approximated to the nearest integer. This leads us to the third implementation for computing a Fibonacci number. To round the result to the nearest integer, we use the function `ceil` (for ceiling):

```
int
deMoivreFib (int n)
{
 return ceil(exp(n*log(1.6180339897) - log(2.2360679775)) - .5);
}
```

Try to justify this implementation using the definition of logarithm.

5.9 BACKTRACKING

In solving some problems, a situation arises where there are different ways leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ascertain that such a return is possible and that all paths can be tried. This technique is called *backtracking* and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. Using backtracking, we can always return to a position which offers other possibilities for successfully solving the problem. This technique is used in artificial intelligence, and one of the problems in which backtracking is very useful is the eight queens problem.

The eight queens problem attempts to place eight queens on a chessboard in such a way that no queen is attacking any other. The rules of chess say that a queen can take another piece if it lies on the same row, on the same column, or on the same diagonal as the queen (see Figure 5.9). To solve this problem, we try to put the first queen on the board,

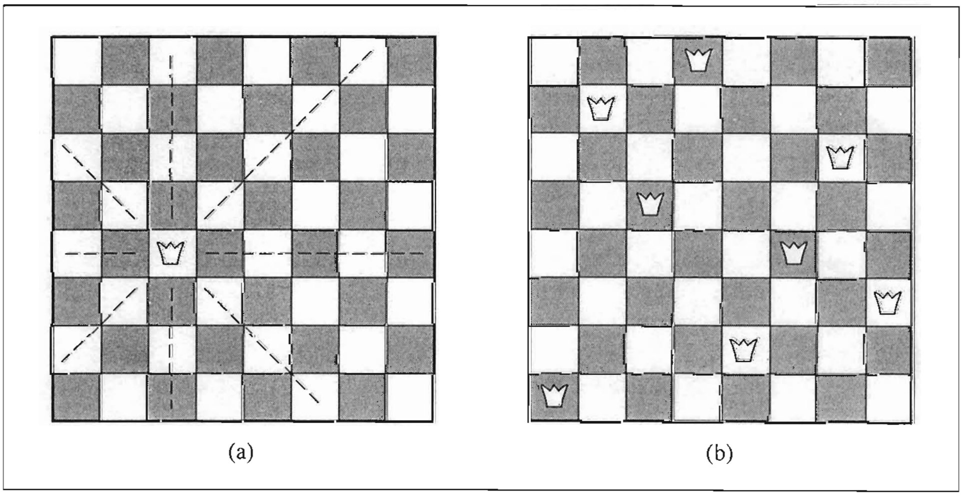

Figure 5.9: **The eight queens problem.**

then the second, so that it cannot take the first, then the third, so that it is not in conflict with the two already placed, and so on, until all of the queens are placed. What happens if, for instance, the sixth queen cannot be placed in a nonconflicting position? We choose another position for the fifth queen and try again with the sixth. If this does not work, the fifth queen is moved again. If all the possible positions for the fifth queen have been tried, the fourth queen is moved and then the process restarts. This process requires a great deal of effort, most of which is spent backtracking to the first crossroads offering some untried avenues. In terms of code, however, the process is rather simple due to the power of recursion which is a natural implementation of backtracking. Pseudocode for this backtracking algorithm is as follows (the last line pertains to backtracking):

```
PutQueen(row)
    for every position col on the same row
       if position col is available
          place the next queen in position col;
          if (row < 8)
             PutQueen(row+1);
          else success;
          remove the queen from position col;
```

This algorithm finds all possible solutions without regard to the fact that some of them are symmetrical.

The most natural approach for implementing this algorithm is to declare an 8×8 array **board** of 1s and 0s representing a chessboard. The array is initialized to 1s and each time a queen is put in a position (r, c), `board[r,c]` is set to 0. Also, a function must set to 0, as not available, all positions on row r, in column c, and on both diagonals that cross each other in position (r, c). When backtracking, the same positions, that is, positions on corresponding row, column and diagonals have to be set back to 1, as again available. Since we can expect hundreds of attempts to find available positions for queens, the setting and resetting process is the most time-consuming part of the implementation; for each queen, between 22 and 28 positions would have to be set and then reset, 15 for row and column, and between 7 and 13 for diagonals.

In this approach, the board is viewed from the perspective of the player who sees the entire board along with all the pieces at the same time. However, if we focus solely on the queens, we can consider the chessboard from their perspective. For the queens, the board is not divided into squares, but into rows, columns, and diagonals. If a queen is placed on a single square, it resides not only on this field square, but on the entire row, column, and diagonal, treating them as its own temporary property. A different data structure can be utilized to represent this.

To simplify the problem for the first solution, we use a 4×4 chessboard instead of the regular 8×8 board. Later, we can make the rather obvious changes in the program to accommodate it to the regular board.

Figure 5.10 contains the 4×4 chessboard. Notice that indexes in all fields in the indicated left diagonal all add up to three, $r + c = 3$; this number is associated with this diagonal. There are seven left diagonals, 0 through 6. Indexes in the fields of the indicated right diagonal all have the same difference, $r - c = -1$, and this number is unique among all right diagonals. Therefore, right diagonals are assigned numbers -3 through 3. The