Helvetic Coding Contest 2016
Solution Sketches

July 10, 2016

A Collective Mindsets

Author: **Christian Kauth**

A1 Easy

Strategy: In a bottom-up approach, we can determine how many brains a zombie of a given rank N needs to offer to each sub-ranked fellow, in order to obtain his consent on the offer. This number is shown in the table below. The cheapest way to collect $N/2$ votes is to distribute the indicated number of brains to the $N/2$ zombies (including the highest-ranked zombie) who are content with the smallest number of brains. Those numbers are marked in bold, they sum up in the right-hand side column and all other zombies get no brains at all.

	to #1	to #2	to #3	to #4	to #5	necessary brains
Offer of #1	**1**					1
Offer of #2	2	**1**				1
Offer of #3	**1**	2	**1**			2
Offer of #4	2	**1**	2	**1**		2
Offer of #5	**1**	2	**1**	2	**1**	3

Complexity: Form the above strategy you gain the insight that the answer is $\lfloor (N+1)/2 \rfloor$, which you must compute in constant time $\mathcal{O}(1)$ to solve this subtask.

Illustration: If Heidi has rank 5 ($N = 5$), she wants 1 brain for herself and needs two more to convince zombies 1 and 3. These will accept a single brain each, since if Heidi died, zombie 4 would make the next offer, which would be accepted if he offered a single brain to zombie 2 and kept the rest for himself, thus leaving zombies 1 and 3 empty-handed.

A2 Medium

Strategy: Let's see who would survive if he got to make an offer, but there were no brains at all. Zombie 1 would survive and so would zombie 2. Zombie 3 on the other hand would die because he has no brains to offer to any fellow to make him accept the offer. Hence zombie 3 knows that he will die if he gets to make an offer, and so he will do everything he can to avoid that he ever needs to make an offer: By unconditionally accepting any offer made by zombie 4, he can save his own life and the life of zombie 4! Zombies 5, 6 and 7 are doomed to die, which brings zombie 8 into a great position, since he will get the votes of zombies 5, 6 and 7, who can save their own lives by keeping zombie 8 alive. Let's recap: zombies 1, 2, 4, 8, ... and any power of 2 will survive if the chest is empty!

Who would survive if there was 1 brain? By using a bottom-up strategy, as for the easy subtask, one can show that zombies 1 to 4 can survive. Zombie 5 has however not enough brains to collect two more votes and dies, unless he unconditionally accepts the offer of zombie 6. This in turn keeps zombie 6 alive! 7, 8, 9 die if they do not unconditionally accept an offer made by 10 and thus keep 10 alive. The pattern is the same as in the previous case now.

Complexity: We now have a correct strategy, but we need to find a pattern in the ranks of the survivors in order to compute the solution fast enough (e.g. in constant time $\mathcal{O}(1)$).

- For 0 brains, the survivors are: $1, 2, 4, 8, ...$
- For 1 brain, the survivors are: $1, 2, 3, 4, 6, 10 = \{1, 2\} \cup (2 + \{1, 2, 4, 8, ...\})$
- For 2 brains, the survivors are: $1, 2, 3, 4, 5, 6, 8, 12 = \{1, 2, 3, 4\} \cup (4 + \{1, 2, 4, 8, ...\})$
- For B brains, the survivors are: $\{1, 2, .., 2 \times B\} \cup (2 \times B + \{1, 2, 4, 8, ...\})$

Notice that the right-hand side component of the union contains only even numbers! Hence if N is odd, it takes $\lfloor (N-1)/2 \rfloor$ brains for N to appear in the left-hand side component of the union!

Also note that the right-hand side member of the union holds all numbers which are twice the number of brains plus any power of two. Or stated the other way round, for zombie N, even, to survive, the number of brains needed is N minus any power of 2, divided by two. Since we ask for the smallest such number, you want to subtract the largest power of two smaller than N.

Illustration: If Heidi has rank 99 ($N = 99$), then there would need to be 49 brains in the chest for her to survive!

But if Heidi has rank 100 ($N = 100$), we may subtract 64, the smallest power of two, smaller than 100, to reach 36, and then divide by 2, leading to 18. If there are 18 brains in the chest and Heidi has rank 100, she will survive!

A3 Hard

Strategy:

- By observing the numbers on the $N-1$ fellows' foreheads, every zombie can determine the sum of all the zombies' numbers minus his own number.
- He knows his own number is between 1 and N.
- For every number between 1 and N that he would guess, the implied total sum would take N different values and exactly one of them is guaranteed to be correct! Every sum modulo N would be distinct on the one hand, and take all possible values between 0 and $N-1$ on the other hand!
- A winning strategy is for every zombie is to guess a number such that the implied sum modulo N is distinct from the ones implied by his fellows. Since the sum modulo N can take N values and there are N zombies, we guarantee that exactly one zombie makes a correct guess!
- The value of the modulo of the implied sum each zombie shall target may for instance be his rank.

Complexity: Summing the $N-1$ numbers on the fellows' foreheads, subtracting the guessing zombie's ID, and returning this number modulo N, plus 1, is done in linear time, implying an overall $\mathcal{O}(N \times T)$ complexity for the T tests.

Illustration: If there were $N = 10$ zombies, every zombie would guess his number such that the unit digit of the implied sum of all numbers equals his rank minus one.

B Recover Polygon

Author: **Andrii Maksai**

B1 Easy

Statement requires us to find a rectangle of non-zero area such that the Zombie Contamination (ZC) level we observed was produced by it, or say that this is impossible.

First, we should observe that if the levels of ZC are indeed produced by a rectangular lair, we will observe a picture similar to the one in Fig. 1, left:

- Four cells around the corners of the polygon will have a ZC level of 1.
- All other cells that touch the border of the polygon on the outside will have a ZC level of 2.
- All cells inside the polygon will have a ZC level of 4, and all other cells will have a ZC level of 0.

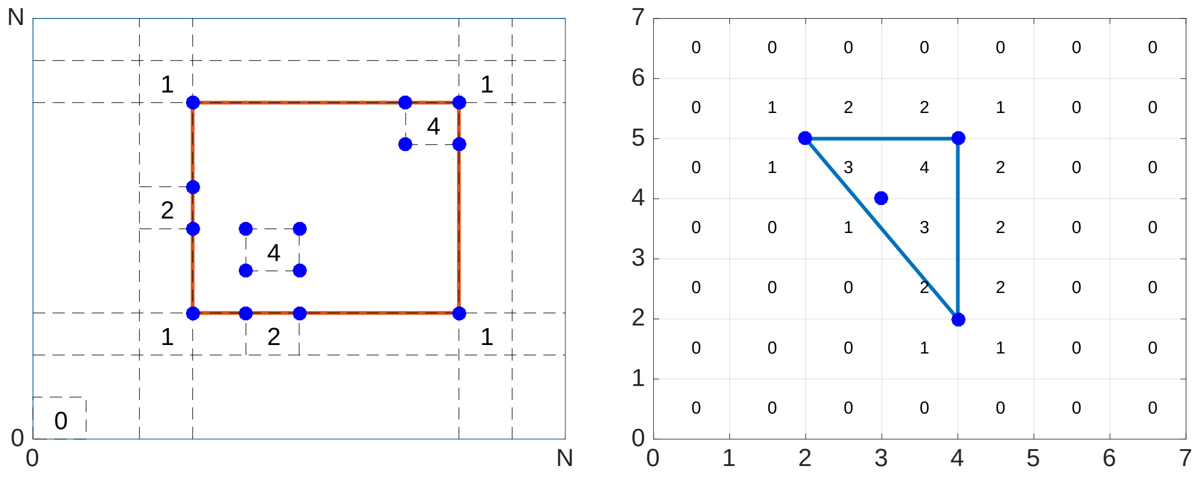

Figure 1: **Left:** Easy problem. Big blue dots mark points inside or on the border of the polygon. Examples of ZC level in some cells are provided. **Right:** Medium problem. Big blue dots mark candidates for the corners of the polygon, computed based on the corners of the cells that contain '1'.

This gives two possible ways of solving this problem. First is to locate all '1' on the grid, confirm that there are 4 of them and they lie in the corners of the non-zero area rectangle. After that one needs to verify that all other cells on the border are indeed '2', inside all cells contain '4', and all the other cells contain '0'. This yields an $O(N^2)$ solution.

Another approach was to backtrack through all possible rectangles on the grid, by looping through all possible coordinates of upper right and bottom left corner. For each such rectangle, a similar check can be done. This yields an $O(N^6)$ solution.

B2 Medium

Key observation for this problem was the following: one of the cells that touches the corner of the polygon always contains a '1'. This is ensured by the fact that polygon is strictly convex, therefore an angle is less than 180 degrees, and there should exist a cell that touches the corner yet is completely on the outside of the rectangle.

We can find out which corner of the cell containing ‘1‘ is indeed on the border or inside of the polygon. To do that, we need for each corner of the cell look at the *other* cells that touch that corner. If at least one of the other cells contains a ‘0‘, then the corner in question is not on the border on inside of the polygon. Otherwise, it is. Example can be seen in Fig. 1, right.

By doing such checking, we can compile a list of candidate points. This list will contain all of the corners and possibly some other points on the border or inside of the polygon. All that remains is to compute the convex hull of these points to remove all points that are not really the corners of the polygon.

There are at most $O(N)$ cells that contain ‘1‘, and the convex hull can be built in $O(N \log N)$ or $O(N^2)$. The complexity is therefore dominated by reading the input, $O(N^2)$.

B3 Hard

One can generalize here the strategy from the previous problem. Now we compose the candidate list by going through all of the cells and looking at other points that touch the corners of the cell. We again need to check that all of the other neighbors of the corner contain not a ‘0‘ and if it is so, we add the corner in question to the candidate list.

To check that a cell contains not a ‘0‘, we need to first look at our input, as it contains all the cells with ‘1‘, ‘2‘, and ‘3‘. All that remains is to check if a given cell contains a ‘4‘. This can be done in several ways.

One is to check whether a center of the point lies inside of a convex hull of centers of all input points. This way, we are checking whether the cell is ‘inside‘ or ‘outside‘ of the polygon we are trying to recover. Querying whether a point is inside of a polygon or not can be done in $O(\log N)$ by first doing an $O(N \log N)$ preprocessing.

Another heuristic approach is as follows. First we compute the center of mass of all input cells. This center of mass lies inside of the polygon that we try to recover. We then check whether the ray from the center of mass to our given cell intersects any of the input cells after it goes through the cell in question. If so, we can again conclude that our cell in question lies ‘inside‘ of the polygon and the value in it is ‘4‘.

Several other heuristics are also possible. Complexity of the solution is $O(N \log N)$, as the number of input points M scales linearly with the growth of the size of the grid N.

C Brain Network

Author: **Damian Straszak**

C1 Easy

It is not hard to see that that in this problem we actually have to deal with undirected graphs. Indeed brains correspond to vertices and brain connectors are edges. The problem we need to solve is equivalent to the following: given a graph G, find out whether it is connected and it has no cycles. Such graphs are called *trees* and one characterize them differently by saying: an n-vertex graph is a tree if and only if it is connected and the number of edges is equal to $n - 1$. Both of these conditions are very simple to verify, to check connectivity we just need to run any search algorithm (like DFS or BFS) or Union-Find. The time complexity is linear $O(n + m)$.

C2 Medium

In this problem we are given a tree on the input. The task is to compute the diameter of the tree, i.e. the distance between the pair of furthest vertices. One solution could be to run a shortest path algorithm, like Floyd-Warshall or Dijkstra, compute all pairwise distances and output the maximum. However, the issue is that with this approach we cannot obviously go below $O(n^2)$ running time, which is too slow, since $n = 100000$. It turns out that since the graph is a tree we can compute the diameter much faster.

Suppose we are given a vertex v, then we can easily compute all distances from v to all the remaining vertices in linear time! Make v the root of the tree and compute recursively the heights of all the vertices (i.e. the children of the root have height 1, their children height 2, etc.). Clearly the heights are just the distances.

Now we just run a magic algorithm which solves the problem! Your task will be to prove that it works. Take any vertex v, compute distances to the remaining ones, let u be a vertex which is furthest away from v. Now, take u and do the same: find the vertex w which is furthest away from u. And now... output the distance from u to w! Note however that such an algorithm works for trees only.

This problem can also be solved in a much more standard way, using dynamic programming. We select a root for the tree arbitrarily. Then we have a rooted tree and, for every vertex, we can consider the subtree rooted at this vertex. Using dynamic programming we can compute the depth of each such subtree. Now, the diameter of the tree is a path between two leaves. Consider the highest point in the tree crossed by this path; the path is a union of two paths from this point to a leaf. So, for each possible vertex (to serve as the highest point), look at all pairs of its children and their depths. (Some care is required to make this run in linear time.)

C3 Hard

This problem is a dynamic version of the tree diameter problem. There is a tree, initially empty, vertices are added one by one (and immediately connected to a previously added one), and after every such operation we are required to output the diameter of the current graph. Of course, we cannot use algorithm from the previous problem, because it takes $O(n)$ time for a tree of size n, hence $\Omega(n^2)$ in total!

Instead of just computing the diameters at every stage we will keep a pair of vertices (u, v) which actually achieve the largest distance. Now comes the crucial observation: if (u, v) is the pair of furthest vertices in a tree T, and another vertex w is added to it, then in the new tree T' to find the furthest pair of vertices we only need to consider $(u, v), (u, w), (v, w)$. This is really helpful. Instead of recomputing everything from scratch, we just need to compute 3 distances in the tree $d(u, v)$, $d(u, w)$, $d(v, w)$ and pick the largest among them!

The final problem is: how to find distances in a tree quickly? This is in fact a well known problem and has an efficient solution. First root the tree arbitrarily and prepare the so-called LCA structure. We can do it in $O(n \log n)$ time, so that then we can answer questions about lowest common ancestors in $O(\log n)$ time. Using this structure, we can easily answer distance queries in $O(\log n)$ time.

The final complexity of our solution is $O(n \log n)$.

D Walls

Author: **Slobodan Mitrović**

D1 Easy

By the definition, two columns c_1 and c_2, $c_1 \leq c_2$, are part of the same wall if and only if every column c, $c_1 \leq c \leq c_2$, contains at least one brick. On the other hand, if column c contains a brick, then at least the first row of column c contains a brick. Therefore, in order to count the number of walls, it suffices to count the number of groups of consecutive columns that contain brick in their first rows. This can be easily done in linear time.

D2 Medium

In this problem we are given n bricks and want to count the number of ways in which we can build walls. Or in other words, our task is to count the number of ways, summing over all $1 \leq i \leq n$, in which we can place i bricks into C columns.

First, if we decide to place t bricks in one column, than there is only one way to do it, i.e. all bricks are the same. Let x_i be the number of bricks in column i, and x_{pile} be the number of bricks out of total n that are not placed in any of the columns. Following that, we can phrase the problem as follows. In how many ways we can place n bricks such that

- a very technical condition – every pile contains a non-negative number of bricks, i.e. $x_i \geq 0$ for all $1 \leq i \leq C$;
- the columns contain at least 1 and no more than the number of available bricks, i.e. $1 \leq \sum_{i=1}^{C} x_i \leq n$, and
- the number of unplaced and the bricks placed in the columns is equal to the total number of bricks, i.e. $x_{pile} + \sum_{i=1}^{C} x_i = n$.

Observe that the above conditions imply $x_{pile} \geq 0$.

Now, the above question has a very nice solution, which is

$$\binom{n+C}{C} - 1 = \frac{(n+C)!}{n! \cdot C!} - 1,$$

where -1 comes from the fact that $x_{pile} = n$ and $x_i = 0$, for every $1 \leq i \leq C$, is the only placing of the bricks such that no column has any brick.

To compute the above binomial coefficient, we use the fact that we are asking for a count modulo a prime number. This allows us to easily obtain an inverse of $n!$ and $C!$ by using Fermat's Little theorem that gives us

$$a^{p-2} \equiv a^{-1} (\text{mod } p),$$

for any integer a not divisible by p. As $(n+C)!$, $n!$ and $C!$ are not divisible by p, we can use the theorem along with a fast exponentiation to obtain the final result.

D3 Hard

Consider the following dynamic programming table dp

$$dp[pos] := \text{ the number of useless walls on } pos \text{ columns.}$$

Then, we have the following (recurrent) definitions

$$\begin{aligned} dp[i] &= 0, \forall i < 0, \\ dp[0] &= 1 \text{ – base cases,} \\ dp[i] &= \sum_{w=0}^{W} \left(dp[i-(w+1)] \cdot H^w \right), \forall i \geq 1 \text{ – the inductive step.} \end{aligned} \tag{1}$$

Term $dp[i-(w+1)]$ in the sum above corresponds to asking :"How many useless walls there are if the wall that ends at position i has width w?" Note that, in that case, column $i-w$ contains no brick. Term H^w is the number of walls such that each wall has width w and height at most H. Although this approach outputs correctly what the task asks for, it is too slow as it runs in $\Theta(CW)$.

However, our recurrent definition depends only on "the last" $W+1$ entries of dp table. Fortunately, we can leverage that to obtain much faster solution, well at least much faster in C. For the sake of brevity (and clarity), let us look at the case $W = 2$. From the recurrent formula (1), it is sufficient to keep track of the last $W+1 = 3$ table entries. In other words, to obtain $dp[i+1]$, it is enough to know $dp[i], dp[i-1]$, and $dp[i-2]$. So, having $dp[i], dp[i-1]$, and $dp[i-2]$, let us write in a matrix-vector form how to get $dp[i+1], dp[i]$, and $dp[i-1]$

$$A \cdot b = \begin{pmatrix} dp[i+1] \\ dp[i] \\ dp[i-1] \end{pmatrix},$$

where A is the following matrix

$$A = \begin{pmatrix} 1 & H & H^2 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix},$$

and b is vector

$$b = \begin{pmatrix} dp[i] \\ dp[i-1] \\ dp[i-2] \end{pmatrix}.$$

Now, to get $dp[i+2]$ we can simply compute $A \cdot (A \cdot b)$. In general, to obtain $dp[i+k]$ we can evaluate expression $\underbrace{A \cdot (\ldots (A \cdot b) \ldots)}_{A \text{ appears } k \text{ times}}$. But, multiplying $(W+1) \times (W+1)$ matrices C times is even slower than the solution we had before. However, matrix multiplication is associative, and therefore we have

$$\underbrace{A \cdot (\ldots (A \cdot b) \ldots)}_{A \text{ appears } k \text{ times}} = A^k \cdot b.$$

Hence, using fast matrix exponentiation we can compute A^k in time $O(W^3 \log k)$. So, we set

$$b = \begin{pmatrix} dp[0] \\ dp[-1] \\ \vdots \\ dp[-W] \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

and evaluate $A^C \cdot b$ in order to obtain $dp[C]$, which now we can do in $O(W^3 \log C)$.

E Photographs

Author: **Jakub Tarnawski**

This was an ad-hoc problem, and both subtasks have many possible solutions. For the easy version, just comparing the sum of absolute differences between the boundary lines (on the two edges of the picture vs. on the two middle rows) was enough to solve 90% testcases correctly.

In the second subproblem, one needed to be more creative (especially since most instances were not even pleasant to solve for humans). A basis of most solutions would be a function which, given two stripes, returns some measure of their dissimilarity (as in: how badly do they fit together?). The same function as for the easy version will not work so well here because of the Gaussian noise; instead one should design a function which averages out several neighboring pixels, thus making the noise much less problematic.

Having a nice function like this, one can define a complete graph with vertices being the stripes and edges having weight equal to their dissimilarity. We are then looking for a Hamiltonian path of minimum weight in this graph. Since there are at most $k = 16$ stripes, an $O(2^k k^2)$ exact solution (using dynamic programming on bitmasks[1]) is the best thing one can implement. However, depending on how good the dissimilarity measure is, one can sometimes also get away with using a greedy algorithm (for example: repeatedly find the two most similar stripes and merge them).

F Tree of Life

Author: **Paweł Gawrychowski**

[1]See e.g. `http://codeforces.com/blog/entry/337`.

F1 Easy

We are given a tree and are asked to compute the number of paths of length 2. To this end, we try out all vertices as candidates for the middle-point of the path. For a fixed vertex, the number of such paths is $\frac{d(d-1)}{2}$, where d is the degree of the vertex (the number of its neighbors). We just take a sum over all vertices.

F2 Medium

The medium and hard versions are the hardest problems in the set.

For a tree T we define its *deck* as the (multi)set of forests (considered up to isomorphism) obtained by deleting exactly one vertex. The objective of the medium version is to check whether a given multiset of forests is the deck of some tree T. What's more, $n \leq 100$, so we can afford to be frivolous with the time complexity of our solution. The first useful observation is: in a deck, there must be a forest consisting of one tree only (just consider removing a leaf – every tree has at least one leaf). Conversely, every single-tree forest must have been obtained by removing a leaf. This leads to a pretty simple solution idea: we choose a single-tree forest (if there is none, just answer `NO`) and then try to "adjoin" a single vertex to it in each of the $n-1$ possible ways. We just have to smartly check whether such an adjoining results in a T with the same deck as the one in the input. How to do it? It's easiest to construct the deck and check if it's the same as in the input (as a multiset of forests up to isomorphism)...

At this stage, it's clear that we will need a clever way of checking whether two trees are the same. While this problem is quite hard for general graphs, for trees there is a simple (though not at all obvious) linear-time solution. What's more, using it, one can assign a unique identifier (a natural number up to n) to any tree, so that two trees are isomorphic if and only if their identifiers are the same. Having these identifiers, we can easily check whether two decks are the same: it's enough to sort the identifiers of trees in every forest, and then sort the resulting tuples lexicographically. What is the overall running time? By adjoining a leaf in each of $n-1$ ways, and then removing a vertex in each of n ways, we get n^2 trees (each on $n-1$ vertices). If we use the linear-time algorithm to compute their identifiers, the overall complexity is $O(n^3)$. But it is also fine to do that in log-linear time, giving $O(n^3 \log n)$ overall.

F3 Hard

The situation is similar, but now we get only two forests. If one of them is a tree, we can use the idea from the previous version to solve the problem. But, of course, this might not be the case...

A first idea could be to guess the neighbours of the deleted vertices. To this end, it is enough to choose exactly one vertex in each tree (in both forests). Unfortunately, there can be quite a lot of these trees and such a method leads to nothing more than a headache. But we can view this guessing process differently. Let the first forest be created by deleting u and the second by deleting v from T. The information about the neighbours of u (v, respectively) is in some sense encoded in $T \setminus \{v\}$ ($T \setminus \{u\}$, respectively), but how to recover it? Let us imagine that we know which vertex in $T \setminus \{v\}$ ($T \setminus \{u\}$, respectively) corresponds to u (v, respectively). Then, after their deletion (from both forests), we need to obtain two isomorphic sets of forests – see Figure 2.

So we have our first idea for a solution: try removing one vertex from each forest so as to get isomorphic forests. However, we get (at least) two issues:

- we have $(n-1)^2$ possibilities of removing one vertex per forest, and for each of them we should verify tree isomorphism, so the whole thing is $O(n^3)$, which is a lot...
- it is clear that the tree isomorphism is a necessary condition, but is it also sufficient? Or, to put it bluntly, is this solution correct?

Fortunately, dealing with the first issue can be postponed (indefinitely), because the solution is not correct! It turns out that the order of trees in the obtained forests does (somewhat) matter. More precisely, let

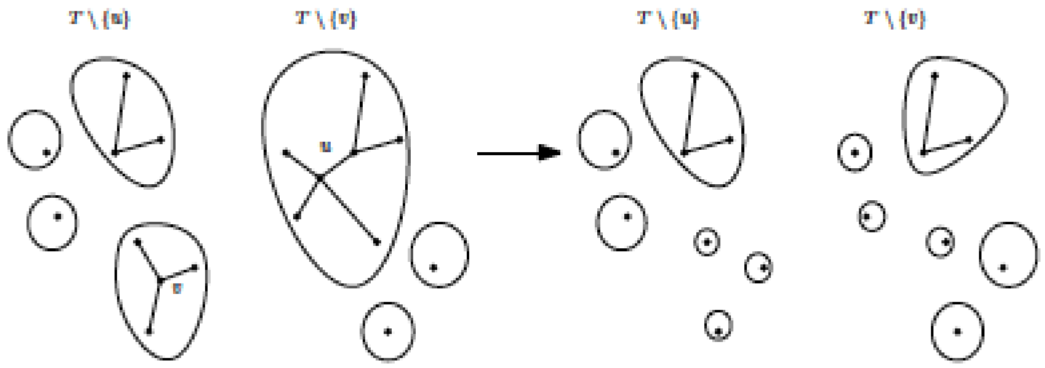

Figure 2: The forests after removing v and u.

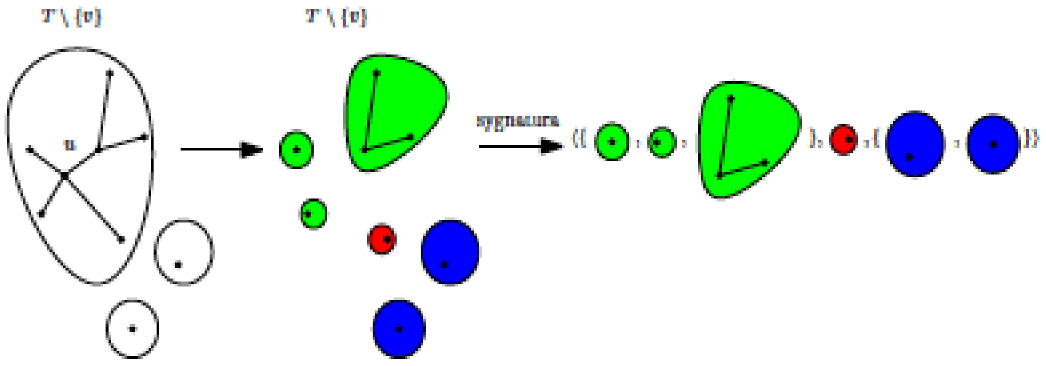

Figure 3: One of the signatures obtained from $T \setminus \{v\}$.

$T \setminus \{u\} = \{T_1, ..., T_k\}$ and assume that vertex $v \in T_k$ is removed, producing $T \setminus \{v\} = \{T'_1, ..., T'_{k'}\}$. Moreover assume that u was connected to a vertex in $T'_{k'}$. Then by the *signature* we denote the triplet $\{T_1, T_2, ..., T_{k-1}\}$, T'_k, $\{T'_1, T'_2, ..., T'_{k'-1}\}$ – see Figure 3. Similarly we define the signature for the forest obtained from $T \setminus \{v\}$, but here the first and third elements of the triplet are swapped. It can be proved that the signatures of $T \setminus \{u\}$ and $T \setminus \{v\}$ are the same if and only if the input in fact corresponds to some tree T. However, one problem remains: we have assumed that we know the trees connected to u and v. Fortunately, our idea is not really going to be very fast anyway: we delete one vertex from each forest in every possible way (there are $n-1$ ways). We can just as well already guess one neighbour for each of them (this increases the number of combinations in each forest only to $2(n-1)$).

Okay, so we have $O(n^2)$ subproblems, in each of which we need to check tree isomorphism. So the total running time is $O(n^3)$. So, even with correct implementation, we will still get TLE. How to cope with this? The following observation is key: what we are really doing is checking whether two sets have an element in common. Instead of guessing one element from each of them and checking whether this is really the same element, a much better idea is to generate and store the first set and then browse the other one. For each of its elements we check whether it has been stored before. With some care (and noticing we have forgotten about one special case) this gives a solution running in time $O(n^2 \log n)$.