Recursion and Recursive Backtracking

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.

Iteration

- When we encounter a problem that requires repetition, we often use *iteration* – i.e., some type of loop.
- Sample problem: printing the series of integers from n1 to n2, where n1 <= n2.
  - example: `printSeries(5, 10)` should print the following:

    `5, 6, 7, 8, 9, 10`
- Here's an iterative solution to this problem:

```
public static void printSeries(int n1, int n2) {
    for (int i = n1; i < n2; i++) {
        System.out.print(i + ", ");
    }
    System.out.println(n2);
}
```

Recursion

- An alternative approach to problems that require repetition is to solve them using *recursion.*
- A recursive method is a method that calls itself.
- Applying this approach to the print-series problem gives:

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

Tracing a Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
        printSeries(7, 7):
            System.out.print(7);
            return
        return
    return
```

Recursive Problem-Solving

- When we use recursion, we solve a problem by reducing it to a simpler problem of the same kind.
- We keep doing this until we reach a problem that is simple enough to be solved directly.
- This simplest problem is known as the *base case*.

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {           // base case
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- The base case stops the recursion, because it doesn't make another call to the method.

Recursive Problem-Solving (cont.)

- If the base case hasn't been reached, we execute the *recursive case.*

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {           // base case
        System.out.println(n2);
    } else {                  // recursive case
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- The recursive case:
  - reduces the overall problem to one or more simpler problems of the same kind
  - makes recursive calls to solve the simpler problems

Structure of a Recursive Method

```
recursiveMethod(parameters) {
    if (stopping condition) {
        // handle the base case
    } else {
        // recursive case:
        // possibly do something here

        recursiveMethod(modified parameters);

        // possibly do something here
    }
}
```

- There can be multiple base cases and recursive cases.
- When we make the recursive call, we typically use parameters that bring us closer to a base case.

Tracing a Recursive Method: Second Example

```
public static void mystery(int i) {
    if (i <= 0) {     // base case
        return;
    }
    // recursive case
    System.out.println(i);
    mystery(i - 1);
    System.out.println(i);
}
```

- What happens when we execute `mystery(2)`?

A Recursive Method That Returns a Value

- Simple example: summing the integers from 1 to n

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

- Example of this approach to computing the sum:

sum(6) = 6 + sum(5)
= 6 + 5 + sum(4)
…

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

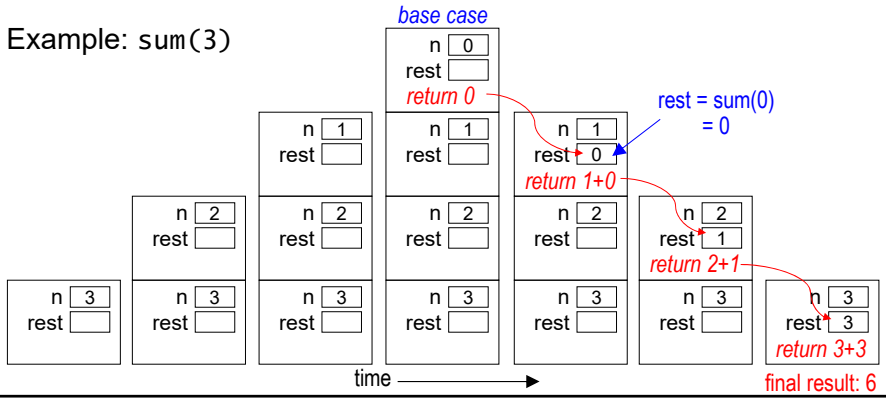
```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
        sum(2) returns 2 + 1 or 3
    sum(3) returns 3 + 3 or 6
main() assigns 6 to x
```

Tracing a Recursive Method on the Stack

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

The final result gets built up ***on the way back*** from the base case!

Example: `sum(3)`



Infinite Recursion

- We have to ensure that a recursive method will eventually reach a base case, regardless of the initial input.
- Otherwise, we can get *infinite recursion*.
  - produces *stack overflow* - there's no room for more frames on the stack!
- Example: here's a version of our `sum()` method that uses a different test for the base case:

```
public static int sum(int n) {
    if (n == 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

  - what values of n would cause infinite recursion?

Designing a Recursive Method

1. Start by programming the base case(s).
   - *What instance(s) of this problem can I solve directly (without looking at anything smaller)?*
2. Find the recursive substructure.
   - *How could I use the solution to **any smaller version** of the problem to solve the overall problem?*
3. Solve the smaller problem using a recursive call!
   - store its result in a variable
4. Do your one step.
   - build your solution from the result of the recursive call
   - use *concrete cases* to figure out what you need to do

Processing a String Recursively

- A string is a recursive data structure. It is either:
  - empty ("")
  - a single character, followed by a string
- Thus, we can easily use recursion to process a string.
  - process one or two of the characters ourselves
  - make a recursive call to process the rest of the string
- Example: print a string vertically, one character per line:

```
public static void printVertical(String str) {
    if (str == null || str.equals("")) {
        return;
    }

    System.out.println(str.charAt(0)); // first char
    printVertical(str.substring(1));   // rest of string
}
```

Short-Circuited Evaluation

- The second operand of both the && and || operators will not be evaluated if the result can be determined on the basis of the first operand alone.

- `expr1 || expr2`
  if `expr1` evaluates to `true`, `expr2` is not evaluated, because we already know that `expr1 || expr2` is `true`
  - example from the last slide:

```
if (str == null || str.equals("")) {
    return;
}
// if str is null, we won't check for empty string.
// This prevents a null pointer exception!
```

- `expr1 && expr2`
  if `expr1` evaluates to _____, `expr2` is not evaluated, because we already know that `expr1 && expr2` is _____.

Counting Occurrences of a Character in a String

- `numOccur(c, s)` should return the number of times that the character `c` appears in the string `s`
  - `numOccur('n', "banana")` should return 2
  - `numOccur('a', "banana")` should return 3

- Take the approach outlined earlier:
  - base case: empty string (or null)
  - delegate `s.substring(1)` to the recursive call
  - we're responsible for handling `s.charAt(0)`

Recursive Counting

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        // do our one step!
        ...
```

Determining Our One Step

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        // do our one step!
```

- In our one step, we take care of `s.charAt(0)`.
  - we build the solution to the larger problem on the solution to the smaller problem (in this case, `rest`)
  - does what we do depend on the value of `s.charAt(0)`?

- ***Use concrete cases to figure out the logic!***

Consider this concrete case…

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        // do our one step!
        ...

numOccur('r', "recurse")
```

What value is eventually assigned to `rest`? (i.e., what does the recursive call return?)

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        // do our one step!
        ...

numOccur('r', "recurse")
```

---

numOccur('r', "recurse")
c = 'r', s = "recurse"
int rest = ???

Consider Concrete Cases

```
numOccur('r', "recurse")     # first char is a match
```

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

```
numOccur('a', "banana")     # first char is not a match
```

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

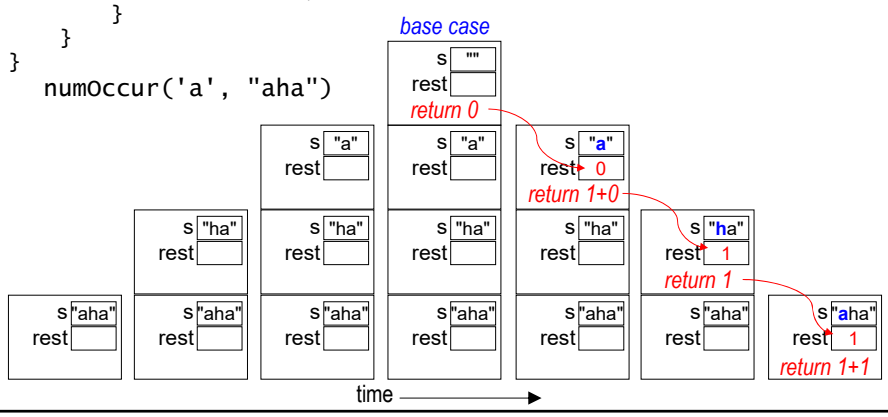
Now complete the method!

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        if (s.charAt(0) == c) {

            return ____________________;
        } else {

            return ____________________;
        }
    }
}
```

Tracing a Recursive Method on the Stack

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        if (s.charAt(0) == c) {
            return 1 + rest;
        } else {
            return rest;
        }
    }
}
   numOccur('a', "aha")
```

The final result gets built up ***on the way back*** from the base case!



Common Mistake

- This version of the method does *not* work:

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    }

    int count = 0;
    if (s.charAt(0) == c) {
        count++;
    }

    numOccur(c, s.substring(1));
    return count;
}
```

Another Faulty Approach

- Some people make `count` "global" to fix the prior version:

```
public static int count = 0;

public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    }

    if (s.charAt(0) == c) {
        count++;
    }

    numOccur(c, s.substring(1));
    return count;
}
```

- Not recommended, and not allowed on the problem sets!
- Problems with this approach?

Removing Vowels From a String

- `removeVowels(s)` - removes the vowels from the string `s`, returning its "vowel-less" version!

  `removeVowels("recursive")` should return `"rcrsv"`

  `removeVowels("vowel")` should return `"vwl"`

- Can we take the usual approach to recursive string processing?
  - base case: empty string
  - delegate `s.substring(1)` to the recursive call
  - we're responsible for handling `s.charAt(0)`

Applying the String-Processing Template

```
public static String removeVowels(String s) {
    if (s.equals("")) {     // base case

        return __________;
    } else {                // recursive case

        String rem_rest = ____________________;
        // do our one step!
    }
}
```

Consider Concrete Cases

```
removeVowels("after")     # first char is a vowel
```

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

```
removeVowels("recurse")   # first char is not a vowel
```

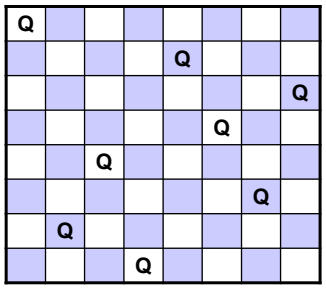
- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

removeVowels()

```
public static String removeVowels(String s) {
    if (s.equals("")) {       // base case
        return "";
    } else {                  // recursive case
        String rem_rest = removeVowels(s.substring(1));
        if ("aeiou".indexOf(s.charAt(0)) != -1) {

            _____________________________________

        } else {

            _____________________________________

        }
    }
}
```

The n-Queens Problem

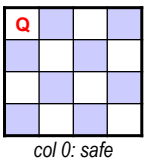
- **Goal:** to place n queens on an n x n chessboard so that no two queens occupy:
  - the same row
  - the same column
  - the same diagonal.
- Sample solution for n = 8:



- This problem can be solved using a technique called *recursive backtracking*.

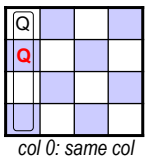
Recursive Strategy for n-Queens

- `findSolution(row)` – to place a queen in the specified `row`:
  - try one column at a time, looking for a "safe" one
  - if we find one: – place the queen there
    – *make a recursive call* to go to the next row
  - if we can't find one: – *backtrack* by returning from the call
    – try to find another safe column in the previous row
- Example:
  - row 0:

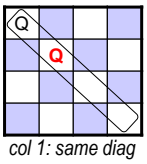


col 0: safe

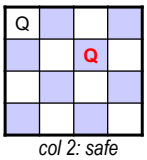
  - row 1:



col 0: same col



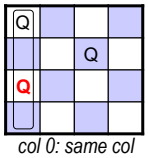
col 1: same diag



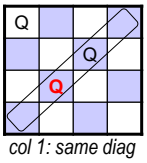
col 2: safe

4-Queens Example (cont.)

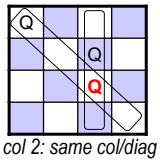
- row 2:



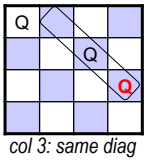
col 0: same col



col 1: same diag

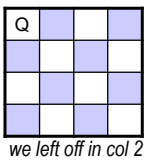


col 2: same col/diag

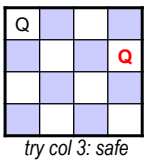


col 3: same diag

  - We've run out of columns in row 2!
  - *Backtrack* to row 1 by returning from the recursive call.
    - pick up where we left off
    - we had already tried columns 0-2, so now we try column 3:



we left off in col 2

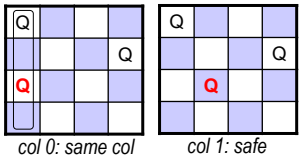


try col 3: safe

  - Continue the recursion as before.

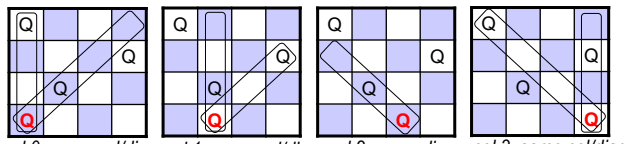
4-Queens Example (cont.)

- row 2:



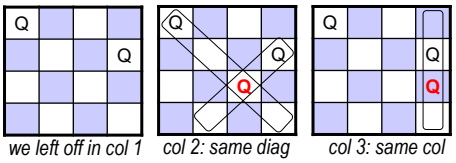
col 0: same col *col 1: safe*

- row 3:



col 0: same col/diag *col 1: same col/diag* *col 2: same diag* *col 3: same col/diag*

- Backtrack to row 2:

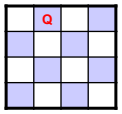


we left off in col 1 *col 2: same diag* *col 3: same col*

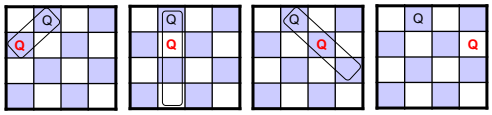
- Backtrack to row 1. No columns left, so backtrack to row 0!

4-Queens Example (cont.)

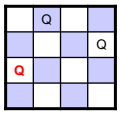
- row 0:



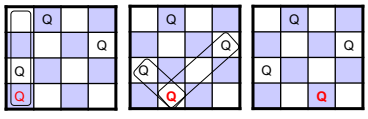
- row 1:



- row 2:



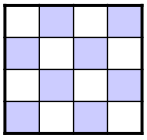
- row 3:



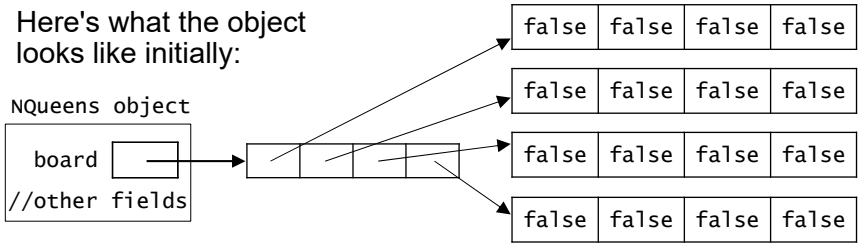
A solution!

A Blueprint Class for an N-Queens Solver

```
public class NQueens {
    private boolean[][] board;  // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    ...
```

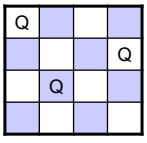


- Here's what the object looks like initially:

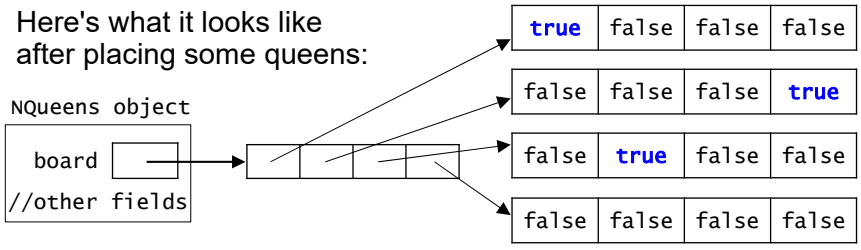


A Blueprint Class for an N-Queens Solver

```
public class NQueens {
    private boolean[][] board;  // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    private void placeQueen(int row, int col) {
        this.board[row][col] = true;
        // modify other fields here...
    }
```



- Here's what it looks like after placing some queens:



A Blueprint Class for an N-Queens Solver

```
public class NQueens {
    private boolean[][] board;  // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    private void placeQueen(int row, int col) {
        this.board[row][col] = true;
        // modify other fields here...
    }

    private void removeQueen(int row, int col){
        this.board[row][col] = false;
        // modify other fields here...
    }

    private boolean isSafe(int row, int col) {
        // returns true if [row][col] is "safe", else false
    }

    private boolean findSolution(int row) {
        // see next slide!
    ...
```

private helper methods that will only be called by code within the class.

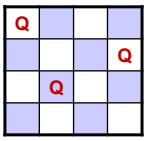
Making them private means we don't need to do error-checking!

Recursive-Backtracking Method

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```

- takes the index of a row (initially 0)
- uses a loop to consider all possible columns in that row
- makes a recursive call to move onto the next row
- returns `true` if a solution has been found; `false` otherwise

Tracing findSolution()




```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        // code to process a solution goes here...
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```

Note: row++ will *not* work here!

We can pick up where we left off, because row *and* col *are stored in the stack frame!*

backtrack!

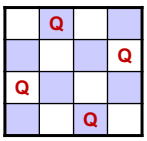
backtrack!

row: 0 col: 0	row: 0 col: 0	row: 0 col: 0	row: 0 col: 0	row: 0 col: 0	row: 0 col: 0	row: 0 col: 0
	row: 1 col: 0,1,2	row: 1 col: 0,1,2	row: 1 col: 0,1,2	row: 1 col: 0,1,2,**3**	row: 1 col: 0,1,2,3	row: 1 col: 0,1,2,3
		row: 2 col:0,1,2,3,4 return false			row: 2 col: 0,1	row: 2 col: 0,1
						row: 3 col:**0,1,2,3,4** return false

…

time →

Once we place a queen in the last row…



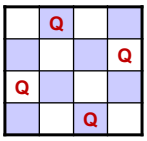

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```

…

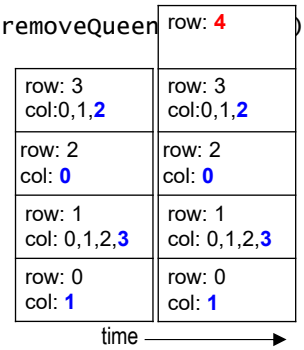
row: 3 col:0,1,**2**
row: 2 col: **0**
row: 1 col: 0,1,2,**3**
row: 0 col: **1**

time →

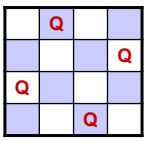
…we make one more recursive call…



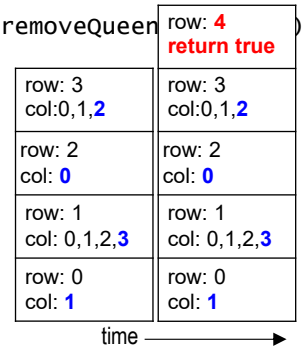

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen                    );
        }
    }
    return false;
}
```



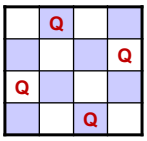
…and hit the base case!



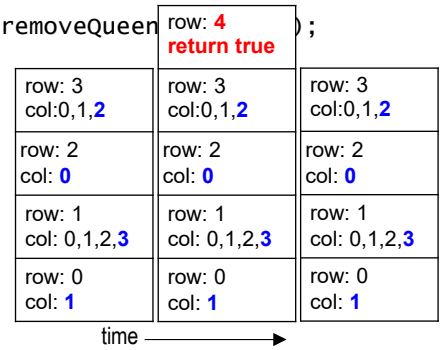

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen                    );
        }
    }
    return false;
}
```



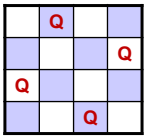
`true` is sent back…



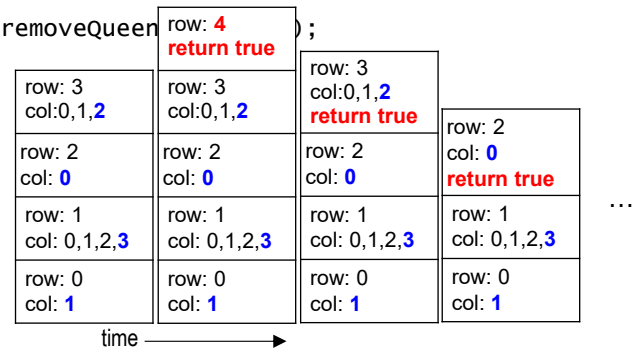

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) { // if (true)
                return true;
            }
            this.removeQueen                    );
        }
    }
    return false;
}
```



…and all the earlier calls also return `true`!




```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) { // if (true)
                return true;
            }
            this.removeQueen                    );
        }
    }
    return false;
}
```



Using a "Wrapper" Method

- The key recursive method is private:

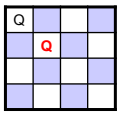
```
private boolean findSolution(int row) {
    ...
}
```

- We use a separate, public *"wrapper" method* to start the recursion:

```
public boolean findSolution() {
    return this.findSolution(0);
}
```

  - an example of overloading – two methods with the same name, but different parameters
  - this method takes no parameters
  - it makes the initial call to the recursive method and returns whatever that call returns
  - it allows us to ensure that the correct initial value is passed into the recursive method

Recursive Backtracking in General

- Useful for *constraint satisfaction problems*
  - involve assigning values to variables according to a set of constraints
  - n-Queens: variables = Queen's position in each row
    constraints = no two queens in same row/col/diag
  - many others: factory scheduling, room scheduling, etc.
- Backtracking greatly reduces the number of possible solutions that we consider.
  - ex: 
    - there are 16 possible solutions that begin with queens in these two positions
    - backtracking doesn't consider any of them!
- Recursion makes it easy to handle an arbitrary problem size.
  - stores the state of each variable in a separate stack frame

Template for Recursive Backtracking

```
// n is the number of the variable that the current
// call of the method is responsible for
boolean findSolution(int n, possibly other params) {
    if (found a solution) {
        this.displaySolution();
        return true;
    }
    // loop over possible values for the nth variable
    for (val = first to last) {
        if (this.isValid(val, n)) {
            this.applyValue(val, n);
            if (this.findSolution(n + 1, other params)) {
                return true;
            }
            this.removeValue(val, n);
        }
    }
    return false;     // backtrack!
}
```

Note: n++ will *not* work here!

Template for Finding Multiple Solutions

(up to some target number of solutions)

```
boolean findSolutions(int n, possibly other params) {
    if (found a solution) {
        this.displaySolution();
        this.solutionsFound++;
        return (this.solutionsFound >= this.target);
    }

    // loop over possible values for the nth variable
    for (val = first to last) {
        if (isValid(val, n)) {
            this.applyValue(val, n);
            if (this.findSolutions(n + 1, other params)) {
                return true;
            }
            this.removeValue(val, n);
        }
    }

    return false;
}
```

Data Structures for n-Queens

- Three key operations:
  - `isSafe(row, col)`: check to see if a position is safe
  - `placeQueen(row, col)`
  - `removeQueen(row, col)`
- In theory, our 2-D array of booleans would be sufficient:

```
public class NQueens {
    private boolean[][] board;
```

- It's easy to place or remove a queen:

```
    private void placeQueen(int row, int col) {
        this.board[row][col] = true;
    }
    private void removeQueen(int row, int col) {
        this.board[row][col] = false;
    }
…
```

- Problem: `isSafe()` takes a lot of steps. What matters more?

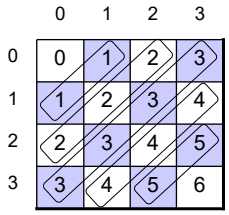
Additional Data Structures for n-Queens

- To facilitate `isSafe()`, add three arrays of booleans:

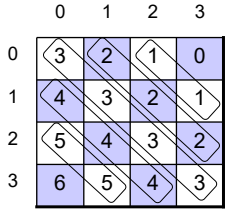
```
private boolean[] colEmpty;
private boolean[] upDiagEmpty;
private boolean[] downDiagEmpty;
```

- An entry in one of these arrays is:
  - `true` if there are no queens in the column or diagonal
  - `false` otherwise
- Numbering diagonals to get the indices into the arrays:

upDiag = row + col



downDiag = (boardSize – 1) + row – col



Using the Additional Arrays

- Placing and removing a queen now involve updating four arrays instead of just one. For example:

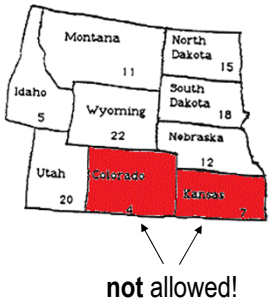
```
private void placeQueen(int row, int col) {
    this.board[row][col] = true;
    this.colEmpty[col] = false;
    this.upDiagEmpty[row + col] = false;
    this.downDiagEmpty[
         (this.board.length - 1) + row - col] = false;
}
```

- However, checking if a square is safe is now more efficient:

```
private boolean isSafe(int row, int col) {
    return (this.colEmpty[col]
      && this.upDiagEmpty[row + col]
      && this.downDiagEmpty[
              (this.board.length - 1) + row - col]);
}
```

Recursive Backtracking II: Map Coloring

- We want to color a map using **only four colors**.
- Bordering states or countries ***cannot*** have the same color.
  - example:



not allowed!

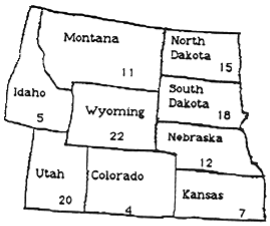
Applying the Template to Map Coloring

```
boolean findSolution(n, perhaps other params) {
    if (found a solution) {
        this.displaySolution();
        return true;
    }
    for (val = first to last) {
        if (this.isValid(val, n)) {
            this.applyValue(val, n);
            if (this.findSolution(n + 1, other params)) {
                return true;
            }
            this.removeValue(val, n);
        }
    }
    return false;
}
```

template element	*meaning in map coloring*
n	
found a solution	
val	
isValid(val, n)	
applyValue(val, n)	
removeValue(val, n)	

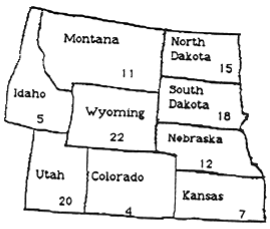
Map Coloring Example

consider the states in alphabetical order. colors = { red, yellow, green, blue }.



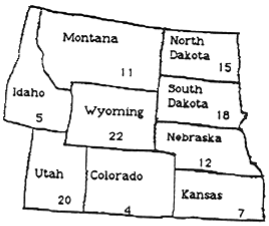
We color Colorado through Utah without a problem.

Colorado:
Idaho:
Kansas:
Montana:
Nebraska:
North Dakota:
South Dakota:
Utah:



No color works for Wyoming, so we backtrack…

Map Coloring Example (cont.)



Now we can complete the coloring:

Recursion vs. Iteration

- Some problems are much easier to solve using recursion.
- Other problems are just as easy to solve using iteration.
- Recursion is a bit more costly because of the overhead involved in invoking a method.
  - also: in some cases, there may not be room on the stack
- Rule of thumb:
  - if it's easier to formulate a solution recursively, use recursion, unless the cost of doing so is too high
  - otherwise, use iteration