Exercise 3

In this exercise you are allowed to import only NumPy and use only the random.choice function.

<u>Part A – Recursion:</u>

This part should be submitted in a separate code file named ex3a.py.

a. Write a <u>recursive</u> function *maze_solver*. The function receives a maze and it returns whether the maze has a solution or not.
   The function will receive a maze which is a list of lists containing strings. The are 3 optional strings in those lists: 'S' for the starting location, 'T' for target and 'X' for walls.
   The function returns *True* if there is a solution to the Maze and *False* otherwise.
   As always you are allowed to use additional functions to implement this function.
   For example:
   Maze = [['X', '', '', 'S'], ['X', '', 'X', ''], ['T', '', 'X', ''], ['X', 'X', 'X', '']]

X			S
X		X	
T		X	
X	X	X	

   *Maze_solver(*Maze) = True

b. Write a class *Maze*. The constructor of the class will receive a path of a csv file containing a maze. The constructor will read the file and convert it to the list of lists format as explained in the function. The constructor should check if the maze has one starting point, one ending point and all the rest are walls ('X'). Only the uppercase letters mentioned in the previous section are valid. The constructor will return a ValueError if the maze is invalid.
   After implementing the constructor, add the function written in section 1 as a method of the class.
   Checking whether the letters 'S' and 'T' (only uppercase counts) appear in the maze is mandatory. Other validation tests are considered as bonus (4 points).
   Example for the CSV file:

maze_example.csv

   The maze in this CSV file is the same maze as in the example appeared in section 1a.

<u>What is a CSV file</u>?

A CSV file is a comma separated file. It means that the file is a simple text file containing only the cell values separated by commas. In addition, in the end of each line there is a char or two (depending on the operating system that you use) indicating a line drop.

For example, the first line from the CSV file attached above will look like:

f = open(file_path)

print(f.readline()) -> "X,,,S\n"

Part B – Snakes and ladders:

This part should be submitted in a separate code file named ex3b.py

In this part you will implement a unique version of snakes and ladders in a class named *SnakesAndLadders*. The game rules are similar to the original game and the target is the same: you should reach the last square, the 100th square. We will play our game using only one dice. A player starts the game on square 1. There are three modifications in our version:

a. You are allowed to choose the probability of each value of the dice.
b. If you are reaching the bottom of a ladder, you do not automatically climb it. First, you need to roll the dice once again and if the value is lower than 4 (1, 2 or 3) you climb it.
c. If you are reaching a snake head, you do not automatically move to its tail. First, you need to roll the dice once again and if the value is higher than 3 (4,5, or 6) you go back to the snake's tail.

The *SnakesAndLadders* class will have the following methods:

a. Constructor – the constructor receives two lists: one for the snakes and one for the ladders. Both lists are given in the ex4b.py file. The ladders list contains tuples in which the first value is the bottom of the ladder and the second one is the end of the ladder. The snakes list contains tuples in which the first value is the head of the snake and the second value is the tail.
b. *update_dice(prob_list)* – the method updates the probabilities of the values of the dice. The method receives a list of six fractions representing the probabilities for the dice values. In other words, the first value is the probability of the value 1 and the second probability is for the value 2 etc.
   The method does not return anything.
c. *roll_dice()* – The method implements a dice roll. Practically, you sample a number according to the probabilities of the dice as updated using *update_dice*.
   The method returns a value randomly sampled.
d. *play_game()* – The method implements a game of one player according to the rules mentioned above.
   The method returns a tuple which contains the number of turns until the game ends as the first value and the dice roll results in a list. You should not save the extra rolls when arriving to a snake or a ladder.
   Limit the game to 900 turns. After the 900th turn it returns the number of turns (900) and the dice roll results as well.
   For example:
   game = *SnakesAndLadders(SNAKES, LADDERS)*
   res = game.play_game()
   res[0] will be the number of turns, res[1] will be a list of values between 1 to 6.

Part C – Playing snakes and ladders **(bonus – 6 points)**:

In this part you are going to play the game you implemented. All the answers to this part should be written in the answer file. Add the code lines you used to the answers file as well.

a. Choose 2 dice with different sets of probabilities. Write the probabilities in the answers file.

b. Run the game with the example board game 100 times with each dice you chose. Compute the average "number of turns" for each dice, the longest and the shortest game. Write the results in the answers file.
   For example:
   I chose the dice [1/6, 1/6, 1/6, 1/6, 1/6, 1/6]
   The number of turns [31,54,62,41,51,56,48,52,68, 49] – the list should include 100 values, one for each stimulation.
   The average number of turns of this die is 51.2.
   Shortest game: 31
   Longest game: 62

c. Run another game (only one) for one of you dice. Calculate the probability of each value of the die. Write the original probability and the one you calculated in the answers file. Is it similar to the probabilities you chose?