2596. Check Knight Tour Configuration

Medium Depth-First Search Breadth-First Search Array Matrix Simulation

### **Problem Description**

chessboard is represented by an integer matrix grid, with each integer being unique and ranging from 0 to n \* n - 1. The grid shows the order in which a knight has visited each cell on the chessboard, starting from the top-left cell (which should be 0). The knight is supposed to visit every cell exactly once.

The task is to verify if the given grid represents a valid knight's tour. In a knight's tour, the knight starts at the top-left corner and

In this problem, you are given a chessboard of size  $n \times n$ , where n represents both the height and width of the board. The

The task is to verify if the given grid represents a valid knight's tour. In a knight's tour, the knight starts at the top-left corner and makes moves in an "L" shape pattern: either moving two squares vertically and one square horizontally, or two squares horizontally and one square vertically. Every cell must be visited exactly once in the tour. The problem asks you to return true if

the grid is a valid tour, or false if it is not.

Intuition

To approach this solution, you can simulate the knight's path using the given grid and check if each move is valid. Begin by

## ensuring that the knight starts at the top-left corner, which would be the 0th position in the tour—any other starting position makes the grid invalid immediately.

Next, create an array pos that will store the coordinates (x, y) for each step the knight takes. It is necessary because the grid is a 2D array representing the tour sequence, and you need to derive the actual coordinates visited at each step.

Once you have the coordinates for the entire path the knight took, go through them in pairs and check if the difference between

two adjacent coordinates corresponds to a legal knight move. A legal knight move is only valid if the change in x (dx) and change in y (dy) match either (1, 2) or (2, 1). If at any point you find that the move does not match these conditions, the tour is

invalid, and you return false.

If all the moves are legal, once you complete the traversal of the path, you return true indicating the grid represents a valid knight's tour.

Solution Approach

The solution begins by creating a list pos that will record the coordinates (i, j) corresponding to each step of the knight's tour, with i being the row index and j being the column index in the grid. The grid holds integers that represent the move sequences, and the pos list will be used to store the actual (x, y) positions for these moves.

Next, a for loop iterates over each cell (i, j) of the grid to fill out the pos list where pos[grid[i][j]] will be the tuple (i,

### j), effectively mapping the move sequence to coordinates. If the knight did not start at the top-left corner (grid[0][0] should

invalid knight's tour.

**Example Walkthrough** 

3 20 12

24 1 14 21 7

13 22 19 2

16

These differences correspond to the movements made by the knight.

index increments, making the code more readable and less error-prone.

be 0), then the grid is immediately deemed invalid.

After the pos list is populated, another for loop iterates through pos in pairs using the pairwise function. For each pair of adjacent positions (x1, y1) and (x2, y2), we calculate the absolute differences dx as abs(x1 - x2) and dy as abs(y1 - y2).

We check if the move is a legal knight's move—a move that results in dx being 1 and dy being 2 or vice versa. The dx variable holds a Boolean representing whether the move is valid dx = 1 and dy = 2 or dx = 2 and dy = 1.

If at any point, a pair of positions does not form a legal knight's move, the function returns false because this would indicate an

the function returns true, which signifies that the provided grid corresponds to a valid configuration of the knight's tour.

This approach relies on the idea of transforming the problem into a simpler data structure (pos list) that can be more easily traversed and verified. The pairwise iterator is a pattern that helps in checking the adjacent elements without manually handling

Finally, if all adjacent pairs fulfill the criteria of a knight's valid move, the loop completes without finding any invalid moves, and

Let's consider a 5×5 chessboard and a sequence that we need to verify. Let's assume the given grid is as follows:

10 17 6 23
5 5 8 11 18

• ...
• grid[4][4] is 2, so pos[2] becomes (4, 4).

Our pos list now represents the sequence in which the knight moves on the chessboard.

The pairwise iteration helps us quickly check the moves one after the other:

from itertools import pairwise # Python 3.10 introduced pairwise function in itertools

""" Check if the numbers in the grid follow a knight's move pattern

# Initialize a list to hold the positions of the numbers in the grid

for (row1, col1), (row2, col2) in pairwise(positions):

# Calculate the absolute deltas between the positions

delta\_row, delta\_col = abs(row1 - row2), abs(col1 - col2)

positions[grid[row\_index][col\_index] - 1] = (row\_index, col\_index)

# Use pairwise from itertools to iterate over consecutive pairs of number positions

def check valid grid(self, grid: List[List[int]]) -> bool:

grid: A 2D list representing a square grid

According to our solution approach, we need to first map each move sequence number to its coordinates on the grid.

• For the start: pos[0] is (0, 0) and pos[1] is (2, 1). The move from (0, 0) to (2, 1) has differences dx = 2 and dy = 1, which is a legal knight's move.

1. To create the pos list, we iterate through the grid:

grid[0][0] is 0, so pos[0] becomes (0, 0).

grid[0][1] is 10, so pos[10] becomes (0, 1).

2. Next, we iterate through the pos list to verify if each move is valid.

knight. Thus, according to our algorithm, we return false.

Current Position = (0, 0), Next Position = (2, 1), Move = Legal
Current Position = (2, 1), Next Position = (4, 4), Move = Illegal, returns false.

need to check the entire grid.

tour.

**Python** 

class Solution:

from typing import List

Aras:

Solution Implementation

• We then look at pos[1] to pos[2]: moving from (2, 1) to (4, 4). The differences dx = 2 and dy = 3 do not match the legal moves of a

We immediately know that the sequence is not a valid knight's tour since the second move is not legal for a knight, so we don't

This example walkthrough effectively illustrates how our algorithm works to verify a knight's tour on the chessboard. By

converting move sequences into coordinates and verifying each move one by one, we can efficiently determine the validity of the

Returns:
A boolean value indicating whether the grid is valid under the condition
that each consecutive number must be a knight's move away from the previous.
"""

# Ensure the first element is 0 as required by the problem's conditions
if grid[0][0] != 0:

```
# Populate the positions list with coordinates of numbers from the grid
for row index in range(size):
    for col index in range(size):
        # Note that the numbers have to be decreased by 1 to become 0-indexed positions
```

return False

size = len(grid)

# Get the size of the grid

positions = [None] \* (size \* size)

```
# Check for valid knight's move: either 1 by 2 steps or 2 by 1 step
            is valid knight move = (delta_row == 1 and delta_col == 2) or (delta_row == 2 and delta_col == 1)
            if not is valid knight move:
                # If any pair of numbers is not separated by a knight's move, the grid is invalid
                return False
        # If all pairs of numbers are separated by a knight's move, the grid is valid
        return True
Java
class Solution {
    // Function to check if a given grid represents a valid grid for the given conditions
    public boolean checkValidGrid(int[][] grid) {
        // Check if the first element is 0 as required
        if (arid[0][0] != 0) {
            return false;
        // Calculate the size of the grid
        int gridSize = grid.length;
        // Create a position array to store positions of each number in the grid
        int[][] positions = new int[gridSize * gridSize][2];
        for (int row = 0; row < gridSize; ++row) {</pre>
            for (int col = 0; col < gridSize; ++col) {</pre>
                // Storing the current number's position
                positions[grid[row][col]] = new int[] {row, col};
        // Loop to check the validity of the grid based on the position of consecutive numbers
        for (int i = 1; i < gridSize * gridSize; ++i) {</pre>
            // Get the positions of the current and previous numbers
            int[] previousPosition = positions[i - 1];
            int[] currentPosition = positions[i];
            // Calculate the distance between the current number and the previous number
            int dx = Math.abs(previousPosition[0] - currentPosition[0]);
            int dy = Math.abs(previousPosition[1] - currentPosition[1]);
            // Check if the distance satisfies the condition for a knight's move in chess
            boolean is Valid Move = (dx == 1 \& dx dy == 2) || (dx == 2 \& dx dy == 1);
            if (!isValidMove) {
                // If the move is not valid, the grid is not valid
                return false;
        // If all moves are valid, the grid is valid
        return true;
```

C++

public:

#include <vector>

class Solution {

#include <cmath> // Include <cmath> for abs function

if (grid[0][0] != 0) {

return false;

// Size of the arid

int gridSize = grid.size();

// The Solution class containing a method to check if a grid is valid

bool checkValidGrid(std::vector<std::vector<int>>& grid) {

for (int row = 0; row < gridSize; ++row) {</pre>

for (int col = 0: col < gridSize: ++col) {</pre>

// Get the current and previous positions

int deltaX = std::abs(previousX - currentX);

def check valid grid(self, grid: List[List[int]]) -> bool:

grid: A 2D list representing a square grid

""" Check if the numbers in the grid follow a knight's move pattern

A boolean value indicating whether the grid is valid under the condition

# Ensure the first element is 0 as required by the problem's conditions

# Initialize a list to hold the positions of the numbers in the grid

# Populate the positions list with coordinates of numbers from the grid

positions[grid[row\_index][col\_index] - 1] = (row\_index, col\_index)

# Use pairwise from itertools to iterate over consecutive pairs of number positions

# If all pairs of numbers are separated by a knight's move, the grid is valid

# Note that the numbers have to be decreased by 1 to become 0-indexed positions

that each consecutive number must be a knight's move away from the previous.

// The start position must be 0, if not return false

// Vector to hold the positions of the numbers in the arid

numberPositions[grid[row][col]] = {row, col};

for (int number = 1; number < gridSize \* gridSize; ++number) {</pre>

auto [currentX, currentY] = numberPositions[number];

// Calculate the differences in x and y coordinates

auto [previousX, previousY] = numberPositions[number - 1];

// Method to check if the moves in the grid represent a valid knight's tour

std::vector<std::pair<int, int>> numberPositions(gridSize \* gridSize);

// Populate numberPositions with coordinates of each number in the grid

// Iterate over the numbers in the grid and check if each move is a valid knight move

```
int deltaY = std::abs(previousY - currentY);
            // Check if the move is a valid knight's move (L shape move)
            bool isValidKnightMove = (deltaX == 1 \& \& deltaY == 2) | (deltaX == 2 \& \& deltaY == 1);
            // If the move isn't valid, return false
            if (!isValidKnightMove) {
                return false;
        // If all moves are valid, the grid represents a valid knight's tour
        return true;
};
TypeScript
function checkValidGrid(grid: number[][]): boolean {
    // Ensure the first element is 0 as expected
    if (arid[0][0] !== 0) {
        return false;
    // Find the size of the grid
    const gridSize = grid.length;
    // Initialize an array to track the positions of the numbers in the grid
    const positions = Array.from({ length: gridSize * gridSize }, () => new Array(2).fill(0));
    // Populate positions array with the coordinates of each number in the grid
    for (let row = 0; row < gridSize; ++row) {</pre>
        for (let col = 0; col < gridSize; ++col) {</pre>
            positions[grid[row][col]] = [row, col];
    // Validate the positions of all numbers from 1 to n*n-1
    for (let i = 1; i < gridSize * gridSize; ++i) {</pre>
        // Get the position of the current and previous number
        const previousPos = positions[i - 1];
        const currentPos = positions[i];
        // Calculate the absolute differences in x and y directions
        const deltaX = Math.abs(previousPos[0] - currentPos[0]);
        const deltaY = Math.abs(previousPos[1] - currentPos[1]);
        // Check for knight-like move (L-shape): 2 by 1 or 1 by 2 steps
        const isValidMove = (deltaX === 1 && deltaY === 2) || (deltaX === 2 && deltaY === 1);
        // If the move is not valid, return false
        if (!isValidMove) {
            return false;
    // If all positions are valid, return true
    return true;
from typing import List
from itertools import pairwise # Python 3.10 introduced pairwise function in itertools
class Solution:
```

# # Check for valid knight's move: either 1 by 2 steps or 2 by 1 step is valid knight move = (delta\_row == 1 and delta\_col == 2) or (delta\_row == 2 and delta\_col == 1) if not is valid knight move: # If any pair of numbers is not separated by a knight's move, the grid is invalid return False

Time and Space Complexity

return True

Args:

Returns:

if grid[0][0] != 0:

size = len(grid)

return False

# Get the size of the grid

positions = [None] \* (size \* size)

for col index in range(size):

for (row1, col1), (row2, col2) in pairwise(positions):

# Calculate the absolute deltas between the positions

delta\_row, delta\_col = abs(row1 - row2), abs(col1 - col2)

for row index in range(size):

The time complexity of the code is  $O(n^2)$ . This is because the main computations are in the nested for-loops, which iterate over every cell in the given grid. Since the grid is n by n, the iterations run for  $n^2$  times.

The space complexity of the code is  $0(n^2)$  as well. This stems from creating a list called pos of size n \* n, which is used to store the positions of the integers from the grid in a 1D array. Since the grid stores  $n^2$  elements, and we're converting it to a 1D array, the space taken by pos will also be  $n^2$ .