1222. Queens That Can Attack the King

Medium Array Matrix Simulation

Leetcode Link

Problem Description

On an 8×8 chessboard, the game setup includes multiple black queens and one white king. The chessboard aligns with a 0-indexed grid, akin to a coordinate plane, where the intersection of each horizontal row and vertical column denotes a square on the chessboard.

We are provided with two inputs:

- A 2D integer array `queens`, where each element `queens[i] = [xQueen_i, yQueen_i]` represents the position of the i-th black queen on the chessboard.
- An integer array `king` of length 2, with `king = [xKing, yKing]` standing for the position of the white king on the chessboard.

The challenge lies in identifying all the black queens that are in a direct position to attack the white king. The queens can attack in straight lines horizontally, vertically, or diagonally. The positions of the black queens capable of launching an attack on the king must be returned and can be listed in any sequence.

Intuition

The essence of finding a solution resides in understanding how queens move and attack in chess. A queen attacks along rows, columns, and diagonals until it's obstructed by another piece or the edge of the chessboard.

So, the intuitive approach is to inspect the chessboard row by row, column by column, and along each diagonal emanating from the king's position. We want to locate the closest queen in each of these directions, as only those queens would be able to attack the king without being blocked by other queens.

The solution involves the following steps:

1. Transform the list of queen positions into a set for efficient presence checking.
2. Define the 8 possible directions in which the king can be attacked based on queen movements: vertical, horizontal, and the four diagonals.
3. Iterate over these directions, "walking" from the king's position step by step in one direction until an edge of the board is reached or a queen is found.
4. When a queen is located in one of these directions, we record her position as she can attack the king, then move on to the next direction.

By proceeding as such, we collect the positions of all queens that pose a direct threat to the king.

Solution Approach

The implementation of the solution leverages simple yet effective programming concepts to iterate through the grid and find the attacking queens. Here's a walk-through of the solution:

1. `n` is set to 8, which represents the size of the chessboard.
2. A set `s` is created to store the positions of the queens. This allows for O(1) look-up times when checking if a given position on the chessboard has a queen or not.
3. We define an empty list `ans` to eventually hold the positions of the queens that can attack the king.
4. The algorithm uses two nested loops, each iterating from -1 to 1, to check all 8 possible directions a queen can attack from (excluding the stationary position `(0, 0)`). These two numbers `(a, b)` represent the "step" in each direction — vertically, horizontally, and diagonally.
5. For each direction, we start at the king's position and "move" one step at a time in that direction by incrementing `x` by `a` and `y` by `b`. We continue this step-wise movement until we reach the edge of the board (`0 <= x + a < n and 0 <= y + b < n`).
6. During each step, we check if the new position `(x, y)` has a queen (`if (x, y) in s`). Once we find a queen, we add her position to the `ans` list using `ans.append([x, y])`. We then break out of the inner while loop to stop looking in that direction because it's not possible for another queen to attack the king from the same direction without being obstructed by the first one found.
7. After inspecting all 8 directions, we return the list `ans` which contains the positions of all queens that can directly attack the king.

By using sets for constant-time look-ups and only searching along each direction once, we ensure that the algorithm runs efficiently. It avoids redundant checks and follows the movement of queens in chess closely to guarantee that only the queen closest to the king in any given direction is considered a direct threat.

Example Walkthrough

Let's consider a small example based on the solution approach described:

Suppose the 2D board has only one queen located at `[3, 5]` and the white king is at `[4, 3]`. We want to use the algorithm to determine if this queen can actually attack the king.

1. The chessboard size `n` is 8 (0-indexed from 0 to 7).
2. The queen's position would be added to a set: `s = {(3, 5)}`.
3. We initiate an empty list `ans` to keep track of attacking queens.

Now, we look at all 8 possible directions one by one:

- Horizontally and Vertically (four directions): left, right, up, down
- Diagonally (four directions): upper-left, upper-right, lower-left, lower-right

For each direction, we start from the king's position `[4, 3]` and move in steps.

Let's pick two directions to illustrate:

- Direction (0, 1) - This represents checking to the right of the king.
- Direction (1, 1) - This represents checking diagonally down-right from the king.

Now, the step-by-step process is as follows:

1. Starting with the right direction, we move from `[4, 3]` to `[4, 4]`, `[4, 5]`, ... and we keep checking if there's a queen on these squares. At `[4, 5]` we exit the board, and since the queen is not in that direct path, we do not find a threat from that direction.
2. Checking diagonally down-right, we go from `[4, 3]` to `[5, 4]` to `[6, 5]` and see there is no queen at these positions (since the set `s` only contains `[3, 5]`, and this does not match our current position `[6, 5]`), and we continue until we either find a queen or exit the board boundaries. Since we neither find a queen nor exit the boundaries, we move no further.
3. Continuing this for all 8 directions, we eventually check the up-left diagonal direction (-1, -1). We would step from `[4, 3]` to `[3, 2]`, `[2, 1]`, `[1, 0]`, but we don't find a queen in this path using our set `s`.

As we can see from our direction checks, the single queen at `[3, 5]` is not in a position to attack the king at `[4, 3]` in any of the 8 directions investigated by our algorithm. Therefore, the `ans` list remains empty, indicating that the king is not under attack from the given position of the queen. At the end of the iterating through all possible directions, we would return the unchanged `ans` list, which signifies that no queens can attack the king in the given board setup.

Python Solution

```python
class Solution:
    def queensAttacktheKing(self, queens: List[List[int]], king: List[int]) -> List[List[int]]:
        # Size of the chess board
        board_size = 8
        # Set representation of queen positions for constant time accessibility
        queen_positions = {(i, j) for i, j in queens}
        # List to hold the final positions of queens that can attack the king
        attack_positions = []

        # Explore all 8 directions from the king's position:
        # These directions are combinations of moving vertically (a), horizontally (b),
        # and diagonally on the chess board.
        for delta_row in range(-1, 2):
            for delta_col in range(-1, 2):
                # Skip (0, 0) to prevent standing still
                if delta_row or delta_col:
                    current_row, current_col = king
                    # Move in the direction until hitting the edge of the board or finding a queen
                    while 0 <= current_row + delta_row < board_size \
                            and 0 <= current_col + delta_col < board_size:
                        # Update current position
                        current_row, current_col = current_row + delta_row, current_col + delta_col
                        # Check if current position is occupied by a queen
                        if (current_row, current_col) in queen_positions:
                            # If found, add the position to the answer list and stop searching this direction
                            attack_positions.append([current_row, current_col])
                            break
        # Return all the unique positions from which the queens can attack the king directly
        return attack_positions

```

Java Solution

```java
import java.util.ArrayList;
import java.util.List;

class Solution {
    // Function that returns a list of queens that can attack the king.
    public List<List<Integer>> queensAttacktheKing(int[][] queens, int[] king) {
        // Define the size of the chessboard
        final int SIZE = 8;

        // Board to track the positions of the queens
        boolean[][] board = new boolean[SIZE][SIZE];

        // Place queens on the board based on given positions
        for (int[] queen : queens) {
            board[queen[0]][queen[1]] = true;
        }

        // List to store positions of queens that can attack the king
        List<List<Integer>> attackPositions = new ArrayList<>();

        // Directions - The pair (a, b) represents all 8 possible directions from a cell
        //              (0, -1)  (-1, -1)  (-1, 0)
        //              (0, 1)             (1, 0)
        //             (1, 1)     (1, -1)
        for (int rowDir = -1; rowDir <= 1; rowDir++) {
            for (int colDir = -1; colDir <= 1; colDir++) {

                // Skip standstill direction as the king is not moving
                if (rowDir != 0 || colDir != 0) {
                    // Starting position for searching in a specific direction
                    int x = king[0] + rowDir, y = king[1] + colDir;

                    // Traverse in the direction until a queen is found or edge of board is reached
                    while (x >= 0 && x < SIZE && y >= 0 && y < SIZE) {
                        // Check if queen is found
                        if (board[x][y]) {
                            attackPositions.add(List.of(x, y));
                            break; // Break out of the loop as we're only looking for the closest queen
                        }
                        // Move to next position in the direction
                        x += rowDir;
                        y += colDir;
                    }
                }
            }
        }

        return attackPositions;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    vector<vector<int>> queensAttacktheKing(vector<vector<int>>& queens, vector<int>& king) {
        const int boardSize = 8; // The size of the chessboard
        bool squares[boardSize][boardSize]{ false }; // Initialize all squares to 'false'

        // Mark positions where queens are located on the chessboard
        for (const auto& queen : queens) {
            squares[queen[0]][queen[1]] = true;
        }

        vector<vector<int>> attackedQueens; // To hold the positions of queens that can attack the king
        // Directions that need to be checked: Horizontal, Vertical, Diagonal (8 directions)
        for (int deltaX = -1; deltaX <= 1; ++deltaX) {
            for (int deltaY = -1; deltaY <= 1; ++deltaY) {
                // Skip checking the direction where both deltaX and deltaY are zero (the king's position)
                if (deltaX == 0 && deltaY == 0) {
                    continue;
                }

                // Start from the king's position and check in the current direction
                int x = king[0] + deltaX, y = king[1] + deltaY;
                // Continue until out of bounds
                while (x >= 0 && x < boardSize && y >= 0 && y < boardSize) {
                    // If a queen is found on the current square
                    if (squares[x][y]) {
                        // Add the queen's position to the result vector and break out of the loop to check the next direction
                        attackedQueens.push_back({x, y});
                        break;
                    }
                    // Move to the next square in the current direction
                    x += deltaX;
                    y += deltaY;
                }
            }
        }

        return attackedQueens; // Return the list of queens that can attack the king
    }
};

```

Typescript Solution

```typescript
function queensAttacktheKing(queens: number[][], king: number[]): number[][] {
    const boardSize = 8; // Chessboard size is 8x8
    // Create a boolean matrix to record the positions of queens
    const board: boolean[][] = Array.from({ length: boardSize }, () => Array.from({ length: boardSize }, () => false));

    // Mark the positions of all queens on the board
    queens.forEach(([row, col]) => (board[row][col] = true));

    // Initialize an array to store queens that can attack the king
    const attackingQueens: number[][] = [];

    // Iterate over all possible directions from the king: 8 surrounding cells (horizontal, vertical, and diagonal)
    for (let rowDirection = -1; rowDirection <= 1; ++rowDirection) {
        for (let colDirection = -1; colDirection <= 1; ++colDirection) {
            // Skip checking the same cell where the king is positioned
            if (rowDirection || colDirection) {
                // Start from the king's position and check in the direction specified by rowDirection and colDirection
                let [currentRow, currentCol] = [king[0] + rowDirection, king[1] + colDirection];

                // Continue checking the cells within the board limits
                while (currentRow >= 0 && currentRow < boardSize && currentCol >= 0 && currentCol < boardSize) {
                    // If a queen is found in the current direction, add it to the attackingQueens array and break the loop
                    if (board[currentRow][currentCol]) {
                        attackingQueens.push([currentRow, currentCol]);
                        break;
                    }
                    // Move to the next cell in the current direction
                    currentRow += rowDirection;
                    currentCol += colDirection;
                }
            }
        }
    }
    // Return the array containing the positions of the attacking queens
    return attackingQueens;
}

```

Time and Space Complexity

The code performs a search in each of the 8 directions from the king's position until it either hits the end of the board or finds a queen. Let's analyze the time and space complexity:

Time Complexity

The time complexity is `O(n)`, where `n` is the number of cells in the board (in this case, `n` is 64, since it's an 8×8 chessboard). We iterate over each direction only once and in the worst case, we traverse the entire length of the board. However, since the board size is fixed, we can consider this time complexity to be `O(1)` in terms of the input size, because the board doesn't grow with the input.

Space Complexity

The space complexity is `O(q)`, where `q` is the number of queens, because we store the positions of the queens in a set `s`. The board size is fixed and does not influence the space complexity beyond the storage of queens. In the worst case where every cell contains a queen, space complexity would be `O(n)`, but since the board's size is constant and doesn't scale with the input, this can also be referred to as `O(1)` from a practical perspective.