1. I would start by having the player look at each element adjacent to it. This would include looking up (1 row above), down (1 row below), right (+1 column), and left (-1 column). The algorithm would check for S in all adjacent elements and finish the maze if the S was adjacent by going onto it. To brute force solve a maze, you can follow one side of a wall forever, until you find the exit. In this example algorithm I will choose going right (the equivalent of putting your hand on the right wall and moving like this the entire time) but this could work by trying to follow the left wall the entire time as well.

After checking the adjacent elements for S and failing to find it, I would have the movement be prioritized by moving right first. This would continue until an X is seen in the next right position, which would cause the player to prioritize going up next, and if an X is seen in that position, then going down unless there is an X, and lastly going back the way we came or left. The only thing we would have to do now is define which way right is. For this, we can create a piece of code that defines going right by incrementing the column of the 2D array and going left by decrementing the column of the 2D array. Then, if the player must go left a column instead of right, we can flip this definition to say right is now decrementing the column and left is now incrementing until the player is forced to move to the original right again, then the definition goes back to right being an increment to the column and left being a decrement. This effectively changes the players priority to going left (new right), up, down, and then lastly right.

The order of priority for up and down must also change. If the player needs to move down a row, the priority of the player needs to go right, down, left, up. If the player needs to go up a row, the priority must change to going down last. For this example, we will try to go right, up, left, down. Essentially, anytime you are going one direction you prioritize going the opposite direction as the very last thing. And we will always prioritize going “right” which changes depending on if the players last column change was positive or negative. If the last column change was neither positive or negative, which means the player is moving up or down in the maze (decrementing rows is up, incrementing rows is down), we will say that right is normal or incrementing the column. This is a brute force way that will solve any maze that the player is put into so long as there is an exit path.

2.

Maze Criteria:

The criteria for a maze needs several things, first it needs to have an exit. Then it needs to have a branching path where one path ends at the exit. No path can trap the player from reaching the exit. Next there needs to be a defined area of the maze, so we need an outer boundary. Next, we need walls that surround our paths.

For the algorithm:

First, we will need to generate a 2D array to define the size of our maze. The array size can be random, but for the sake of this example we will make it a “square” maze where both the column size and row size are equivalent. After the length of the column and rows are decided, we must put an S in a semi random position, where we can choose to put it either in a row where the column element is 0 or the last element of the column or a column where the row is 0 or the last row element. This will ensure our maze has an exit which will be on the outer boundary of the maze, thus when you reach the exit you will have escaped. The rest of the positions our exit could of gone will be filled with X’s. Next, we can start to create a path where elements are assigned to a space character. This path can be randomly generated with a few rules. We will go through each row and column starting at the exit but when randomly deciding where to assign a space, it must be adjacent to another space or the exit. There can be only 1 space adjacent to the exit, effectively making a single path be the key to escape. After each space is assigned, X's must fill in the other adjacent rows or columns that are unfilled without breaking the rule that spaces must have an adjacent space and that the path to the exit cannot be blocked. The path can branch and be traced randomly in any direction this way, and the X walls will create a single branching path.