999. Available Captures for Rook

Easy Arrays Matrix Simulation

Leetcode Link

Problem Description

The problem is set on an 8×8 chessboard with one white rook 'R', several white bishops 'B', black pawns 'p', and empty squares '.'. The rook can move in any of the four cardinal directions - north, east, south, or west. The move of the rook is only stopped by either reaching the board's edge, capturing a black pawn, or being blocked by a white bishop. The task is to calculate the number of black pawns that the rook can capture in its next move. A pawn is considered capturable or "under attack" if the rook can reach it without being obstructed by a bishop or the edge of the board. The desired output is the count of such pawns.

Intuition

To approach the problem efficiently, the solution first identifies the rook's position on the board. Once that's done, the solution needs to check each of the four cardinal directions from the rook's position to see if there's a pawn that can be captured.

This involves moving outward from the rook's position in the given direction, step by step, until hitting the edge of the board, a bishop, or a pawn. When encountering a pawn, it's important to increase the capture count and stop checking that direction as the rook would stop after a capture. Upon encountering a bishop, checking that direction should also stop as the bishop blocks the rook's path. By iterating through each direction, the solution checks all possible captures the rook can make.

The code uses a 'dirs' tuple which contains the changes in coordinates corresponding to the four cardinal directions - up, right, down, and left, given in pairs using the pairwise function. By looping through these direction pairs, the code checks each direction sequentially, applying the direction changes to the rook's position to move along the chessboard.

Solution Approach

The solution is straightforward and involves iterating through the chessboard to identify the rook's position and then, from there, examining all four cardinal directions to check for capturable pawns. Here's the breakdown of the implementation steps:

1. **Iteration to Find the Rook's Position:**
   - Start by iterating over each cell of the 8×8 chessboard using a nested loop.
   - Identify the rook's position by checking if the character in the current cell is "R".
2. **Directional Checks for Captures:**
   - Once the position of the rook is found, use a directional array `dirs = (-1, 0, 1, 0, -1)` that represents the possible moves in the north, east, south, and west directions sequentially.
   - The `pairwise` function (not built-in) is implied to create pairs of directions (`dx`, `dy`), obtaining the directions for the rook to check. Normally we would need to create these pairs manually or through iteration.
3. **Loop Through Directions:**
   - For each direction pair obtained from `dirs`, initialize coordinates (`x`, `y`) to the rook's position.
   - Move in the current direction until a pawn 'p', bishop 'B', or the edge of the board is encountered.
4. **Edge and Piece Encounters:**
   - Use a while loop to repeatedly apply the direction to the coordinates (`x`, `y`) until one of the following occurs:
     - The new coordinates are out of bounds (beyond the edges of an 8×8 board).
     - A "B" (bishop) is encountered, which blocks the rook's movement on that path.
     - A "p" (pawn) is encountered, in which case increment the capture count `ans` and stop checking that path since the pawn will be captured.
5. **Terminating Conditions and Incrementing Capture Count:**
   - If the cell under the current direction is a pawn, increment the answer (`ans`) as a pawn is capturable.
   - If the cell is a bishop, it acts as a blocking piece, and the rook cannot move further in that direction, so break out of the while loop and proceed to the next direction.
6. **Returning the Result:**
   - After all four directions are checked, the resulting capture count `ans` is returned, which is the total number of pawns the rook can capture according to the game rules outlined in the problem description.

Example Walkthrough

Let's consider a small example to illustrate the solution approach described above. Suppose we have the following 8×8 chessboard layout:

```
1 . . . . . . .
2 . . p . . . .
3 . B . . . . .
4 . . . R . . .
5 . . . . . . .
6 . . . . . . .
7 . . . . . . .
8 . . p . . . .
```

Here, the rook `R` is located at coordinates (3,3), using a 0-based index. We want to determine how many black pawns `p` the rook can capture in its next move without being obstructed by any white bishops `B`. Now, let's walk through the solution steps:

1. **Iteration to Find the Rook's Position:**
   - We iterate through the board and find the rook at coordinates (3,3).
2. **Directional Checks for Captures:**
   - We identify the direction array `dirs = (-1, 0, 1, 0, -1)` and use the implied `pairwise` technique to get direction pairs. The direction pairs would be `(-1, 0)`, `(0, 1)`, `(1, 0)`, and `(0, -1)` representing north, east, south, and west respectively.
3. **Loop Through Directions:**
   - We start iterating through these direction pairs applying them one by one.
4. **Edge and Piece Encounters:**
   - When going north (up), we encounter the edge of the board. No pawns or bishops block the path, but the rook can't move past the boundary; therefore, no pawns are capturable in this direction.
   - Moving east (right), the first piece the rook encounters is a bishop at (3,7). This blocks any further movement in that direction. No pawns are capturable.
   - Heading south (down), the rook encounters a pawn at (4,3). This pawn is capturable, and the movement in this direction stops with an increment in the capture count `ans`.
   - Finally, going west (left), the rook reaches the edge of the board without encountering any pawns or bishops, so no captures can be made.
5. **Terminating Conditions and Incrementing Capture Count:**
   - From these explorations, only one pawn at (4,3) is capturable. So we increment the answer `ans` by 1.
6. **Returning the Result:**
   - Having checked all four directions, we conclude that there's only one pawn the rook can capture. So the function returns `ans` which is 1.

Python Solution

```python
class Solution:
    def numRookCaptures(self, board: List[List[str]]) -> int:
        # Initialize the number of captures to 0
        captures = 0
        # This tuple represents the four directions: up, right, down, left
        directions = (-1, 0), (0, 1), (1, 0), (0, -1)

        # Locate the Rook on the board
        for i in range(8):
            for j in range(8):
                if board[i][j] == "R":
                    # Search for possible captures in all four directions
                    for d in directions:
                        x, y = i, j
                        # Keep moving in the current direction until hitting a boundary or a piece
                        while 0 <= x + d[0] < 8 and 0 <= y + d[1] < 8:
                            x += d[0]
                            y += d[1]
                            # If a pawn is encountered, capture it and break out of the loop
                            if board[x][y] == "p":
                                captures += 1
                                break
                            # If there is a bishop or any other piece, the Rook cannot move past it
                            if board[x][y] != ".":
                                break
        # Return the number of pawns the Rook can capture
        return captures
```

Java Solution

```java
class Solution {
    // This method calculates the number of pawns the rook can capture.
    public int numRookCaptures(char[][] board) {
        int captures = 0; // Store the number of captures by the rook
        // Directions array to help move up, right, down, left (clockwise)
        // It represents the 4 possible directions for the rook to move.
        int[] directions = {-1, 0, 1, 0, -1};

        // Loop through the board to find the rook's position
        for (int i = 0; i < 8; ++i) {
            for (int j = 0; j < 8; ++j) {
                // Check if we found the rook
                if (board[i][j] == 'R') {
                    // Check all four directions for captures
                    for (int k = 0; k < 4; ++k) {
                        // Start position for the rook
                        int x = i, y = j;
                        // x and y direction for the current search
                        int deltaX = directions[k], deltaY = directions[k + 1];

                        // Keep moving in the direction unless a boundary or a bishop is hit
                        while (x + deltaX >= 0 && x + deltaX < 8 &&
                               y + deltaY >= 0 && y + deltaY < 8 &&
                               board[x + deltaX][y + deltaY] != 'B') {
                            x += deltaX;
                            y += deltaY;

                            // Check if a pawn is captured by the rook
                            if (board[x][y] == 'p') {
                                captures++; // Increase the number of captures
                                break; // Move to next direction after capture
                            }
                        }
                    }
                    // No need to check the rest of the board after finding the rook
                    break;
                }
            }
        }

        return captures; // Return the number of pawns captured
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the number of pawns that a rook can capture
    int numRookCaptures(vector<vector<char>>& board) {
        // Initialize the answer to zero.
        int numCaptures = 0;
        // Define the direction vectors for the rook to move: up, right, down, left.
        int directions[5] = {-1, 0, 1, 0, -1};

        // Traverse the 8x8 board to find the rook's position.
        for (int row = 0; row < 8; ++row) {
            for (int col = 0; col < 8; ++col) {
                // When the rook is found ('R')...
                if (board[row][col] == 'R') {
                    // Check all four directions.
                    for (int k = 0; k < 4; ++k) {
                        // Start from the rook's position.
                        int x = row, y = col;
                        // Get the current direction's deltas.
                        int deltaX = directions[k], deltaY = directions[k + 1];
                        // Move in the current direction until hitting a boundary or a 'B' (bishop).
                        while (x + deltaX >= 0 && x + deltaX < 8 &&
                               y + deltaY >= 0 && y + deltaY < 8 &&
                               board[x + deltaX][y + deltaY] != 'B') {
                            // Advance to the next cell in the current direction.
                            x += deltaX;
                            y += deltaY;
                            // If a pawn ('p') is encountered, increment the capture count.
                            if (board[x][y] == 'p') {
                                ++numCaptures;
                                // Break out of this loop as a pawn has been captured in this direction.
                                break;
                            }
                        }
                    }
                    // Since the rook can only be in one position on the board,
                    // we can return the number of captures immediately.
                    return numCaptures;
                }
            }
        }
        // Return the number of pawns the rook can capture.
        return numCaptures;
    }
};
```

Typescript Solution

```typescript
// Define the board as a 2D array of characters.
type Board = char[][];

// Function to calculate the number of pawns that a rook can capture.
function numRookCaptures(board: Board): number {
    // Initialize the answer to zero.
    let numCaptures: number = 0;
    // Define the direction vectors for the rook to move: up, right, down, left.
    const directions: number[] = [-1, 0, 1, 0, -1];

    // Traverse the 8x8 board to find the rook's position.
    for (let row: number = 0; row < 8; ++row) {
        for (let col: number = 0; col < 8; ++col) {
            // When the rook is found ('R')...
            if (board[row][col] === 'R') {
                // Check all four directions.
                for (let k: number = 0; k < 4; ++k) {
                    // Start from the rook's position.
                    let x: number = row, y: number = col;
                    // Get the current direction's deltas.
                    const deltaX: number = directions[k], deltaY: number = directions[k + 1];
                    // Move in the current direction until hitting a boundary or a 'B' (bishop).
                    while (x + deltaX >= 0 && x + deltaX < 8 &&
                           y + deltaY >= 0 && y + deltaY < 8 &&
                           board[x + deltaX][y + deltaY] !== 'B') {
                        // Advance to the next cell in the current direction.
                        x += deltaX;
                        y += deltaY;
                        // If a pawn ('p') is encountered, increment the capture count.
                        if (board[x][y] === 'p') {
                            numCaptures++;
                            // Break out of this loop as a pawn has been captured in this direction.
                            break;
                        }
                    }
                }
                // Since the rook can only be in one position on the board,
                // return the number of captures immediately.
                return numCaptures;
            }
        }
    }
    // Return the number of pawns the rook can capture.
    return numCaptures;
}
```

Time and Space Complexity

Time Complexity

The time complexity for this code is $O(n^2)$, where n is the dimension of the chessboard; since the board is 8×8, the time complexity can also be considered $O(1)$ as the size of the board is constant and does not scale with input size. The algorithm goes through every cell of the chessboard in the worst case (`for` iterations), and in each cell where the rook is found, it iterates in four directions until it encounters a bishop, pawn, or the edge of the board. The maximum distance the inner while-loop can cover in any direction is 7 cells meaning $O(n)$. But, since `n=8` is a constant, iterating through the entire direction in a fixed-size board doesn't depend on input size and thus contributes a constant factor.

Space Complexity

The space complexity of the code is $O(1)$ because the space used does not scale with the size of the input; the `dirs` tuple and the few variables used for iteration are all that is needed. The space requirements remain constant regardless of the size of the board or configuration of pieces on the chessboard.