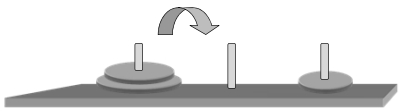


# CSIS 3103

Ch 5: Recursion

## Chapter Objectives

- Learn how to think recursively
- Trace recursive method calls
- Improve skills in writing recursive algorithms and methods
- Learn about recursive data structures

## What Is Recursion?

**Recursive call:** A method call which directly or indirectly calls itself

**Direct recursion:** A method directly calls itself

**Indirect recursion:** A chain of two or more method calls returns to the method that originated the chain

## Facts About Recursion

- Recursion is "divide-and-conquer"
  - Split a problem into one or more simpler versions of itself
- Many problems lend themselves to simple, elegant, recursive solutions
- Recursive solutions are sometimes less efficient than iterative solutions
  … but sometimes more efficient
- Recursion solutions must be designed carefully

## Definitions

**Base case:** Case(s) for which the solution can be stated non-recursively

**General (recursive) case:** The case(s) for which the solution is expressed in terms of a smaller version of itself

**Recursive algorithm:** A solution that is expressed in terms of

(a) base cases and

(b) smaller instances of itself

## Factorial

Two definitions:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$ Non-recursive

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$ Recursive

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

## The Factorial Algorithm

- Familiar
- Easy to visualize
- But in practice, one would not want to solve this problem using recursion
- The iterative solution is just as simple and requires less computing resources

## Iterative Factorial

```
private static int factorial(int n) {
  int fact = 1;
  for (int count = 2; count <= n; count++)

      fact = fact * count;

  return fact;
}
```

## Verifying Recursion

The Three-Question Technique

1. *The Base-Case Question:* What is the non-recursive way out of the method, and does the method work correctly for this base case?
2. *The Smaller-Caller Question:* Does each recursive call to the method invoke a "smaller" version of the original problem, leading inescapably to the base case?
3. *The General-Case Question:* If the recursive call(s) works correctly, does the whole method work correctly?

This is essentially an induction proof

## Writing Recursive Methods

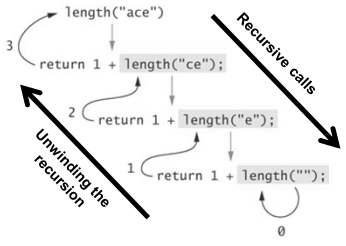
1. Develop an exact definition of the problem to be solved
2. Determine the size of the problem to be solved on this call to the method
3. Identify and solve the base case(s) in which the problem can be expressed non-recursively
4. Identify and solve the general case(s) correctly in terms of a smaller case of the same problem - a recursive call

## String Length Algorithm

**if** the string is empty (has no characters)
    the length is 0
**else**
    the length is 1 plus the length of the string that excludes the first character

```
public int length(String str) {
   if (str == null || str.equals(""))
      return 0;
   else
      return 1 + length(str.substring(1));
}
```

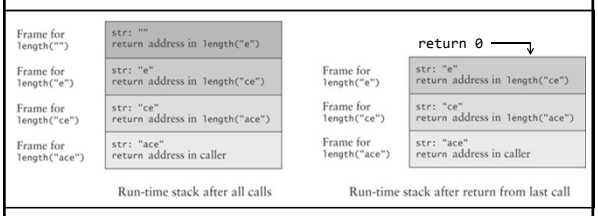
## Tracing a Recursive Method



## Run-Time Stack and Activation Frames

- Java maintains a run-time stack where it saves new information in an *activation frame*
- Activation frame stores
  - method arguments
  - local variables (if any)
  - the return address of the instruction that called the method
- Whenever a method is called, Java pushes a new activation frame onto the run-time stack

## Run-Time Stack and Activation Frames




## Recursive Algorithm for gcd

The greatest common divisor (gcd) of two numbers is the largest integer that divides both numbers

- The gcd of 20 and 15 is 5
- The gcd of 36 and 24 is 12
- The gcd of 36 and 18 is 18

## Recursive Algorithm for gcd

Given 2 positive integers $m$ and $n$ ($m > n$)

**if** n is a divisor of m
    gcd(m, n) = n
**else**
    gcd (m, n) = gcd (n, m % n)

## Recursive Algorithm for gcd

```
/** Recursive gcd method
    pre: m > 0 and n > 0
    @param m The first number
    @param n The second number
    @return Greatest common divisor of m and n
*/
public static double gcd(int m, int n) {
    if (m % n == 0)
       return n;
    else if (m < n)
       return gcd(n, m); // Transpose arguments.
    else
       return gcd(n, m % n);
}
```

## Efficient Recursive Exponentiation

Algorithm:

Example:

## Java Implementation

```
/** Returns x to the n power
  * @param x Base
  * @param n Exponent
  */
public static double exp(double x, int n) {
  if (n == 0) return 1.0;   // base
  if (n == 1) return x;     // base
  if (n%2 != 0) return x*exp(x, n-1);
  double y = exp(x, n/2);
  return y*y;
}
```

## Fibonacci Numbers

Fibonacci numbers were invented to model the growth of a rabbit colony

$fib_1 = 1$

$fib_2 = 1$

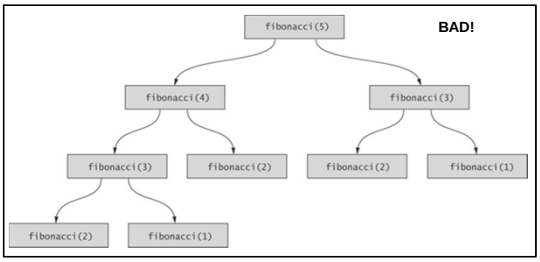
$fib_n = fib_{n-1} + fib_{n-2}$

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

## O($2^n$) (Exponential) Recursive `fibonacci` Method

```
/** Recursive method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param n The position of the Fibonacci number being calculated
    @return The Fibonacci number
*/
public static int fibonacci(int n) {
    if (n <= 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

## Exponential `fibonacci`




## O(n) Recursive `fibonacci` Method

```
/** Recursive O(n) method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param fibCurrent The current Fibonacci number
    @param fibPrevious The previous Fibonacci number
    @param n The count of Fibonacci numbers left to calculate
    @return The value of the Fibonacci number calculated so far
*/
private static int fibo(int fibCurrent, int fibPrevious, int n) {
    if (n == 1)
        return fibCurrent;
    else
        return fibo(fibCurrent + fibPrevious, fibCurrent, n - 1);
}
```

## O(n) Recursive `fibonacci` Method

To start the method executing, we provide a non-recursive wrapper method:

```
/** pre: n >= 1
    @param n The desired Fibonacci number
    @return  The value of the nth Fibonacci
             number
*/
public static int fibonacciStart(int n) {
     return fibo(1, 0, n);
}
```

## Linear `fibonacci`