

Theoretical analysis

\*Modeling the Game:\*

In the Snakes and Ladders game, we can model the game as a directed graph, where each node represents a possible board position, and each edge represents a possible move. This graph-based modeling is effective because it allows us to use graph traversal algorithms like Depth-First Search (DFS) to solve the tasks efficiently.

\*Representation of the Game:\*

1. \*Graph Representation:\* We represent the game board as a graph, where each cell is a node in the graph, and the edges between nodes represent possible moves (either through dice rolls or via snakes and ladders). We use an adjacency list to represent the graph, where each node (cell) stores a list of neighboring nodes that can be reached from that cell.

2. \*Snakes and Ladders Representation:\* We maintain a dictionary (`self.snakes\_and\_ladders`) that maps the start positions of snakes or ladders to their respective end positions. This representation allows us to quickly determine where a player should move when encountering a snake or ladder.

\*Justification for the Modeling:\*

- Using a graph-based model allows us to leverage graph algorithms like BFS, which are efficient for solving problems involving connectivity and shortest paths.

- The adjacency list representation is memory-efficient, as it stores only the necessary information about possible moves.

- The representation of snakes and ladders as a dictionary provides easy access to their endpoints during gameplay.

\*\*Explanation of Code:\*\*

1. `SnakesAndLadders` Class Initialization:

- The `\_\_init\_\_` method initializes the game board.

- `n` represents the board size (nxn).

- `graph` is a defaultdict of lists used to represent the game board as a graph.

- `snakes\_and\_ladders` is a dictionary that stores the mappings of start positions to end positions for snakes and ladders.

- `goal` represents the last cell on the board (n\*n).

- `start` represents the initial position of the player, which is always 1.

2. `add\_snake\_or\_ladder` Function:

- This function allows adding snakes and ladders to the game.

- It checks if the provided start and end positions are within the board boundaries.

- If they are, it adds the snake or ladder to the `snakes\_and\_ladders` dictionary.

3. `build\_graph` Function:

- This function constructs the game board as a graph.

- It iterates through each cell (node) on the board and adds edges (connections) to neighboring cells for possible moves.

4. `can\_reach\_goal` Function:

- This function checks whether it's possible to reach the goal (last cell) starting from the initial position (cell 1).

- It uses a breadth-first search (BFS) approach to explore the graph and mark visited cells.

- If the goal cell is reached, it returns `True`, indicating that it's possible to reach the goal; otherwise, it returns `False`.

5. `has\_cycles` Function:

- This function checks if there are cycles in the game board.

- It uses a depth-first search (DFS) approach to explore the graph while keeping track of the current path.

- If a node is encountered that is already in the current path, it indicates the presence of a cycle, and the function returns `True`.

- If no cycles are found, it returns `False`.

6. `verify\_board` Function:

- This function verifies if the game board meets the specified conditions:

- It checks if it's possible to reach the goal using `can\_reach\_goal`.

- It checks for the presence of cycles using `has\_cycles`.

- If both conditions are met, it returns `True`, indicating that the board conditions are satisfied; otherwise, it returns `False`.

7. `shortest\_sequence` Function:

- This function finds the shortest sequence of dice rolls required to reach the goal.

- It uses BFS to explore the graph, starting from the initial position.

- It maintains a count of the number of moves (dice rolls) made.

- When the goal cell is reached, it returns the count of moves as the shortest sequence.

- If it's not possible to reach the goal, it returns `None`.

\*\*Complexity Analysis:\*\*

1. `add\_snake\_or\_ladder` Function:

- Time Complexity: O(1) - Adding a snake or ladder is a constant-time operation.

- Space Complexity: O(1) - The function does not use additional space proportional to the board size.

2. `build\_graph` Function:

- Time Complexity: O(n^2) - Iterating through each cell and considering up to 6 neighboring cells.

- Space Complexity: O(1) - The graph is represented in the `graph` data structure, which is not dependent on the board size.

3. `can\_reach\_goal` Function:

- Time Complexity: O(n^2) - In the worst case, it may explore all cells.

- Space Complexity: O(n) - Queue and visited array store at most n elements.

4. `has\_cycles` Function:

- Time Complexity: O(n^2) - In the worst case, it may explore all cells.

- Space Complexity: O(n) - Stack and visited array store at most n elements.

5. `verify\_board` Function:

- Time Complexity: O(n^2) - Combining the time complexity of `can\_reach\_goal` and `has\_cycles`.

- Space Complexity: O(n) - Combining the space complexity of `can\_reach\_goal` and `has\_cycles`.

6. `shortest\_sequence` Function:

- Time Complexity: O(n^2) - In the worst case, it may explore all cells.

- Space Complexity: O(n) - Queue and visited array store at most n elements.

Overall, the code is efficient and can handle reasonably sized game boards.