Chapter 18 Recursion

CS1: Java Programming
Colorado State University

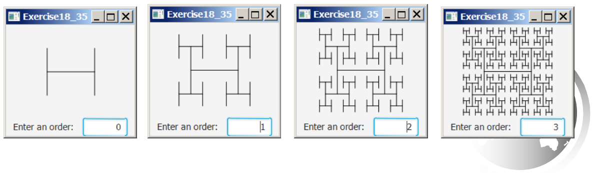
Original slides by Daniel Liang
Modified slides by Chris Wilcox

Motivations

Suppose you want to find all the files under a directory that contains a particular word. How do you solve this problem? There are several ways to solve this problem. An intuitive solution is to use recursion by searching the files in the subdirectories recursively.

Motivations

H-trees, depicted in Figure 18.1, are used in a very large-scale integration (VLSI) design as a clock distribution network for routing timing signals to all parts of a chip with equal propagation delays. How do you write a program to display H-trees? A good approach is to use recursion.



Objectives

- ❑ To describe what a recursive method is and the benefits of using recursion (§18.1).
- ❑ To develop recursive methods for recursive mathematical functions (§§18.2–18.3).
- ❑ To explain how recursive method calls are handled in a call stack (§§18.2–18.3).
- ❑ To solve problems using recursion (§18.4).
- ❑ To use an overloaded helper method to derive a recursive method (§18.5).
- ❑ To implement a selection sort using recursion (§18.5.1).
- ❑ To implement a binary search using recursion (§18.5.2).
- ❑ To get the directory size using recursion (§18.6).
- ❑ To solve the Tower of Hanoi problem using recursion (§18.7).
- ❑ To draw fractals using recursion (§18.8).
- ❑ To discover the relationship and difference between recursion and iteration (§18.9).
- ❑ To know tail-recursive methods and why they are desirable (§18.10).

Computing Factorial

factorial(0) = 1;

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

n! = n * (n-1)!

0! = 1

ComputeFactorial Run

---

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4)

---

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

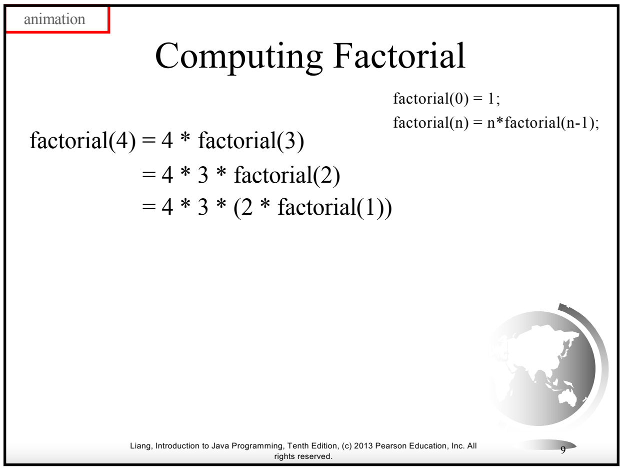
---

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

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

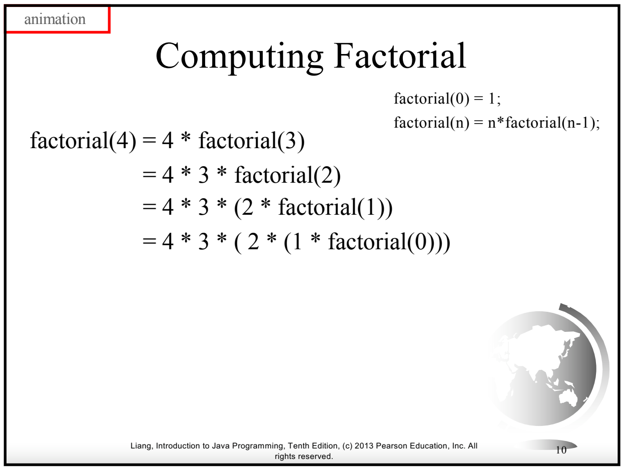


animation

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))

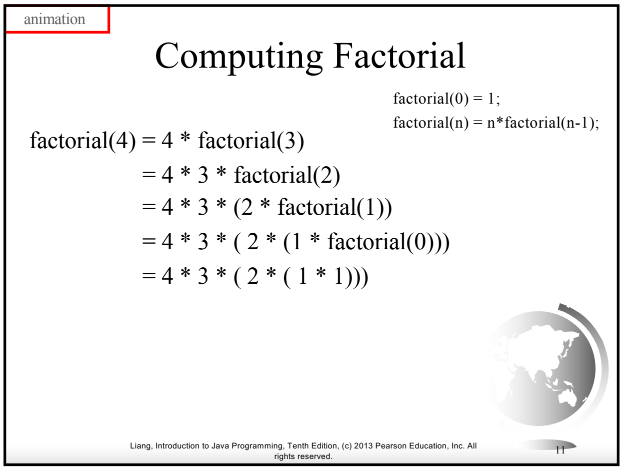


animation

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)))

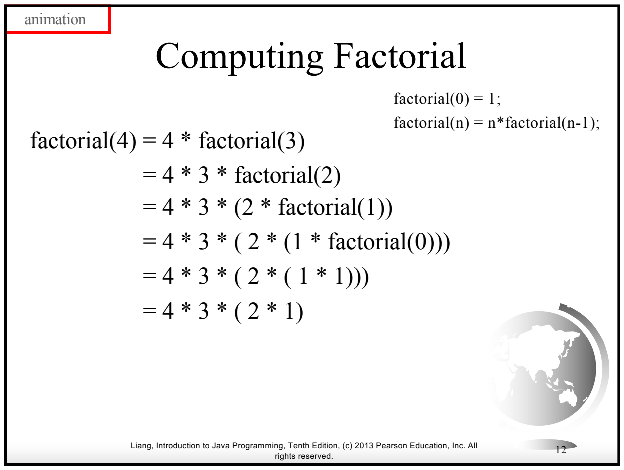


animation

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)))

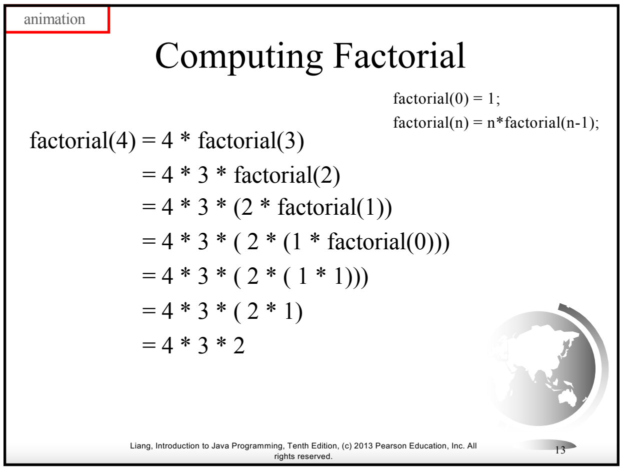


animation

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)

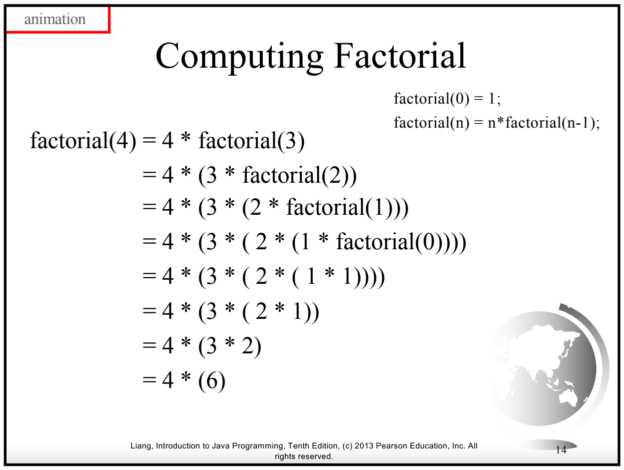


animation

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

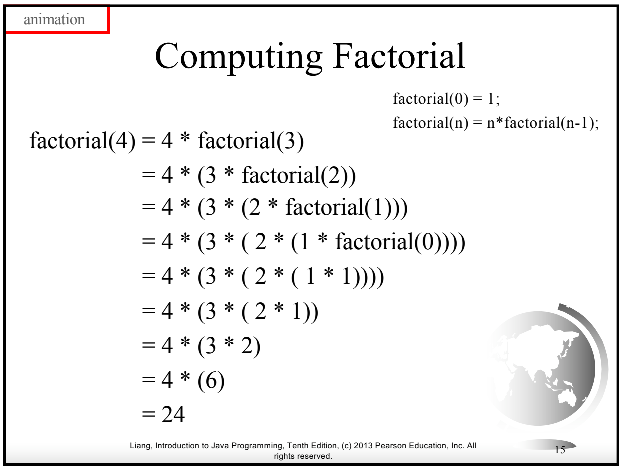


animation

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)

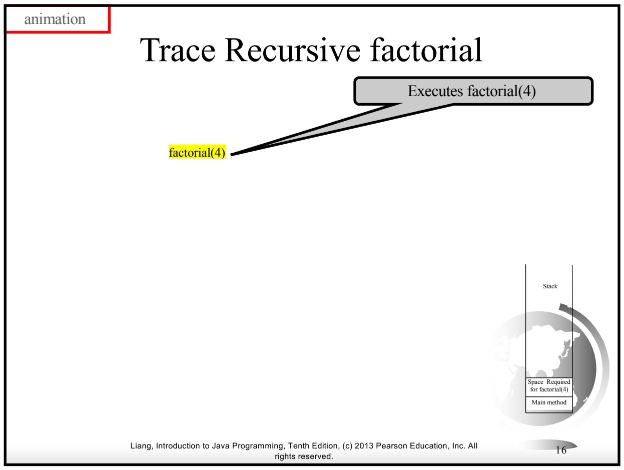


animation

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)
= 24

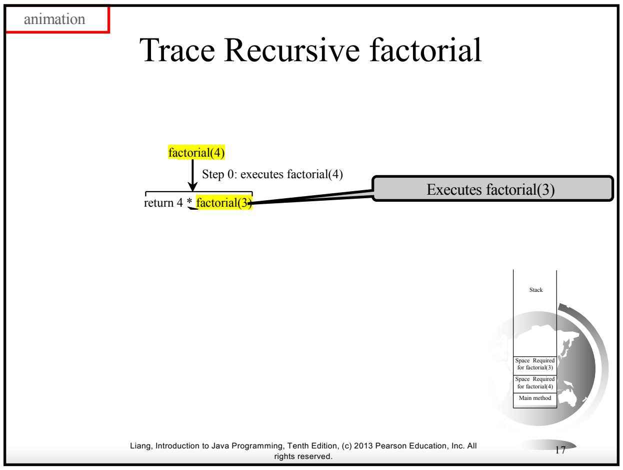


animation

Trace Recursive factorial

Executes factorial(4)

factorial(4)



animation

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

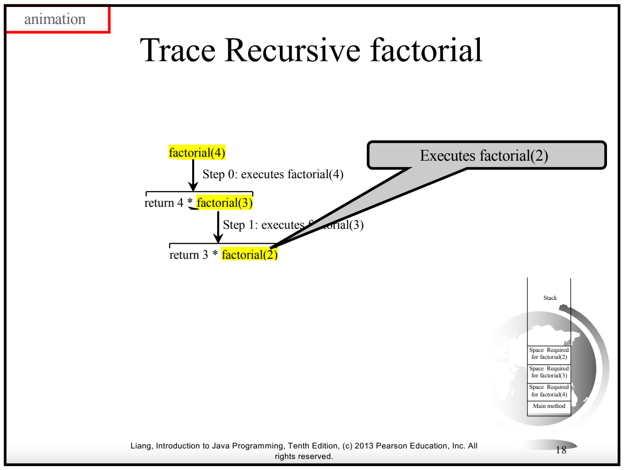
Executes factorial(3)

Stack

Space Required for factorial(3)

Space Required for factorial(4)

Main method



animation

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes f orial(3)

return 3 * factorial(2)

Executes factorial(2)

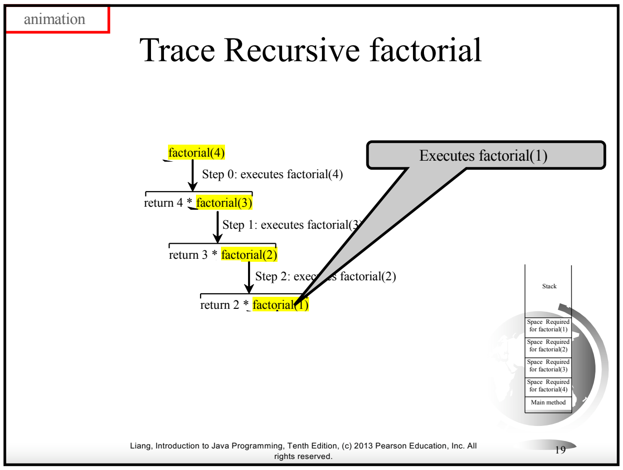
Stack

Space Required for factorial(2)

Space Required for factorial(3)

Space Required for factorial(4)

Main method



animation

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: exec s factorial(2)

return 2 * factorial(1)

Executes factorial(1)

Stack

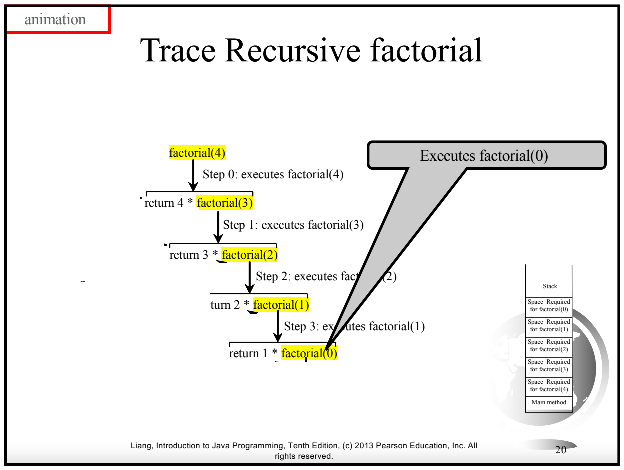
Space Required for factorial(1)

Space Required for factorial(2)

Space Required for factorial(3)

Space Required for factorial(4)

Main method



animation

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 fac (2)

turn 2 * factorial(1)

Step 3: ex utes factorial(1)

return 1 * factorial(0)

Executes factorial(0)

Stack

Space Required for factorial(0)

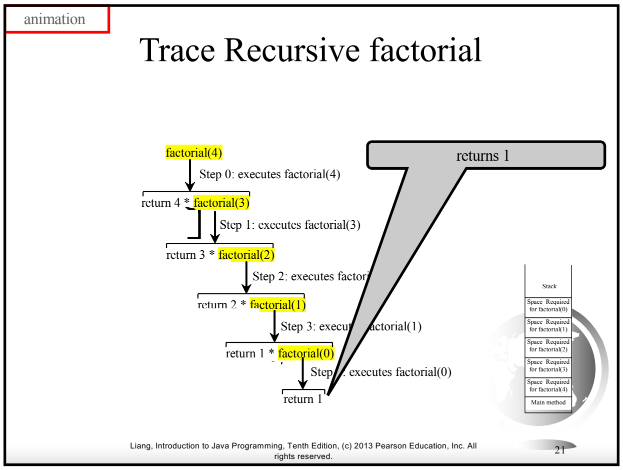
Space Required for factorial(1)

Space Required for factorial(2)

Space Required for factorial(3)

Space Required for factorial(4)

Main method



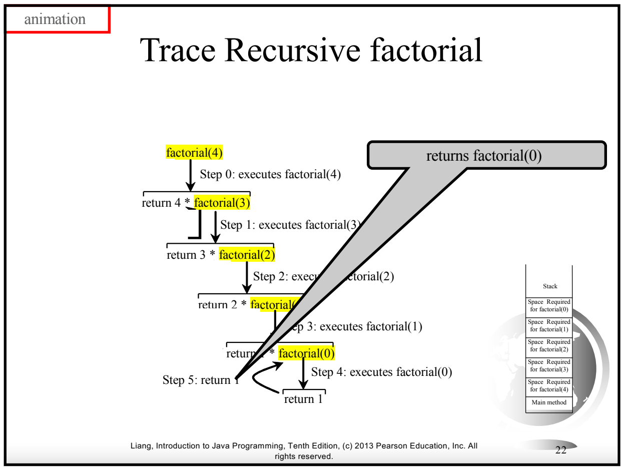
animation

Trace Recursive factorial

returns 1

factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factori
return 2 * factorial(1)
Step 3: execu actorial(1)
return 1 * factorial(0)
Step executes factorial(0)
return 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



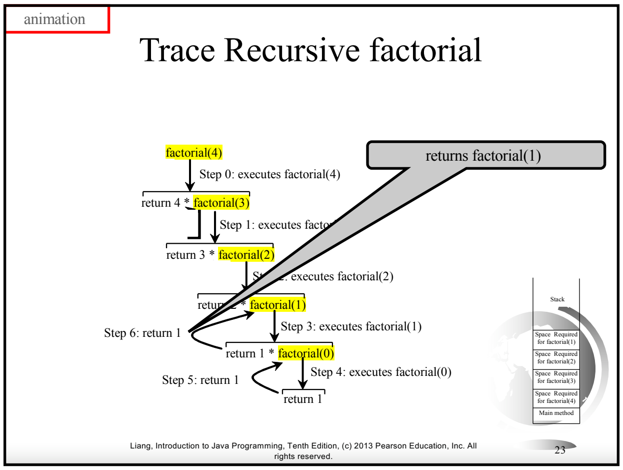
animation

Trace Recursive factorial

returns factorial(0)

factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3
return 3 * factorial(2)
Step 2: exec ctorial(2)
return 2 * factorial(
p 3: executes factorial(1)
return * factorial(0)
Step 4: executes factorial(0)
Step 5: return 1
return 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



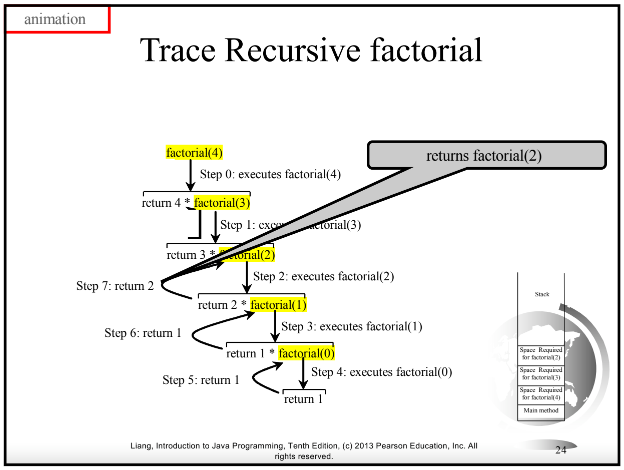
animation

Trace Recursive factorial

returns factorial(1)

factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes facto
return 3 * factorial(2)
St 2: executes factorial(2)
retur 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(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method



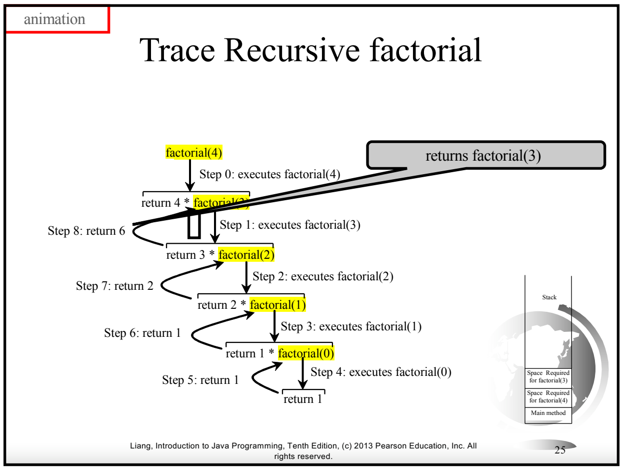
animation

Trace Recursive factorial

returns factorial(2)

factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: exec ctorial(3)
return 3 * torial(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(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method



animation

Trace Recursive factorial

returns factorial(3)

factorial(4)

Step 0: executes factorial(4)

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)

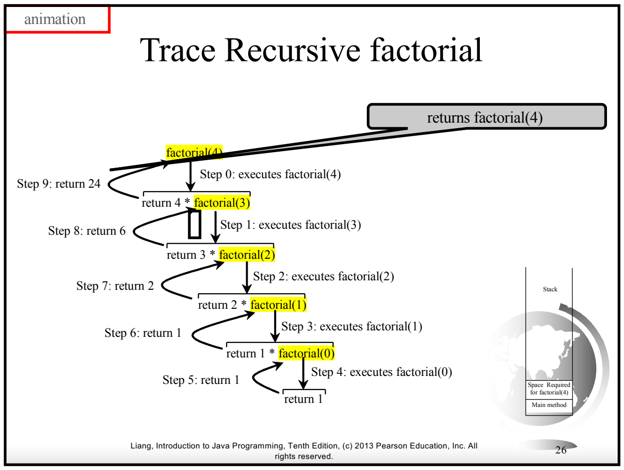
return 1

Stack

Space Required for factorial(3)

Space Required for factorial(4)

Main method



animation

Trace Recursive factorial

returns factorial(4)

factorial(4)

Step 9: return 24

Step 0: executes factorial(4)

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

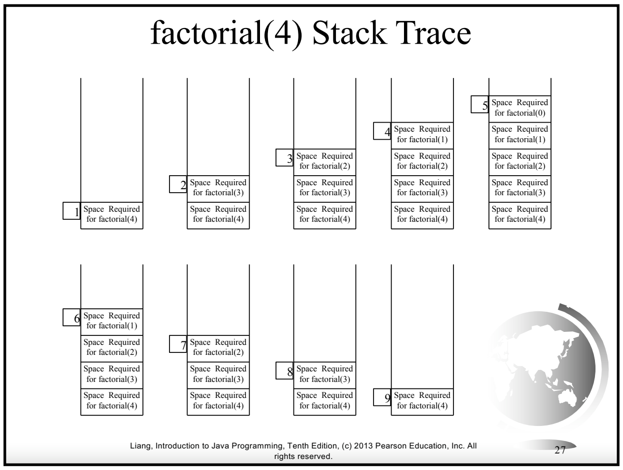
Step 4: executes factorial(0)

return 1

Stack

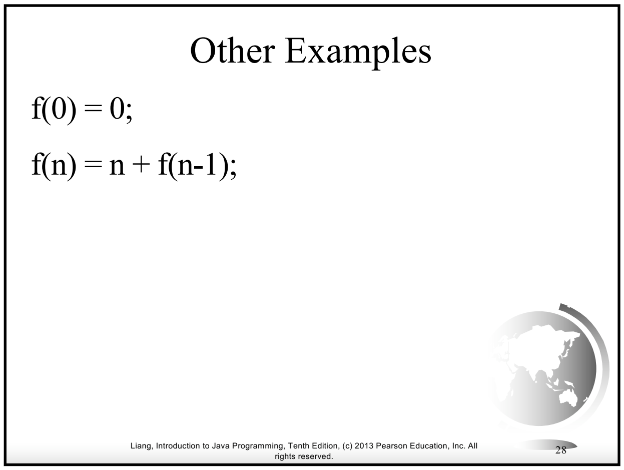
Space Required for factorial(4)

Main method



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)



Other Examples

f(0) = 0;

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

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;

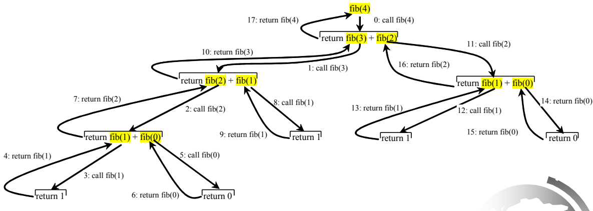
fib(1) = 1;

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

ComputeFibonacci Run

Fibonnaci Numbers, cont.


fib(4)
17: return fib(4)
0: call fib(4)
return fib(3) + fib(2)
10: return fib(3)
11: call fib(2)
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)
return fib(1) + fib(0)
9: return fib(1)
return 1
return 1
15: return fib(0)
return 0
4: return fib(1)
5: call fib(0)
3: call fib(1)
return 1
6: return fib(0)
return 0

Characteristics of Recursion

All recursive methods have the following characteristics:

- One or more base cases (the simplest case) are used to stop recursion.
- 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, you break it into subproblems. 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 with a smaller size.

Problem Solving Using Recursion

Let us consider a simple 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 second problem is the same as the original problem with a smaller size. The base case for the problem is n==0. You can solve this problem using recursion as follows:

nPrintln("Welcome", 5);

```
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
}
```

Think Recursively

Many of the problems presented in the early chapters can be solved using recursion if you *think recursively*. For example, the palindrome problem can be solved recursively as follows:

```
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));
}
```

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:

```
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);
}
```

Recursive Selection Sort

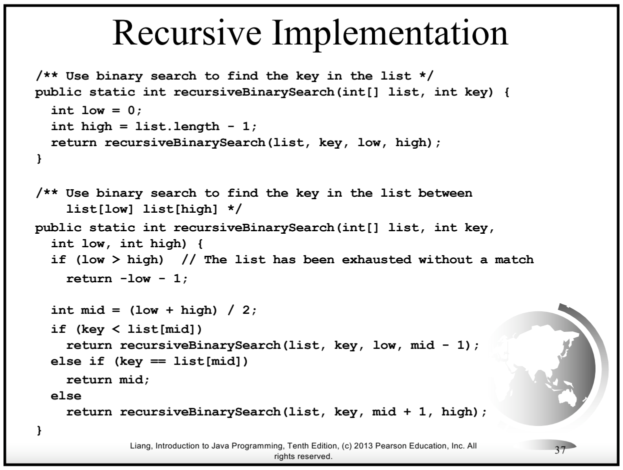
1. Find the smallest number in the list and swaps it with the first number.
2. Ignore the first number and sort the remaining smaller list recursively.

RecursiveSelectionSort

Recursive Binary Search

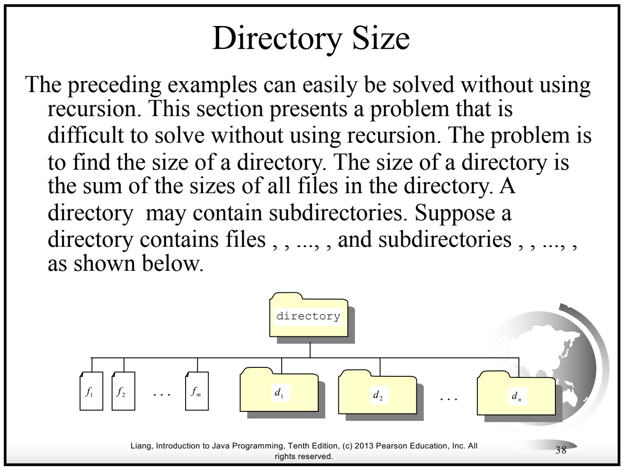
1. Case 1: If the key is less than the middle element, recursively search the key in the first half of the array.
2. Case 2: If the key is equal to the middle element, the search ends with a match.
3. Case 3: If the key is greater than the middle element, recursively search the key in the second half of the array.

RecursiveBinarySearch



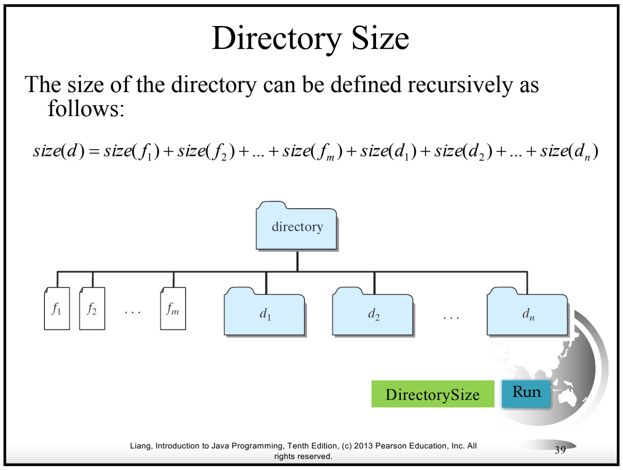
Recursive Implementation

```
/** Use binary search to find the key in the list */
public static int recursiveBinarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;
  return recursiveBinarySearch(list, key, low, high);
}

/** Use binary search to find the key in the list between
    list[low] list[high] */
public static int recursiveBinarySearch(int[] list, int key,
  int low, int high) {
  if (low > high)  // The list has been exhausted without a match
    return -low - 1;

  int mid = (low + high) / 2;
  if (key < list[mid])
    return recursiveBinarySearch(list, key, low, mid - 1);
  else if (key == list[mid])
    return mid;
  else
    return recursiveBinarySearch(list, key, mid + 1, high);
}
```



Directory Size

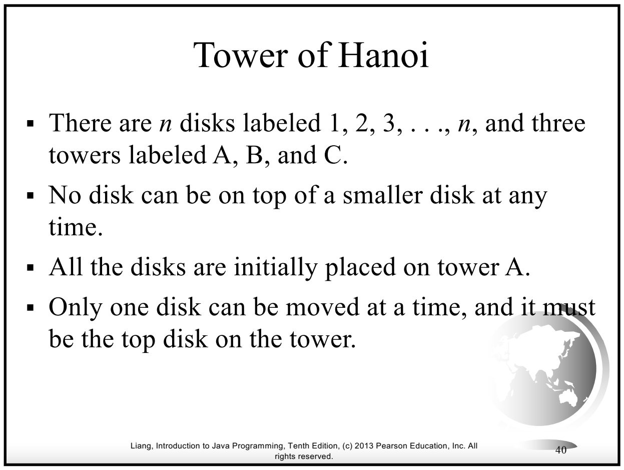
The preceding examples can easily be solved without using recursion. This section presents a problem that is difficult to solve without using recursion. The problem is to find the size of a directory. The size of a directory is the sum of the sizes of all files in the directory. A directory may contain subdirectories. Suppose a directory contains files , , ..., , and subdirectories , , ..., , as shown below.



Directory Size

The size of the directory can be defined recursively as follows:

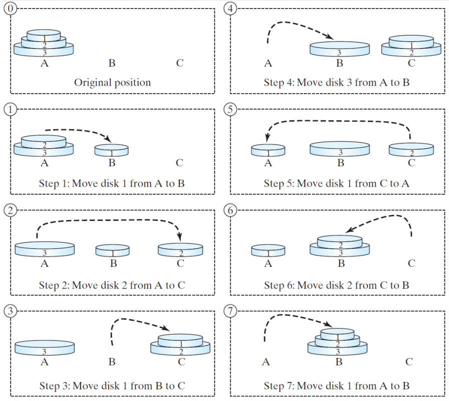
$$size(d) = size(f_1) + size(f_2) + ... + size(f_m) + size(d_1) + size(d_2) + ... + size(d_n)$$



Tower of Hanoi

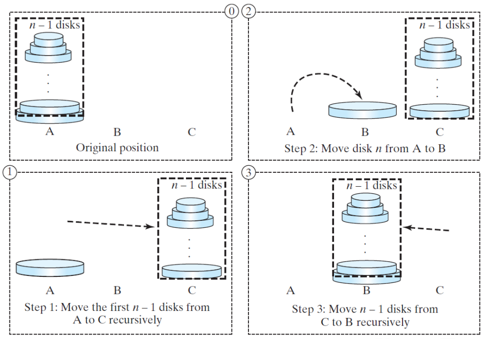
- There are *n* disks labeled 1, 2, 3, . . ., *n*, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

Tower of Hanoi, cont.


0 Original position
A B C
1 Step 1: Move disk 1 from A to B
2 Step 2: Move disk 2 from A to C
3 Step 3: Move disk 1 from B to C
4 Step 4: Move disk 3 from A to B
5 Step 5: Move disk 1 from C to A
6 Step 6: Move disk 2 from C to B
7 Step 7: Move disk 1 from A to B

Solution to Tower of Hanoi

The Tower of Hanoi problem can be decomposed into three subproblems


0 Original position
n – 1 disks
A B C
1 Step 1: Move the first n – 1 disks from A to C recursively
2 Step 2: Move disk n from A to B
3 Step 3: Move n – 1 disks from C to B recursively

Solution to Tower of Hanoi

- ☐ Move the first n - 1 disks from A to C with the assistance of tower B.
- ☐ Move disk n from A to B.
- ☐ Move n - 1 disks from C to B with the assistance of tower A.

TowerOfHanoi Run

Exercise 18.3 GCD

gcd(2, 3) = 1

gcd(2, 10) = 2

gcd(25, 35) = 5

gcd(205, 301) = 5

gcd(m, n)

Approach 1: Brute-force, start from min(n, m) down to 1, to check if a number is common divisor for both m and n, if so, it is the greatest common divisor.

Approach 2: `Euclid's` algorithm

Approach 3: Recursive method

Approach 2: `Euclid's` algorithm

```
// Get absolute value of m and n;
t1 = Math.abs(m); t2 = Math.abs(n);
// r is the remainder of t1 divided by t2;
r = t1 % t2;
while (r != 0) {
  t1 = t2;
  t2 = r;
  r = t1 % t2;
}

// When r is 0, t2 is the greatest common
// divisor between t1 and t2
return t2;
```

Approach 3: `Recursive Method`

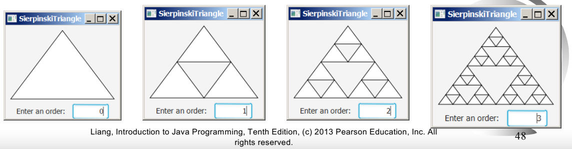
gcd(m, n) = n if m % n = 0;
gcd(m, n) = gcd(n, m % n); otherwise;

Fractals?

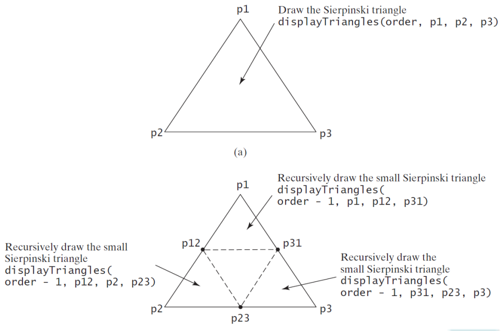
A fractal is a geometrical figure just like triangles, circles, and rectangles, but fractals can be divided into parts, each of which is a reduced-size copy of the whole. There are many interesting examples of fractals. This section introduces a simple fractal, called *Sierpinski triangle*, named after a famous Polish mathematician.

Sierpinski Triangle

1. It begins with an equilateral triangle, which is considered to be the Sierpinski fractal of order (or level) 0, as shown in Figure (a).
2. Connect the midpoints of the sides of the triangle of order 0 to create a Sierpinski triangle of order 1, as shown in Figure (b).
3. Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski of order 2, as shown in Figure (c).
4. You can repeat the same process recursively to create a Sierpinski triangle of order 3, 4, ..., and so on, as shown in Figure (d).



Sierpinski Triangle Solution


p1
Draw the Sierpinski triangle
displayTriangles(order, p1, p2, p3)
p2
p3
(a)
p1
Recursively draw the small Sierpinski triangle
displayTriangles(
order - 1, p1, p12, p31)
p12
p31
Recursively draw the small
Sierpinski triangle
displayTriangles(
order - 1, p12, p2, p23)
Recursively draw the
small Sierpinski triangle
displayTriangles(
order - 1, p31, p23, p3)
p2
p23
p3
(b)

SierpinskiTriangle Run

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 space for all of the method’s local variables and parameters. This can consume considerable memory and requires extra time to manage the additional space.

Advantages of Using Recursion

Recursion is good for solving the problems that are inherently recursive.

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.

Non-tail recursive ComputeFactorial

Tail recursive ComputeFactorialTailRecursion