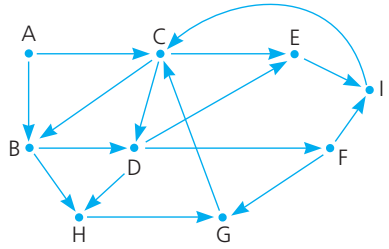
```
        while (!done && (nextCity != NO_CITY))
        {
            done = isPath(nextCity, destinationCity);
            if (!done)
                nextCity = getNextCity(originCity);
        }  // end while

        result = done;
    }  // end if

    return result;
}  // end isPath
```

Note: The subproblems that a recursive solution generates eventually must reach a base case. Failure to do so could result in an algorithm that does not terminate. Solutions that involve backtracking are particularly subject to this kind of error.

Question 6 Trace the method `isPath` with the map in Figure 5-6 for the following requests. Show the recursive calls and the returns from each.

- Fly from *A* to *B*.
- Fly from *A* to *D*.
- Fly from *C* to *G*.

FIGURE 5-6 Flight map for Checkpoint Question 6



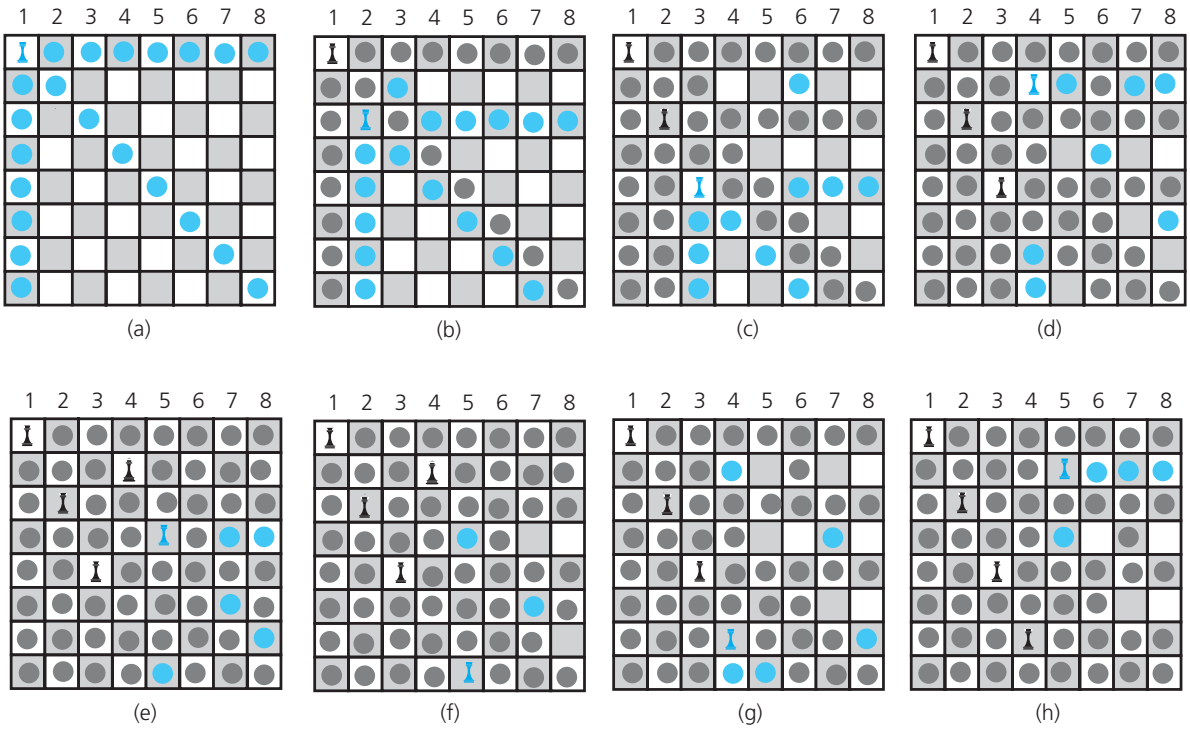
5.3.2 The Eight Queens Problem Start Here

Place eight queens on the chessboard so that no queen can attack any other queen

A chessboard contains 64 squares that form eight rows and eight columns. The most powerful piece in the game of chess is the queen, because it can attack any other piece within its row, within its column, or along its diagonal. The Eight Queens problem asks you to place eight queens on the chessboard so that no queen can attack any other queen.

One strategy is to guess at a solution. However, according to Section 2.6.3 of Chapter 2, there are $g(64, 8) = 4{,}426{,}165{,}368$ ways to arrange eight queens on a chessboard of 64 squares—so many that it would be exhausting to check all of them for a solution to this problem. Nevertheless, a simple observation eliminates many arrangements from consideration: No queen can reside in a row or a column that contains another queen. Alternatively, each row and column can contain exactly one queen. Thus, attacks along rows or columns are eliminated, leaving only $8! = 40{,}320$ arrangements of queens to be checked for attacks along diagonals. A solution now appears more feasible.

FIGURE 5-7 Placing one queen at a time in each column, and the placed queens' range of attack: (a) the first queen in column 1; (b) the second queen in column 2; (c) the third queen in column 3; (d) the fourth queen in column 4; (e) the five queens can attack all of column 6; (f) backtracking to column 5 to try another square for the queen; (g) backtracking to column 4 to try another square for the queen; (h) considering column 5 again



Place queens one column at a time

Suppose that you provide some organization for the guessing strategy by placing one queen per column, beginning with the first square of column 1. Figure 5-7a shows this queen and its range of attack. When you consider column 2, you eliminate its first square because row 1 contains a queen, you eliminate its second square because of a diagonal attack, and you finally place a queen in the third square of column 2, as Figure 5-7b illustrates. The black dots in the figure indicate squares that are rejected because a queen in that square is subject to attack by another queen in an earlier column. The blue dots indicate the additional squares that the new queen can attack.

If you reach an impasse, backtrack to the previous column

We continue to place queens in this manner until we get to column 6, as Figure 5-7e shows. Although the five placed queens cannot attack each other, they can attack any square in column 6, and therefore, you cannot place a queen in column 6. You must back up to column 5 and move its queen to the next possible square in column 5, which is in the last row, as Figure 5-7f indicates. When you consider column 6 once again, there are still no choices for a queen in that column. Because you have exhausted the possibilities in column 5, you must back up to column 4. As Figure 5-7g shows, the next possible square in column 4 is in row 7. You then consider column 5 again and place a queen in row 2 (Figure 5-7h).

How can you use recursion in the process that was just described? Consider an algorithm that places a queen in a column, given that you have placed queens correctly in the preceding columns.

First, if there are no more columns to consider, you are finished; this is the base case. Otherwise, after you successfully place a queen in the current column, you need to consider the next column. That is, you need to solve the same problem with one fewer column; this is the recursive step. Thus, you begin with eight columns, consider smaller problems that decrease in size by one column at each recursive step, and reach the base case when you have a problem with no columns.

This solution appears to satisfy the criteria for a recursive solution. However, you do not know whether you can successfully place a queen in the current column. If you can, you recursively consider the next column. If you cannot place a queen in the current column, you need to backtrack, as has already been described. Stop Here

The Eight Queens problem can be solved in a variety of ways. The solution in this chapter uses two classes: a `Board` class to represent the chessboard and a `Queen` class to represent a queen on the board. A `Queen` object keeps track of its row and column placement and contains a static pointer to the `Board`. It also has operations to move to the next row and to see whether it is subject to attack. A `Board` object keeps track of the `Queen` objects currently on the board and contains operations—such as `placeQueens`—to perform the Eight Queens problem and display the solution.

The following pseudocode describes the algorithm for placing queens in columns, given that the previous columns contain queens that cannot attack one another:

The solution combines recursion with backtracking

```
// Places queens in eight columns.
placeQueens(queen: Queen): void

    if (queen's column is greater than the last column)
        The problem is solved
    else
    {
        while (unconsidered squares exist in queen's column and
                the problem is unsolved)
        {
            Find the next square in queen's column that is
              not under attack by a queen in an earlier column
            if (such a square exists)
            {
                Place a queen in the square

                // Try next column
                placeQueens(new Queen(firstRow, queen's column + 1))
                if (no queen is possible in the next column)
                {
                    Delete the new queen
                    Remove the last queen placed on the board and
                      consider the next square in that column
                }
            }
        }
    }
```

The Eight Queens problem is initiated by the method `doEightQueens`, which calls `placeQueens` with a new queen in the upper-left corner of the board:

```
doEightQueens()
{
    placeQueens(new Queen(firstRow, firstColumn))
}
```

After `doEightQueens` has completed, the board may display the solution, if one was found.