timings On one computer, this algorithm requires 930 milliseconds to generate the 40,320 permutations of 8 objects, whereas the linked-list algorithm accomplishes the task in 660 milliseconds, an improvement of about 30 percent. With other implementations these numbers will differ, of course, but it is safe to conclude that the linked-list algorithm is at least comparable in efficiency. The correctness of the linked-list method, moreover, is obvious, whereas a proof that this other method actually generates all $n!$ distinct permutations of n objects is much more involved.

8.2.2 Backtracking: Nonattacking Queens

For our second example of an algorithm where recursion allows the postponement of all but one case, let us consider the puzzle of how to place eight queens on a chessboard so that no queen can take another. Recall that a queen can take another piece that lies on the same row, the same column, or the same diagonal (either direction) as the queen. The chessboard has eight rows and columns.

It is by no means obvious how to solve this puzzle, and its complete solution defied even the great C. F. GAUSS, who attempted it in 1850. It is typical of puzzles that do not seem amenable to analytic solutions, but require either luck coupled with trial and error, or else much exhaustive (and exhausting) computation. To convince you that solutions to this problem really do exist, two of them are shown in Figure 8.5.

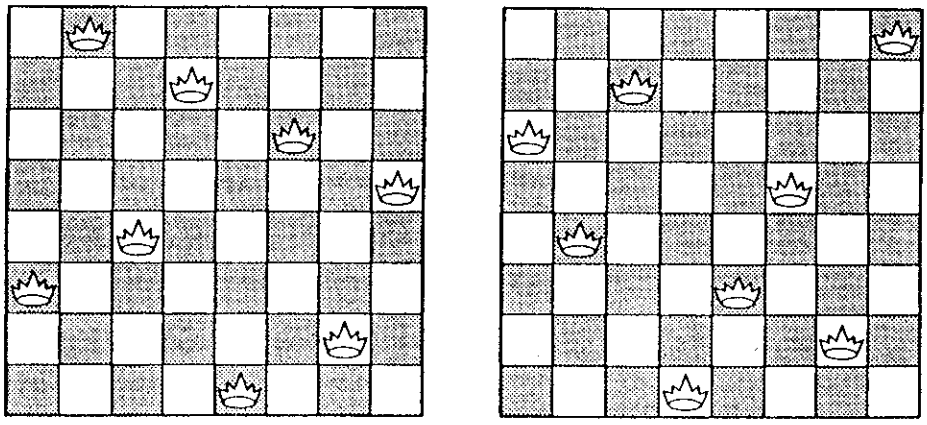


Figure 8.5. Two configurations showing eight nonattacking queens

1. Solving the Puzzle

A person attempting to solve the Eight Queens problem will usually soon abandon attempts to find all (or even one) of the solutions by being clever and will start to put queens on the board, perhaps randomly or perhaps in some logical order, but always making sure that no queen placed can take another already on the board. If the person is lucky enough to get eight queens on the board by proceeding in this way, then he has found a solution; if not, then one or more of the queens must be removed and placed

elsewhere to continue the search for a solution. To start formulating a program let us sketch this method in algorithmic form. We denote by n the number of queens on the board; initially $n = 0$. The key step is described as follows.

outline

```
void AddQueen(void)
{
   for (every unguarded position p on the board) {
      Place a queen in position p;
      n++;
      if (n == 8)
         Print the configuration;
      else
         AddQueen();
      Remove the queen from position p;
      n--;
   }
}
```

This sketch illustrates the use of recursion to mean "Continue to the next stage and repeat the task." Placing a queen in position p is only tentative; we leave it there only if we can continue adding queens until we have eight. Whether we reach eight or not, the function will return when it finds that it has finished or there are no further possibilities to investigate. After the inner call has returned, then, it is time to remove the queen from position p, because all possibilities with it there have been investigated.

2. Backtracking

This function is typical of a broad class called ***backtracking algorithms*** that attempt to complete a search for a solution to a problem by constructing partial solutions, always ensuring that the partial solutions remain consistent with the requirements of the problem. The algorithm then attempts to extend a partial solution toward completion, but when an inconsistency with the requirements of the problem occurs, the algorithm backs up (***backtracks***) by removing the most recently constructed part of the solution and trying another possibility.

Backtracking proves useful in situations where many possibilities may first appear, but few survive further tests. In scheduling problems, for example, it will likely be easy to assign the first few matches, but as further matches are made, the constraints drastically reduce the number of possibilities. Or consider the problem of designing a compiler. In some languages it is impossible to determine the meaning of a statement until almost all of it has been read. Consider, for example, the pair of FORTRAN statements

```
DO 17 K = 1, 6
DO 17 K = 1. 6
```

Both of these are legal: the first initiates a loop, and the second assigns the number 1.6 to

the variable DO17K. In such cases where the meaning cannot be deduced immediately, backtracking is a useful method in *parsing* (that is, splitting apart to decipher) the text of a program.

parsing

3. Refinement: Choosing the Data Structures

To fill in the details of our algorithm for the Eight Queens problem, we must first decide how we will determine which positions are unguarded at each stage and how we will loop through the unguarded positions. This amounts to reaching some decisions about the representation of data in the program.

A person working on the Eight Queens puzzle with an actual chessboard will probably proceed to put queens into the squares one at a time. We can do the same in a computer by introducing an 8×8 array with Boolean entries and by defining an entry to be true if a queen is there and false if not. To determine if a position is guarded, the person would scan the board to see if a queen is guarding the position, and we could do the same, but doing so would involve considerable searching.

square Boolean array

A person working the puzzle on paper or on a blackboard often observes that when a queen is put on the board, time will be saved in the next stage if all the squares that the new queen guards are marked off, so that it is only necessary to look for an unmarked square to find an unguarded position for the next queen. Again, we could do the same by defining each entry of our array to be true if it is free and false if it is guarded. A problem now arises, however, when we wish to remove a queen. We should not necessarily change a position that she has guarded from false to true, since it may well be that some other queen still guards that position. We can solve this problem by making the entries of our array integers rather than Boolean, each entry denoting the number of queens guarding the position. Thus to add a queen we increase the count by 1 for each position on the same row, column, or diagonal as the queen, and to remove a queen we reduce the appropriate counts by 1. A position is unguarded if and only if it has a count of 0.

square integer array

In spite of its obvious advantages over the previous attempt, this method still involves some searching to find unguarded positions and some calculation to change all the counts at each stage. The algorithm will be adding and removing queens a great many times, so that this calculation and searching may prove expensive. A person working on this puzzle soon makes another observation that saves even more work.

Once a queen has been put in the first row, no person would waste time searching to find a place to put another queen in the same row, since the row is fully guarded by the first queen. There can never be more than one queen in each row. But our goal is to put eight queens on the board, and there are only eight rows. It follows that there must be a queen, exactly one queen, in every one of the rows. (This is called the ***pigeonhole principle***: If you have n pigeons and n pigeonholes, and no more than one pigeon ever goes in the same hole, then there must be a pigeon in every hole.)

pigeonhole principle

Thus we can proceed by placing the queens on the board one row at a time, starting with the first row, and we can keep track of where they are with a single array

array of locations

```
int col[8];
```

where col[i] gives the column containing the queen in row i. To make sure that no

two queens are on the same column or the same diagonal, we need not keep and search through an 8×8 array, but we need only keep track of whether each column is free or guarded, and whether each diagonal is likewise. We can do this with three Boolean arrays, colfree, upfree, and downfree, where diagonals from the lower left to the upper right are considered upward and those from the upper left to lower right are considered downward.

guards

How do we identify the positions along a single diagonal? Along the main (downward) diagonal the entries are

$$[0,0],\ [1,1],\ \ldots,\ [7,7];$$

which have the property that the row and column indices are equal; that, is, their difference is 0. It turns out that along any downward diagonal the row and column indices will have a constant difference. This difference is 0 for the main diagonal, and ranges from $0 - 7 = -7$ for the downward diagonal of length 1 in the upper right corner, to $7 - 0 = 7$ for the one in the lower left corner. Similarly, along upward diagonals the sum of the row and column indices is constant, ranging from $0 + 0 = 0$ to $7 + 7 = 14$.

After making all these decisions, we can now define all our data structures formally, and, at the same time, we can write the main program.

```
int col[8];                          /* column with the queen                */
Boolean_type colfree[8];             /* Is the column free?                  */
Boolean_type upfree[15];             /* Is the upward diagonal free?         */
Boolean_type downfree[15];           /* Is the downward diagonal free?       */

int row = -1;                        /* row whose queen is currently placed  */
int sol = 0;                         /* number of solutions found            */
```

main program

```
/* Solve the Eight Queens problem. */
void main(void)
{
    int i;

    for (i = 0; i < 8; i++)
        colfree[i] = TRUE;
    for (i = 0; i < 15; i++) {
        upfree[i] = TRUE;
        downfree[i] = TRUE;
    }
    AddQueen();
}
```

Translation of the sketch of the function AddQueen into a program is straightforward, given the use of the arrays that have now been defined.

recursive function

```
/* AddQueen: attempts to place queens, backtracking when needed */
void AddQueen(void)
{
    int c;                                  /* column being tried for the queen      */
    row++;
    for (c = 0; c < 8; c++)
        if (colfree[c] && upfree[row + c] && downfree[row - c + 7]) {
            col[row] = c;                   /* Put a queen in (row, c).              */
            colfree[c] = FALSE;
            upfree[row + c] = FALSE;
            downfree[row - c + 7] = FALSE;
            if (row == 7)                   /* termination condition                 */
                WriteBoard();
            else
                AddQueen();                 /* Proceed recursively.                  */
            colfree[c] = TRUE;              /* Now backtrack by removing the queen.  */
            upfree[row + c] = TRUE;
            downfree[row - c + 7] = TRUE;
        }
    row--;
}
```

4. Local and Global Variables

Note that in the Eight Queens program almost all the variables and arrays are declared globally, whereas in the Towers of Hanoi program the variables were declared in the recursive function. If variables are declared within a function, then they are local to the function and not available outside it. In particular, variables declared in a recursive function are local to a single occurrence of the function, so that if the function is called again recursively, the variables are new and different, and the original variables will be remembered after the function returns. The copies of variables set up in an outer call are not available to the function during an inner recursive call. In the Eight Queens program we wish the same information about guarded rows, columns, and diagonals to be available to all the recursive occurrences of the function, and to do this, the appropriate arrays are declared not in the function but in the main program. The only reason for the array col[ ] is to communicate the positions of the queens to the function WriteBoard. The information in this array is also preserved in the eight local copies of the variable c set up during the recursive calls, but only one of these local copies is available to the program at a given time.

5. Analysis of Backtracking

Finally, let us estimate the amount of work that our program will do. If we had taken the naive approach by writing a program that first placed all eight queens on the board and

then rejected the illegal configurations, we would be investigating as many configurations as choosing eight places out of sixty-four, which is

$$\binom{64}{8} = 4{,}426{,}165{,}368.$$

The observation that there can be only one queen in each row immediately cuts this number to

$$8^8 = 16{,}777{,}216.$$

This number is still large, but our program will not investigate nearly this many positions. Instead, it rejects positions whose column or diagonals are guarded. The requirement that there be only one queen in each column reduces the number to

reduced count

$$8! = 40{,}320$$

which is quite manageable by computer, and the actual number of cases the program considers will be much less than this (see projects), since positions with guarded diagonals in the early rows will be rejected immediately, with no need to make the fruitless attempt to fill the later rows.

effectiveness of backtracking

This behavior summarizes the effectiveness of backtracking: positions that are early discovered to be impossible prevent the later investigation of many fruitless paths.

Another way to express this behavior of backtracking is to consider the tree of recursive calls to function AddQueen, part of which is shown in Figure 8.6. It appears formally that each vertex might have up to eight children corresponding to the recursive

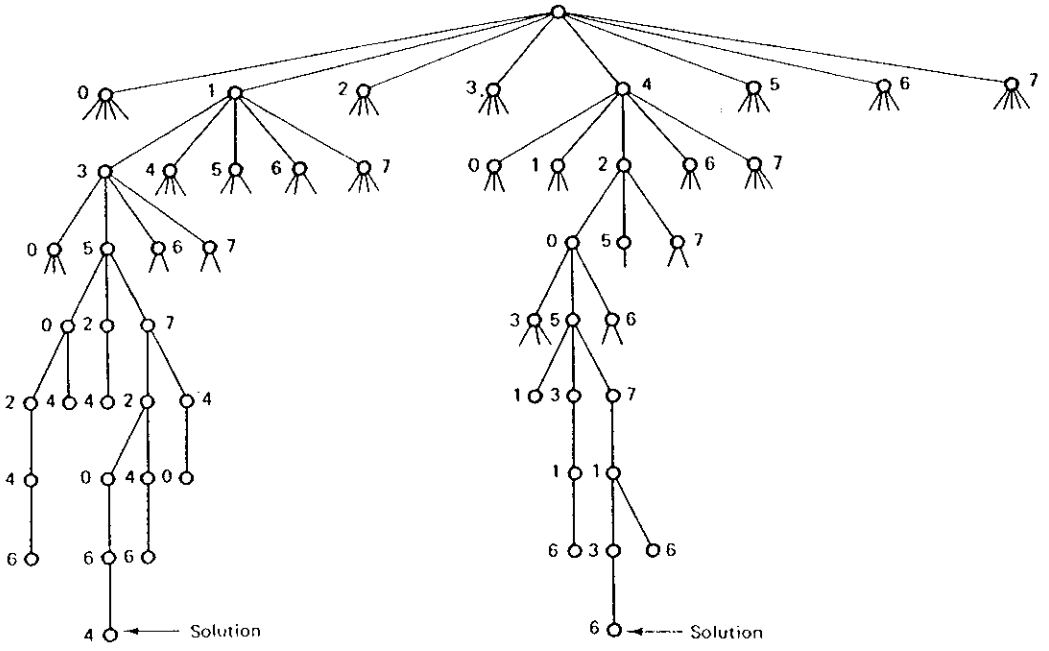


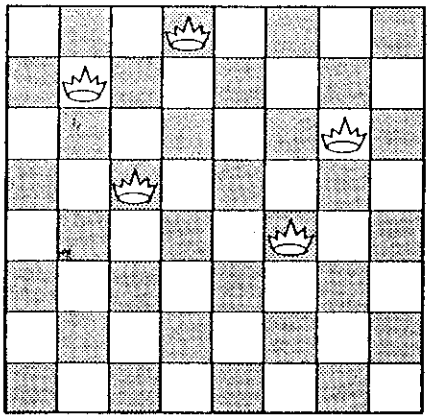
Figure 8.6. Part of the recursion tree, Eight Queens problem

calls to AddQueen for the eight iterations of the for loop. Even at levels near the root, however, most of these branches are found to be impossible, and the removal of one vertex on an upper level removes a multitude of its descendents. Backtracking is a most effective tool to prune a recursion tree to manageable size.

Exercises 8.2

E1. What is the maximum depth of recursion in the Eight Queens program?

E2. Starting with the following partial configuration of five queens on the board, construct the recursion tree of all situations that the Eight Queens program will consider in trying to add the remaining three queens. Stop drawing the tree at the point where the program will backtrack and remove one of the original five queens.



E3. Modify the linked-list algorithm for generating permutations so that the position occupied by each number does not change by more than one to the left or to the right from any permutation to the next one generated. [This is a simplified form of one rule for *campanology* (ringing changes on church bells).]

Programming Projects 8.2

P1. Run the Queen program on your computer. You will need to write function WriteBoard to do the output. In addition, find out exactly how many positions are investigated by including a counter that is incremented every time function AddQueen is started. [Note that a method that placed all eight queens before checking for guarded squares would be equivalent to eight calls to AddQueen.]

molecular weight

P2. Write a program that will read a molecular formula such as H_2SO_4 and will write out the molecular weight of the compound that it represents. Your program should be able to handle bracketed radicals such as in $Al_2(SO_4)_3$. [*Hint*: Use recursion to find the molecular weight of a bracketed radical. *Simplifications:* You may find it helpful to enclose the whole formula in parentheses(...). You will need to set up a table of atomic weights of elements, indexed by their abbreviations. For simplicity the table may be restricted to the more common elements. Some elements have one-letter abbreviations, and some two. For uniformity you may add blanks to the one-letter abbreviations.]