Overview

The goal of this lab is to explore recursion and algorithm analysis. The first two checkpoints work at the beginning level, while the last two involve a deeper use of recursion.

Checkpoints

1. Write the following two recursive functions:
   
   (a) Add the contents of a vector
   
   (b) Find the maximum value in a vector

   We have provided a simple driver main program to help you test the functions. Download the following file and add your functions to it:

   http://www.cs.rpi.edu/~stewart/cs2/labs/week13/q1_main.cpp

2. Ignore your laptops for the moment. Write by hand non-recursive functions to do the following and then give an “O” estimate of the number of operations required in each:

   (a) Add the contents of a vector
   
   (b) Find the maximum value in a vector
   
   (c) Find the maximum value in a vector that is already sorted.

   Justify your answers briefly.

3. The N-queens problem is the question of how to place queens on an \( N \times N \) chess board in such a way that no queen is “attacking” any other queen (for \( N \leq 3 \) no solutions exist). A queen is attacking another queen (or any other chess piece) if it is in the same row or column or diagonal. Here is an example of non-attacking queens for a \( 6 \times 6 \) board
Here is a different example where 4 queens have been placed in rows 0 through 3, but no queen may be placed safely anywhere in the 4th row.

The N-queens problem is most easily solved using recursion. Each recursive call is responsible for placing a queen in the next row (call it \( r \)). Each column, \( c \), of \( r \) is tested and if board position \((r, c)\) is not attacked by any of the queens in rows 0 through \( r - 1 \), then a queen is placed at \( (r, c) \) and a recursive call is made for row \( r + 1 \). If that recursive call returns false (failure), then the queen is removed from \( (r, c) \) and the next column, \( c + 1 \), of row \( r \) is tried. If the recursive call for row \( r + 1 \) returns true, however, then the recursive call for row \( r \) also immediately returns true because all \( N \) rows will have been filled. The recursive call for row \( r \) returns false if it has tried all columns of row \( r \) without success.
We have written the main program and the recursive function, `place_queens`. Your job is to write the functions `print_board` (this checkpoint) and `non_attacking` (next checkpoint).

Start by downloading the file

http://www.cs.rpi.edu/~stewart/cs2/labs/week13/queens.cpp

and read the code:

- The board is represented as a
  
  ```cpp
  vector< vector< bool > >
  ```

  using a `typedef` name of `board_type` as a short-hand.

- The main program reads the size of the board from the command-line (just one argument), initializes the board, and initiates the recursive search for a placement of the queens. If this succeeds, it outputs the board.

- The function `place_queens` is the recursive function that does most of the work.

To complete this 3rd checkpoint, write the function `output_board`. It should output a blank line before the board and then output each row of the board in one row of output. It should output a space and a `. ` for locations that don’t have a queen and a space and a `Q` for locations that do. Here’s an example for the $6 \times 6$ board above that has only 4 queens.

```
. Q . . .
. . . Q .
Q . . . .
. . . Q .
. . . . .
. . . . .
```

To complete this checkpoint, add code to the main program that creates a separate board, places the queens on the board in exactly the above configuration, and outputs the board. To get the program to compile, you will also have to add a “dummy” version of function `non_attacking` that just returns false.
4. Write the function `non_attacking`, whose prototype is shown in the code. The function should determine if it is safe to place a queen in the given `row` and `col` (row and column numbering both start at 0) assuming that there are queens in rows 0 through `row-1`, but none (so far) in rows `row` and higher. As an example, given the board

![Chess board with queens](image)

The following 5 function calls should all return false:

```cpp
    non_attacking( 4, 0, board );
    non_attacking( 4, 1, board );
    non_attacking( 4, 2, board );
    non_attacking( 4, 3, board );
    non_attacking( 4, 5, board );
```

and only the call

```cpp
    non_attacking( 4, 4, board );
```

should return true.

Complete this checkpoint and the entire lab by testing the program on boards of size 3, 4 and 5. Note that the code in `place_queens` calls `print_board` so that you can see each stage of the recursion.

**Final note:** The standard library does something that at first looks strange. A vector of bools does not use an internal representation that is an array of bools. It is more specialized. Therefore, code like
vector<bool> v;
    // insert some bools into v
bool *ptr = &v[0];

    if ( *ptr )
         // etc.

is illegal.