🎉 Master algorithms together on Binary Search! Create a room, invite your friends, and race to finish the problems.

Daily Coding Problem #267

Problem

This problem was asked by Oracle.

You are presented with an 8 by 8 matrix representing the positions of pieces on a chess board. The only pieces on the board are the black king and various white pieces. Given this matrix, determine whether the king is in check.

For details on how each piece moves, see here.

For example, given the following matrix:

```
...K....
........
.B......
......P.
.......R
..N.....
........
.....Q..
```

You should return `True`, since the bishop is attacking the king diagonally.

Solution

Solution

In chess, the king can be attacked horizontally or vertically (by the rook), diagonally (by the bishop or queen), or in an `L`-shape (by the knight). The pawn also attacks diagonally, but only from a single square away.

Therefore, our solution can be to identify the possible attacking squares along these dimensions, and check whether a relevant piece is occupying any of them. If so, the king is indeed in check, and we can return `True`.

Here is the skeleton of our code.

```
def check(board):
    location = find_king(board)

    pawns = get_pawn_moves(board, location)
    crosses = get_cross_moves(board, location)
    diagonals = get_diagonal_moves(board, location)
    knights = get_knight_moves(board, location)

    if 'P' in pawns or \
       'B' in diagonals or 'Q' in diagonals or \
       'R' in crosses or 'Q' in crosses or \
       'N' in knights:
        return True

    return False
```

To find the king, we can simply search through the matrix until we find the `'K'` symbol.

```
def find_king(board):
    for row in range(8):
        for col in range(8):
            if board[row][col] == 'K':
                return row, col
```

A white pawn can attack the king from one of two squares: down and to the left, and down and to the right. In matrix representation, we are printing the board from top to bottom, so we must take care to increment the row count instead of decrementing.

```
def get_pawn_moves(board, row, col):
```

```
def get_pawn_moves(board, row, col):
    if row < 6:
        return [board[row + 1][col - 1], board[row + 1][col + 1]]
    else:
        return []
```

To obtain the horizontal and vertical attacking squares, it is not sufficient to return the entire row and column corresponding to the king's location. For if there is a third piece in between some piece and the king, the line of attack is cut off.

Instead, starting with the king's location, we can move outward to the left, to the right, to the bottom, and to the top, and take the first piece we find in each direction. This will return a maximum of four pieces.

```
def get_cross_moves(board, row, col):
    pieces = []

    for c in range(col - 1, 0, -1):
        if board[row][c] != '.':
            pieces.append(board[row][c])
            break
    for c in range(col + 1, 8, 1):
        if board[row][c] != '.':
            pieces.append(board[row][c])
            break
    for r in range(row + 1, 8, 1):
        if board[row][c] != '.':
            pieces.append(board[r][col])
            break

    for r in range(row - 1, 0, -1):
        if board[row][c] != '.':
            pieces.append(board[r][col])
            break

    return pieces
```

The process is similar for finding diagonal attacking squares. To generate the locations we should check for a given diagonal, we can `zip` together a range of rows and columns between the starting location and the edge of the board.

For example, if the king is sitting on `(3, 4)`, and we want to traverse the northeast diagonal, we would `zip` together `[4, 5, 6, 7]` and `[5, 6, 7]` to get `[(4, 5), (5, 6), (6, 7)]`.

```
def diagonal_moves(board, row, col):
    pieces = []

    for r, c in zip(range(row + 1, 8, 1), range(col + 1, 8, 1)):
        if board[r][c] != '.':
            pieces.append(board[r][c])
            break
    for r, c in zip(range(row + 1, 8, 1), range(col - 1, -1, -1)):
        if board[r][c] != '.':
            pieces.append(board[r][c])
            break
    for r, c in zip(range(row - 1, -1, -1), range(col + 1, 8, 1)):
        if board[r][c] != '.':
            pieces.append(board[r][c])
            break
    for r, c in zip(range(row - 1, -1, -1), range(col - 1, -1, -1)):
        if board[r][c] != '.':
            pieces.append(board[r][c])
            break

    return pieces
```

Finally, to obtain the squares that are one knight move away, we can enumerate all the possible ways of moving away from our start square, and use these differences to find each

destination. We should only return destinations that remain on the board, so we will use a helper function that removes invalid locations.

```
def is_legal(row, col):
    return 0 <= row <= 7 and 0 <= col <= 7

def knight_moves(board, row, col):
    moves = [(-2, -1), (-2, +1), (+2, -1), (+2, +1), (-1, -2), (-1, +2), (+1, -2), (+1, +2)]
    destinations = [(row + i, col + j) for (i, j) in moves]
    return [board[row][col] for row, col in destinations if is_legal(row, col)]
```

© Daily Coding Problem 2019

Privacy Policy

Terms of Service

Press