positioning queens on a chess board is one of the classic problems in mathematics and computer science. This long-standing problem goes back even before Carl Gauss (1777–1855), and is based on the chessboard. It is the problem of finding all of the ways to position eight queens on the chessboard so that none of them is under attack by any other. Remember, the queen can move horizontally, vertically, and in the two diagonal directions; for convenience I’ll call the direction down and to the right (and its reverse) the diagonal direction, then call the direction up and to the right the antidiagonal direction.

You could approach this problem by looking at all possible ways of placing eight queens in the 64 available cells—and there are 64!/56! ways to do that (the permutations of 64 things taken eight at a time), but you don’t have to look at all 1.78E+14 permutations. (If you could figure out a way of just generating the combinations of 64 things taken eight at a time, the number goes down to 4.43E+09.) You can use some natural intelligence because you know that the queens can attack horizontally. That means that you can only have one queen to a row. Because there are eight positions on each of the eight rows, the total candidate configurations is reduced to 8^8—1.68E+07—a nasty number but not nearly as nasty as the earlier one. You can, however, toss out massive numbers of these.

As a programming exercise, this is a classical problem solved by backtracking. You start by positioning a queen in the top row of the board. As she sits in her cell, you move to the next row and try positioning a queen there. If you find that a queen above is attacking the queen below, you don’t have to proceed any farther. All board configurations that include this start will be illegal. So you simply position that queen on the next available cell. You back up to the earlier row when you have finished positioning a queen on all available cells of the current row and try the next cell in that row for its queen.

Eight queens is a bit challenging to start with, so you can rephrase this as the N Queens problem: Positioning N queens on a grid made up of N rows of N squares to the row. We know that there is no solution for the two-queens problem. Necessarily, each queen is attacking the other either vertically or diagonally. If you play around with paper and pencil, you’ll see that there is no possible solution for the three-queens problem either—there is necessarily a diagonal attack. So the four-queens problem is the first one that has a solution. For pseudocode purposes I’m going to use the C and Java conventions of numbering rows from zero on up. This means that when you hit N, you’ve already positioned N queens.

To position a queen in row j:
1. If j has reached N, you have a valid solution: Process it as valid.
2. Otherwise, for each column, position k in this row:
   a. Position a queen in the (j,k) cell.
   b. Check for attack by all the queens above row j.
   c. If there is no attack, position a queen in row j+1.

Listing One implements this algorithm in C. The big question is how to perform the check for attack from above, and this is going to be the source for one of the optimizations. In Fundamentals of Computer Algorithms (Computer Science Press, 1978), Ellis Horowitz and Sartaj Sahni gave a simple test based on the assumption that the board above the current position is valid. They chose to represent the board as a one-dimensional array. Each element represents a row on the board, and the number in that element represents the column position that the queen is occupying. That means that you can easily check for vertical attack: An earlier row has the same column position as your current row. They also noticed that you can easily check for diagonal attack, as shown in the following pseudocode.

To check for a valid board[] filled from row 0 to row j:
1. For each row k from 0 to j–1:
   a. If board[k] = board[j], return False.
   b. Else if abs(board[row]– board[k])=(row–k), return False.
2. If the loop terminates normally, return True.
Listing Two implements this algorithm in C.

You can easily see that the time required for the check increases with the size of the board. In computer jargon, it is an “order-N2 algorithm.”

In Algorithms and Data Structures (1986), Niklaus Wirth showed that you can perform the check in constant time, provided that you use some additional arrays to hold information. (His first proposal of this technique, however, was in the April 1971 issue of the Communications of the ACM.) For instance, you can have an array indicating which columns have already been filled. To check whether you can position a queen in a particular column, you just look at that cell of the array to see if it is in use. Similarly, you can have additional arrays to hold information about the diagonals and antidiagonals. You can see that along the diagonals, the difference of the row and column subscripts is a constant, while along the antidiagonals the sum of the row and column subscripts is a constant. Listing Three implements Wirth’s algorithm in C.

Wirth’s algorithm dramatically speeds up the processing, requiring only about half the time as compared with the code using Horowitz and Sahni’s order-N validity check.

There is, however, another optimization possible. You’re positioning only one queen on a row because of the horizontal attack. You know exactly the same thing about the columns—for each column there can be only one queen. This means that all successful solutions are just going to be permutations of the column subscripts: Each successive row has one fewer candidate position than the previous row. Thus, the problem space (before backtracking) has come down from 8N to 8!—from 1.68E+07 down to 403E+04. All you have to do is generate candidate permutations. As a bonus, the validity check becomes a little easier, since you only need to check for diagonal and antidiagonal attacks.

(In Fundamentals of Algorithmics, Gilles Brassard and Paul Bratley take note of using this method, but they approach it as examining all 8! possible permutations without combining the method with backtracking during permutation generation.)

You first fill the entire array with all of the column positions—a massively illegal configuration with all the queens lined up along the diagonal—but then you march the available column positions through each row cell. You can do this with swaps: After you have evaluated the initial partial permutation, you just swap the value in the front cell with the values in the remaining cells and then evaluate the resulting partial permutation. In the end, you have all the values in the same order except that the value from the last cell is now at the front. So you can regenerate the original configuration by doing a circular leftward rotation in the array. (This was discussed in my article in “Backtracking Algorithms,” DDJ, May 2004.)

To position a queen in row j:

1. If j has reached N, you have a valid solution. Process it as valid.
2. Otherwise:
   a. Loop as k takes on values from j to N–1:
      i. Swap entries j and k.
      ii. Check for attack by all the queens above row j.
   b. If there is no attack, position a queen in row j+1.
   c. Restore the initial state of the array:
      i. Save the value in position j.
      ii. As k goes from j+1 to the end, move elements from [k] to [k–1].
      iii. Position the saved value at the end of the array.

Listing Four implements this algorithm in C.

This optimization even more dramatically speeds up the processing, requiring only a fifth of the time as compared with the code using the row-filling implementation. The comparison code uses Horowitz and Sahni’s order-N validity check so that speed-up comes only from the permutation vector optimization.

Combining both optimizations, of course, really speeds up the processing.

The final optimizations are to remove the benchmarking superstructure, and then to use inline code for the most frequently performed operations: Marking the Boolean arrays for diagonal attack and performing the validity checks based on those Boolean arrays. This roughly doubles the speed.

Figure 1 shows the time required in a benchmarking run on a Dell desktop computer with a 2-GHz Pentium 4 processor for boards of sizes 12 up to 18. The Excel workbook and code are available electronically (see “Resource Center,” page 5). While the figure shows the C language results, the ZIP file also includes the Java implementations of the benchmarking code and of the final optimization code. Note that Figure 1 is using logarithmic scaling for its y-axis, and that the x-axis runs from 12 to 18.

Rejecting Equivalent Solutions

You may want to restrict the acceptable solutions to the unique solutions. Some solutions possess what is called “rotational symmetry”: If you rotate the board through 180 degrees—or perhaps even 90 degrees—you end up with exactly the same configuration. Figure 2 shows two solutions, one with the 180-degree rotational symmetry, the other with the 90-degree rotational symmetry. The numbering in the figure shows the queens that are equivalent upon the rotation.

If a solution does not possess such a rotational symmetry, then successive rotations through 90 degrees generate other solutions that will be discovered as you process the solutions. In addition to the rotations, there is another symmetry operation: reflection in a mirror. By the very nature of the N-Queens problem, a valid solution cannot have mirror symmetry. That means that each valid solution also has mirror images that turn up in the processing. Figure 3 shows the solution for the Five-Queens problem that lacks rotational symmetry (Figure 2 shows the

![Figure 1: N-Queens optimization results.](http://www.ddj.com)
symmetric solution). Consequently, there are eight solutions that are equivalent, being simply rotations or reflections of an initial board configuration.

You might think that rejecting equivalent solutions would require searching through previously accepted solutions, but there is an alternative approach that does not require saving the earlier solutions. You are representing the board by an array of column positions. All you need to do is to consider lexicographic ordering of the solutions, thinking of the array as an N-digit number. If you have the rule that you will only accept the first solution in this ordering of equivalent solutions, then the rejection of all the rest is straightforward. For a candidate solution, rotate it by successive 90-degree increments. If at any time the result compares as "smaller," reject the candidate solution. For the mirror images, you can generate one mirror image and then rotate that through three successive 90-degree increments to check for the four mirror images.

Listing Five is the C implementation of these symmetry checks. The function intncmp mimics the Standard C Library function strncmp, but with an array of ints rather than chars.

The timing results in Figure 1 are from running programs that implement this rejection of candidate solutions equivalent to the first on rotations and reflections.

You might think that the very fastest generation of all solutions to the N-Queens problem would be achieved by stripping out the symmetry checks. There are, however, several issues that need to be faced. Because of the vertical mirror plane, the optimized version of the Nqueens procedure only processes cells in the initial row in the first half of the array. If the mirror images are being counted separately, then you may think you need to go all the way across, doubling the amount of work. You can, however, just go half-way across and then adjust the result. If N is an even number, simply double the result, but if N is an odd number, then you would over-count the number of solutions in which the queen is placed in the center of the first row: You will count the mirror images twice. So you need to detect that case and only count those solutions once.

For all this grief, and with a significant loss of information, you get less than a 10 percent speed-up.

Table 1 summarizes the benchmarking results reflected in Figure 1. It gives the times and the time ratios for the benchmarking results running N from 12 to 18 in all six cases.

With enough work, you can sometimes reduce the time required for a calculation by a factor of 20. Too bad there isn’t a commercial application for the solutions to the N-Queens problem!

Parallel Threads in Java

The Queens problem belongs to the class of problems called “Embarrassingly parallel”—The solution to one subproblem (here represented by the queen’s position in the first row) is completely independent of another subproblem (a different position in the first row). Consequently, if you have a computer with multiple processors, it is easy to keep them busy solving the problem in parallel by using Java threads. (See, for instance, my article “Bargain-Basement Parallelism,” DDJ, February 2003.)

The only requirement is this: You must ensure that each thread works on a different first-row position, and that the threads take turns in building the complete solution (the total number of solutions and of unique solutions). You can do this by taking advantage of the Java keyword synchronized. In the absence of wait(), only one process at a time can execute the entirety of a synchronized method, so that (in operating-system parlance) the method represents a critical section of code.

Save 30% — more Wrox for less

Four topics, four titles, one great sale at Borders. Wrox is the one that helps you get the job done!

Details in store.
Listing Six shows the class Board, which controls the initial board positions that the threads work from and also accumulates the sums for total solutions and unique solutions. The synchronized method nextJob( ) receives the partial results from a thread (receiving zeroes on the first invocation) and sends the column position from which the thread should begin the next job — returning a negative number as the end-of-job message.

Coarser synchronization (waiting for thread completion) is handled by the Java method Thread.join( ). To ease the waiting game, you can daisy-chain thread creation: Each thread generates its own child thread until all required threads are active. The main program can then simply execute child.join( ) and be certain that all threads have completed before it resumes execution because every thread with a child will itself execute child.join( ) before terminating.

Listing Seven shows the constructor for the class WorkEngine that extends Thread. The child thread creation and start is part of the constructor itself. This allows the run( ) method to contain just the logic to dialog with the Board object to work subproblems and then terminate after receiving the end-of-job message and executing child.join( ), if appropriate. Listing Eight shows that method.

Table 2 shows the statistics from a number of runs on a quad-processor Xeon computer under Linux. Since the Xeon processor is itself a dual processor, Linux sees eight available 1.5-MHz processors. From that table, you can easily see some

<table>
<thead>
<tr>
<th>Description</th>
<th>Time Required</th>
<th>Ratio With No Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Optimization</td>
<td>20:59:51</td>
<td>1.000</td>
</tr>
<tr>
<td>Wirth's Validity Check</td>
<td>8:51:50</td>
<td>2.369</td>
</tr>
<tr>
<td>Permutation Vector</td>
<td>3:52:36</td>
<td>5.416</td>
</tr>
<tr>
<td>Both Optimizations</td>
<td>1:54:54</td>
<td>10.965</td>
</tr>
<tr>
<td>Inline Code Optimization</td>
<td>1:03:28</td>
<td>19.850</td>
</tr>
<tr>
<td>Without Symmetry Checks</td>
<td>0:58:18</td>
<td>21.609</td>
</tr>
</tbody>
</table>

Table 1: Benchmarking results.

<table>
<thead>
<tr>
<th>Original</th>
<th>Vertical mirror</th>
<th>1 . . . . .</th>
<th>. . . . 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . 2 .  .</td>
<td>. . . . 2 . .</td>
<td>3 . . . . \</td>
<td>. . . . 3 . .</td>
</tr>
<tr>
<td>. . . 4 .</td>
<td>. . . . . . 4 .</td>
<td>. . . . . . 4 .</td>
<td>. . . . . . 4 .</td>
</tr>
<tr>
<td>. . . . . 5</td>
<td>. . . . . . . 5</td>
<td>. . . . . . . 5</td>
<td>. . . . . . . 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>90 degree rotation</th>
<th>Antidiagonal mirror</th>
<th>. . . . . 1</th>
<th>. . . . . 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . . . . . . . 4</td>
<td>. . . . . . . . 5</td>
<td>. . . . . . . . 5</td>
<td>. . . . . . . . 5</td>
</tr>
<tr>
<td>. . . . . . . . . 2</td>
<td>. . . . . . . . . 5</td>
<td>. . . . . . . . . 5</td>
<td>. . . . . . . . . 5</td>
</tr>
<tr>
<td>. . . . . . . . . . 5</td>
<td>. . . . . . . . . . 5</td>
<td>. . . . . . . . . . 5</td>
<td>. . . . . . . . . . 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>180 degree rotation</th>
<th>Horizontal mirror</th>
<th>. . . . . . 5</th>
<th>. . . . . . 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . . . . . . . . . 3</td>
<td>. . . . . . . . . . 3</td>
<td>. . . . . . . . . . 3</td>
<td>. . . . . . . . . . 3</td>
</tr>
<tr>
<td>. . . . . . . . . . . 2</td>
<td>. . . . . . . . . . . 2</td>
<td>. . . . . . . . . . . 2</td>
<td>. . . . . . . . . . . 2</td>
</tr>
<tr>
<td>. . . . . . . . . . . . 1</td>
<td>. . . . . . . . . . . . 1</td>
<td>. . . . . . . . . . . . 1</td>
<td>. . . . . . . . . . . . 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>270 degree rotation</th>
<th>Diagonal mirror</th>
<th>. . . . . . 3</th>
<th>. . . . . . 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . . . . . . . . . . 1</td>
<td>. . . . . . . . . . . 1</td>
<td>. . . . . . . . . . . 1</td>
<td>. . . . . . . . . . . 1</td>
</tr>
<tr>
<td>. . . . . . . . . . . . 5</td>
<td>. . . . . . . . . . . . 5</td>
<td>. . . . . . . . . . . . 5</td>
<td>. . . . . . . . . . . . 5</td>
</tr>
<tr>
<td>. . . . . . . . . . . . . 2</td>
<td>. . . . . . . . . . . . . 2</td>
<td>. . . . . . . . . . . . . 2</td>
<td>. . . . . . . . . . . . . 2</td>
</tr>
<tr>
<td>. . . . . . . . . . . . . . 4</td>
<td>. . . . . . . . . . . . . . 4</td>
<td>. . . . . . . . . . . . . . 4</td>
<td>. . . . . . . . . . . . . . 4</td>
</tr>
</tbody>
</table>

Table 2: Examples of rotational symmetry in solutions.

Figure 2: Examples of rotational symmetry in solutions.

DynamicPDF™ components will revolutionize the way your websites and applications handle printable output. DynamicPDF™ components are available natively for .NET, Java and COM/ActiveX.

DynamicPDF™ Merger v3.0
Our popular and highly efficient class library for real time PDF document manipulation is available natively for .NET and Java • 100% managed and safe code • Intuitive object model • Merging • Appendding • Form-filling • Page placing, rotating and scaling • Splitting • Stamping • Seamless integration with Generator API

DynamicPDF™ Generator v3.0
Our flexible and highly efficient class library for real time PDF creation is available natively for .NET, Java and COM/ActiveX • Intuitive object model • 18 bar code symbologies • HTML Text • Unicode and CJK font support • Font embedding and subsets • Flexible document templating • Custom page element API

DynamicPDF™ ReportWriter v1.6
COM/ActiveX class library that integrates with ADO allowing for quick and easy generation of PDF reports • Five section reports • Group-by functionality • Dynamic text sizing • Supports 18 reporting events • Font embedding • Layout mode • Custom format function • Page and section numbering
Listing One

```c
void Nqueens (int Board[], int Trial[], int Size, int Row)
{
    if (Row == Size)
        Process(Board, Size);
    else
        for (int Col = 0; Col < Size; Col++)
            Board[Row] = Col;
            if (Valid (Board, Size, Row))
                Nqueens (Board, Trial, Size, Row+1);
}
```

Listing Two

```c
int Valid (int Board[], int Size, int Row)
{
    for (int Idx = 0; Idx < Row; Idx++)
        if (Board[Idx] == Board[Row] ||
            abs(Board[Row]-Board[Idx]) == (Row-Idx))
            return 0;    // boolean false
    return 1;          // boolean true
}
```

Listing Three

```c
int Valid (int Board[], int Size, int Row,
            int Col[], int diag[], int antiD[])
{
    int Idx;        /* Index into Diag[] / AntiD[] */
    int Chk;        /* Occupied flag               */
    Chk = Col[Board[Row]];
    /* diagonal:  Row-Col == constant */
    Idx = Row - Board[Row] + Size-1;
    Chk = Chk || Diag[Idx];
    /* anti-diagonal:  Row+Col == constant */
    Idx = Row + Board[Row];
    Chk = Chk || AntiD[Idx];
    return !Chk;    /* Valid if NOT any occupied  */
}
```

Listing Four

```c
void Nqueens (int Board[], int Size, int Row)
{
    int Idx, Lim, Vtemp;
    /* Check for a partial board. */
    if (Row < Size-1)
        { if (Valid (Board, Size, Row)
          Nqueens (Board, Trial, Size, Row+1);
     for (Idx = Row+1; Idx < Size; Idx++)
         { Vtemp = Board[Idx];
        Board[Idx] = Board[Row];
        Board[Row] = Vtemp;
         if (Valid (Board, Size, Row))
             Nqueens (Board, Trial, Size, Row+1); }
    /* Regenerate original vector from Row to Size-1: */
    Vtemp = Board[Row];
    for (Idx = Row; Idx < Size; Idx++)
        Board[Idx] = Board[Row];
    Board[Row] = Vtemp;
    /* This is a complete board. Final validity check */
    if (Valid (Board, Size, Row))
        Process(Board, Size);
    } /* end Nqueens */
}
```

**Acknowledgments**

The benchmarking runs reported here were performed during an academic vacation period on equipment owned by the State of Washington and located within the Computer Science Department at Eastern Washington University. This article contains material first presented at the Small College Computing Symposium at Augustana College (Sioux Falls, South Dakota), 21–22 April 1995. It was published in *SCCS: Proceedings of the 28th Annual Small College Computing Symposium* (1995), 201–10. The article is available online through [http://penguin.ewu.edu/~trolfe/SCCS-95/index.html](http://penguin.ewu.edu/~trolfe/SCCS-95/index.html).

For even more optimizations, see [http://www.jsomers.com/nqueens/nqueens.html](http://www.jsomers.com/nqueens/nqueens.html). I have felt it appropriate as I am working on this article not to work through his solution myself, but he reports a 10-fold speed-up compared with the code available through the SCCS-95 web page referenced earlier.

---

**Table 2: Thread Timing Results.** *Speed-up is the average uniprocessor time divided by the threaded time.*
/* WE HAVE A GOOD ONE! */
/* Reflect -- vertical mirror */
/* 90 degrees --- clockwise (4th parameter of Rotate is FALSE)*/
/* -270 degrees --- equiv. to anti-diagonal mirror */
/* -180 degrees --- equiv. to horizontal mirror */
/* -90 degrees --- equiv. to diagonal mirror */

http://www.ddj.com Dr. Dobb's Journal, May 2005

{  this.size = size;
}

public WorkEngine(int size, int nMore, Board info)
Listing Seven

Listing Six

Listing Eight

DDJ

We've got problems with your name on them.

At Google, we process the world’s information and make it accessible to the world’s population. As you might imagine, this task poses considerable challenges. Maybe you can help.

We’re looking for experienced software engineers with superb design and implementation skills and expertise in the following areas:

• high-performance distributed systems
• operating systems
• data mining
• information retrieval
• machine learning
• and/or related areas

If you have a proven track record based on cutting-edge research and/or large-scale systems development in these areas, we have brain-bursting projects with your name on them in Mountain View, Santa Monica, New York, Bangalore, Hyderabad, Zurich and Tokyo.


http://www.adlj.com

Dr. Dobb's Journal, May 2005

37