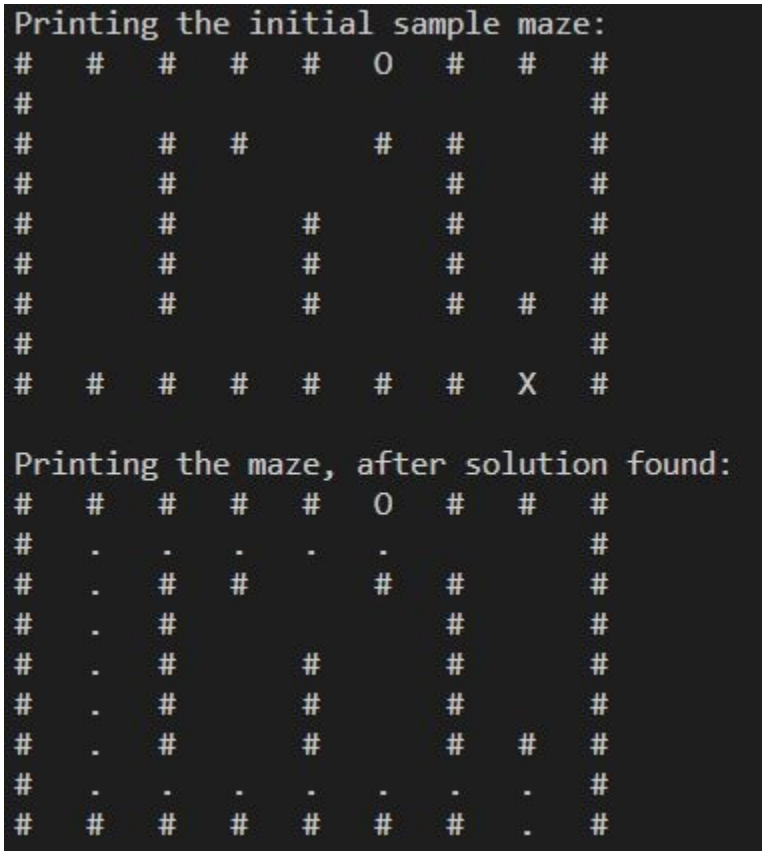
Searching for an Exit using DFS in Python

This problem will help us solve some recursive problems using a technique referred to as a depth-first search (DFS) to look for an exit for a given maze. The maze will be sent to the function as a list of lists, so there is no specific way to get the maze data; here's an example of the expected output:



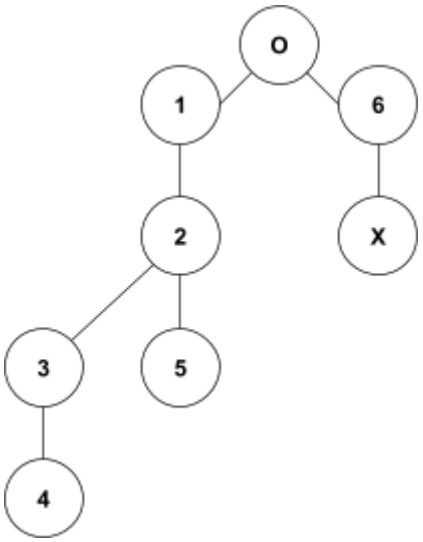
About Depth-first search: Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking.

In a file named maze.py, we will include a function called dfs(list(list(string)), tuple(int, int), list(tuple(int, int)) = []), or as a simpler example of what it might look like in code: dfs(maze, position, explored=[]).

Searching for an Exit (cont.)

Depth-First Search

The DFS is used a lot in computer science to search trees or graphs for specific nodes and many other purposes, but it was originally developed as a maze-solving strategy!



DFS works by following one path as far as it can - let's say it always goes left first. Every time it visits a spot, it adds that position to a list of "explored" nodes so that we know not to visit it again. When it gets to the end (no more places to go) it returns "False" to say it didn't find anything. If it finds what it's looking for, it reports back "True" to say it found it!

In the case of the graph to the right, I've numbered it in the order our DFS would look at the nodes, seeking from O to X. It doesn't take the best path, but it finds the end eventually.

DFS in our Maze

The logic of DFS remains the same, but we will need to work out some extra logic to make it work how we need it to. We can consider our maze a "graph" where each spot is connected to up to four other spots; the spot above, left, right, and below. It's considered "adjacent" if it's one of those four positions, within bounds, and not a wall ("#"). A function has been provided to give us all adjacent spots - don't worry! We're starting from "O" (position supplied in the function) and going to "X".

Pseudocode for a Recursive Depth-First Search:

```
dfs(list_of_connected_points, current_point, explored=[])
    Add current_point to explored
    For each point adjacent to current_point:
        If that point is not in the explored list:
            Call DFS on the list of connected points with the new point End if
    End For
```

Searching for an Exit (cont.)

The maze_helper.py file contains several functions to help us avoid dealing with spatial issues when writing our algorithm. We will review the purpose of these functions, how to use them, and where they might help us:

sample_maze(): This function provides us with one sample maze.

get_adjacent_positions(maze, position): This function will check each location around the position provided and return a list of tuples containing the positions of each location that is not a wall.

symbol_at(maze, position): This function returns the symbol in the maze at the provided position. While these are simple, these functions have been provided so that we do not need to worry about treating the list as row/col vs x/y.

add_walk_symbol(maze, position): This function will place a dot at the correct position in the maze.

print_maze(maze): Prints the provided.