Recursion

Lecture 10a

Topics

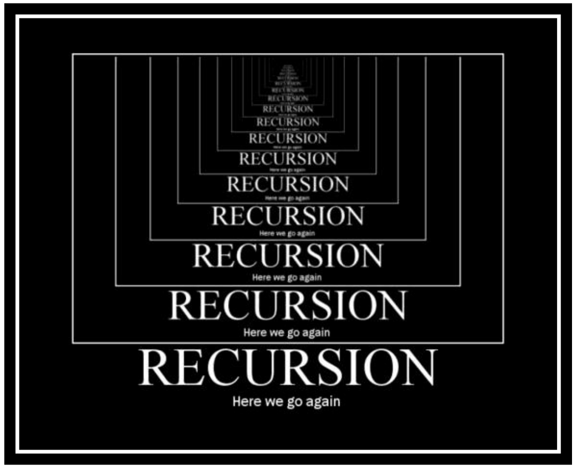
- Introduction to Recursion
- Recursive Methods
- Solving Problems with Recursion
- Simple Examples of Recursive Methods
- Direct and Indirect Recursion
- Summing a Range of Array Elements
- The Fibonacci Series
- Greatest Common Divisor
- A Recursive Binary Search Method
- The Towers of Hanoi

Introduction to Recursion (1 of 10)

- What is recursion?

Powerful technique for breaking up computational problems into simpler ones.

The term “recursion” means: the same computation **recurs**, or occurs repeatedly, as the problem is solved.



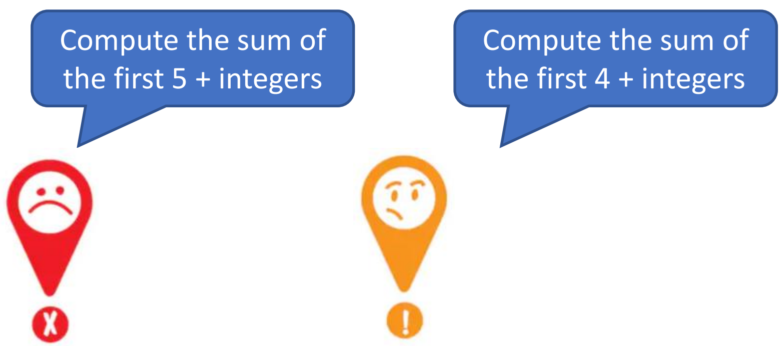
Recursion is often the most natural way of thinking about a problem, and some computations are difficult to perform without recursion.

Introduction to Recursion (2 of 10)

- Thinking Recursively

  - Suppose that you can solve the problem below by solving an identical but smaller problem. What problem would that be?

Step 1



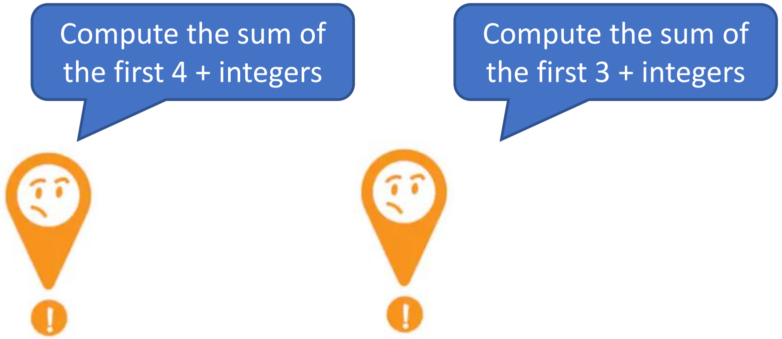
Step 2

Introduction to Recursion (3 of 10)

- Thinking Recursively

  - If you use this strategy again, you will need to solve an even smaller problem that is also similar to the original problem.

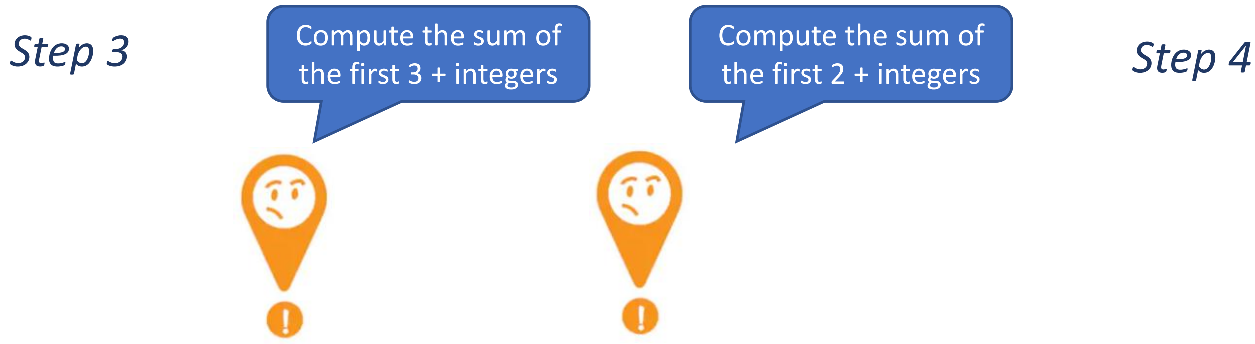
Step 2



Step 3

Introduction to Recursion (4 of 10)

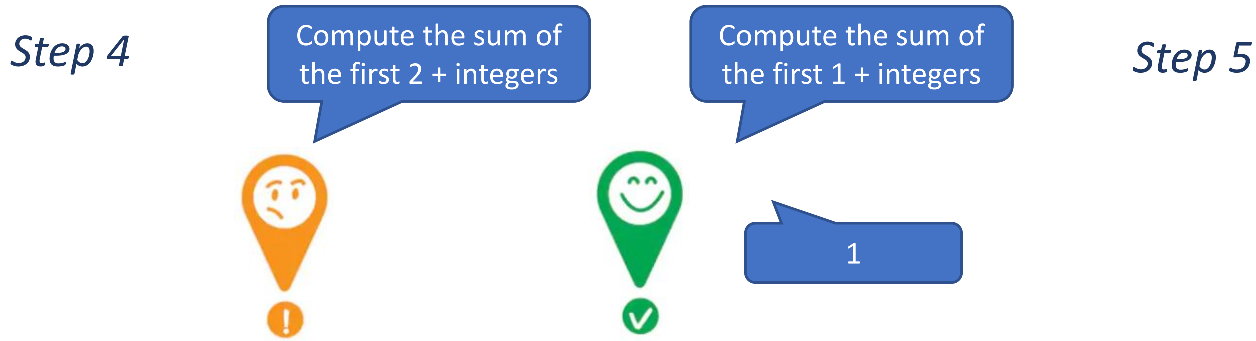
- Thinking Recursively
    - If you continue, you will need to solve an even smaller problem that is also similar to the original problem. How will replacing a problem with another one ever lead to a solution?



Introduction to Recursion (5 of 10)

- Thinking Recursively

  - One aspect to the success of this recursive strategy is that eventually you will reach a smaller problem whose solution you know because either it is obvious, or it is given.



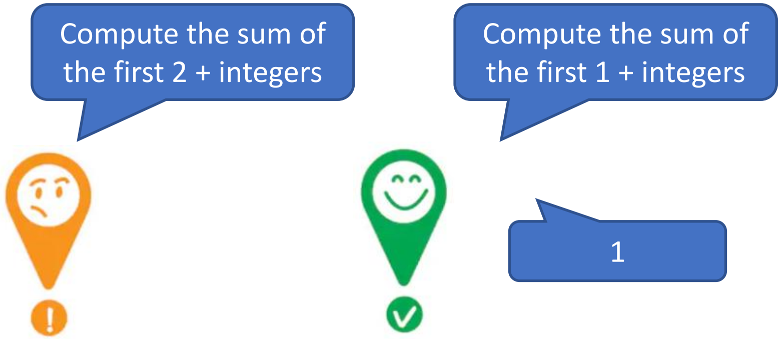
Introduction to Recursion (6 of 10)

- Thinking Recursively

  - The solution to this smallest problem is probably not the solution to your original problem, but it can help you to reach it. Can you guess how?

Step 4

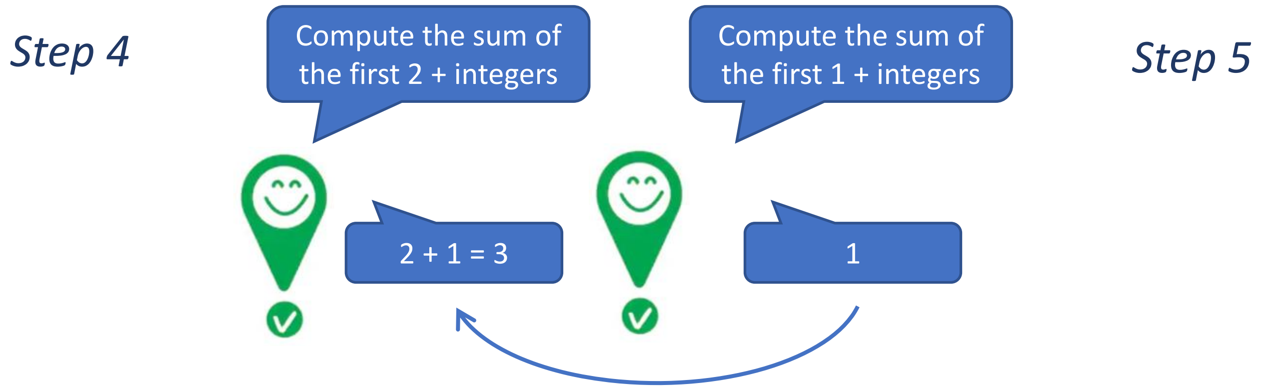
Step 5



Introduction to Recursion (7 of 10)

- Thinking Recursively

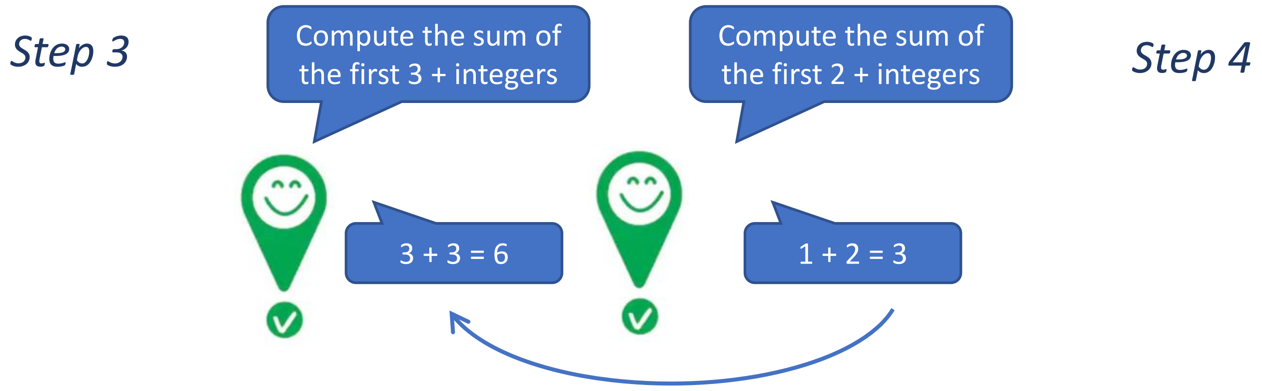
    ○ Either just before or just after you solve a smaller problem, you usually contribute a portion of the solution.



Introduction to Recursion (8 of 10)

- Thinking Recursively

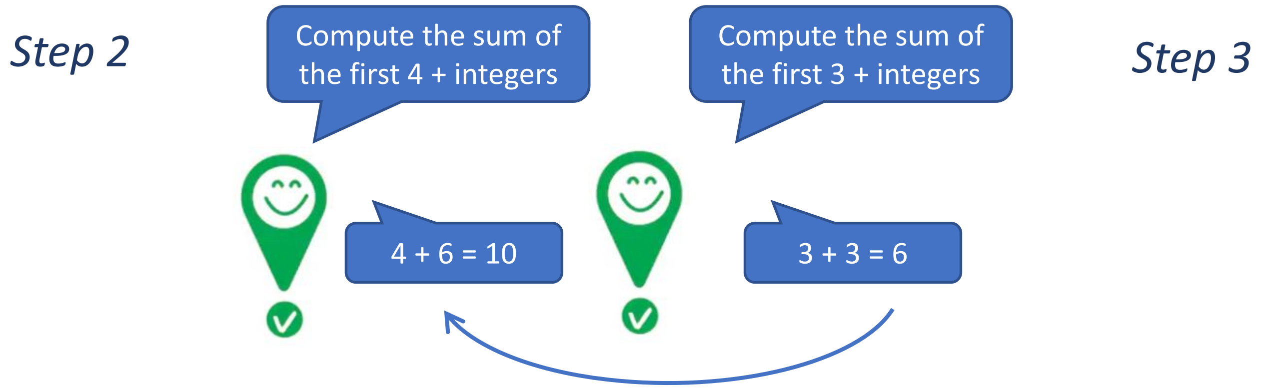
○ Either just before or just after you solve a smaller problem, you usually contribute a portion of the solution.



Introduction to Recursion (9 of 10)

- Thinking Recursively

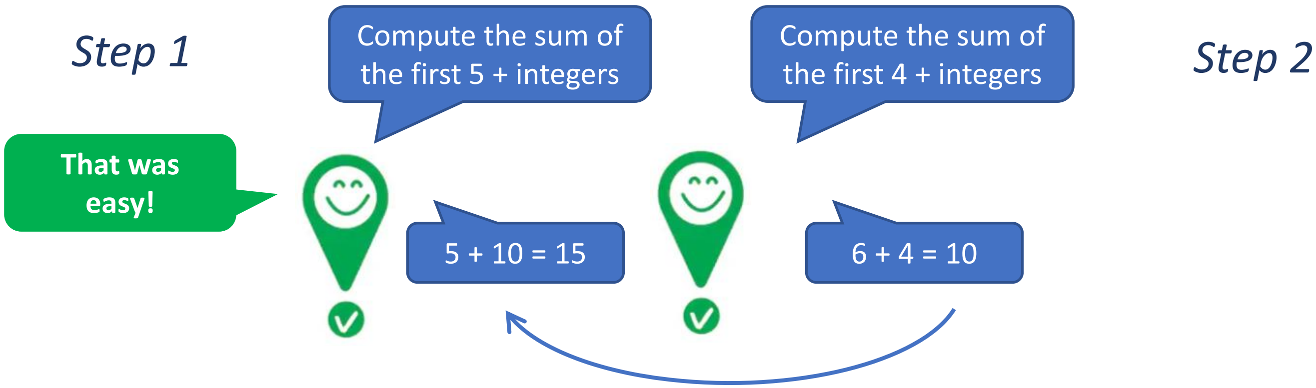
◦ Either just before or just after you solve a smaller problem, you usually contribute a portion of the solution.



Introduction to Recursion (10 of 10)

- Thinking Recursively

  ○ This portion, together with the solutions to the other, smaller problems, provides the solution to a larger problem.



Formally, if $S(N)$ is the sum of the first N^+ integers, then $S(1) = 1$ and $S(N) = \mathrm{N} + S(N-1)$, a recursive mathematical function.

Recursive Methods (1 of 18)

- So far, we have been calling other methods from a method. For instance, method A can call method B, which can then call method C.
- It's also possible for a method to call itself.
- A method that calls itself is a *recursive method*.
- The number of times that a method calls itself is known as the depth of recursion.

Recursive Methods (2 of 18)

Example: [EndlessRecursion.java]

```
/**
This class has a recursive method.
*/

public class EndlessRecursion
{
  public static void message()
  {
    System.out.println("This is a recursive method.");
    message();
  }
}
```

Recursive Methods (3 of 18)

- This method in the example displays the string “This is a recursive method.”, and then calls itself.
- Each time it calls itself, the cycle is repeated endlessly.
- Like a loop, a recursive method must have some way to control the number of times it repeats.
- Example: Recursive.java, RecursionDemo.java

Recursive Methods (4 of 18)

The method is first called from the main method of the RecursionDemo class.

First call of the method
n = 5

The second through sixth calls are recursive (number of times a method calls itself).

Second call of the method
n = 4

Third call of the method
n = 3

Fourth call of the method
n = 2

Fifth call of the method
n = 1

Sixth call of the method
n = 0

Each time the method is called, a new instance of the n parameter is created in memory!

Depth of recursion = 5.

Recursive Methods (5 of 18)

For instance. The following method shows a straightforward implementation of the $S(N)$ recursive function.

```
1 // Evaluate the sum of  the first n + integers
2
3 public static long s(int n)
4 {
5     if (n==1)
6           return 1;
7     else
8           return n + s(n-1);
9 }
```

- If $N = 1$, then we have the trivial case for which we know the solution $S(1) = 1$ (lines 5, 6). No recursion needed to solve it.
- Otherwise, we follow the recursive definition $S(N) = N + S(N - 1)$ (line 8).

Recursive Methods (6 of 18)

- Tracing recursive calls

```
//client
public static void main(...)
{
   print s(3);
   ...
}//end main
```

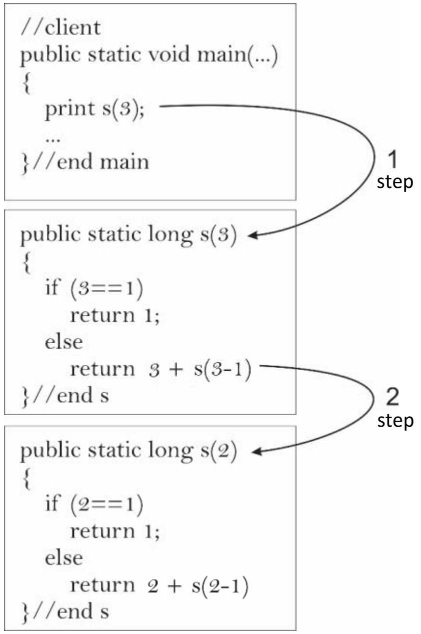
1
step

```
public static long s(3)
{
   if (3==1)
      return 1;
   else
      return  3 + s(3-1)
}//end s
```

1. The driver method main calls the s() method (first call). The argument 3 is copied into the parameter n of the method s().

Recursive Methods (7 of 18)

- Tracing recursive calls

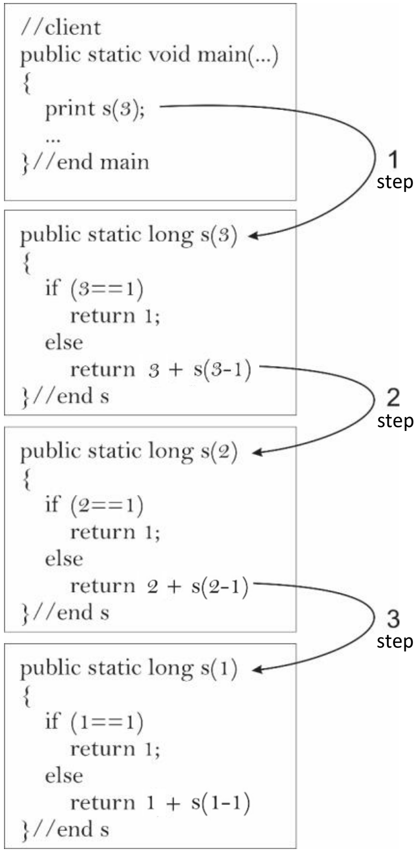


1. The driver method main calls the s() method (first call). The argument 3 is copied into the parameter n of the method s().
2. As the test if (3==1) fails, the statement return 3 + s(3-1) is executed (first recursive call). The execution of s(3) is then suspended until the results of s(3-1) are known. The argument 2 is copied into the parameter *n* of the method s().

Recursive Methods (8 of 18)

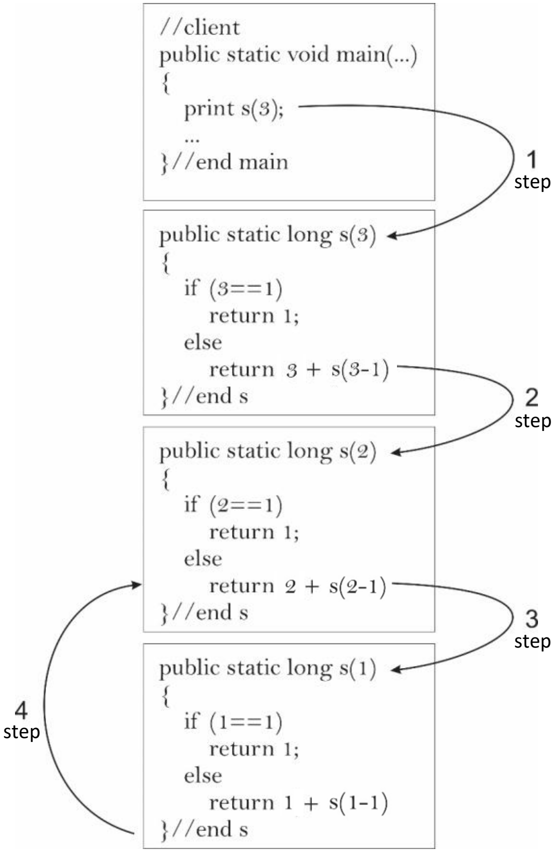
- Tracing recursive calls

1. The driver method main calls the s() method (first call). The argument 3 is copied into the parameter n of the method s().
2. As the test if (3==1) fails, the statement return s(3-1) + 3 is executed (first recursive call). The execution of s(3) is then suspended until the results of s(3-1) are known. The argument 2 is copied into the parameter *n* of the method s().
3. As the test if (2==1) fails, the statement return 2 + s(2-1) is executed (second recursive call). The execution of s(2) is then suspended until the results of s(2-1) occurs. The argument 1 is copied into the parameter n of the method s().



Recursive Methods (9 of 18)

- Tracing recursive calls



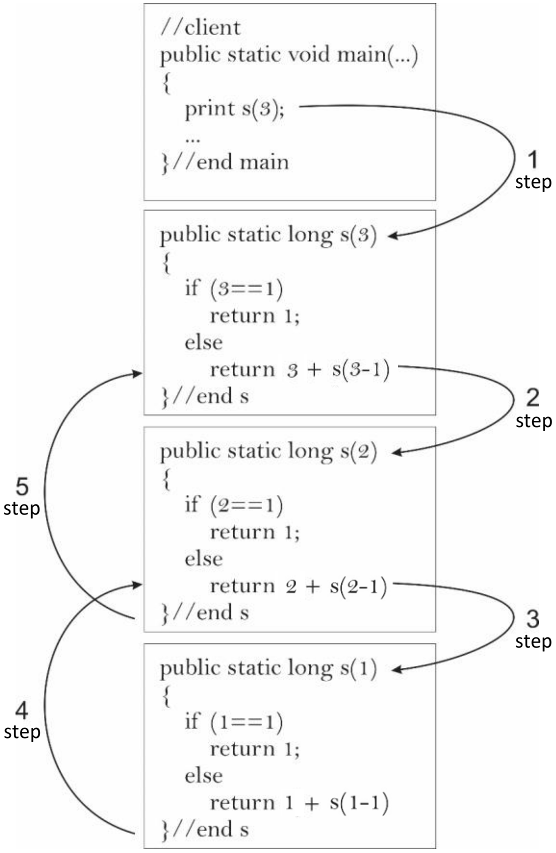
4. As the test if (1==1) returns true, the statement return 1 is executed and no other recursive calls occurs. The method completes execution and returns to the call s(2-1). Then, the execution of s(2) resumes returning 2 + 1.

Recursive Methods (10 of 18)

- Tracing recursive calls

4. As the test if (1==1) returns true, the statement return 1 is executed and no other recursive calls occurs. The method completes execution and returns to the call s(2-1). Then, the execution of s(2) resumes returning 1 + 2.

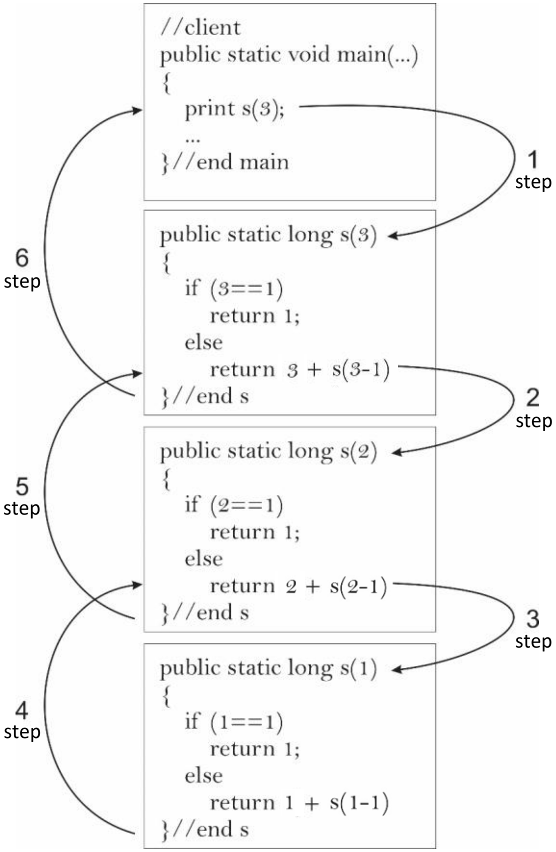
5. Then, the execution of s(3) resumes returning 3 + 3.



Recursive Methods (11 of 18)

- Tracing recursive calls

4. As the test if (1==1) returns true, the statement return 1 is executed and no other recursive calls occurs. The method completes execution and returns to the call s(2-1). Then, the execution of s(2) resumes returning 1 + 2.
5. Then, the execution of s(3) resumes returning 3 + 3.
6. Finally, a return to the main method occurs and 6 is printed.

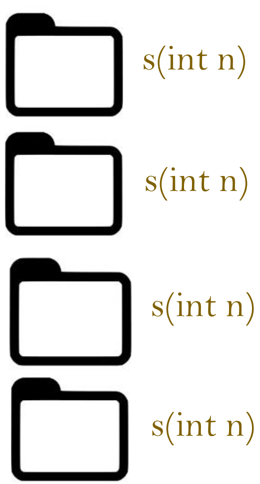


Recursive Methods (12 of 18)

- The stack of activation records

Although it seems that the recursive method s(int n) is calling itself, in reality it is calling a clone to itself. That clone is simply another method with different parameters, local variables, and location of the current instruction.

Java implements methods by using an internal stack of activation records, where each record provides a snapshot of a method’s state during its execution. At any instant, only one clone is active; the rest are pending. Can you tell which clone?

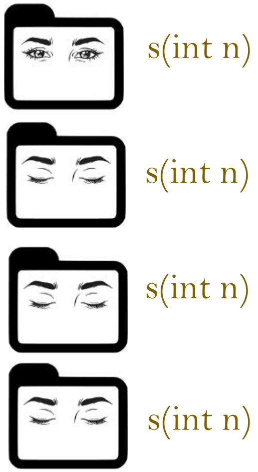


Recursive Methods (13 of 18)

- The stack of activation records

Although it seems that the recursive method s(int n) is calling itself, in reality it is calling a clone to itself. That clone is simply another method with different parameters, local variables, and location of the current instruction.

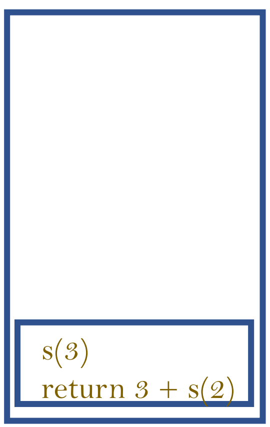
Java implements methods by using an internal stack of activation records, where each record provides a snapshot of a method's state during its execution. At any instant, only one clone is active; the rest are pending. Can you tell which clone?



Recursive Methods (14 of 18)

- The stack of activation records

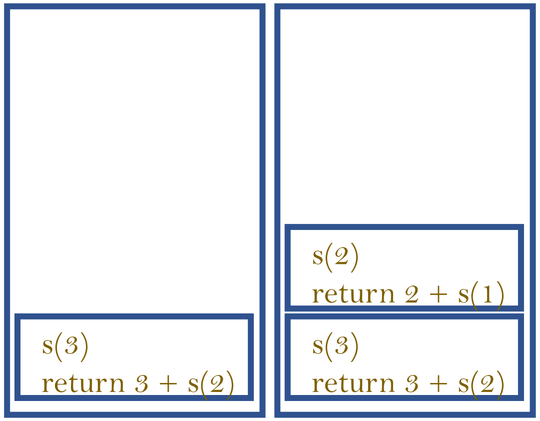
The stack of activation records is used because methods return in reverse order of invocation.



Recursive Methods (15 of 18)

- The stack of activation records

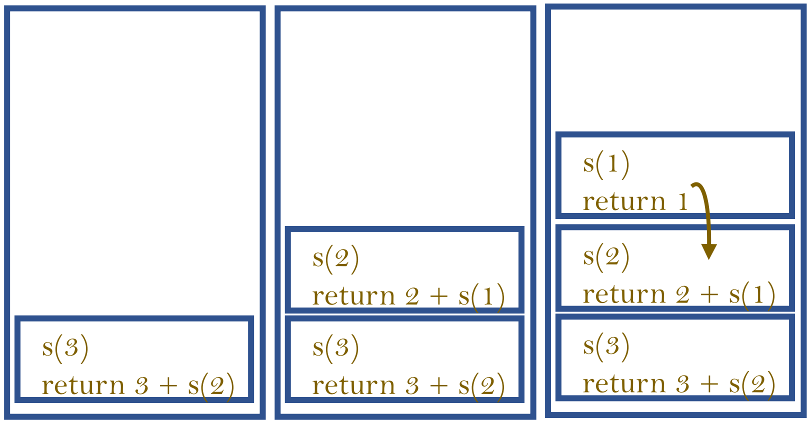
The stack of activation records is used because methods return in reverse order of invocation.



Recursive Methods (16 of 18)

- The stack of activation records

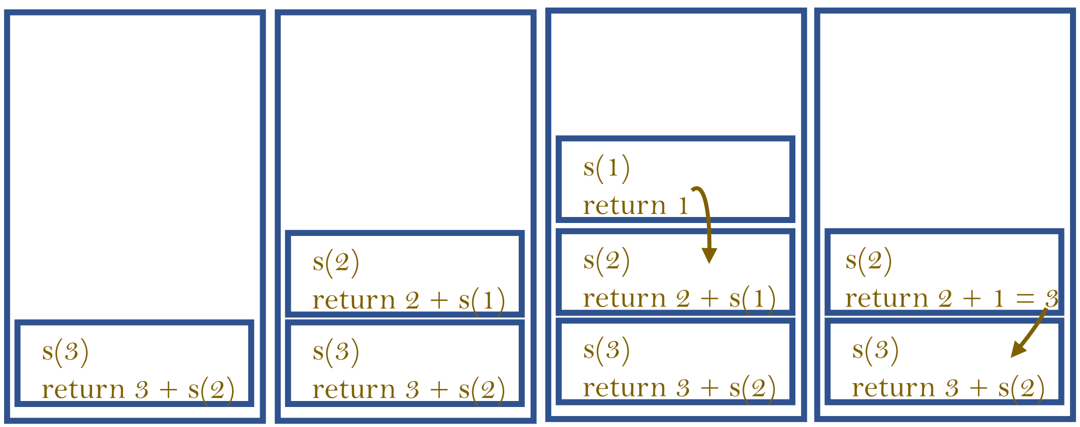
The stack of activation records is used because methods return in reverse order of invocation.



Recursive Methods (17 of 18)

- The stack of activation records

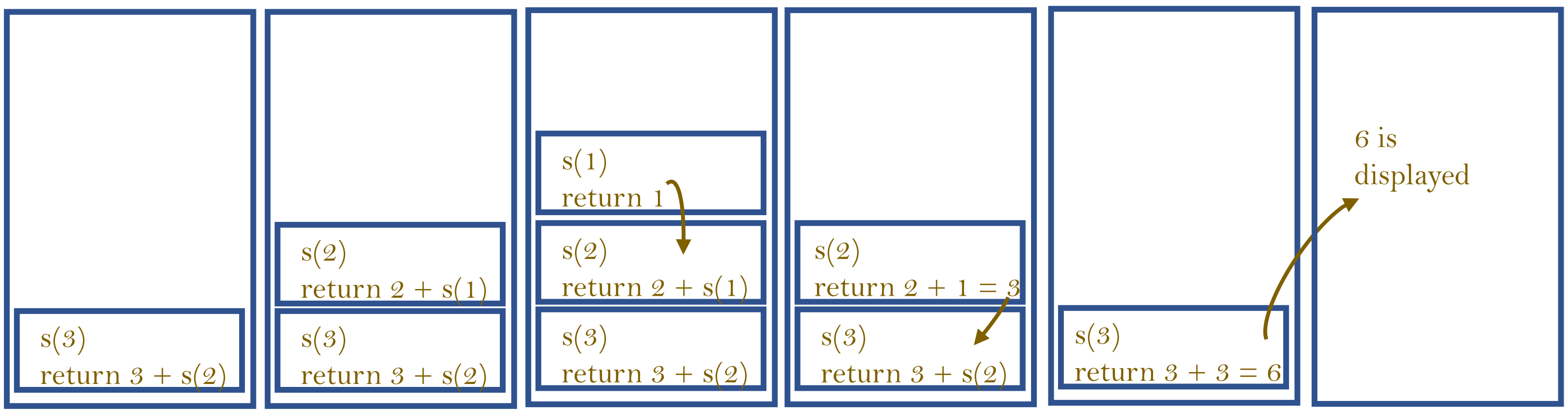
The stack of activation records is used because methods return in reverse order of invocation.



Recursive Methods (18 of 18)

- The stack of activation records

The stack of activation records is used because methods return in reverse order of invocation.



Solving Problems With Recursion (1 of 8)

- A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem.
- Recursion can be a powerful tool for solving repetitive problems.
- Recursion is never absolutely required to solve a problem.
- Any problem that can be solved recursively can also be solved iteratively, with a loop.
- In many cases, recursive algorithms are less efficient than iterative algorithms.

Solving Problems With Recursion (2 of 8)

- Recursive solutions repetitively:
  - allocate memory for parameters and local variables, and
  - store the address of where control returns after the method terminates.
- These actions are called *overhead* and take place with each method call.
- This overhead does not occur with a loop.
- Some repetitive problems are more easily solved with recursion than with iteration.
  - Iterative algorithms might execute faster; however,
  - a recursive algorithm might be designed faster.

Solving Problems With Recursion (3 of 8)

- Point to ponder #1:

What does this equation tell you? $S(N) = N(N+1)/2$?

The sum of the first N^+ integers (closed form)

It is easier to write a formula recursively than in closed form!

$S(1) = 1$ and $S(N) = N + S(N-1)$.

Solving Problems With Recursion (4 of 8)

- Recursion works like this:
  - A base case is established.
    - If matched, the method solves it and returns.
  - If the base case cannot be solved now:
    - the method reduces it to a smaller problem (recursive case) and calls itself to solve the smaller problem.
- By reducing the problem with each recursive call, the base case will eventually be reached, and the recursion will stop.

Solving Problems With Recursion (5 of 8)

- In mathematics, the notation $n!$ represents the factorial of the number n.
- The factorial of a nonnegative number can be defined by the following rules:
  - If $n = 0$ then $n! = 1$
  - If $n > 0$ then $n! = 1 \times 2 \times 3 \times \ldots \times n$
- Let's replace the notation $n!$ with factorial(n), which looks a bit more like computer code, and rewrite these rules as:
  - If $n = 0$ then factorial(n) = 1
  - If $n > 0$ then factorial(n) = $1 \times 2 \times 3 \times \ldots \times n$

Solving Problems With Recursion (6 of 8)

- These rules state that:
  - when *n* is 0, its factorial is 1, and
  - when *n* greater than 0, its factorial is the product of all the positive integers from 1 up to *n*.
- Factorial(6) is calculated as
  - 1 × 2 × 3 × 4 × 5 × 6.
- The base case is where *n* is equal to 0:
  - `if n = 0 then factorial(n) = 1`
- The recursive case, or the part of the problem that we use recursion to solve is:
  - `if n > 0 then factorial(n) = n × factorial(n – 1)`

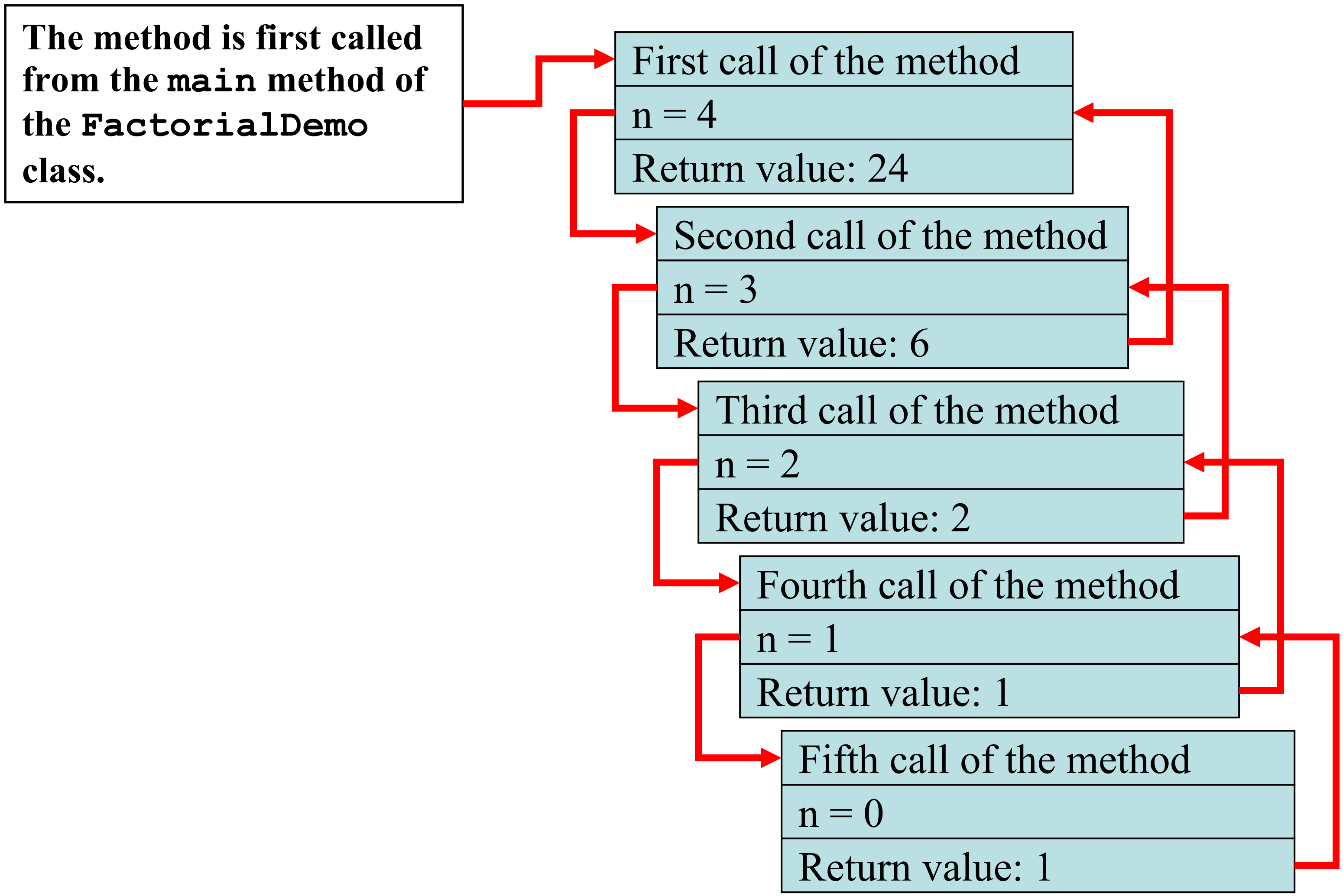
Solving Problems With Recursion (7 of 8)

- The recursive call works on a reduced version of the problem, *n* – 1.
- The recursive rule for calculating the factorial:
  - If n = 0 then factorial(n) = 1
  - If n > 0 then factorial(n) = n × factorial(n – 1)
- A Java based solution:

```
private static int factorial(int n)
{
  if (n == 0) return 1; // Base case
  else return n * factorial(n - 1);
}
```

- Example: [FactorialDemo.java]

Solving Problems With Recursion (8 of 9)



Solving Problems With Recursion (9 of 9)

- Things to be avoided (1)
  - Infinite recursion: a recursive method that does not check for the base case, misses the base case, or does not get simpler will execute “forever”.

```
public static long r(int n)
{
    return n+ r(n-1);
}
```

```
public static long t(int n)
{
    if (n==1)
        return 1;
    else
        return n + t(n-2);
}
```

```
public static long u(int n)
{
    if (n==1)
        return 1;
    else
        return n + u(n);
}
```

Point to ponder #1:

What will happen here after some number of calls?

The program shuts down and report a stack overflow .

Recursion

Lecture 10b

Topics

- Introduction to Recursion
- Recursive Methods
- Solving Problems with Recursion
- Simple Examples of Recursive Methods
- Direct and Indirect Recursion
- Summing a Range of Array Elements
- The Fibonacci Series
- Greatest Common Divisor
- A Recursive Binary Search Method
- The Towers of Hanoi

Direct and Indirect Recursion (1 of 2)

- When recursive methods directly call themselves, it is known as *direct recursion*.
- *Indirect recursion* is when method A calls method B, which in turn calls method A.
- There can even be several methods involved in the recursion.
- Example, method A could call method B, which could call method C, which calls method A.
- Care must be used in indirect recursion to ensure that the proper base cases and return values are handled.

Direct and Indirect Recursion (2 of 2)

Direct Recursion

```
void directRecFun()
{
    // Some code....

    directRecFun();

    // Some code...
}
```

Indirect Recursion

```
void indirectRecFun1()
{
    // Some code...

    indirectRecFun2();

    // Some code...
}

void indirectRecFun2()
{
    // Some code...

    indirectRecFun1();

    // Some code...
}
```

Summing a Range of Array Elements (1 of 23)

- Recursion can be used to sum a range of array elements.
- A method, `rangeSum` takes following arguments:
  - an `int` array,
  - an `int` specifying the starting element of the range, and
  - an `int` specifying the ending element of the range.
  - How it might be called:

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int sum;
sum = rangeSum(numbers, 3, 7);
```

Summing a Range of Array Elements (2 of 23)

- The definition of the `rangeSum` method:

```
public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

- Example: [RangeSum.java]

Summing a Range of Array Elements (3 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	

Summing a Range of Array Elements (4 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	

Summing a Range of Array Elements (5 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	

Summing a Range of Array Elements (6 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	

Summing a Range of Array Elements (7 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	

Summing a Range of Array Elements (8 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	

Summing a Range of Array Elements (9 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	
4 (rec)	6	7	

Summing a Range of Array Elements (10 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	
4 (rec)	6	7	

Summing a Range of Array Elements (11 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	
4 (rec)	6	7	
5 (rec)	7	7	

Summing a Range of Array Elements (12 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	
4 (rec)	6	7	
5 (rec)	7	7	

Summing a Range of Array Elements (13 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	
4 (rec)	6	7	
5 (rec)	7	7	
6 (rec)	8	7	

Summing a Range of Array Elements (14 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	
4 (rec)	6	7	
5 (rec)	7	7	
6 (rec)	8	7	

Summing a Range of Array Elements (15 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	
4 (rec)	6	7	
5 (rec)	7	7	
6 (rec)	8	7	0

Summing a Range of Array Elements (16 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	
4 (rec)	6	7	
5 (rec)	7	7	8
6 (rec)	8	7	0

Summing a Range of Array Elements (17 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	
4 (rec)	6	7	15
5 (rec)	7	7	8
6 (rec)	8	7	0

Summing a Range of Array Elements (18 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	
3 (rec)	5	7	21
4 (rec)	6	7	15
5 (rec)	7	7	8
6 (rec)	8	7	0

Summing a Range of Array Elements (19 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	
2 (rec)	4	7	26
3 (rec)	5	7	21
4 (rec)	6	7	15
5 (rec)	7	7	8
6 (rec)	8	7	0

Summing a Range of Array Elements (20 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	30
2 (rec)	4	7	26
3 (rec)	5	7	21
4 (rec)	6	7	15
5 (rec)	7	7	8
6 (rec)	8	7	0

Summing a Range of Array Elements (21 of 23)

- Debugging the `rangeSum` method:

```
public static void main(String[] args)
{
  int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int sum;
  sum = rangeSum(numbers, 3, 7);
}

public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

call	start	end	return
1	3	7	30
2 (rec)	4	7	26
3 (rec)	5	7	21
4 (rec)	6	7	15
5 (rec)	7	7	8
6 (rec)	8	7	0

Output = 30

Summing a Range of Array Elements (22 of 23)

- Debugging the `rangeSum` method:

```
int [] array = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

seq	calls	start	end	return
1	`(first call)rangeSum(array,3,7)`	3	7	30
1	`(rec call) array[3] + rangeSum(array,4,7)`	4	7	30
2	`(rec call) array[4] + rangeSum(array,5,7)`	5	7	26
3	`(rec call) array[5] + rangeSum(array,6,7)`	6	7	21
4	`(rec call) array[6] + rangeSum(array,7,7)`	7	7	15
5	`(rec call) array[7] + rangeSum(array,8,7)`	8	7	8

Output = 30

Summing a Range of Array Elements (23 of 23)

- Point to ponder #1:

  How can we decrease the depth of this recursion from 5 to 4?

```
public static int rangeSum(int[] array, int start, int end)
{
  if (start > end)
    return 0;
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

```
public static int rangeSum(int[] array, int start, int end)
{
  if (start == end)
    array[start];
  else
    return array[start] + rangeSum(array, start + 1, end);
}
```

The Fibonacci Series (1 of 2)

- Some mathematical problems are designed to be solved recursively.
- One well known example is the calculation of *Fibonacci numbers.*:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,…
- After the second number, each number in the series is the sum of the two previous numbers.
- The Fibonacci series can be defined as:
  - $Fib(0) = 0$
  - $Fib(1) = 1$
  - $Fib(n) = Fib(n-1) + Fib(n–2)$ *for* $n \geq 2$.

The Fibonacci Series (2 of 2)

```
public static int fib(int n)
{
  if (n == 0)
    return 0;
  else if (n == 1)
      return 1;
  else
      return fib(n - 1) + fib(n - 2);
}
```

- This method has two base cases:
  - when n is equal to 0, and
  - when n is equal to 1.
- Example: [FibNumbers.java]

Greatest Common Divisor (GCD) (1 of 2)

- The GCD of two positive integers, x and y, is as follows:

  - if y divides x evenly, then gcd(x, y) = y
  - Otherwise, gcd(x, y) = gcd(y, remainder of x/y)

- For instance:

gcd(18,10) -> gcd(10,8) -> gcd(8,2) = 2
gcd(18,15) -> gcd(15,3) = 3
gcd(18,5) -> gcd(5,3) -> gcd(3,2) -> gcd(2,1) = 1

Greatest Common Divisor (GCD) (2 of 2)

- The definition of the `gcd` method:

```
public static int gcd(int x, int y)
{
      if (x % y == 0)
         return y;
      else
         return gcd(y, x % y);
}
```

- Example: GCDdemo.java

Recursive Binary Search

- The binary search algorithm can be implemented recursively.
- The procedure can be expressed as:

```
if array[middle] equals the search value, then
   the value is found.
Else
if array[middle] is less than the search value,
   do a binary search on the upper half of the
   array.
Else
if array[middle] is greater than the search
   value, perform a binary search on the lower
   half of the array.
```

- Example: [RecursiveBinarySearch.java]()

The Towers of Hanoi (1 of 4)

- The Towers of Hanoi is a mathematical game that uses:
  - three pegs and
  - a set of discs with holes through their centers.
- The discs are stacked on the leftmost peg, in order of size with the largest disc at the bottom.
- The goal of the game is to move the pegs from the left peg to the right peg by these rules:
  - Only one disk may be moved at a time.
  - A disk cannot be placed on top of a smaller disc.
  - All discs must be stored on a peg except while being moved.

The Towers of Hanoi (2 of 4)

- The overall solution to the problem is to move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.
- This algorithm solves the game.

```
If n > 0 Then
    Move n – 1 discs from peg A to peg B, using peg C
     as a temporary peg.
    Move the remaining disc from the peg A to peg C.
    Move n – 1 discs from peg B to peg C, using peg A
     as a temporary peg.
End If
```

- The base case for the algorithm is reached when there are no more discs to move.
- Example: Hanoi.java, HanoiDemo.java

The Towers of Hanoi (3 of 4)

- num The number of discs to move.
- fromPeg The peg to move the discs from.
- toPeg The peg to move the discs to.
- tempPeg The peg to use as a temporary peg.

```
private static void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
{
  if (num > 0) {
     moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
     System.out.println("Move a disc from peg " + fromPeg + " to peg " + toPeg);
     moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
  }
}
```

The Towers of Hanoi (4 of 4)

```
If n > 0 Then
     Move n - 1 discs from peg A to peg B, using peg C as a temporary peg.
     Move the remaining disc from the peg A to peg C.
     Move n - 1 discs from peg B to peg C, using peg A as a temporary peg.
End If
```