909. Snakes and Ladders Medium Breadth-First Search Array Matrix **Leetcode Link**

Problem Description In this problem, we are given a square board of $n \times n$ size, labeled with consecutive numbers from 1 to n^2 in a Boustrophedon style,

alternates with each row. The player begins at square 1 and the goal is to reach square n^2. On each turn, the player can move to a square in the range of 1 to 6 squares ahead from their current position, as if rolling a six-sided die. Some squares contain the start of a snake or ladder, indicated by a non-negative number in board[r][c]. If a player lands on such a

meaning that the numbers start from the bottom left corner and snake back and forth row by row; the direction of numbering

square, they must follow it to the designated end square. The start and end squares of the board don't contain snakes or ladders. The player only follows a snake or ladder for a single move, not chaining multiple snakes or ladders in a single turn.

The objective is to find the minimum number of moves needed to reach the end of the board. We have to consider the effect of dice

rolls, the presence of snakes and ladders which can alter the player's position significantly, and the unique movement pattern due to

the Boustrophedon style numbering. It's important to track visited squares to avoid infinite loops (visiting the same squares over and over) and to make sure not to follow the same snake or ladder twice in a single move. Intuition

To solve this problem, we employ a Breadth-First Search (BFS) algorithm. BFS is a suitable approach here because it is excellent for

finding the shortest path in an unweighted graph, which essentially our game board is. Each roll of the die represents a potential transition from one node (square) to another. Moreover, since snakes and ladders can move the player to non-adjacent nodes, BFS can handle these jumps seamlessly while keeping track of the shortest path. We maintain a queue to represent game states we need to visit (game states are just positions on the board). We also keep track of

visited positions to avoid processing the same state multiple times. For each turn, we consider all reachable positions within a roll of

possibilities, we ensure we don't miss any potential paths that might lead to the goal in fewer moves, and once we reach n^2, we can

a six-sided die and follow snakes and ladders immediately as they only affect the current move. By gradually exploring all

The function get(x) in the solution converts the label of a square to its coordinates on the board, taking into account the alternating direction of the rows. The use of a set vis ensures that each square is visited no more than once, optimizing the search and avoiding cycles. The BFS loop increments a counter ans which represents the number of moves taken so far until it either finds the end or exhausts all possible paths.

Solution Approach The implemented solution approach uses Breadth-First Search (BFS), a popular search algorithm, to efficiently find the shortest path to reach the end of the board from the start. Here's the walkthrough of how the solution functions:

1. BFS Queue: We use a deque (a queue that allows inserting and removing from both ends) to store the squares we need to visit.

2. Visited Set: A set vis is used to track the squares we have already visited to prevent revisiting and consequently, getting stuck in loops.

Initially, the queue contains just the first square 1.

of that snake or ladder as described by the board (next = board[i][j]).

(ans) since this would represent the shortest path to the end.

Let's apply the solution approach to a small example to illustrate how it works.

reached - the function returns -1.

Example Walkthrough

Example Board

return the number of moves taken.

3. Move Count: ans is a counter that keeps track of the number of moves made so far. It gets incremented after exploring all possible moves within one round of a die roll.

4. Main Loop: The BFS is implemented in a while loop that continues as long as there are squares to visit in the queue.

5. Exploring Moves: For each current position, we look at all possible positions we could land on with a single die roll (six

- possibilities at most), which are the squares numbered from curr + 1 to min(curr + 6, n^2). We use the get(x) function to calculate the corresponding board coordinates from a square number. 6. Following Snakes or Ladders: If the calculated position has a snake or ladder (board[i][j] != -1), we move to the destination
- mark it visited. 8. Checking for Completion: If, during the BFS, we reach the last square n^2, we immediately return the number of moves made

7. Visiting and Queuing: If the next square (factoring in any snakes or ladders) has not been visited, we add it to the queue and

The BFS algorithm, coupled with the correct handling of the unique numbering and presence of snakes and ladders, ensures an optimized search for the quickest path to victory.

9. Return Case: If it is not possible to reach the end - which we know if the queue gets exhausted and the end has not been

Suppose we have a 3x3 board, which means n=3 and our board looks like this:

The -2 at position 2 means there is a ladder that takes the player directly from square 2 to square 3. The goal is to reach square 9 in

2. Begin the BFS loop.

board.

Python Solution

class Solution:

5

6

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

35

36

37

38

39

40

40

41

42

43

44

45

46

47

48

49

50

52

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

51 }

C++ Solution

2 public:

1 class Solution {

from collections import deque

while queue:

def snakesAndLadders(self, board: List[List[int]]) -> int:

def get_square_coordinates(square_number):

return board_size - 1 - row, col

board_size = len(board) # n by n board

queue = deque([1]) # Start from square 1

steps = 0 # Counter for number of moves

for _ in range(len(queue)):

return steps

visited = {1} # Keep track of visited squares

Process all squares at the current depth.

Win condition: reached the last square

next_square = board[i][j]

queue.append(next_square)

visited.add(next_square)

if next_square not in visited:

if current_square == board_size * board_size:

Check all possible next moves by dice roll (1-6)

i, j = get_square_coordinates(next_square)

If next square has not been visited, add it to the queue

If there's a ladder or snake, take it.

current_square = queue.popleft()

if board[i][j] != -1:

// Convert the square number to board coordinates (i, j)

// In even rows (from the top), the order is right to left

// Convert row to the actual row in the board from the bottom

int boardSize; // The size of the board is stored at the class level.

queue<int> positionsQueue; // Queue to hold positions to explore

vector<bool> visited(boardSize * boardSize + 1, false); // Visited squares

// The main function to play the game on the given board.

boardSize = board.size(); // Initial board size

int snakesAndLadders(vector<vector<int>>& board) {

positionsQueue.push(1); // Starting position

visited[1] = true; // Start square is visited

int moves = 0; // Counter for number of moves

int levelSize = positionsQueue.size();

for (int i = 0; i < levelSize; ++i) {</pre>

int target = nextPos;

if (!visited[target]) {

visited[target] = true;

positionsQueue.push(target);

int current = positionsQueue.front();

// Check if we've reached the last square

if (current == boardSize * boardSize) return moves;

// Explore each possibility from the current position

int row = position.first, col = position.second;

// If there is a ladder or snake, move to its end.

if (board[row][col] != -1) target = board[row][col];

// If the target square is not visited, add it to the queue.

if (!visited[target]) {

visited[target] = true;

return -1; // If we are here, there is no solution.

positionsQueue.push(target);

moves++; // Increment the move count after each level of BFS.

// If the target square is not visited, add it to queue

// BFS to find the shortest path

while (!positionsQueue.empty()) {

positionsQueue.pop();

private int[] convertToPosition(int squareNum) {

return new int[] {boardSize - 1 - row, col};

int row = (squareNum - 1) / boardSize;

int col = (squareNum - 1) % boardSize;

col = boardSize - 1 - col;

if (row % 2 == 1) {

Helper function to map the board number to board coordinates (i,j)

row, col = (square_number - 1) // board_size, (square_number - 1) % board_size

Walkthrough Steps

the minimum number of moves.

Dequeue 1, the current square.

Add 3 to the visited set and enqueue it.

3. Increment ans to 1 (we've made one move).

 The end square 9 cannot be reached in this turn. 5. Increment ans to 2 (we've made two moves).

1. Initialize the BFS queue with the starting square [1], add 1 to the visited set, and set ans (the move counter) to 0.

Explore all possible moves (squares 2 through 6, but since our board is 3x3, squares 4 through 6 do not exist).

4. Dequeue 3 and explore the next moves (4 through 9, but since we can reach at most 8 with a roll of six, we only consider up to 8).

None of the squares from 4 to 8 contain a snake or ladder, so we enqueue squares 4, 5, 6, 7, and 8 and mark them visited.

Square 2 has a ladder to square 3, so we move to 3 instead of stopping at 2.

style snake and ladder board is 3.

Hence, using this method, we determine that the minimum number of moves needed to reach the end of this 3x3 Boustrophedon

6. Dequeue 4, 5, 6, 7, and 8 in the next iterations. From any of these squares, a roll of one or more will reach the end square 9.

7. Once 9 is reached, return the current ans value, which now is 3 as the minimum number of moves taken to reach the end of the

if row % 2 == 1: 8 # On odd rows, the counting is from right to left. 9 10 col = board_size - 1 - col 11 # Transform row to start from bottom of board

for next_square in range(current_square + 1, min(current_square + 7, board_size * board_size + 1)):

31 32 33 34

```
41
                 # Increment the number of moves after expanding all possible moves at current depth.
 42
                 steps += 1
 43
             # If we have exited the loop without reaching the last square, it's not possible to win.
 44
 45
             return -1
 46
Java Solution
  1 class Solution {
         private int boardSize;
  3
         // Method to find the shortest path to reach the last square
  4
         public int snakesAndLadders(int[][] board) {
             boardSize = board.length; // Get the size of the board (n x n)
             Deque<Integer> queue = new ArrayDeque<>(); // A queue to perform BFS
             queue.offer(1); // Start from the first square
  8
             boolean[] visited = new boolean[boardSize * boardSize + 1]; // Visited array to keep track of visited squares
  9
 10
             visited[1] = true; // Mark the first square as visited
 11
             int moves = 0; // Moves counter
 12
 13
             // Perform BFS to reach the last square
             while (!queue.isEmpty()) {
 14
 15
                 for (int i = queue.size(); i > 0; --i) { // Iterate over current level
 16
                     int current = queue.poll(); // Get the current square
                     if (current == boardSize * boardSize) { // Check if reached the end
 17
 18
                         return moves;
 19
 20
                     // Explore the next 6 possible moves
 21
                     for (int k = current + 1; k <= Math.min(current + 6, boardSize * boardSize); ++k) {</pre>
 22
                         int[] position = convertToPosition(k); // Convert square number to board coordinates
 23
                         int next = k; // Next square
 24
                         // Check if there's a snake or ladder in the square
 25
                         if (board[position[0]][position[1]] != -1) {
                             next = board[position[0]][position[1]]; // Go to the new square
 26
 27
 28
                         // If it's not visited, mark as visited and add to the queue
                         if (!visited[next]) {
 29
 30
                             visited[next] = true;
 31
                             queue.offer(next);
 32
 33
 34
 35
                 moves++; // Increment move count after finishing one level in BFS
 36
 37
             return -1; // Return -1 if it's impossible to reach the end
 38
 39
```

37 38 39 40

```
36
                 ++moves; // Increment move count after each level of BFS.
             return -1; // If we are here, there is no solution.
 41
 42
 43
         // Converts a 1D array index to 2D grid indices considering snaking rows.
 44
         pair<int, int> convertTo2D(int pos) {
             int row = (pos - 1) / boardSize;
 45
 46
             int col = (pos - 1) % boardSize;
 47
             if (row % 2 == 1) { // For odd rows we reverse the column index.
 48
                 col = boardSize - 1 - col;
 49
             // Convert row from bottom to top because the last row of the board is board grid 1D index 1.
 50
             row = boardSize - 1 - row;
 51
             return {row, col};
 52
 53
 54 };
 55
Typescript Solution
  1 // The size of the board is stored as a global variable.
  2 let boardSize: number;
  4 // Converts a 1D array index to 2D grid indices considering snaking rows.
    function convertTo2D(pos: number): [number, number] {
         const row = Math.floor((pos - 1) / boardSize);
         let col = (pos - 1) % boardSize;
         if (row % 2 === 1) {
  8
             // For odd rows (according to zero-index), we reverse the column index.
  9
             col = boardSize - 1 - col;
 10
 11
 12
         // Convert row from bottom to top as the last row of the board corresponds to the first 1D index.
         const convertedRow = boardSize - 1 - row;
 13
 14
         return [convertedRow, col];
 15 }
 16
 17 // The main function to play the game on the given board.
    function snakesAndLadders(board: number[][]): number {
         boardSize = board.length; // Initialize board size
 19
 20
         const positionsQueue: number[] = []; // Queue to hold positions to explore
 21
         positionsQueue.push(1); // Starting position
 22
         const visited = new Array(boardSize * boardSize + 1).fill(false); // Visited squares
 23
         visited[1] = true; // Start square is visited
 24
         let moves = 0; // Counter for number of moves
 25
 26
        // BFS to find the shortest path.
 27
         while (positionsQueue.length > 0) {
 28
             const levelSize = positionsQueue.length;
 29
             for (let i = 0; i < levelSize; ++i) {</pre>
                 const current = positionsQueue.shift()!;
 30
 31
 32
                 // Check if we've reached the last square.
 33
                 if (current === boardSize * boardSize) return moves;
 34
 35
                 // Explore each possibility from the current position.
                 for (let nextPos = current + 1; nextPos <= Math.min(current + 6, boardSize * boardSize); ++nextPos) {</pre>
 36
                     const [row, col] = convertTo2D(nextPos); // Convert 1D position to 2D grid indices.
 37
 38
                     let target = nextPos;
 39
 40
                     // If there is a ladder or snake, move to its end.
                     if (board[row][col] !== -1) target = board[row][col];
 41
```

for (int nextPos = current + 1; nextPos <= min(current + 6, boardSize * boardSize); ++nextPos) {</pre>

auto position = convertTo2D(nextPos); // Convert 1D position to 2D grid indices

performed by the code. **Time Complexity**

Time and Space Complexity

43

44

45

46

47

48

49

50

51

52

53

54

once.

• The outer while loop runs as long as there are elements in the queue q. In the worst case, every cell could be enqueued once, which corresponds to the number of cells in the board, n * n. For each element in the queue, the inner for loop considers up to 6 possible moves (representing rolling a die).

The time complexity of the solution is determined by the number of vertices in the graph and the number of edges that are explored

in the BFS algorithm. Each cell in the board can be considered a vertex, and the possible moves from each cell form the edges.

The given Python code represents a solution for finding the minimum number of moves to reach the end of a snake and ladder board

game using a Breadth-First-Search (BFS) algorithm. To analyze the time and space complexity, I will break down the operations

The complexity for each layer of vertices explored is 0(6) due to the six possible moves (the die roll), resulting in a total time complexity of $0(n^2 * 6)$. Since constant factors are dropped in Big O notation, the overall time complexity simplifies to $0(n^2)$.

• The helper function get, which translates a linear board position to a matrix position, is constant time, 0(1), for each call.

- **Space Complexity** The space complexity of the solution is determined by the additional space required by data structures that store intermediate
- results or states during the execution of the algorithm. • The queue q can hold all the vertices in the worst case, so its space complexity is 0(n * n), since each cell could be visited

• The set vis also has a space complexity of O(n * n) as it may contain a unique entry for every cell in the worst-case scenario.

 The get function uses a constant amount of space. Considering both the queue and visited set, the space complexity is 0(n^2).

Therefore, the overall time complexity is $0(n^2)$, and overall space complexity is $0(n^2)$.