Problem Decomposition
ECE2893

Lecture 6
Solving Problems by Decomposition

Given a large problem, it is often convenient to decompose the problem into several smaller sub–problems. It is not necessary yet to be able to solve the sub–problems, but rather just to be able to describe the requirements for the sub–problems. Then think about the larger problem in terms of the (not yet solved, but assumed to be solvable) sub–problems.

In programming, we can even implement a “solution”, using subroutine calls to functions that represent the sub–problems. These functions need to be defined, but not necessarily implemented.

Also, when thinking about programming a solution in C/C++, we need to think abstractly about What data do I need to represent in my program?

Finally, we need to think of an Algorithm that uses the set of sub–problems and the data representations to solve the larger problem. Again, at this point, solutions to the sub–problems are not needed.

We will illustrate this approach with a detailed example.
Given a large problem, it is often convenient to decompose the problem into several smaller sub-problems.

It is not necessary yet to be able to solve the sub-problems, but rather just to be able to describe the requirements for the sub-problems. Then think about the larger problem in terms of the (not yet solved, but assumed to be solvable) sub-problems.

In programming, we can even implement a “solution”, using subroutine calls to functions that represent the sub-problems. These functions need to be defined, but not necessarily implemented.

Also, when thinking about programming a solution in C/C++, we need to think abstractly about What data do I need to represent in my program?

Finally, we need to think of an Algorithm that uses the set of sub-problems and the data representations to solve the larger problem. Again, at this point, solutions to the sub-problems are not needed.

We will illustrate this approach with a detailed example.
Solving Problems by Decomposition

1. Given a large problem, it is often convenient to *decompose* the problem into several smaller sub–problems.

2. It is not necessary yet to be able to solve the sub–problems, but rather just to be able to describe the requirements for the sub–problems.
Solving Problems by Decomposition

1. Given a large problem, it is often convenient to decompose the problem into several smaller sub–problems.

2. It is not necessary yet to be able to solve the sub–problems, but rather just to be able to describe the requirements for the sub–problems.

3. Then think about the larger problem in terms of the (not yet solved, but assumed to be solvable) sub–problems.

In programming, we can even implement a “solution”, using subroutine calls to functions that represent the sub–problems. These functions need to be defined, but not necessarily implemented.

Also, when thinking about programming a solution in C/C++, we need to think abstractly about What data do I need to represent in my program?

Finally, we need to think of an Algorithm that uses the set of sub–problems and the data representations to solve the larger problem. Again, at this point, solutions to the sub–problems are not needed.

We will illustrate this approach with a detailed example.
Solving Problems by Decomposition

1. Given a large problem, it is often convenient to *decompose* the problem into several smaller sub–problems.
2. It is not necessary yet to be able to solve the sub–problems, but rather just to be able to describe the requirements for the sub–problems.
3. Then think about the larger problem in terms of the (not yet solved, but assumed to be solvable) sub–problems.
4. In programming, we can even implement a "solution", using subroutine calls to functions that represent the sub–problems. These functions need to be defined, but not necessarily implemented.
Solving Problems by Decomposition

1. Given a large problem, it is often convenient to *decompose* the problem into several smaller sub–problems.

2. It is not necessary yet to be able to solve the sub–problems, but rather just to be able to describe the requirements for the sub–problems.

3. Then think about the larger problem in terms of the (not yet solved, but assumed to be solvable) sub–problems.

4. In programming, we can even implement a "solution", using subroutine calls to functions that represent the sub–problems. These functions need to be defined, but not necessarily implemented.

5. Also, when thinking about programming a solution in C/C++, we need to think abstractly about *What data do I need to represent in my program?*
Solving Problems by Decomposition

1. Given a large problem, it is often convenient to decompose the problem into several smaller sub–problems.
2. It is not necessary yet to be able to solve the sub–problems, but rather just to be able to describe the requirements for the sub–problems.
3. Then think about the larger problem in terms of the (not yet solved, but assumed to be solvable) sub–problems.
4. In programming, we can even implement a "solution", using subroutine calls to functions that represent the sub–problems. These functions need to be defined, but not necessarily implemented.
5. Also, when thinking about programming a solution in C/C++, we need to think abstractly about What data do I need to represent in my program?
6. Finally, we need to think of an Algorithm that uses the set of sub–problems and the data representations to solve the larger problem. Again, at this point, solutions to the sub–problems are not needed.
Solving Problems by Decomposition

1. Given a large problem, it is often convenient to *decompose* the problem into several smaller sub–problems.
2. It is not necessary yet to be able to solve the sub–problems, but rather just to be able to describe the requirements for the sub–problems.
3. Then think about the larger problem in terms of the (not yet solved, but assumed to be solvable) sub–problems.
4. In programming, we can even implement a "solution", using subroutine calls to functions that represent the sub–problems. These functions need to be defined, but not necessarily implemented.
5. Also, when thinking about programming a solution in C/C++, we need to think abstractly about *What data do I need to represent in my program?*
6. Finally, we need to think of an *Algorithm* that uses the set of sub–problems and the data representations to solve the larger problem. Again, at this point, solutions to the sub–problems are not needed.
7. We will illustrate this approach with a detailed example.
The Eight Queens Problem

Given a chessboard of 8 rows and 8 columns, place eight Queen pieces on the board such that no queen attacks any other Queen.

If you are not familiar with chess, a Queen placed on any row \( r \) and column \( c \) attacks all squares on the same row, same column, and both diagonals starting at \( r, c \).

One solution for the problem is shown below. There are in fact exactly 92 solutions. In case you are interested, there are 14,772,512 solutions for a 16 x 16 chessboard!
The Eight Queens Problem

Given a chessboard of 8 rows and 8 columns, place eight Queen pieces on the board such that no queen attacks any other Queen.
The Eight Queens Problem

1. Given a chessboard of 8 rows and 8 columns, place eight Queen pieces on the board such that no queen attacks any other Queen.

2. If you are not familiar with chess, a Queen placed on any row $r$ and column $c$ attacks all squares on the same row, same column, and both diagonals starting at $r, c$. 

One solution for the problem is shown below. There are in fact exactly 92 solutions. In case you are interested, there are 14,772,512 solutions for a 16 x 16 chessboard!
The Eight Queens Problem

1. Given a chessboard of 8 rows and 8 columns, place eight Queen pieces on the board such that no queen attacks any other Queen.

2. If you are not familiar with chess, a Queen placed on any row $r$ and column $c$ attacks all squares on the same row, same column, and both diagonals starting at $r, c$.

3. One solution for the problem is shown below. There are in fact exactly 92 solutions. *In case you are interested, there are 14,772,512 solutions for a 16 x 16 chessboard!*.
The Eight Queens Sub–Problems

What sub–problems can we define for the Eight Queens problem?
The Eight Queens Sub–Problems

What sub–problems can we define for the Eight Queens problem?

1. Given a board with zero or more Queens already placed, is it legal to place another Queen on square \( r, c \).

2. Place a new Queen on the board at square \( r, c \).

3. Remove an existing Queen from the board at square \( r, c \).

Note. It is necessary to remove pieces, since placing a Queen legally on a square may in fact (and often does) lead to a situation where no solution can be found from that position. When this happens we have to remove that piece and find another square to place the next piece.

4. Has a solution been found?

5. Print the solution.
The Eight Queens Sub–Problems

What sub–problems can we define for the Eight Queens problem?

1. Given a board with zero or more Queens already placed, is it legal to place another Queen on square \( r, c \).
2. Place a new Queen on the board at square \( r, c \).
The Eight Queens Sub–Problems

What sub–problems can we define for the Eight Queens problem?

1. Given a board with zero or more Queens already placed, is it legal to place another Queen on square \( r, c \).
2. Place a new Queen on the board at square \( r, c \).
3. Remove an existing Queen from the board at square \( r, c \).

Note. It is necessary to remove pieces, since placing a Queen legally on a square may in fact (and often does) lead to a situation where no solution can be found from that position. When this happens we have to remove that piece and find another square to place the next piece.
The Eight Queens Sub–Problems

What sub–problems can we define for the Eight Queens problem?

1. Given a board with zero or more Queens already placed, is it legal to place another Queen on square \( r, c \).
2. Place a new Queen on the board at square \( r, c \).
3. Remove an existing Queen from the board at square \( r, c \).
   Note. It is necessary to remove pieces, since placing a Queen legally on a square may in fact (and often does) lead to a situation where no solution can be found from that position. When this happens we have to remove that piece and find another square to place the next piece.
4. Has a solution been found?
The Eight Queens Sub–Problems

What sub–problems can we define for the Eight Queens problem?

1. Given a board with zero or more Queens already placed, is it legal to place another Queen on square \( r, c \).
2. Place a new Queen on the board at square \( r, c \).
3. Remove an existing Queen from the board at square \( r, c \).
   
   Note. It is necessary to remove pieces, since placing a Queen legally on a square may in fact (and often does) lead to a situation where no solution can be found from that position. When this happens we have to remove that piece and find another square to place the next piece.
4. Has a solution been found?
5. Print the solution
The Eight Queens Abstract Data

What data do we need for the Eight Queens problem?
The Eight Queens Abstract Data

What data do we need for the Eight Queens problem?

1. A representation of the state of the chessboard.
The Eight Queens Abstract Data

What data do we need for the Eight Queens problem?

1. A representation of the state of the chessboard.
2. For each square on the board, is there already a Queen there?

3. If a square is empty, is it attacked by another Queen?
4. If a square is attacked, how many other Queens attack it?

Nothing else comes to mind at the moment, but we may decide later that more information is needed.
The Eight Queens Abstract Data

What data do we need for the Eight Queens problem?

1. A representation of the state of the chessboard.
   1. For each square on the board, is there already a Queen there?
   2. If a square is empty, is it attacked by another Queen.

   Nothing else comes to mind at the moment, but we may decide later that more information is needed.
The Eight Queens Abstract Data

What data do we need for the Eight Queens problem?

1. A representation of the state of the chessboard.
   1. For each square on the board, is there already a Queen there?
   2. If a square is empty, is it attacked by another Queen?
   3. If a square is attacked, how many other Queens attack it?

Nothing else comes to mind at the moment, but we may decide later that more information is needed.
The Eight Queens Abstract Data

What data do we need for the Eight Queens problem?

1. A representation of the state of the chessboard.
   1. For each square on the board, is there already a Queen there?
   2. If a square is empty, is it attacked by another Queen
   3. If a square is attacked, how many other Queens attack it?

2. Nothing else comes to mind at the moment, but we may decide later that more information is needed.
An Eight Queens Algorithm (flawed!)

**Algorithm Q** *Find a solution to the eight Queens problem.* Given an eight by eight chessboard, place eight Queens on the board such that no Queen attacks any other Queen.
An Eight Queens Algorithm (flawed!)

Algorithm Q Find a solution to the eight Queens problem. Given an eight by eight chessboard, place eight Queens on the board such that no Queen attacks any other Queen.

Q1 \textit{SET} col \leftarrow 0
An Eight Queens Algorithm (flawed!)

**Algorithm Q** *Find a solution to the eight Queens problem.* Given an eight by eight chessboard, place eight Queens on the board such that no Queen attacks any other Queen.

**Q1**  
*SET col ← 0*

**Q2**  
*WHILE col < 8*
An Eight Queens Algorithm (flawed!)

**Algorithm Q** Find a solution to the eight Queens problem. Given an eight by eight chessboard, place eight Queens on the board such that no Queen attacks any other Queen.

**Q1**  
\[ \text{SET } \text{col} \leftarrow 0 \]

**Q2**  
\[ \text{WHILE } \text{col} < 8 \]

\[ \text{Q2.1 } \text{Find the first row in the column } \text{col} \text{ where a Queen can legally be placed.} \]
An Eight Queens Algorithm (flawed!)

**Algorithm Q** Find a solution to the eight Queens problem. Given an eight by eight chessboard, place eight Queens on the board such that no Queen attacks any other Queen.

Q1  \( \text{SET \ } col \leftarrow 0 \)

Q2  \( \text{WHILE } col < 8 \)
    Q2.1  Find the first row in the column \( col \) where a Queen can legally be placed.
    Q.2.2  IF legal location found

---

ECE2893  Problem Decomposition  Spring 2011  6 / 17
An Eight Queens Algorithm (flawed!)

**Algorithm Q** *Find a solution to the eight Queens problem.* Given an eight by eight chessboard, place eight Queens on the board such that no Queen attacks any other Queen.

Q1 \( \text{SET } \text{col} \leftarrow 0 \)
Q2 \( \text{WHILE } \text{col} < 8 \)
   Q2.1 Find the first row in the column \( \text{col} \) where a Queen can legally be placed.
   Q.2.2 \( \text{IF} \) legal location found
      Q.2.2.1 Place a Queen on that square.
An Eight Queens Algorithm (flawed!)

**Algorithm Q** Find a solution to the eight Queens problem. Given an eight by eight chessboard, place eight Queens on the board such that no Queen attacks any other Queen.

Q1 \( SET \ col \leftarrow 0 \)

Q2 \( WHILE \ col < 8 \)
  
  Q2.1 Find the first row in the column \( col \) where a Queen can legally be placed.
  
  Q2.2 **IF** legal location found
    
    Q2.2.1 Place a Queen on that square.
    
    Q2.2.2 \( SET \ col \leftarrow col + 1 \)
An Eight Queens Algorithm (flawed!)

**Algorithm Q** Find a solution to the eight Queens problem. Given an eight by eight chessboard, place eight Queens on the board such that no Queen attacks any other Queen.

Q1  \[ SET \ col \leftarrow 0 \]

Q2  WHILE \ col < 8

Q2.1  Find the first row in the column \ col \ where a Queen can legally be placed.

Q2.2  \textit{IF} legal location found

Q2.2.1  Place a Queen on that square.

Q2.2.2  \[ SET \ col \leftarrow col + 1 \]

Q2.3  ELSE
An Eight Queens Algorithm (flawed!)

Algorithm Q Find a solution to the eight Queens problem. Given an eight by eight chessboard, place eight Queens on the board such that no Queen attacks any other Queen.

Q1 \( \text{SET } \text{col} \leftarrow 0 \)
Q2 \( \text{WHILE } \text{col} < 8 \)
\hspace{1em} Q2.1 Find the first row in the column \( \text{col} \) where a Queen can legally be placed.
\hspace{1em} Q2.2 \text{IF legal location found}
\hspace{1.5em} Q2.2.1 Place a Queen on that square.
\hspace{1.5em} Q2.2.2 \( \text{SET } \text{col} \leftarrow \text{col} + 1 \)
\hspace{1em} Q2.3 \text{ELSE}
\hspace{1.5em} Q2.3.1 No solution is possible, exit algorithm.
An Eight Queens Algorithm (flawed!)

Algorithm Q *Find a solution to the eight Queens problem*. Given an eight by eight chessboard, place eight Queens on the board such that no Queen attacks any other Queen.

Q1 \( SET \ col \leftarrow 0 \)
Q2 \( WHILE \ col < 8 \)
\( \quad Q2.1 \) Find the first row in the column \( col \) where a Queen can legally be placed.
\( \quad Q2.2 \) *IF* legal location found
\( \quad \quad Q2.2.1 \) Place a Queen on that square.
\( \quad \quad Q2.2.2 \) \( SET \ col \leftarrow col + 1 \)
\( \quad Q2.3 \) *ELSE*
\( \quad \quad Q2.3.1 \) No solution is possible, exit algorithm.

Q3 Print The Solution
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)

Diagram of a chessboard with four queens placed in such a way that they are attacking each other.
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)

[Diagram of a chessboard with multiple queens placed in such a way that they do not attack each other]
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (flawed!)
An Eight Queens Algorithm (better)

Algorithm Q  *Find a solution to the eight Queens problem.*
An Eight Queens Algorithm (better)

**Algorithm Q** *Find a solution to the eight Queens problem.*

Q1  \( \text{SET } col \leftarrow 0 \)
An Eight Queens Algorithm (better)

Algorithm Q Find a solution to the eight Queens problem.

Q1 \textit{SET} \textit{col} \leftarrow 0

Q2 \textit{WHILE} \textit{col} < 8
An Eight Queens Algorithm (better)

**Algorithm Q** *Find a solution to the eight Queens problem.*

Q1  **SET** \( col \leftarrow 0 \)
Q2  **WHILE** \( col < 8 \)

Q2.1  Find the *next* row in the current column where a Queen can legally be placed.
An Eight Queens Algorithm (better)

**Algorithm Q** *Find a solution to the eight Queens problem.*

Q1  SET col $\leftarrow 0$
Q2  WHILE col $< 8$

Q2.1  Find the *next* row in the current column where a Queen can legally be placed.

Q.2.2  *IF* legal location found

Q.2.3  *ELSE*

Q.2.3.1  SET col $\leftarrow$ col $- 1$
Q.2.3.2  Backtrack (remove last piece placed and try again)

Q3  Print The Solution
An Eight Queens Algorithm (better)

**Algorithm Q** Find a solution to the eight Queens problem.

Q1 \( \text{SET } \text{col} \leftarrow 0 \)
Q2 \( \text{WHILE } \text{col} < 8 \)

Q2.1 Find the *next* row in the current column where a Queen can legally be placed.
Q.2.2 \( \text{IF} \) legal location found
Q.2.2.1 Place a Queen on that square.
Q.2.3 \( \text{ELSE} \)
Q.2.3.1 \( \text{SET} \) \( \text{col} \leftarrow \text{col} - 1 \)
Q.2.3.2 Backtrack (remove last piece placed and try again)
Q3 Print The Solution
An Eight Queens Algorithm (better)

**Algorithm Q** *Find a solution to the eight Queens problem.*

Q1  \( \text{SET } \text{col} \leftarrow 0 \)
Q2  \( \text{WHILE } \text{col} < 8 \)

  Q2.1  Find the *next* row in the current column where a Queen can legally be placed.

  Q2.2  *IF* legal location found

    Q2.2.1  Place a Queen on that square.

    Q2.2.2  \( \text{SET } \text{col} \leftarrow \text{col} + 1 \)
An Eight Queens Algorithm (better)

**Algorithm Q**  
*Find a solution to the eight Queens problem.*

**Q1**  
`SET col ← 0`

**Q2**  
`WHILE col < 8`

**Q2.1** Find the *next* row in the current column where a Queen can legally be placed.

**Q.2.2** *IF* legal location found

**Q.2.2.1** Place a Queen on that square.

**Q.2.2.2** `SET col ← col + 1`

**Q.2.3** *ELSE*

**Q.2.3.1** `SET col ← col − 1`

**Q.2.3.2** Backtrack (remove last piece placed and try again)

**Q3** Print The Solution
An Eight Queens Algorithm (better)

**Algorithm Q** Find a solution to the eight Queens problem.

Q1 \[\text{SET} \ col \leftarrow 0\]

Q2 \[\text{WHILE} \ col < 8\]

Q2.1 Find the *next* row in the current column where a Queen can legally be placed.

Q2.2 *IF* legal location found

Q2.2.1 Place a Queen on that square.

Q2.2.2 \[\text{SET} \ col \leftarrow col + 1\]

Q2.3 *ELSE*

Q2.3.1 \[\text{SET} \ col \leftarrow col - 1\]

Q2.3.2 *Backtrack* (remove last piece placed and try again)
An Eight Queens Algorithm (better)

**Algorithm Q** *Find a solution to the eight Queens problem.*

Q1  \( \text{SET} \ col \leftarrow 0 \)
Q2  \( \text{WHILE} \ col < 8 \)
    Q2.1  Find the *next* row in the current column where a Queen can legally be placed.
    Q2.2  *IF* legal location found
        Q2.2.1  Place a Queen on that square.
        Q2.2.2  \( \text{SET} \ col \leftarrow col + 1 \)
    Q2.3  *ELSE*
        Q2.3.1  \( \text{SET} \ col \leftarrow col - 1 \)
        Q2.3.2  *Backtrack* (remove last piece placed and try again)
Q3  Print The Solution
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)

![Eight Queens Algorithm Diagram]
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
An Eight Queens Algorithm (better)
Backtracking

1. Back up to previous column.

2. If there is no previous column the algorithm terminates.

3. Remove the Queen presently placed on this row (there will always be one).

4. Find the next row where a Queen can legally be placed on this column, starting from the row where the removed Queen was placed.

5. If none can be found, backtrack again.

6. If one is found, place a Queen there and advance to the next column.
Backtracking

1. Back up to previous column.

2. If there is no previous column the algorithm terminates.

3. Remove the Queen presently placed on this row (there will always be one).

4. Find the next row where a Queen can legally be placed on this column, starting from the row where the removed Queen was placed.

5. If none can be found, backtrack again.

6. If one is found, place a Queen there and advance to the next column.
Backtracking

1. Back up to previous column.
   - If there is no previous column the algorithm terminates.
Backtracking

1. Back up to previous column.
   - If there is no previous column the algorithm terminates.

2. Remove the Queen presently placed on this row (there will always be one)
Backtracking

1. Back up to previous column.
   - If there is no previous column the algorithm terminates.

2. Remove the Queen presently placed on this row (there will always be one)

3. Find the *next* row where a Queen can legally be placed on this column, starting from the row where the removed Queen was placed.
Backtracking

1. Back up to previous column.
   1. If there is no previous column the algorithm terminates.
2. Remove the Queen presently placed on this row (there will always be one)
3. Find the next row where a Queen can legally be placed on this column, starting from the row where the removed Queen was placed.
4. If none can be found, backtrack again.
Backtracking

1. Back up to previous column.
   - If there is no previous column the algorithm terminates.

2. Remove the Queen presently placed on this row (there will always be one)

3. Find the next row where a Queen can legally be placed on this column, starting from the row where the removed Queen was placed.

4. If none can be found, backtrack again.

5. If one is found, place a Queen there and advance to the next column.
Place a Queen on a Given Row/Column

Mark the board location as occupied. Mark all squares on the same row, column, and diagonals as attacked.

Question. Is it possible when marking the squares as attacked we find the square already occupied?

▶ Answer. Not possible. If the square being marked already has a Queen, then the location we just placed was not a legal placement.

Question. Do we need a simple true/false value for the attacked array, or something more complicated?

▶ Answer. More complicated. It is possible that a given square is attacked by more than one Queen. When we remove a Queen, we cannot simply say the square is no longer attacked. Instead, we need a count (integer) of number of pieces attacking that square.
Place a Queen on a Given Row/Column

- Mark the board location as occupied.

Question. Is it possible when marking the squares as attacked we find the square already occupied?

Answer. Not possible. If the square being marked already has a Queen, then the location we just placed was not a legal placement.

Question. Do we need a simple true/false value for the attacked array, or something more complicated?

Answer. More complicated. It is possible that a given square is attacked by more than one Queen. When we remove a Queen, we cannot simply say the square is no longer attacked. Instead, we need a count (integer) of number of pieces attacking that square.
Place a Queen on a Given Row/Column

- Mark the board location as occupied.
- Mark all squares on the same row, column, and diagonals as attacked.

Question. Is it possible when marking the squares as attacked we find the square already occupied?

Answer. Not possible. If the square being marked already has a Queen, then the location we just placed was not a legal placement.

Question. Do we need a simple true/false value for the attacked array, or something more complicated?

Answer. More complicated. It is possible that a given square is attacked by more than one Queen. When we remove a Queen, we cannot simply say the square is no longer attacked. Instead, we need a count (integer) of number of pieces attacking that square.
Place a Queen on a Given Row/Column

- Mark the board location as occupied.
- Mark all squares on the same row, column, and diagonals as *attacked*.
- Question. Is it possible when marking the squares as *attacked* we find the square already occupied?

Answer. Not possible. If the square being marked already has a Queen, then the location we just placed was not a legal placement.

Question. Do we need a simple true/false value for the *attacked* array, or something more complicated?

Answer. More complicated. It is possible that a given square is attacked by more than one Queen. When we remove a Queen, we cannot simply say the square is no longer attacked. Instead, we need a count (integer) of number of pieces attacking that square.
Place a Queen on a Given Row/Column

- Mark the board location as occupied.
- Mark all squares on the same row, column, and diagonals as attacked.

Question. Is it possible when marking the squares as attacked we find the square already occupied?

  Answer. Not possible. If the square being marked already has a Queen, then the location we just placed was not a legal placement.
Place a Queen on a Given Row/Column

- Mark the board location as occupied.
- Mark all squares on the same row, column, and diagonals as *attacked*.

Question. Is it possible when marking the squares as *attacked* we find the square already occupied?
  - Answer. Not possible. If the square being marked already has a Queen, then the location we just placed was not a legal placement.

Question. Do we need a simple true/false value for the *attacked* array, or something more complicated?
Place a Queen on a Given Row/Column

- Mark the board location as occupied.
- Mark all squares on the same row, column, and diagonals as *attacked*.
- Question. Is it possible when marking the squares as *attacked* we find the square already occupied?
  - Answer. Not possible. If the square being marked already has a Queen, then the location we just placed was not a legal placement.
- Question. Do we need a simple true/false value for the *attacked* array, or something more complicated?
  - Answer. More complicated. It is possible that a given square is attacked by more than one Queen. When we remove a Queen, we cannot simply say the square is no longer attacked. Instead, we need a count (integer) of number of pieces attacking that square.
Remove a Queen from a Given Row/Column

Mark the board location as empty (unoccupied).

Decrement the attacked value for all squares on the same row, column, and diagonals.

Question. Is it possible when decrementing the attacked values that we find the square occupied?

▶ Answer. Not possible for same reasons as previously stated.
Remove a Queen from a Given Row/Column

- Mark the board location as empty (unoccupied).

Question. Is it possible when decrementing the attacked values that we find the square occupied?

▶ Answer. Not possible for same reasons as previously stated.
Remove a Queen from a Given Row/Column

- Mark the board location as empty (unoccupied).
- Decrement the *attacked* value for all squares on the same row, column, and diagonals.
Remove a Queen from a Given Row/Column

- Mark the board location as empty (unoccupied).
- Decrement the *attacked* value for all squares on the same row, column, and diagonals.
- Question. Is it possible when decrementing the *attacked* values that we find the square occupied?

▶ Answer. Not possible for same reasons as previously stated.
Remove a Queen from a Given Row/Column

- Mark the board location as empty (unoccupied).
- Decrement the \textit{attacked} value for all squares on the same row, column, and diagonals.
- Question. Is it possible when decrementing the \textit{attacked} values that we find the square occupied?
  - Answer. Not possible for same reasons as previously stated.
Has a Solution Been Found?

A solution has been found if the `col` variable has advanced to 8 (recall that columns are numbered 0 to 7). If, during backtracking, the column decrements to -1 (i.e., we are backtracking from column 0), then no solution is possible.
Has a Solution Been Found?

- A solution has been found if the \textit{col} variable has advanced to 8 (recall that columns are numbered 0 to 7).
Has a Solution Been Found?

- A solution has been found if the `col` variable has advanced to 8 (recall that columns are numbered 0 to 7).
- If, during backtracking, the column decrements to -1 (i.e., we are backtracking from column 0), then no solution is possible.
Suggested Data Structures

const int nCols = 8; // Number of columns
const int nRows = 8; // Number of rows
int Board[nRows][nCols];

▶ Suggested interpretation of the integers in board:
▶ 0 means unoccupied
▶ -1 means a Queen placed on that square
▶ Greater than zero indicates the number of attacks on that square.
Suggested Data Structures

- const int nCols = 8; // Number of columns

- Suggested interpretation of the integers in board:
  - 0 means unoccupied
  - -1 means a Queen placed on that square
  - Greater than zero indicates the number of attacks on that square.
Suggested Data Structures

- `const int nCols = 8; // Number of columns`
- `const int nRows = 8; // Number of rows`
Suggested Data Structures

- `const int nCols = 8; // Number of columns`
- `const int nRows = 8; // Number of rows`
- `int Board[nRows][nCols];`

- Suggested interpretation of the integers in board:
  - 0 means unoccupied
  - -1 means a Queen placed on that square
  - Greater than zero indicates the number of attacks on that square.
Suggested Data Structures

- const int nCols = 8; // Number of columns
- const int nRows = 8; // Number of rows
- int Board[nRows][nCols];

  Suggested interpretation of the integers in board:
Suggested Data Structures

- `const int nCols = 8;` // Number of columns
- `const int nRows = 8;` // Number of rows
- `int Board[nRows][nCols];`
  - Suggested interpretation of the integers in board:
  - 0 means unoccupied
Suggested Data Structures

- const int nCols = 8; // Number of columns
- const int nRows = 8; // Number of rows
- int Board[nRows][nCols];

- Suggested interpretation of the integers in board:
  - 0 means unoccupied
  - -1 means a Queen placed on that square
Suggested Data Structures

- const int nCols = 8; // Number of columns
- const int nRows = 8; // Number of rows
- int Board[nRows][nCols];
  - Suggested interpretation of the integers in board:
    - 0 means unoccupied
    - -1 means a Queen placed on that square
    - Greater than zero indicates the number of attacks on that square.
Marking Squares as *Attacked*

Sub–problem. When placing a Queen on row \( r \) and column \( c \), update the attacked status of all squares attacked by the Queen at location \( r, c \).

- For each column \( c_1 \) in row \( r \), if \( c_1 \neq c \) then ▶ Increment \( \text{Board}[r][c_1] \) by one.
- For each row \( r_1 \) in column \( c \), if \( r_1 \neq r \) then ▶ Increment \( \text{Board}[r_1][c] \) by one.

The diagnoals are harder (next slide)
Marking Squares as *Attacked*

- Sub–problem. When placing a Queen on row \( r \) and column \( c \), update the *attacked* status of all squares attacked by the Queen at location \( r, c \).
Marking Squares as *Attacked*

- Sub–problem. When placing a Queen on row \( r \) and column \( c \), update the *attacked* status of all squares attacked by the Queen at location \( r, c \).
- For each column \( c_1 \) in row \( r \), if \( c_1 \neq c \) then
  
  ▶ Increment \( \text{Board}[r][c_1] \) by one.

  ▶ \( \text{Board}[r][c_1]++ \)

  ▶ Increment \( \text{Board}[r_1][c] \) by one.

  ▶ \( \text{Board}[r_1][c]++ \)

  The diagnoals are harder (next slide)
Marking Squares as *Attacked*

- Sub–problem. When placing a Queen on row \( r \) and column \( c \), update the *attacked* status of all squares attacked by the Queen at location \( r, c \).
- For each column \( c1 \) in row \( r \), if \( c1 \neq c \) then
  - Increment \( \text{Board}[r][c1] \) by one.

The diagonals are harder (next slide)
Marking Squares as *Attacked*

- Sub–problem. When placing a Queen on row $r$ and column $c$, update the *attacked* status of all squares attacked by the Queen at location $r, c$.
- For each column $c1$ in row $r$, if $c1 \neq c$ then
  - Increment `Board[r][c1]` by one.
  - `Board[r][c1]++`;

The diagnoals are harder (next slide)
Marking Squares as *Attacked*

- **Sub-problem.** When placing a Queen on row $r$ and column $c$, update the *attacked* status of all squares attacked by the Queen at location $r, c$.
- For each column $c_1$ in row $r$, if $c_1 \neq c$ then
  >  Increment $\text{Board}[r][c_1]$ by one.
  >  $\text{Board}[r][c_1]++$;
- For each row $r_1$ in column $c$, if $r_1 \neq r$ then
Marking Squares as *Attacked*

- **Sub–problem.** When placing a Queen on row $r$ and column $c$, update the *attacked* status of all squares attacked by the Queen at location $r, c$.

- For each column $c_1$ in row $r$, if $c_1 \neq c$ then
  - Increment $\text{Board}[r][c_1]$ by one.
  - $\text{Board}[r][c_1]++$

- For each row $r_1$ in column $c$, if $r_1 \neq r$ then
  - Increment $\text{Board}[r_1][c]$ by one.
Marking Squares as *Attacked*

- **Sub–problem.** When placing a Queen on row $r$ and column $c$, update the *attacked* status of all squares attacked by the Queen at location $r, c$.
- For each column $c_1$ in row $r$, if $c_1 \neq c$ then
  - Increment `Board[r][c1]` by one.
  - `Board[r][c1]++`;
- For each row $r_1$ in column $c$, if $r_1 \neq r$ then
  - Increment `Board[r1][c]` by one.
  - `Board[r1][c]++`;
Marking Squares as *Attacked*

- Sub–problem. When placing a Queen on row $r$ and column $c$, update the *attacked* status of all squares attacked by the Queen at location $r, c$.

  - For each column $c_1$ in row $r$, if $c_1 \neq c$ then
    - Increment $\text{Board}[r][c_1]$ by one.
    - $\text{Board}[r][c_1]++$;

  - For each row $r_1$ in column $c$, if $r_1 \neq r$ then
    - Increment $\text{Board}[r_1][c]$ by one.
    - $\text{Board}[r_1][c]++$;

- The diagnoals are harder (next slide)
Marking Squares as *Attacked*, Diagonals

There are four diagonals starting at location $r_1, c_1$.

1. Towards the lower right
2. Towards the upper right
3. Towards the lower left
4. Towards the upper left

For each of these, we increment (or decrement) both the row and column numbers by plus or minus one, and then increment the board location to indicate an additional attacker for that square. For example, going towards the upper right, the row decreases by one and the column increases by one.

$$
\begin{align*}
&\text{SET } r_1 = r_1 - 1, c_1 = c_1 + 1, \\
&\text{WHILE } r_1 \geq 0 \text{ AND } c_1 < 8, \\
&\text{SET } \text{Board}[r_1][c_1] = \text{Board}[r_1][c_1] + 1, \\
&\text{SET } r_1 = r_1 - 1, c_1 = c_1 + 1.
\end{align*}
$$

The other three diagonals are similar to above, but moving the $r_1$ and $c_1$ values in different directions.
Marking Squares as *Attacked*, Diagonals

- There are four diagonals starting at location $r, c$.

```plaintext
  ▶ SET $r_1 = r - 1$, $c_1 = c + 1$
  ▶ WHILE $r_1 \geq 0$ AND $c_1 < 8$
    ▶ SET Board[$r_1$][$c_1$] = Board[$r_1$][$c_1$] + 1
    ▶ SET $r_1 = r_1 - 1$, $c_1 = c_1 + 1$

The other three diagonals are similar to above, but moving the $r_1$ and $c_1$ values in different directions.
```
Marking Squares as *Attacked*, Diagonals

- There are four diagonals starting at location \( r, c \).
  - Towards the lower right
Marking Squares as *Attacked*, Diagonals

- There are four diagonals starting at location \( r, c \).
  1. Towards the lower right
  2. Towards the upper right
Marking Squares as *Attacked*, Diagonals

- There are four diagonals starting at location $r, c$.
  1. Towards the lower right
  2. Towards the upper right
  3. Towards the lower left

\[
\begin{align*}
\text{SET} \quad r_1 &= r - 1, \\
\text{SET} \quad c_1 &= c + 1 \quad \text{\(\star\)} \\
\text{SET} \quad \text{Board}[r_1][c_1] &= \text{Board}[r_1][c_1] + 1 \quad \text{\(\star\)} \\
\text{SET} \quad r_1 &= r_1 - 1, \\
\text{SET} \quad c_1 &= c_1 + 1
\end{align*}
\]

The other three diagonals are similar to above, but moving the $r_1$ and $c_1$ values in different directions.
Marking Squares as *Attacked*, Diagonals

There are four diagonals starting at location \( r, c \).

1. Towards the lower right
2. Towards the upper right
3. Towards the lower left
4. Towards the upper left

For each of these, we increment (or decrement) both the row and column numbers by plus or minus one, and then increment the board location to indicate an additional attacker for that square. For example, going towards the upper right, the row decreases by one and the column increases by one.

\[
\begin{align*}
\text{SET } r_1 &= r - 1, \\
\text{SET } c_1 &= c + 1
\end{align*}
\]

The other three diagonals are similar to above, but moving the \( r_1 \) and \( c_1 \) values in different directions.
Marking Squares as *Attacked*, Diagonals

- There are four diagonals starting at location $r, c$.
  1. Towards the lower right
  2. Towards the upper right
  3. Towards the lower left
  4. Towards the upper left

- For each of these, we increment (or decrement) both the row and column numbers by plus or minus one, and then increment the board location to indicate an additional attacker for that square.

\[
\begin{align*}
\text{SET } r_1 &= r - 1, c_1 = c + 1 \\
\text{SET } r_2 &= r_1 - 1, c_2 = c_1 + 1 \\
\text{SET } r_3 &= r_1 + 1, c_3 = c_1 - 1 \\
\text{SET } r_4 &= r_3 + 1, c_4 = c_3 - 1
\end{align*}
\]

The other three diagonals are similar to above, but moving the $r_1$ and $c_1$ values in different directions.
Marking Squares as *Attacked*, Diagonals

- There are four diagonals starting at location $r, c$.
  1. Towards the lower right
  2. Towards the upper right
  3. Towards the lower left
  4. Towards the upper left

- For each of these, we increment (or decrement) both the row and column numbers by plus or minus one, and then increment the board location to indicate an additional attacker for that square.

- For example, going towards the upper right, the row decreases by one and the column increases by one.
Marking Squares as *Attacked*, Diagonals

- There are four diagonals starting at location \( r, c \).
  1. Towards the lower right
  2. Towards the upper right
  3. Towards the lower left
  4. Towards the upper left

- For each of these, we increment (or decrement) both the row and column numbers by plus or minus one, and then increment the board location to indicate an additional attacker for that square.
- For example, going towards the upper right, the row decreases by one and the column increases by one.
  - \[ \text{SET } r_1 = r - 1, c_1 = c + 1 \]
Marking Squares as *Attacked*, Diagonals

- There are four diagonals starting at location \( r, c \).
  1. Towards the lower right
  2. Towards the upper right
  3. Towards the lower left
  4. Towards the upper left

- For each of these, we increment (or decrement) both the row and column numbers by plus or minus one, and then increment the board location to indicate an additional attacker for that square.

- For example, going towards the upper right, the row decreases by one and the column increases by one.
  
  \[
  \text{SET } r_1 = r - 1, \quad c_1 = c + 1
  \]
  \[
  \text{WHILE } r_1 \geq 0 \text{ AND } c_1 < 8
  \]
Marking Squares as *Attacked*, Diagonals

- There are four diagonals starting at location \( r, c \).
  1. Towards the lower right
  2. Towards the upper right
  3. Towards the lower left
  4. Towards the upper left

- For each of these, we increment (or decrement) both the row and column numbers by plus or minus one, and then increment the board location to indicate an additional attacker for that square.

- For example, going towards the upper right, the row decreases by one and the column increases by one.
  - \( SET \ r1 = r - 1, \ c1 = c + 1 \)
  - \( WHILE \ r1 \geq 0 \ AND \ c1 < 8 \)
    - \( SET \ Board[r1][c1] = Board[r1][c1] + 1 \)
Marking Squares as *Attacked*, Diagonals

- There are four diagonals starting at location \( r, c \).
  1. Towards the lower right
  2. Towards the upper right
  3. Towards the lower left
  4. Towards the upper left

- For each of these, we increment (or decrement) both the row and column numbers by plus or minus one, and then increment the board location to indicate an additional attacker for that square.

- For example, going towards the upper right, the row decreases by one and the column increases by one.

  
  - \( SET \ r1 = r - 1, \ c1 = c + 1 \)
  - \( WHILE \ r1 \geq 0 AND c1 < 8 \)
  
  
  - \( SET \ Board[r1][c1] = Board[r1][c1] + 1 \)
  - \( SET \ r1 = r1 - 1, \ c1 = c1 + 1 \)
Marking Squares as *Attacked*, Diagonals

- There are four diagonals starting at location \( r, c \).
  1. Towards the lower right
  2. Towards the upper right
  3. Towards the lower left
  4. Towards the upper left

- For each of these, we increment (or decrement) both the row and column numbers by plus or minus one, and then increment the board location to indicate an additional attacker for that square.

- For example, going towards the upper right, the row decreases by one and the column increases by one.

  - \( SET \ r1 = r - 1, c1 = c + 1 \)
  - \( WHILE \ r1 \geq 0 \ AND \ c1 < 8 \)
    - \( SET \ Board[r1][c1] = Board[r1][c1] + 1 \)
    - \( SET \ r1 = r1 - 1, c1 = c1 + 1 \)

- The other three diagonals are similar to above, but moving the \( r1 \) and \( c1 \) values in different directions.
A Solution!