Recursion Revisited; Recursive Backtracking

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Review: Recursive Problem-Solving

- When we use recursion, we *reduce* a problem 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.

Review: 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

Raising a Number to a Power

- We want to write a recursive method to compute

$$x^n = \underbrace{x*x*x*\ldots*x}_{n \text{ of them}}$$

  where *x* and *n* are both integers and *n* >= 0.

- Examples:
  - $2^{10} = 2*2*2*2*2*2*2*2*2*2 = 1024$
  - $10^5 = 10*10*10*10*10 = 100000$

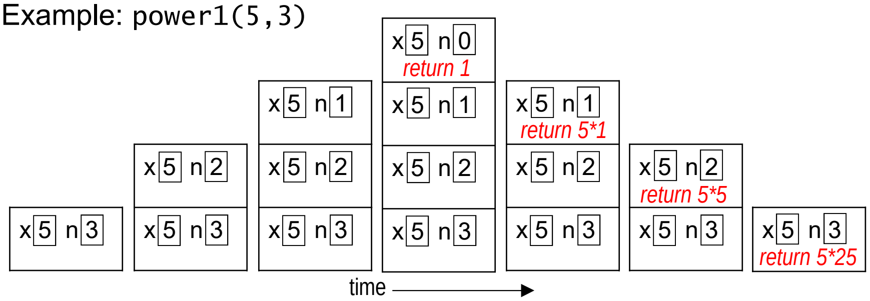
- Computing a power recursively: $2^{10} = 2*2^9$
  $= 2*(2 * 2^8)$
  $= \ldots$

- Recursive definition: $x^n = x * x^{n-1}$ when $n > 0$
  $x^0 = 1$

Power Method: First Try

```
public static int power1(int x, int n) {
    if (n < 0) {
        throw new IllegalArgumentException();
    } else if (n == 0) {
        return 1;
    } else {
        int pow_rest = power1(x, n-1);
        return x * pow_rest;
    }
}
```

Example: `power1(5,3)`



Power Method: Second Try

- There's a better way to break these problems into subproblems.
  For example: $2^{10} = (2*2*2*2*2)*(2*2*2*2*2)$
  $= (2^5) * (2^5) = (2^5)^2$
- A more efficient recursive definition of x^n (when $n > 0$):
  $x^n = (x^{n/2})^2$ when n is even
  $x^n = x * (x^{n/2})^2$ when n is odd (using integer division for n/2)

```
public static int power2(int x, int n) {
    // code to handle n < 0 goes here...








}
```

Analyzing power2

- How many method calls would it take to compute 2^{1000} ?

```
power2(2, 1000)
  power2(2, 500)
    power2(2, 250)
      power2(2, 125)
        power2(2, 62)
          power2(2, 31)
            power2(2, 15)
              power2(2, 7)
                power2(2, 3)
                  power2(2, 1)
                    power2(2, 0)
```

- Much more efficient than power1() for large n.
- It can be shown that it takes approx. $\log_2 n$ method calls.

An Inefficient Version of power2

- What's wrong with the following version of power2()?

```
public static int power2(int x, int n) {
    // code to handle n < 0 goes here...
    if (n == 0) {
        return 1;
    } else {
        // int pow_rest = power2(x, n/2);
        if (n % 2 == 0) {
            return power2(x, n/2) * power2(x, n/2);
        } else {
            return x * power2(x, n/2) * power2(x, n/2);
        }
    }
}
```

Review: 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
}
```

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()

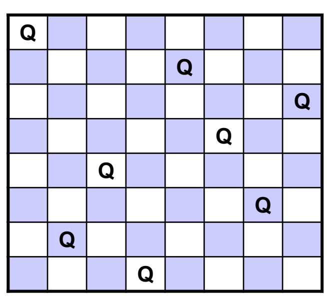
```
public static String removeVowels(String s) {
    if (s.equals("")) {       // base case
        return "";
    } else {                 // recursive case
        String rem_rest = removeVowels(s.substring(1));
        if ("aeiou".indexOf(s.charAt(0)) != -1) {

            ____________________________________
        } else {

            ____________________________________
        }
    }
}
```

The n-Queens Problem

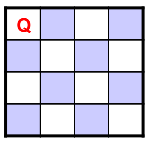
- **Goal:** to place n queens on an n x n chessboard so that no two queens occupy:
  - the same row
  - the same column
  - the same diagonal.
- Sample solution for n = 8:



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

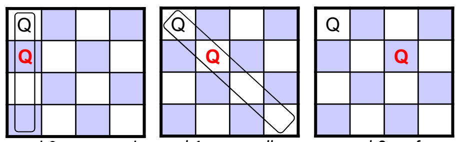
Recursive Strategy for n-Queens

- `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

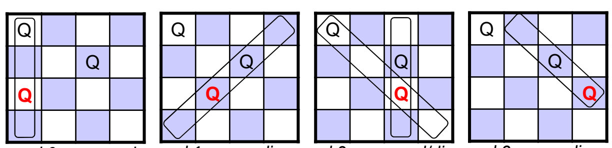
  - row 1:



col 0: same col *col 1: same diag* *col 2: safe*

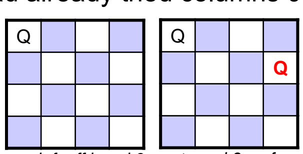
4-Queens Example (cont.)

- 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:

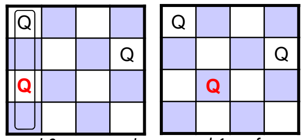


we left off in col 2 *try col 3: safe*

- Continue the recursion as before.

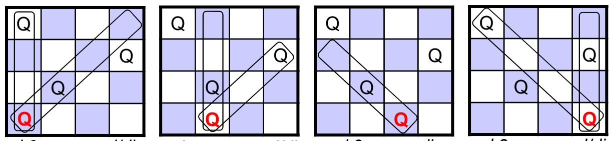
4-Queens Example (cont.)

- row 2:



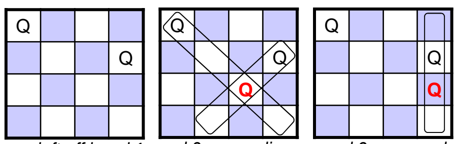
col 0: same col | col 1: safe

- 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:

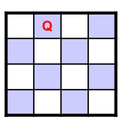


we left off in col 1 | col 2: same diag | col 3: same col

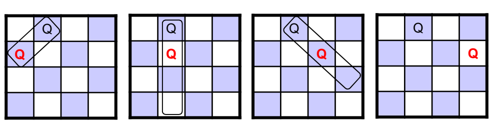
- Backtrack to row 1. No columns left, so backtrack to row 0!

4-Queens Example (cont.)

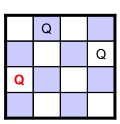
- row 0:



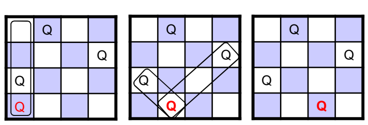
- row 1:



- row 2:



- row 3:

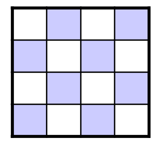


A solution!

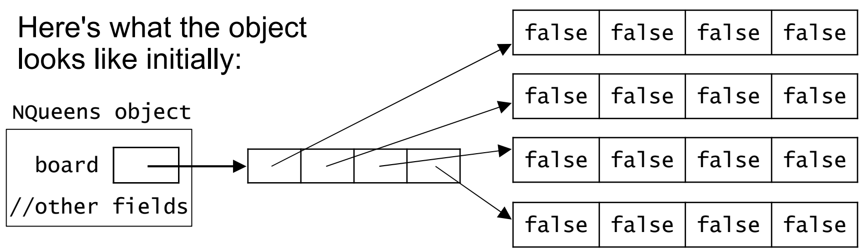
A Blueprint Class for an N-Queens Solver

```
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:

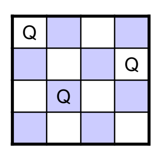


A Blueprint Class for an N-Queens Solver

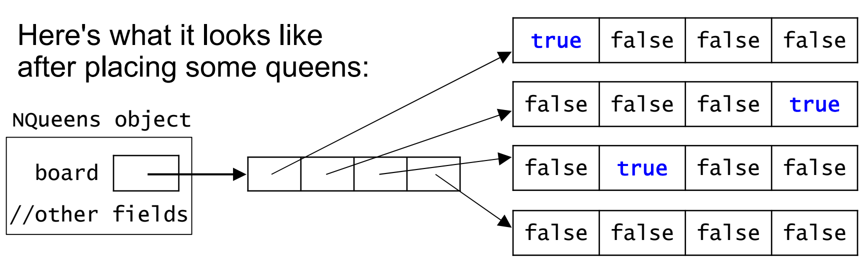
```
public class NQueens {
    private boolean[][] board;  // state of the chessboard
    // other fields go here...

    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.

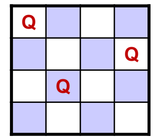
Making them private means we don't need to do error-checking!

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!

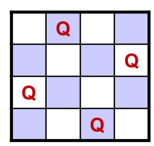
We can pick up where we left off, because row and col are stored in the stack frame!

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

...

time →

Once we place a queen in the last row…



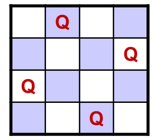

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row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

...

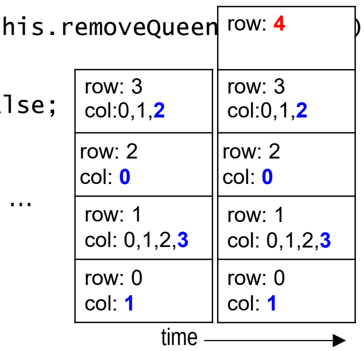
row: 3 col:0,1,**2**
row: 2 col: **0**
row: 1 col: 0,1,2,**3**
row: 0 col: **1**

time →

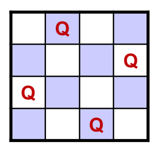
…we make one more recursive call…



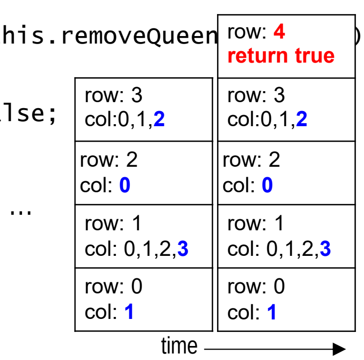
```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row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



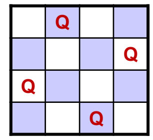
…and hit the base case!



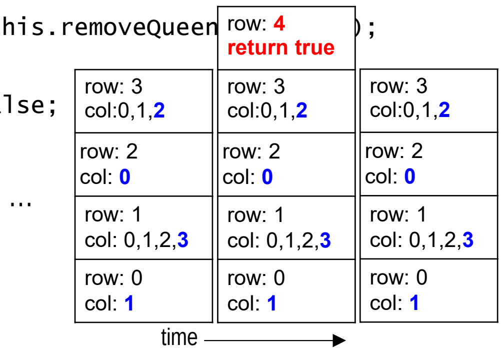
```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row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



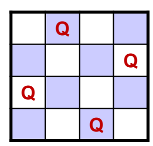
`true` is sent back…



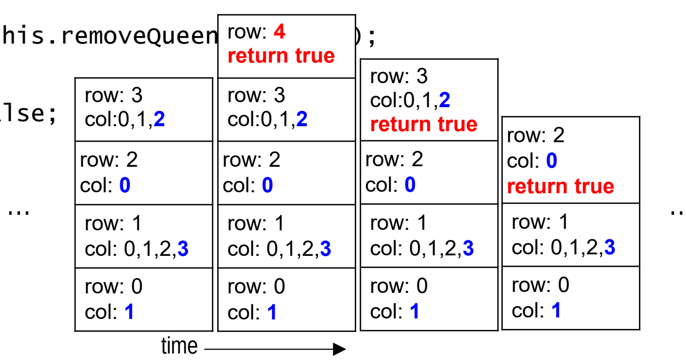

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row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

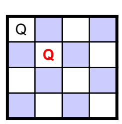
- We use a separate, public *"wrapper" method* to start the recursion:

```
public boolean findSolution() {
    return 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?

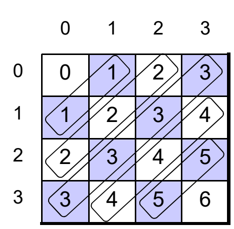
Additional Data Structures for n-Queens

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

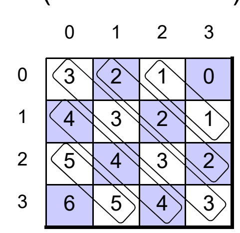
```
    private boolean[] colEmpty;
    private boolean[] upDiagEmpty;
    private boolean[] downDiagEmpty;
```

- 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:

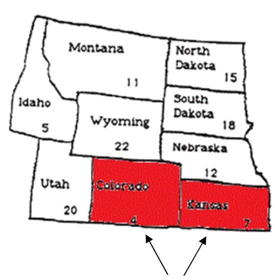
```
private void placeQueen(int row, int col) {
    this.board[row][col] = true;
    this.colEmpty[col] = false;
    this.upDiagEmpty[row + col] = false;
    this.downDiagEmpty[
         (this.board.length - 1) + row - col] = false;
}
```

- 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!

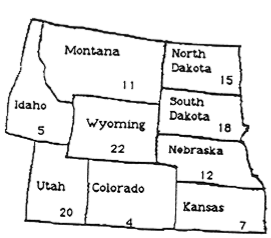
Applying the Template to Map Coloring

```
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)	

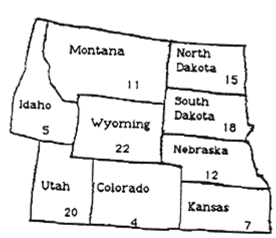
Map Coloring Example

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



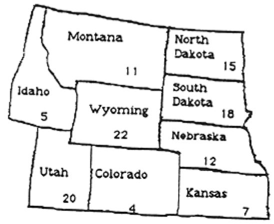
We color Colorado through Utah without a problem.

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: