Chapter 18 - Part 1

Recursion

Why Use Recursion

- Recursion allows solving complex problem with simple solutions.

- Reduces coding.

- Leads to efficient programs.

Example: Computing Factorial

n! = n * n-1 * n-2 * n-3 … * 1

n! = n * (n-1)!

factorial(0) = 1; //special cases

factorial(n) = n*factorial(n-1);

```java
/** Return the factorial for a specified number */
public static long factorial(int n)
{
    if (n == 0) // Base case
        return 1;
    else
        return n * factorial(n - 1); // Recursive call
}
```

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4)

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

Computing Factorial

$factorial(0) = 1;$

$factorial(n) = n*factorial(n-1);$

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

= 4 * 3 * (2 * (1 * factorial(0)))

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

$\qquad$ = 4 * 3 * factorial(2)

$\qquad$ = 4 * 3 * (2 * factorial(1))

$\qquad$ = 4 * 3 * (2 * (1 * factorial(0)))

$\qquad$ = 4 * 3 * (2 * (1 * 1)))

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

$\qquad$ = 4 * 3 * factorial(2)

$\qquad$ = 4 * 3 * (2 * factorial(1))

$\qquad$ = 4 * 3 * (2 * (1 * factorial(0)))

$\qquad$ = 4 * 3 * (2 * (1 * 1)))

$\qquad$ = 4 * 3 * (2 * 1)

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

= 4 * 3 * (2 * (1 * factorial(0)))

= 4 * 3 * (2 * (1 * 1)))

= 4 * 3 * (2 * 1)

= 4 * 3 * 2

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

= 4 * 3 * (2 * factorial(1))

= 4 * 3 * (2 * (1 * factorial(0)))

= 4 * 3 * (2 * (1 * 1)))

= 4 * 3 * (2 * 1)

= 4 * 3 * 2

= 4 * 6

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

$\quad$ = 4 * 3 * factorial(2)

$\quad$ = 4 * 3 * (2 * factorial(1))

$\quad$ = 4 * 3 * (2 * (1 * factorial(0)))

$\quad$ = 4 * 3 * (2 * (1 * 1)))

$\quad$ = 4 * 3 * (2 * 1)

$\quad$ = 4 * 3 * 2

$\quad$ = 4 * 6

$\quad$ = 24

13

Trace Recursive factorial

Executes factorial(4)

factorial(4)

Stack

Space Required
for factorial(4)

Main method

14

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

Executes factorial(3)

return 4 * factorial(3)

Stack

| Space Required for factorial(3) |
| Space Required for factorial(4) |
| Main method |

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Executes factorial(2)

Stack

| Space Required for factorial(2) |
| Space Required for factorial(3) |
| Space Required for factorial(4) |
| Main method |

16

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Executes factorial(1)

Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

turn 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Executes factorial(0)

Stack
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factor

return 2 * factorial(1)

Step 3: execu   factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

return 1

returns 1

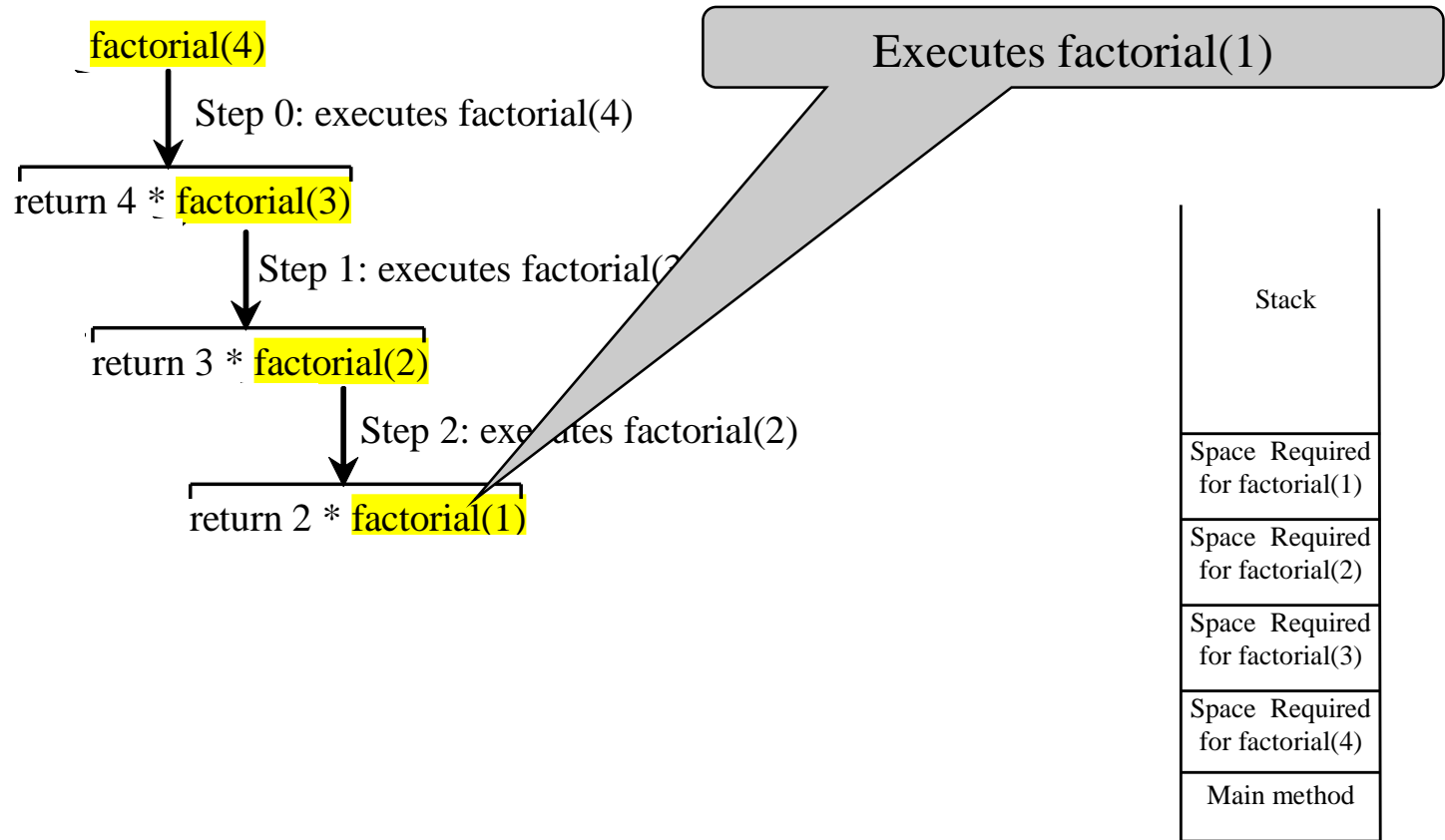| Stack |
| --- |
| Space Required for factorial(0) |
| Space Required for factorial(1) |
| Space Required for factorial(2) |
| Space Required for factorial(3) |
| Space Required for factorial(4) |
| Main method |

19

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

returns factorial(0)

Stack
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

20

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes fact...

returns factorial(1)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

21

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 6: return 1

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 5: return 1

Step 4: executes factorial(0)

return 1

returns factorial(2)

Stack
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

22

Trace Recursive factorial

animation

factorial(4)

Step 0: executes factorial(4)

returns factorial(3)

return 4 * factorial(3)

Step 8: return 6

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 7: return 2

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 6: return 1

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 5: return 1

Step 4: executes factorial(0)

return 1

Stack

Space Required for factorial(3)

Space Required for factorial(4)

Main method

23

Trace Recursive factorial

returns factorial(4)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 8: return 6

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

Stack

Space Required
for factorial(4)

Main method

24

factorial(4) Stack Trace

1	Space Required for factorial(4)

2	Space Required for factorial(3)
	Space Required for factorial(4)

3	Space Required for factorial(2)
	Space Required for factorial(3)
	Space Required for factorial(4)

4	Space Required for factorial(1)
	Space Required for factorial(2)
	Space Required for factorial(3)
	Space Required for factorial(4)

5	Space Required for factorial(0)
	Space Required for factorial(1)
	Space Required for factorial(2)
	Space Required for factorial(3)
	Space Required for factorial(4)

6	Space Required for factorial(1)
	Space Required for factorial(2)
	Space Required for factorial(3)
	Space Required for factorial(4)

7	Space Required for factorial(2)
	Space Required for factorial(3)
	Space Required for factorial(4)

8	Space Required for factorial(3)
	Space Required for factorial(4)

9	Space Required for factorial(4)

Example: Sum Function

sum(n) = n + n-1 + n-2 + n-3 + … + 0

sum(n) = n + sum(n-1);

sum(0) = 0; //special case

```
Sum(5) = 5 + sum(4)
       = 5 + (4 + sum(3))
       = 5 + (4 + (3 +  sum(2)))
       = 5 + (4 + (3 + (2 + sum(1))))
       = 5 + (4 + (3 + (2 + (1 + sum(0)))))
       = 5 +  4 +  3 +  2 +  1 + 0
       = 15
```

Example: Sum Function

```
/** Return the sum for a specified number */
public static int sum (int n)
{
  if (n == 0) // Base case
      return 0;
  else
      return n + sum(n - 1); // Recursive call
}
```

Fibonacci Numbers

```
Fibonacci series:  0 1 1 2 3 5 8 13 21 34 55 89…

        indices:   0 1 2 3 4 5 6 7  8  9  10 11
```

fib(0) = 0;

fib(1) = 1;

fib(index) = fib(index -1) + fib(index -2); index >=2

```
fib(3) = fib(2)              + fib(1)
       = (fib(1) + fib(0))   + fib(1)
       = (1       + 0)       + fib(1)
       = 1                   + fib(1)
       = 1                   + 1
       = 2
```

Fibonnaci Numbers

fib(4)

17: return fib(4)　　0: call fib(4)

return fib(3) + fib(2)

11: call fib(2)

10: return fib(3)

1: call fib(3)

16: return fib(2)

return fib(2) + fib(1)

return fib(1) + fib(0)

7: return fib(2)

2: call fib(2)

8: call fib(1)

13: return fib(1)　　12: call fib(1)

14: return fib(0)

9: return fib(1)

return 1

15: return fib(0)

return 0

return fib(1) + fib(0)

return 1

4: return fib(1)

3: call fib(1)

5: call fib(0)

return 1

6: return fib(0)

return 0

29
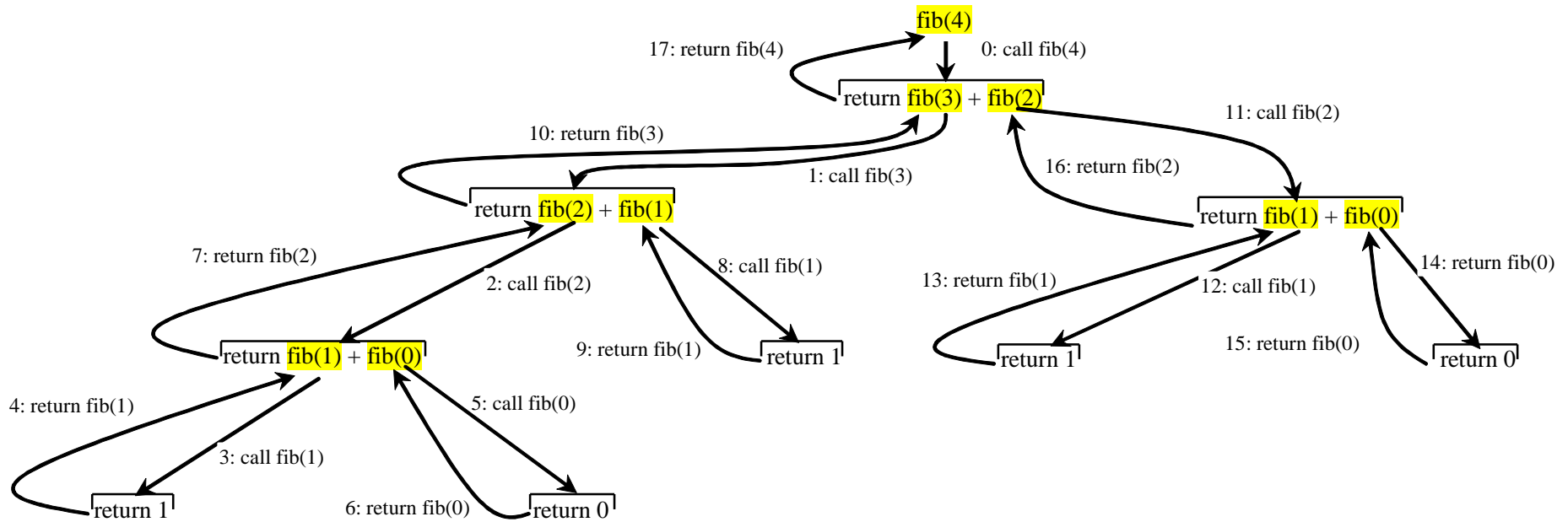
Fibonnaci Numbers

```java
/** Return the fibonacci number */
public static long fib(long index)
{
  if (index == 0) // Base case
     return 0;
  else if (index == 1)
     return 1;
  else
     return fib(index-1) + fib(index-2); // Recursive call
}
```

Characteristics of Recursion

All recursive methods have the following characteristics:

– <u>Base case:</u> One or more base cases (the simplest case) are used to stop/terminate the recursive process.

– <u>Reduction process:</u> Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

In general, to solve a problem using recursion, <u>break it into subproblems</u>. If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively. This subproblem is almost the same as the original problem in nature <u>with a smaller size.</u>

Problem Solving Using Recursion

Consider the problem of printing a message for n times.

You can break the problem into two subproblems: one is to print the message one time and the other is to print the message for n-1 times.

The base case is n==0.

You can solve this problem using recursion as follows:

```java
public static void nPrintln(String message, int times)
{
    if (times >= 1)
    {
        System.out.println(message);
        nPrintln(message, times - 1);
    } // The base case is times == 0
}
```

HW: Re-write it such that counting is in ascending order.

Think Recursively

Many of the problems solved using loops can be solved using recursion if you *think recursively*.

For example, the palindrome problem can be solved recursively.

See non-recursive and recursive solutions next slides.

Non-Recursive Palindrome

```java
// Demonstrates the use of nested while loops.
import java.util.Scanner;
public class PalindromeTester
{
 public static void main (String[] args)
 { String str, another = "y";
   int left, right;
   Scanner scan = new Scanner (System.in);
   while (another.equalsIgnoreCase("y"))  // allows y or Y
   {
     System.out.println ("Enter a potential palindrome string:");
     str = scan.nextLine();
     left = 0;
     right = str.length() - 1;
     while (str.charAt(left) == str.charAt(right) && left < right)
     {
       left = left + 1;
       right = right - 1;
     }
     System.out.println();
     if (left < right)
        System.out.println ("That string is NOT a palindrome.");
     else
        System.out.println ("That string IS a palindrome.");
     System.out.println();
     System.out.print ("Test another palindrome (y/n)? ");
     another = scan.nextLine();
   }
 }
}
```

Recursive Palindrome

```
public static boolean isPalindrome(String s) {
  if (s.length() <= 1) //Base case
    return true;
  else if (s.charAt(0) != s.charAt(s.length()-1)) //Base case
    return false;
  else
    return isPalindrome(s.substring(1, s.length()-1));
} //This solution creates too many substrings!
```

Note: each recursive call passes a new string object since function **substring()** create a new string object. This is inefficient use of memory space.

Recursive Helper Methods

The preceding recursive isPalindrome() method is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, use a helper method (overloaded version of the method):

```java
public static void main (String[] args){
  boolean result = isPalindrome("racecar");
  System.out.println ("result = " + result);
}
//-------------------------------------------------
public static boolean isPalindrome(String s) {
    return isPalindrome(s, 0, s.length() - 1);
}
//-------------------------------------------------
public static boolean isPalindrome(String s, int low, int high) {
   if (high <= low) //Base case
      return true;
   else if (s.charAt(low) != s.charAt(high)) //Base case
      return false;
   else
      return isPalindrome(s, low + 1, high - 1);
}
```

Other Examples to Think About…

Some problems are difficult to solve without using recursion.

Binary search: If the target value is less than the middle element in the list (must be sorted), recursively search first half of the list. If the target value is equal to the middle element, the search ends with a match. If the target value is greater than the middle element in the list, recursively search the second half of the array.

Selection sort: Find the smallest number in the list and swaps it with the first number. Ignore the first number and sort the remaining smaller list recursively as in step 1.

Finding the size of a directory: The size of a directory is the sum of the sizes of all files in the directory. A directory may contain subdirectories.

Towers of Hanoi: Moving the disks from tower A to tower C in the same order.
- There are n disks labeled 1, 2, 3, . . ., n, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

Other Examples to Think About…

Sum of digits in a positive integer: Use the division (/) and remainder (%) operators to extract digits one at a time. The base case would be when (`n/10 == 0`), meaning that all digits have been extracted and added to sum.

Print the content of an array: Traverse the array one element at a time. The base case would be when the array length reaches 0 elements.

Print a string in reverse order: Traverse the string one character at a time. The base case would be when the end of the string is reached (that is, a counter variable is equal to the length of the string).

Return the reverse input string: Parse the input string one character at a time and add it to reverse string backward. The base case would be when the input string is empty (something like: `inputString.equals("")==true`).

Find the Greatest Common Divisor (GCD) of 2 numbers: Use the remainder (%) operator. The base case is when the remainder is zero, you return the second number.

Print the binary representation of an number: Use the division (/) and remainder (%) operators to reduce the number and get the binary digit, respectively. The base case is when the number is either 0 or 1.

Recursion vs. Iteration

Recursion is an alternative form of program control. It is essentially repetition without a loop.

Recursion bears substantial overhead. Each time the program calls a method, the system must assign memory space for all of the method's local variables and parameters (i.e., each call creates a new activation record on the runtime stack). This can consume considerable memory space and requires extra time to manage the additional space.

Tail Recursion

A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call. Otherwise, called *non-tail recursive*

End of Slides