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)`?

Printing a File to the Console

- Here's a method that prints a file using iteration:

```
public static void print(Scanner input) {
    while (input.hasNextLine()) {
        System.out.println(input.nextLine());
    }
}
```

- Here's a method that uses recursion to do the same thing:

```
public static void printRecursive(Scanner input) {
    // base case
    if (!input.hasNextLine()) {
        return;
    }

    // recursive case
    System.out.println(input.nextLine());
    printRecursive(input);  // print the rest
}
```

Printing a File in Reverse Order

- What if we want to print the lines of a file in reverse order?
- It's not easy to do this using iteration. Why not?
- It's easy to do it using recursion!
- How could we modify our previous method to make it print the lines in reverse order?

```
public static void printRecursive(Scanner input) {
    if (!input.hasNextLine()) {   // base case
        return;
    }

    String line = input.nextLine();
    System.out.println(line);
    printRecursive(input);  // print the rest
}
```

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 total = n + sum(n – 1);
    return total;
}
```

- 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 total = n + sum(n – 1);
    return total;
}
```

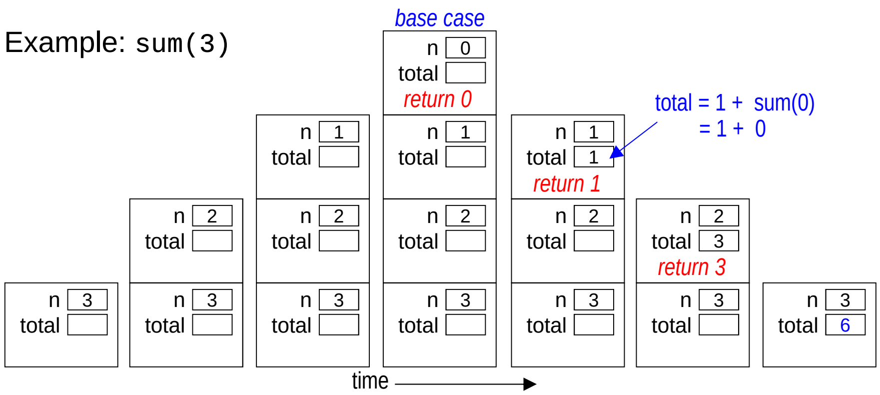
- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
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()
```

Tracing a Recursive Method on the Stack

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

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 total = n + sum(n – 1);
    return total;
}
```

  - what values of n would cause infinite recursion?

Thinking Recursively

- When solving a problem using recursion, ask yourself these questions:
  1. How can I break this problem down into one or more smaller subproblems?
     - make recursive method calls to solve the subproblems
  2. What are the base cases?
     - i.e., which subproblems are small enough to solve directly?
  3. Do I need to combine the solutions to the subproblems? If so, how should I do so?

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:

$$\begin{aligned} 2^{10} &= 2*2^9 \\ &= 2*(2 * 2^8) \\ &= \ldots \end{aligned}$$

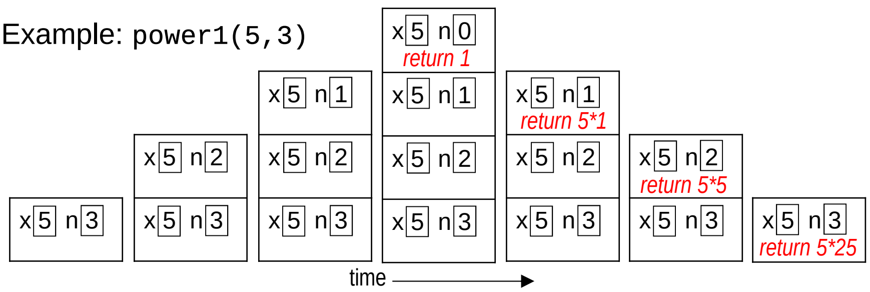
- Recursive definition:

$$x^n = x * x^{n-1} \text{ when } n > 0$$
$$x^0 = 1$$

Power Method: First Try

```
public class Power {
    public static int power1(int x, int n) {
        if (n < 0)
            throw new IllegalArgumentException(
              “n must be >= 0”);
        if (n == 0)
            return 1;
        else
            return x * power1(x, n-1);
    }
}
```

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)
- Let's write the corresponding method together:

```
public static int power2(int x, int n) {









}
```

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 power2Bad(int x, int n) {
    // code to handle n < 0 goes here...
    if (n == 0)
        return 1;
    if ((n % 2) == 0)
        return power2(x, n/2) * power2(x, n/2);
    else
        return x * power2(x, n/2) * power2(x, n/2);
}
```

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
  - 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
}
```

Counting Occurrences of a Character in a String

- Let's design a recursive method called `numOccur()`.
- `numOccur(ch, str)` should return the number of times that the character `ch` appears in the string `str`
- Thinking recursively:

Counting Occurrences of a Character in a String (cont.)

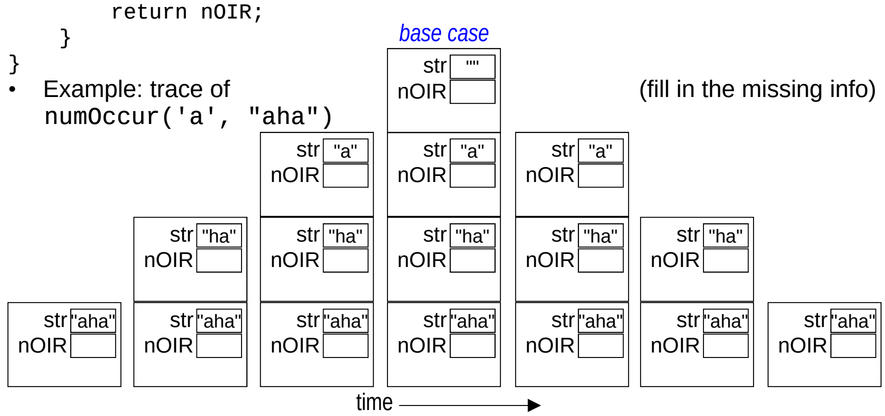
- Put the method definition here:

Tracing a Recursive Method on the Stack

```
public static int numOccur(char ch, String str) {
    if (str == null || str.equals("")) {
        return 0;
    }
    int nOIR = numOccur(ch, str.substring(1));
    if (str.charAt(0) == ch) {
        return 1 + nOIR;
    } else {
        return nOIR;
    }
}
```

- Example: trace of `numOccur('a', "aha")`

(fill in the missing info)



Common Mistake

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

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

    int count = 0;
    if (str.charAt(0) == ch) {
        count++;
    }

    numOccur(ch, str.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 ch, String str) {
    if (str == null || str.equals("")) {
        return 0;
    }
    if (str.charAt(0) == ch) {
        count++;
    }
    numOccur(ch, str.substring(1));
    return count;
}
```

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

Removing Vowels from a String

- Let's design a recursive method called `removeVowels()`.
- `removeVowels(str)` should return a string in which all of the vowels in the string `str` have been removed.
  - example:
    `removeVowels("recurse")`

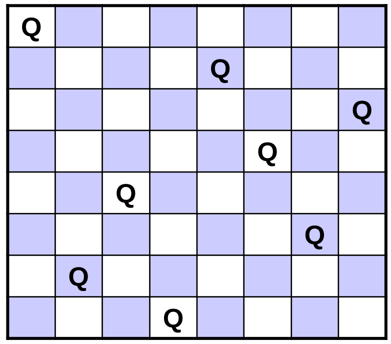
    should return
    `"rcrs"`
- Thinking recursively:

Removing Vowels from a String (cont.)

- Put the method definition here:

Recursive Backtracking: the n-Queens Problem

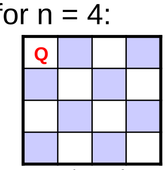
- Find all possible ways of placing n queens on an n x n chessboard so that no two queens occupy the same row, column, or diagonal.
- Sample solution for n = 8:



- This is a classic example of a problem that can be solved using a technique called *recursive backtracking*.

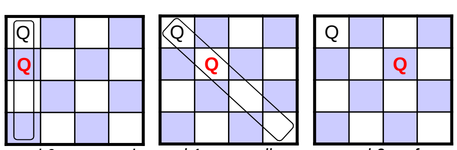
Recursive Strategy for n-Queens

- Consider one row at a time. Within the row, consider one column at a time, looking for a "safe" column to place a queen.
- If we find one, place the queen, and *make a recursive call* to place a queen on the next row.
- If we can't find one, *backtrack* by returning from the recursive call, and try to find another safe column in the previous row.
- Example for n = 4:
  - row 0:



col 0: safe

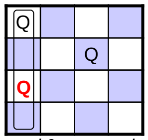
  - row 1:

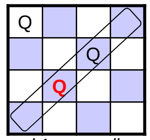


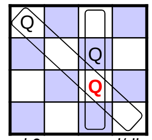
col 0: same col *col 1: same diag* *col 2: safe*

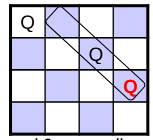
4-Queens Example (cont.)

- row 2:

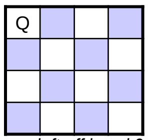

col 0: same col

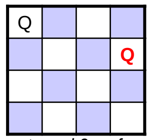

col 1: same diag


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:

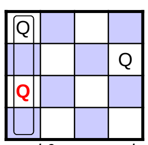

we left off in col 2

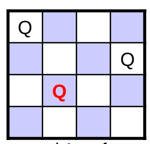

try col 3: safe

- Continue the recursion as before.

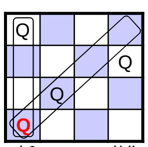
4-Queens Example (cont.)

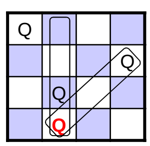
- row 2:

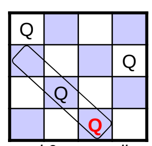

col 0: same col

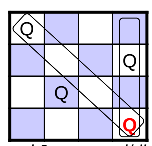

col 1: safe

- row 3:

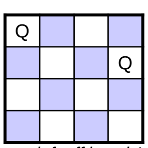

col 0: same col/diag

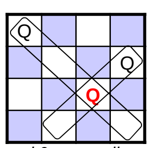

col 1: same col/diag

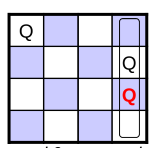

col 2: same diag


col 3: same col/diag

- Backtrack to row 2:


we left off in col 1

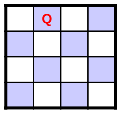

col 2: same diag


col 3: same col

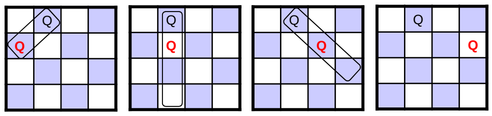
- Backtrack to row 1. No columns left, so backtrack to row 0!

4-Queens Example (cont.)

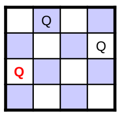
- row 0:



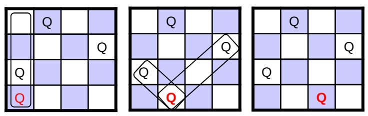
- row 1:



- row 2:



- row 3:



A solution!

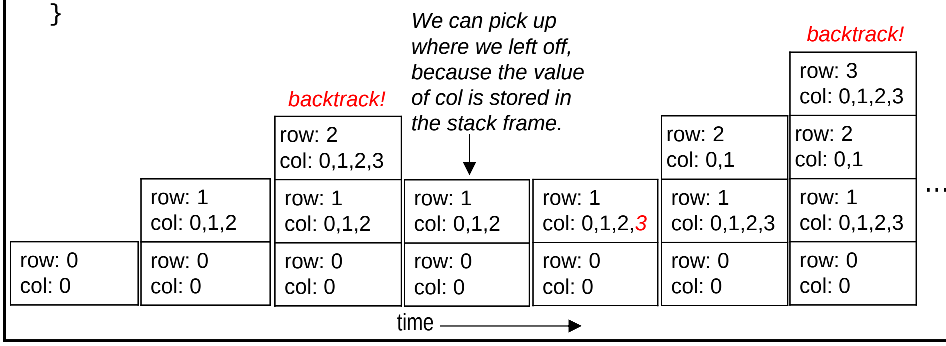
findSafeColumn() Method

```
public void findSafeColumn(int row) {
    if (row == boardSize) {  // base case: a solution!
        solutionsFound++;
        displayBoard();
        if (solutionsFound >= solutionTarget)
            System.exit(0);
        return;
    }

    for (int col = 0; col < boardSize; col++) {
        if (isSafe(row, col)) {
            placeQueen(row, col);

            // Move onto the next row.
            findSafeColumn(row + 1);

            // If we get here, we've backtracked.
            removeQueen(row, col);
        }
    }
}
```

Note: neither row++ nor ++row will work here.

Tracing findSafeColumn()

```
public void findSafeColumn(int row) {
    if (row == boardSize) {
        // code to process a solution goes here...
    }
    for (int col = 0; col < BOARD_SIZE; col++) {
        if (isSafe(row, col)) {
            placeQueen(row, col);
            findSafeColumn(row + 1);
            removeQueen(row, col);
        }
    }
}
```



Template for Recursive Backtracking

```
void findSolutions(n, other params) {
    if (found a solution) {
        solutionsFound++;
        displaySolution();
        if (solutionsFound >= solutionTarget)
            System.exit(0);
        return;
    }

    for (val = first to last) {
        if (isValid(val, n)) {
            applyValue(val, n);
            findSolutions(n + 1, other params);
            removeValue(val, n);
        }
    }
}
```

Template for Finding a Single Solution

```
boolean findSolutions(n, other params) {
    if (found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val, n)) {
            applyValue(val, n);
            if (findSolutions(n + 1, other params))
                return true;
            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)`
- A two-dim. array of booleans would be sufficient:

```
public class Queens {
    private boolean[][] queenOnSquare;
```

- Advantage: easy to place or remove a queen:

```
    public void placeQueen(int row, int col) {
        queenOnSquare[row][col] = true;
    }
    public void removeQueen(int row, int col) {
        queenOnSquare[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

	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

downDiag = (boardSize – 1) + row – col

	0	1	2	3
0	3	2	1	0
1	4	3	2	1
2	5	4	3	2
3	6	5	4	3

Using the Additional Arrays

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

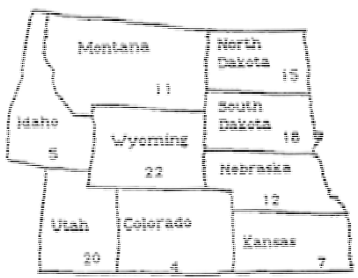
```
public void placeQueen(int row, int col) {
    queenOnSquare[row][col] = true;
    colEmpty[col] = false;
    upDiagEmpty[row + col] = false;
    downDiagEmpty[(boardSize - 1) + row - col] = false;
}
```

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

```
public boolean isSafe(int row, int col) {
    return (colEmpty[col]
      && upDiagEmpty[row + col]
      && downDiagEmpty[(boardSize - 1) + row - col]);
}
```

Recursive Backtracking II: Map Coloring

- Using just four colors (e.g., red, orange, green, and blue), we want color a map so that no two bordering states or countries have the same color.
- Sample map (numbers show alphabetical order in full list of state names):



- This is another example of a problem that can be solved using recursive backtracking.

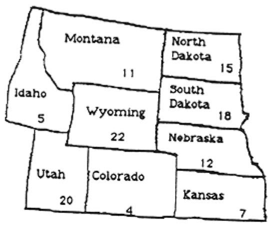
Applying the Template to Map Coloring

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

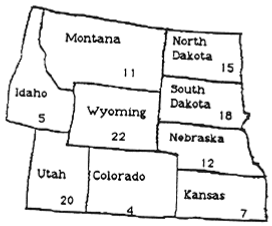
Map Coloring Example

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



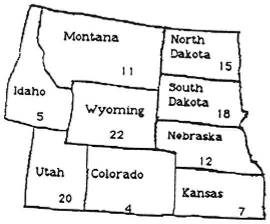
We color Colorado through Utah without a problem.

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:

Recursive Backtracking in General

- Useful for *constraint satisfaction problems* that 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, column, diagonal
  - map coloring
    - variables = each state's color
    - constraints = no two bordering states with the same color
  - many others: factory scheduling, room scheduling, etc.
- Backtracking reduces the # of possible value assignments that we consider, because it never considers invalid assignments....
- Using recursion allows us to easily handle an arbitrary number of variables.
  - stores the state of each variable in a separate stack frame

Recursion vs. Iteration

- Recursive methods can often be easily converted to a non-recursive method that uses iteration.
- This is especially true for methods in which:
  - there is only one recursive call
  - it comes at the end (tail) of the method

  These are known as *tail-recursive* methods.
- Example: an iterative sum() method.

```
public static int sum(n) {
    // handle negative values of n here
    int sum = 0;
    for (int i = 1; i <= n; i++)
        sum += i;
    return sum;
}
```

Recursion vs. Iteration (cont.)

- Once you're comfortable with using recursion, you'll find that some algorithms are easier to implement using recursion.
- We'll also see that some data structures lend themselves to recursive algorithms.
- Recursion is a bit more costly because of the overhead involved in invoking a method.
- 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