PROBLEM 4

Report with the implementation of Problem 4 explained, justification of why the implementation has been chosen and finally some examples with the correct operation of the implementation

Isabel Martínez Gómez

Algorithm and Complexity

2019-2020

SOLUTION EXPLAINED

**BACKTRACKING ALGORITHM**

First, I’m going to explain how I have implemented the backtracking algorithm to get from point 1 of the map to point 2, finding all possible paths and indicating what is the optimal solution.

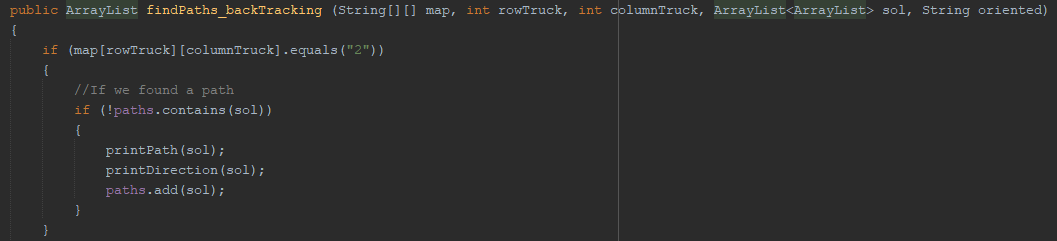
I will start by commenting the main structures that I have used to solve the problem:

* **Paths:** this is an arraylist that is used to save all the paths founded. If for example, we find one path that is already in paths, having this arraylist, we could discard it.
* **Map:** is a String array of two dimensions which we will initialize from the input file example\_backtracking.txt
* **RowTruck:** is the row on the map where the truck is initially located.
* **ColumnTruck:** is the column on the map where the truck is initially located.
* **Sol:** is an arraylist where the positions and directions of the possible path from 1 to 2 will be added. This will be added in the form: [row,column,direction,orientation].
* **Oriented:** contains the orientation of the truck that can be: left, right, up and down.

**FINDPATHS\_BACKTRACKING**

In this function we managed to obtain, through a map of Abecelandia and the position of the truck with its orientation, all the possible paths that the truck would have to get from point 1 to point 2, driving only on straight line and with 90º turns to the right.

The first thing to consider is, if we have reached point 2, we add the path that is saved in sol to the arraylist paths, we also call the functions printPath and printDirection that are responsible for writing in the output file the path found. The first, writes the positions followed by the path and the second, the directions (straight or right) that follows the found path.



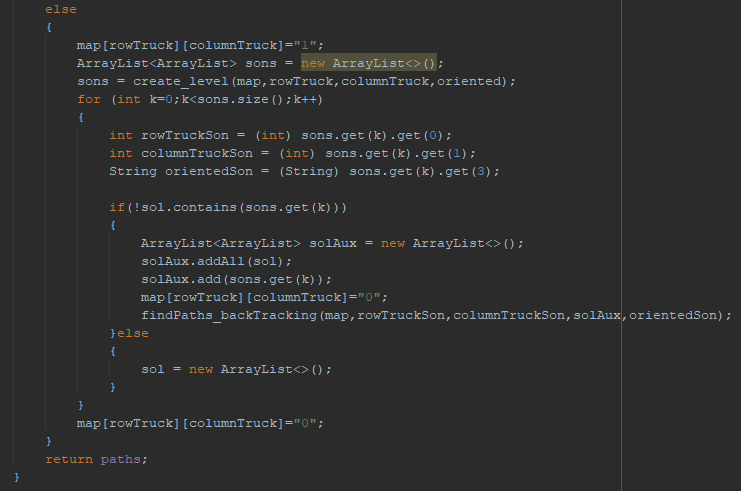
If, on the contrary, we haven’t found point 2, we are going to look for possible positions where the truck can go, adding in each iteration a possible position to the arraylist sol and recursively calling the function with that position.

The first thing we will do here will be to mark the position with a 1, to mark the truck's transition from one point on the map to another. Then, I have created an arraylist sons which will contain the possible positions where the truck can move from the position in which it is located (continue on straight line, go to the right or both), these possible positions will be returned by the create\_level function, which I will explain below.

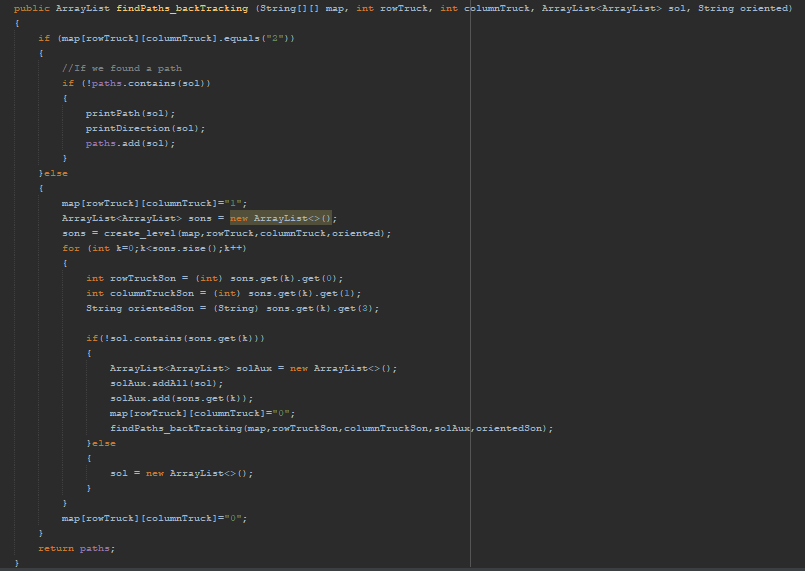
Once we have all the possible positions for the truck at that moment, we go through the sons arraylist and check if that position we had not previously visited, in this case, we will add it to the solution arraylist, we mark the truck position with a 0 , since the truck will move to a new position on the map and we make a recursive call with the new position of the truck.

On the other hand, if that position had already been visited, it will mean that we are passing through a position that we have already passed before, this could mean that we are in a loop, therefore, the solution found so far and that will be stored in the arraylist sol is not correct or is redundant, so we will have to remove what it contains.

Finally, when there are no more possible paths, all recursive functions will end and the arraylist paths will be returned, which will have all the paths found from point 1 to point 2, satisfying the restrictions of going straight and turning right only.



This is the entire function:



**CREATE\_LEVEL**

This function is responsible for returning the positions to which the truck can go in an instant, given the map and the position and orientation of the truck by parameters, following in a straight line or turning to the right. It will be considered the cases where the truck is oriented to the left, right, up or down.

First, I consider the case that the truck is oriented to the left. In that case, two case are verified, if the truck can go straight and if it can turn to the right.

If the truck can continue straight line, the position, the direction and the orientation of this possible path for the truck would been add to the arraylist sons, which will be the returned arraylist, and which will help the above explained function to find a possible path.

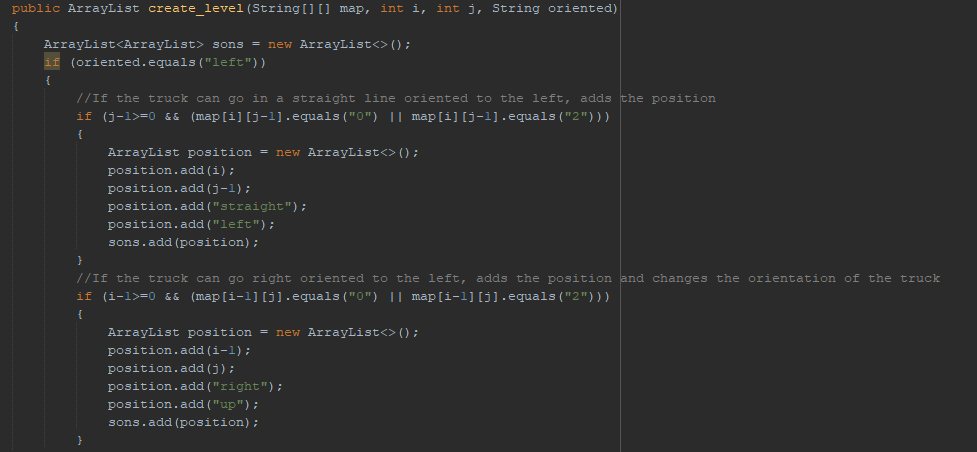
Then, it is checked whether the truck can turn right, either because there is a passable road or because we find the stadium. If it can turn to the right, then position, direction and orientation are added to the arraylist sons.

This works in the same way for all orientations but changing the positions of row and column from the map.

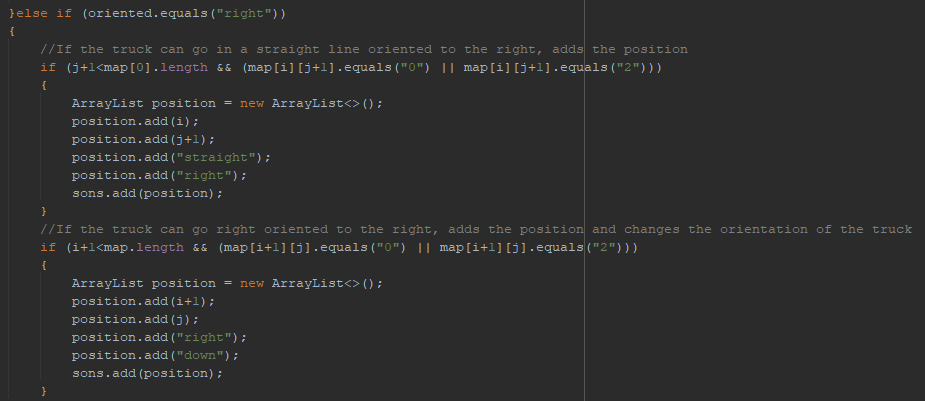
Finally, and once looked according to the orientation, whether the truck can go straight or to the right, the arraylist sons is returned.

**IMPORTANT:** it is important to observe how this function is structured. As you can see, the first condition is always if the truck can go on straight line, so if the findPaths\_backtracking function finds a path, the first path will be always the optimal solution, which will be the path with the fewest turns to the right.

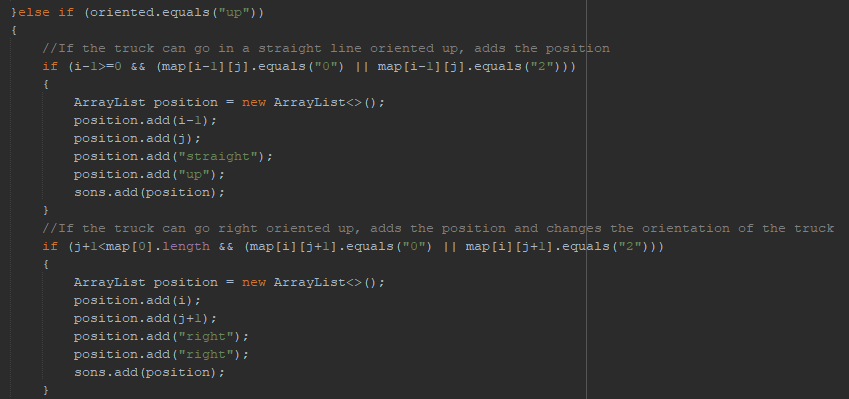
Case the truck is oriented to the left



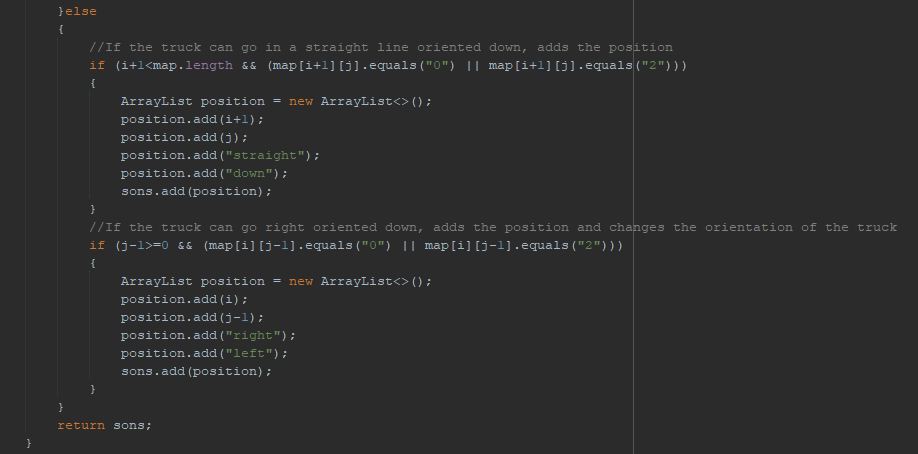
Case the truck is oriented to the right



Case the truck is oriented up

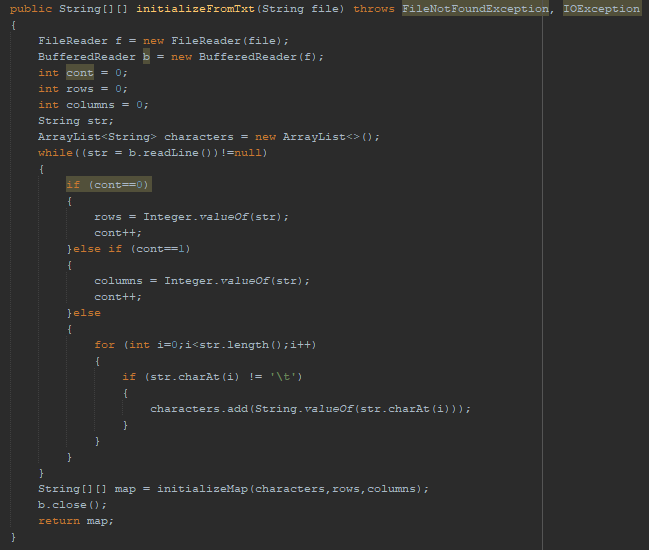


Case the truck is oriented down



**INITIALIZEFROMTXT**

In this function is initialized the data from the input file "example\_backtracking.txt".



To initialize the data, a counter that starts at 0 is used. When the counter is 0, then it is known that the number in the file refers to the rows that the map has, when the counter is 1, then the number of the input file refers to the columns of the map and otherwise is the data of Abecelandia map.

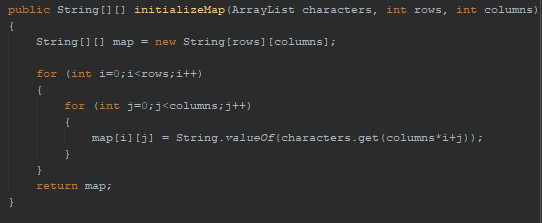
To initialize the map, I have created another function called initializeMap where we pass an arraylist with the characters collected from the map, eliminating the tabs that came in the input file between character and character. Also, we pass as a parameter the number of rows and columns of the map.

Finally, we return the map created by that function.

**INITIALIZEMAP**

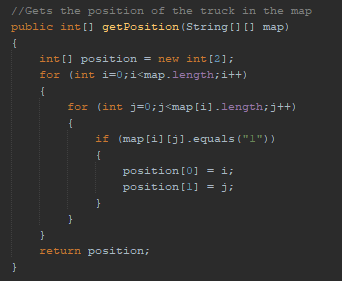
In this function, we create the map in a String[][] structure with the data obtained from the input file.

As characters is a one-dimension arraylist, to correctly initialize the map, we have to access to the equivalent position by doing: *columns \* i + j*   
For example, if we have a 4x4 map and we want to access position (0,0) that the map will have then 4 \* 0 + 0 = 0, however if we want to access position (2,1) on the map, we will do 4 \* 2 + 1 = 9, therefore position 2,1 of the two-dimensional map is position 9 of the one-dimensional arraylist characters.



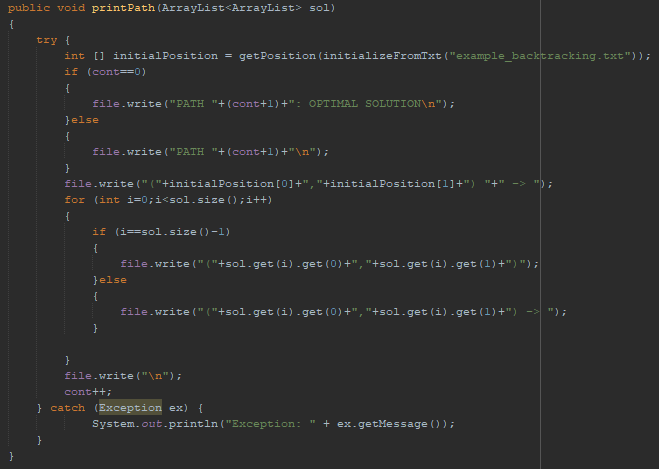
**GETPOSITION**

This function returns the initial position of the truck. We need this function so that when we call the findPaths\_backtracking function with the map and orientation of the truck, we can pass the truck's starting position as parameters.



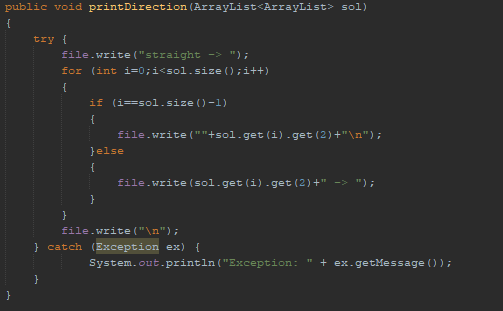
**PRINTPATH**

This function is required to write the solution of the problem in the output file "output\_backtracking.txt". When findPaths\_backtracking function finds a path, this function writes it in the txt with the positions followed by the truck from point 1 to point 2.



**PRINTDIRECTION**

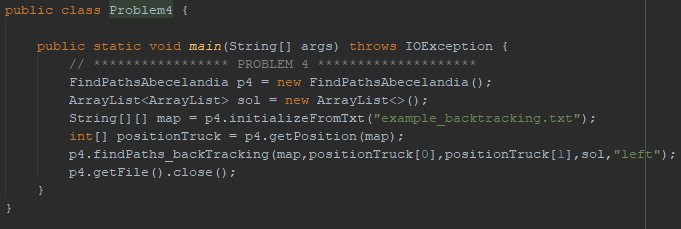
This function is required to write the solution of the problem in the output file "output\_backtracking.txt" When findPaths\_backtracking function finds a path, this function writes the path with the followed directions (straight or right) for the truck from point 1 to point 2.



**MAIN FUNCTION**

This is the main function in which we first initialize from the input txt the rows, the columns and the map of Abecelandia, then we get the initial position of the truck and finally we call to the function findPaths\_backtracking to find all the possible paths from the point 1 to the point 2.

As the problem indicates, the truck is initially oriented to the left.

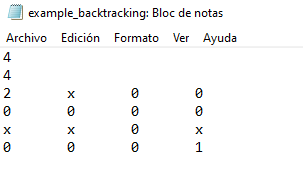


EXAMPLES

**FIRST EXAMPLE**

This is the Example 1 of the statement

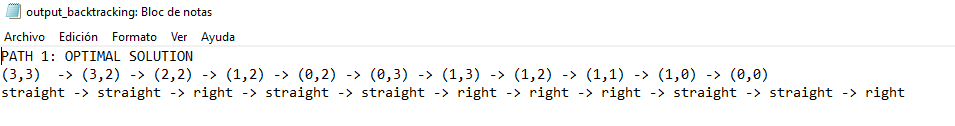
**INPUT FILE**



**OUTPUT FILE**

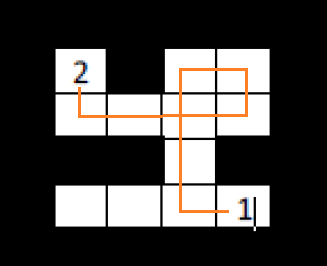
As we can see we only obtain a possible path to get from position 1 to position 2.

As this path is the only solution, it is added to the output file as the optimal solution of the problem.



The path is the following:

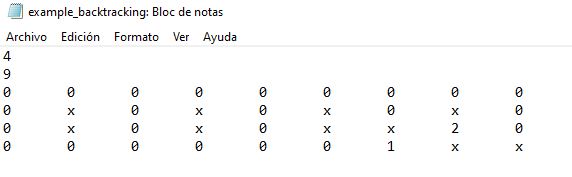
* Path 1



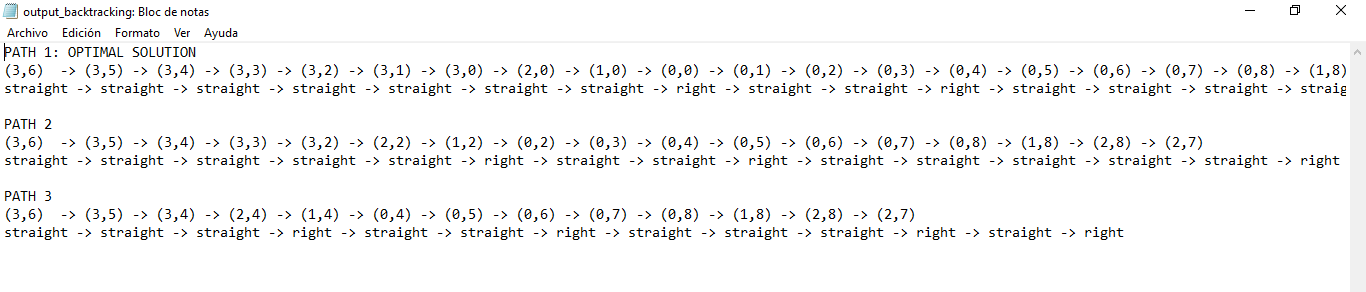
**SECOND EXAMPLE**

This is the Example 2 of the statement

**INPUT FILE**

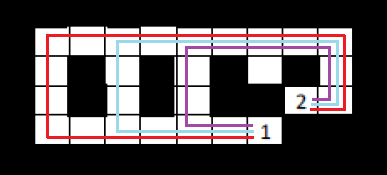


**OUTPUT FILE**

As we can see, there are three possible paths from the starting point of the truck to the stadium. In this case, they all have the same turns to the right, therefore, take the optimal path as the first one found by default.

These are the paths:

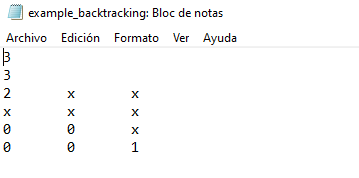
* Path 1
* Path 2
* Path 3

****

**THIRD EXAMPLE**

**INPUT FILE**

This example has no possible path to get from point 1 to point 2.



**OUTPUT FILE**

