CROATIAN OPEN COMPETITION IN INFORMATICS 2011/2012

Round 7

Croatian Olympiad in Informatics

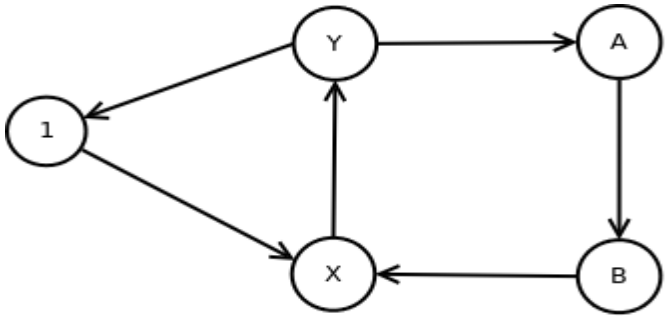
SOLUTIONS

COCI 2011/2012	Task KAMPANJA
Round 7 - COI	

The first step of the solution is computing *distance*[*a*][*b*], the minimum distance from city *a* to city *b*, for all pairs of cities. If there is no possible path between a pair of cities, we will use +inf as the distance. It follows that, if *distance*[1][2] = +inf or *distance*[2][1] = +inf, then no solution exists. This part can easily be implemented using the Floyd-Warshall algorithm.

Let us denote by *dp*[*a*][*b*] the minimum number of cities that need to be monitored so that a path 1 → *a* → *b* → 1 is possible, where cities a and *b* do not necessarily have to be distinct. Then it is possible to move from "state" *dp*[*a*][*b*] to *dp*[*x*][*y*] with the following cost:

$$dp[x][y] = dp[a][b] + distance[b][x] + distance[x][y] + distance[y][a] - 1$$



If we search the state space over all (*a*, *b*) pairs using Dijkstra's algorithm, we will obtain the optimal solution in *dp*[2][2].

COCI 2011/2012	Task MJESEC
Round 7 - COI	

The first thing we need to do is finding the orientation of the robot. It can be done by issuing the command “move **L**”, where **L** is the total length of the track, calculated easily from input data. The command will cause the robot to complete one full circle along the track, counting each turn exactly once. Since the track is a polygon that doesn't intersect itself, it is easy to show that the number of left turns will be greater than the number of right turns if and only if the robot is moving counterclockwise.

The next step is moving the robot to some vertex (turn). It can be solved using binary search. Observe that, if the robot is currently at some position **P** with distance exactly **X** to the closest vertex, then **X** is the smallest number such that “move **Y**” returns (0, 0) for all **Y** < **X** and a value different from (0, 0) for all **Y** ≥ **X**. It follows that we can find **X** by binary search, by following each “move **Y**” command (whose result we search over) by a “move **L**-**Y**” command in order to return to position **P** before the next iteration. After finding **X**, we simply execute “move **X**”, which is guaranteed to place the robot in some polygon vertex - we just don't know which one.

The last step is finding the exact vertex where we have moved the robot. In the beginning, all vertices are possible candidates for the robot's position. Now we need to find, for some two candidates **A** and **B**, a number **Y** such that the command “move **Y**” returns a different value depending on whether the robot is in **A** or in **B**. Then, after executing “move **Y**”, we can eliminate at least one of the vertices as a possible candidate (perhaps even both). Then we can return to our starting vertex using the command “move **L**-**Y**” and repeat the process until only one candidate vertex is left, giving us the exact position of the robot. Now we simply need to find a good value of **Y** for given vertices **A** and **B**: it can be done by simulation, walking along the track and constructing the sequence [segment_length, turn_orientation, segment_length, …] for one complete circuit around the track, with each of the two vertices as a starting position. The sequences for **A** and **B** must differ because the problem statement guarantees that a unique solution exists. The first difference between the two sequences gives us the needed value of **Y**.

Since the first step uses 1 command, the second at most 2*log(max segment length) commands, and the third 2*N commands, we are guaranteed to use less than 5000 commands in total.

COCI 2011/2012	Task SETNJA
Round 7 - COI	

Each of Mirko's friends is described by numbers **X**, **Y**, **P**. Notice that possible meeting points with a given friend form a square centered at (**X**, **Y**) – more precisely, a square with opposing vertices at (**X** – **P**, **Y** – **P**) and (**X** + **P**, **Y** + **P**).

Now the task has been reduced to finding the shortest possible path that touches all squares in order from first to last.

For each **K** from 1 to **N**, we consider all possible shortest paths that visit squares 1, 2, …, **K**, in that order. Let *S*(**K**) be the set of all endpoints of those paths, i.e. the set of all positions where we could have ended up after visiting the first **K** friends in an optimal manner. Let *d*(**K**) be the length of these shortest paths.

Notice that *S*(1) is precisely the square corresponding to the first friend, since we could have started (and ended) a path with length 0 by visiting the first friend in any point of that square.

The solution should proceed to find *S*(2), *S*(3), …, *S*(**N**). As we will show, all the sets will be **rectangles**. Now we have to determine how to find *S*(**K**+1) if we are given *S*(**K**).

Let **D** be the minimum distance (number of steps) from any point of *S*(**K**) to the square belonging to the (**K**+1)st friend (whom we need to visit next). Notice that *d*(**K**+1) = *d*(**K**) + **D**.

If we expand the rectangle *S*(**K**) by **D** units in all four directions, we will obtain all points reachable after optimally visiting the first **K** friends and then moving **D** steps in any direction. The **intersection** of the expanded rectangle and the square corresponding to the (**K**+1)st friend is therefore the set of points where we can meet friend (**K**+1) after making *d*(**K**) + **D** steps (from the definition of **D**, the intersection is guaranteed to be nonempty) – therefore, this intersection is exactly *S*(**K**+1), furthermore it is a rectangle (as an intersection of a square and a rectangle, both aligned with the same axes).

The final solution is simply the sum of all **D** values obtained while finding *S*(2), *S*(3), …, *S*(**N**) – this is precisely *d*(**N**).

COCI 2011/2012	Task TRAMPOLIN
Round 7 - COI	

Depending on Kickass's starting position, we can distinguish two cases:

Case #1 (appearing in 20% of test data): it is impossible to reach any trampoline.

At first glance, we might think that it is enough to simply compare the number of skyscrapers visitable by moving as far as possible to the left versus to the right. However, there are two additional possibilities: moving left visiting all skyscrapers with the same height as the starting one and then moving all the way to the right, as well as the mirror case of this. It is easy to check both possibilities and choose the better one.

Case #2 (appearing in 20% of test data): at least one trampoline is reachable.

Let us assume that A, B, C, ..., M are indices of all skyscrapers which contain a trampoline or have at least one trampoline reachable from them. We will call such skyscrapers beautiful. Notice that the starting skyscraper (**K**) is among them according to the assumption. Let $T(A)$, $T(B)$, $T(C)$, ..., $T(M)$ be the corresponding reachable skyscrapers with trampolines (not necessarily distinct).

Consider the following path:

K -> ... -> T(**K**) -> A -> … -> $T(A)$ -> B -> … -> $T(B)$ -> C -> … -> $T(C)$ -> … … … -> M -> … -> $T(M)$

This path is guaranteed to visit all beautiful skyscrapers. After that, we need to visit the longest possible sequence of non-beautiful skyscrapers (ones with no reachable trampoline); we can reach any such sequence using $T(M)$. The total solution is then the sum of the number of beautiful skyscrapers and the length of the longest non-beautiful skyscraper sequence.

We can implement this solution in several steps:

1) We first mark all skyscrapers containing a trampoline as beautiful.
2) Traversing the skyscrapers from left to right, we mark skyscraper ***i*** as beautiful if skyscraper (***i*** − 1) is beautiful and we can jump from ***i*** to (***i*** − 1).
3) Traversing the skyscrapers from right to left, we mark skyscraper ***i*** as beautiful if skyscraper (***i*** + 1) is beautiful and we can jump from ***i*** to (***i*** + 1).

4) For all skyscrapers that haven't been marked beautiful in one of the previous steps, we have to find the number of skyscrapers we can visit from that skyscraper going to the left. It can be computed dynamically, in a way similar to step 2). Analogously, the number of skyscrapers visitable to the right can be computed similarly to step 3).
5) The final length of the longest non-beautiful sequence is simply the largest of the numbers obtained in step 4).

The complexity of the algorithm is O(**N**) in both cases.