Q1:

# EXPLANATION:

The code generates a random Sudoku puzzle and then solves it using depth-first search (DFS) algorithm.

The **generate\_sudoku()** function first creates a 9x9 Sudoku puzzle board filled with zeros. It then fills in the diagonal boxes with numbers 1-9, shuffles the rows and columns, and finally adds constraints to each 3x3 box. The result is a random Sudoku puzzle.

The **solve\_sudoku\_dfs(board)** function takes a Sudoku puzzle board as input and solves it using DFS algorithm. The **dfs\_helper(board, depth, num\_calls)** function is a recursive function that takes a Sudoku puzzle board, the depth of the search tree, and the number of function calls as input. It returns True if the board is solved and False otherwise.

The **get\_choices(board, row, col)** function takes a Sudoku puzzle board, a row and a column as input and returns a set of valid choices for that cell. It removes numbers that are already in the same row, column, or 3x3 box from the set of choices.

In the **solve\_sudoku\_dfs(board)** function, the DFS algorithm chooses the cell with the fewest number of possible choices and tries each choice recursively until the board is solved. The algorithm backtracks if it reaches a dead end.

After solving the Sudoku puzzle, the **solve\_sudoku\_dfs(board)** function prints the time complexity and space complexity of the algorithm. The time complexity is the time taken to solve the puzzle in seconds, and the space complexity is the number of function calls made by the algorithm.

The modifications made in the depth-first search (DFS) algorithm in the code are specific to solving a Sudoku puzzle.

In a regular DFS, the search algorithm explores each path until it reaches the end of the path, or a goal is found. In a Sudoku solver, the algorithm must find a solution that satisfies the rules of Sudoku. The solver uses a backtracking approach, meaning that when it encounters a contradiction, it returns to the previous cell and tries a different value.

The modifications in the code involve selecting the cell with the fewest number of possible choices first. This is because it reduces the branching factor and makes the algorithm more efficient. Additionally, it uses a set of allowed values for each empty cell that are determined based on the rules of Sudoku.

The **get\_choices** function calculates a set of all the possible values that can be placed in each cell without violating any of the Sudoku rules. The **min** function in the **dfs\_helper** function chooses the next cell to fill based on the length of this set of possible values. This ensures that the algorithm tries to fill cells with the fewest possible choices first, which leads to faster convergence.

Overall, these modifications make the algorithm more efficient for solving Sudoku puzzles by reducing the search space and making the algorithm smarter in selecting which cells to fill first.

Q2:

# EXPLANATION:

This code generates and solves a 3x3 magic square puzzle.

The **generate\_magic\_square()** function first creates an empty 3x3 matrix **puzzle** and then generates a random permutation of the numbers 1 through 9. It then places these numbers in the cells of the **puzzle** matrix.

The **solve\_magic\_square(puzzle)** function takes a **puzzle** as an input and uses the A\* algorithm to find the solution to the magic square puzzle. It initializes a starting state tuple consisting of the **puzzle** matrix as a tuple of tuples, the index of the blank cell, and the number of moves made so far. It then creates an empty set **visited**, which will keep track of the states that have already been visited, and a stack **stack**, which will keep track of the states to be explored. The function continues to explore the states by popping the top state from the **stack** until the **stack** is empty or a magic square is found. The function generates all possible moves from the current blank cell position and applies each move to the current state to generate a new state. The new state is added to the **stack** if it has not been visited before. The function then returns the solution state and the number of moves made to reach the solution.

The **find\_blank(puzzle)** function finds the index of the blank cell in the **puzzle** matrix.

The **is\_magic\_square(puzzle)** function takes a **puzzle** as an input and checks if it is a magic square by checking if the sum of each row, column, and diagonal is equal to 15.

The **possible\_moves(blank\_idx)** function takes the index of the blank cell as input and returns the indices of the cells that the blank cell can be moved to.

The main code generates a magic square puzzle using the **generate\_magic\_square()** function and prints it. It then solves the puzzle using the **solve\_magic\_square(puzzle)** function and prints the solution and the number of moves made to reach the solution. Finally, the code prints the time taken to solve the puzzle.