, but a moment's thought reveals that the problem is more combinatorial than geometric in nature. Linearizing the coordinates, we see that what we get are a bunch of intervals (on a circle) and we need to pick the smallest number of new intervals such that each input interval crosses at least one of the constructed intervals. This still looks pretty hard though, and the key observation is that the geometry of the problem actually should not be ignored completely: the fact that all wires connect two different sides of the door means that it is always possible to cut all wires using two diagonal cuts.

This means that the problem now reduces to figuring out if we can cut all wires using a single cut or not (hence the problem name). If not, using the diagonals is an optimal cut. Checking if a single cut suffices can be done in linear time after sorting all the wire end points. We keep two pointers s and t (indicating that the cut we make is from s to t), initially pointing to the same position. Then, we repeat the following, until the s pointer has gone a whole lap around the date:

1. while we can advance t without making any wire contained within the interval (s, t), do so.
2. advance the s pointer.

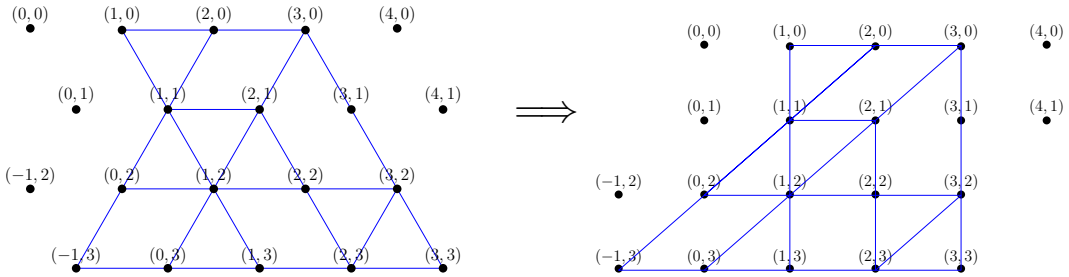
(Here, "advance" means moving the pointer past the next wire end point.) During this process we keep track for each wire whether it is completely outside, partially inside, or completely inside the interval (s, t) and keep a count of how many intervals are partially inside. If at any point during the process all n wires are partially inside, we have found our single cut.

One small caveat is to make sure that the cut we make goes between two different sides of the door.

Problem I: Triangles

Shortest judge solution: 2136 bytes. Shortest team solution (during contest): 1824 bytes.

Let us focus on counting $\triangle$-shaped triangles – a solution for that easily generalizes to $\triangledown$-shaped triangles (e.g., we can simply turn the entire picture upside down and then count $\triangle$-shaped triangles again). There are many reasonable coordinate systems one can pick for the grid, but for the purposes of this discussion, let us apply the one shown on the left below, where points (x, y) with a fixed x-coordinate correspond to a negative-slope diagonal. Shifting the rows horizontally to make these diagonals vertical, we get the picture shown on the right, and are now trying to count $\triangle$-shaped triangles there.



Now we process the grid row by row, then column by column. For each point (x, y), we compute the following quantities:

1. $L(x, y) =$ how many steps does the horizontal line extend to the left from (x, y). E.g. in the figure above, $L(2, 3) = 3$ (because the line extends to $(-1, 3)$.
2. $U(x, y) =$ how many steps does the vertical line extend upwards from (x, y). E.g. in the figure above, $U(2, 3) = 2$ (because the line extends to $(2, 1)$.
3. $D(x, y) =$ how many steps does the diagonal line extend upwards from (x, y). E.g. in the figure above, $D(2, 3) = 1$ (because the line extends to $(3, 2)$.

Computing these in constant time for each new grid square is easy, e.g. if the horizontal edge between $(x - 1, y)$ and (x, y) exists then $L(x, y) = 1 + L(x - 1, y)$ (and we have already computed $L(x - 1, y)$) otherwise it is 0.

To compute the number of triangles with lower right corner at (x, y), we would now like to answer the following type of queries: let $s = \min(L(x, y), U(x, y))$. For how many values of i between 1 and s (inclusive) does it hold that $D(x - i, y) \geq i$? Each such i means that we have the triangle between $(x - i, y)$, (x, y), and $(x, y + i)$, and you should convince yourself that these are indeed the only triangles with lower right corner at (x, y).

A naive way of answering these queries would be to simply loop over all i from 1 to s, but this results in $\Omega(n^3)$ time and should time out (where for notational simplicity we here use $n = r + c$ to denote the side lengths – technically the bound is a bit inaccurate and it is actually $\Omega(r \cdot c \cdot \min(r, c))$ time but the worst case is when the grid is quadratic and we will just give bounds in terms of n). However, fixing this is a not too difficult dynamic data structure exercise involving some form of range queries. As a concrete suggestion, suppose that when processing a row y we keep an additional array $A(x)$, where, when we are at point (x, y) the value of $A(x')$ for $x' < x$ is 1 if $D(x', y) \geq x$ and 0 otherwise. In other words, when we process (x, y) we set $A(x) = 1$, and after we have processed $(x + D(x, y), y)$ we set $A(x)$ to 0 (which we do by keeping a list for each x coordinate of which A-values should be zeroed out after we have processed this x coordinate). Now the number of i's between 1 and s such that

$D(x-i,y) \geq i$ equals the sum of $A(x')$ from $x' = x-s$ to $x' = x-1$. Doing updates to the A array while being able to answer partial sum queries like that is exactly what a Fenwick tree does, in $O(\log n)$ time per update and per query. This results in an $O(n \log n)$ time per row or $O(n^2 \log n)$ time in total for the entire grid.

This problem had rather tight time limits, so one did need to be careful about constant factors in the implementation, and to make sure not to waste too much time on reading the input. The reason for this was not that we really wanted teams to make the best implementation, but simply that a reasonably optimized $\Omega(n^3)$ solution was rather fast for the grid size used (which already results in rather large input files, so we were hesitant to increase it even further), and we did want the $\Omega(n^3)$ solution to time out (the n^3 solutions we had were about 80% slower than the time limit and we wanted a decent margin to account for teams being better at low-level optimizations than us). Ultimately, we did succeed in preventing n^3 time solutions, but we also timed out several $n^2 \log n$ solutions forcing teams to optimize their constants a bit too much.

Problem J: Uncrossed Knight's Tour

Shortest judge solution: "1105" bytes. Shortest team solution (during contest): N/A bytes.

This was the hardest problem in this set, and it was actually an open problem in mathematics; with people not knowing the optimal tour lengths for some of the chessboard sizes for which we posed this problem.

The first observation is that if the limits for n were smaller (like, 50), we could attack this with a DP. The reason is that in order to extend a partial tour that is constructed up to some row, we don't need the full information on what happened previous to that row. The full state required to extend the tour, for a given row, is the following state:

- a representation of squares in this row: we either visited this square, both entering from and leaving to a square above this row, or visited entering from a square above this row, but not leaving above this row, or not yet visited this square
- a representation of moves from the row above to the row below (jumps of vertical length 2): we can have one of the two possible jumps (vertical and to the left, or vertical and to the right), or no jump at all, between any two cells.
- the connectivity information: we have to remember how the squares entered from above without leaving and the vertical jumps crossing our row are connected into pairs by the parts of the tour above our row.

In theory, this representation means that we hold one of three possible values for each of the 8 squares in a row, and each of the three possible values for each of the 7 spaces between squares, and additionally we have to connect up to 14 elements into pairs (which can be done in 13!! = 135135 different ways), amounting to over 10^12 states. However, in practice, most of these states are unreachable, and a program starting from the empty row will reach several hundred thousand states.

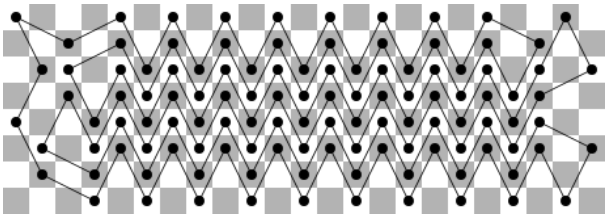
Representing the connectivity information in this state is a bit tricky, because we want to canonicalize it (that is, we want to make sure that we have only one valid representation for a given connectivity).

For any such state, we can then calculate what are the possible states in the next row, and how many visited squares do we add in such a transition. This can be done by a recursive

algorithm, where for each line between squares we just add a visited square in the next row, for each visited square with one incoming edge we branch out into the four options for the other edge from this square, and for each empty square we branch out into seven options – either it is really empty, or we visit it with both edges going downwards. We need to do this carefully to make sure the edges we add do not intersect, and that we maintain the connectivity information correctly.

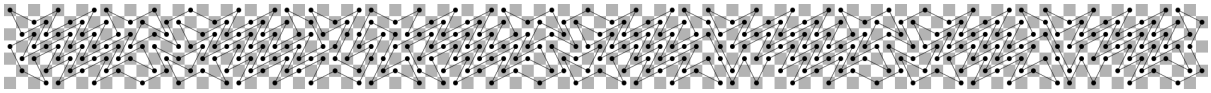
There are many technical details in the implementation of this algorithm (for instance, one has to distinguish the "empty row" state of "we have not started yet, there were no visited squares" from "we have already finished the tour", in order not to allow multiple disconnected tours as a solution). With an efficient implementation, an 8 by 50 chessboard can be solved in several seconds, and an unoptimized program will perhaps run for several minutes.

Now, the crucial observation is that the solutions actually start getting repetitive after a while. For instance, for board size 8 by n, after a while you get a pattern of 6 knights in a row all jumping left, and then all jumping right, adding 12 visited squares with each two added rows, as in the picture below.



This suggests that we should try to run the DP described above for a reasonable range of n, and look for this cyclic behaviour – that adding some k rows to n increases the answer by some a. And indeed – after running up to, say, $n \leq 200$, we will discover cycles for all possible values of m. Thus we can precompute these values locally and then submit a simple solution which has them hard-coded.

We found it interesting that for $m = 7$ one finds a somewhat surprising cycle length of 33 – there is a pattern of size 7×33 that can be repeated over and over in the solution. See, for instance, this diagram:



Problem K: Wireless is the New Fiber

Shortest judge solution: 709 bytes. Shortest team solution (during contest): 851 bytes.

This is one of the easier problems in this problem set. After reading through the statement, we can see the problem is about constructing a tree where as many vertices as possibe have a given degree (note that the only information we care about from the input are the degrees of vertices in the input graph, and not the exact shape of the input graph).

The choice of which vertices preserve their degree can be made greedily. Since we have to construct a tree, the sum of degrees of all the vertices will be $2n - 2$, and each vertex will have degree at least 1. Thus, we are left with $n - 2$ spare degree increases to assign. In order to satisfy as many vertices as possible, we should assign these increments to the vertices with the

smallest expected degree first. In this way we arrive with a degree assignment that satisfies as many degrees as possible.

The second part of the problem is to construct a tree with given degree values. This can be done in a number of ways. One such way is to order the vertices by degree, and add them to the tree in order of decreasing degree, starting with the highest degree vertex as the root, and maintaining a list of "outstanding degrees" – that is vertices with a larger expected degree than the number of edges already connected. When adding a new vertex to the tree, we connect it to any of the vertices with positive outstanding degree (and decrease the outstanding degree of both vertices by one). Note that until the very end, there will always be a vertex with positive outstanding degree in the already constructed tree. This is because a tree with k vertices has outstanding degree zero if the sum of expected degrees in the tree is $2k-2$, and so the average expected degree is $(2k-2)/k$. However, the average expected degree among all vertices is $(2n-2)/n$, which is larger than $(2k-2)/k$ since $k < n$, and the average expected degree in any partially constructed tree is at least equal to the average expected degree among all vertices, because we take the vertices from the largest expected degree.