Chapter 18 Recursion

CS1: Java Programming
Colorado State University

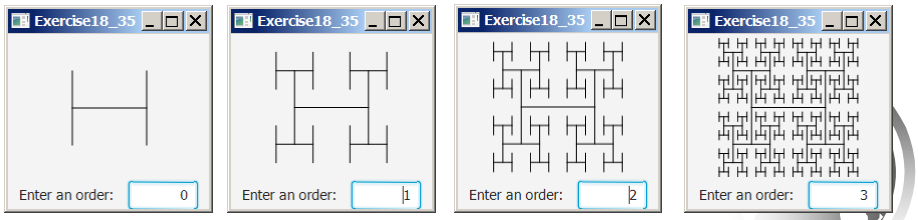
Original slides by Daniel Liang
Modified slides by Chris Wilcox

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.



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

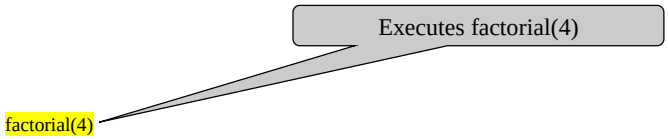
Computing Factorial

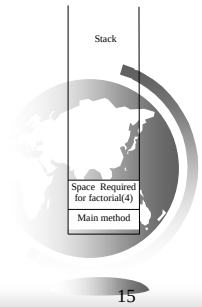
factorial(0) = 1;
factorial(n) = n*factorial(n-1);

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

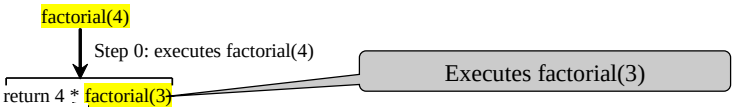
Trace Recursive factorial

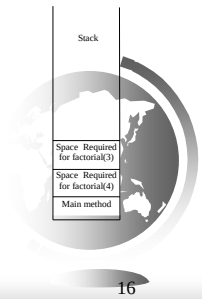




animation

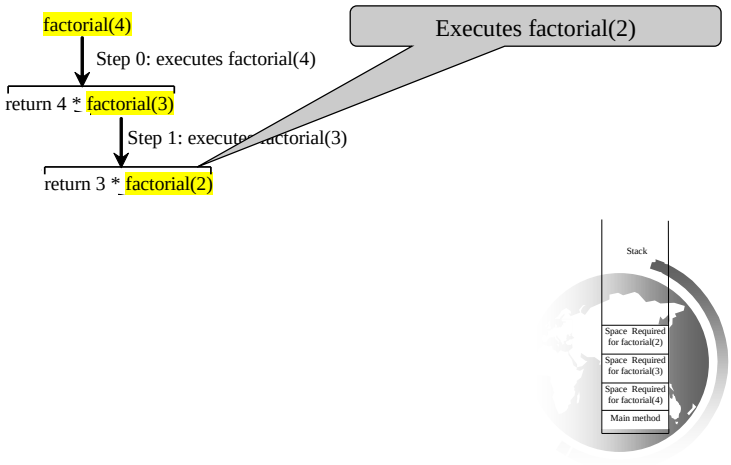
Trace Recursive factorial





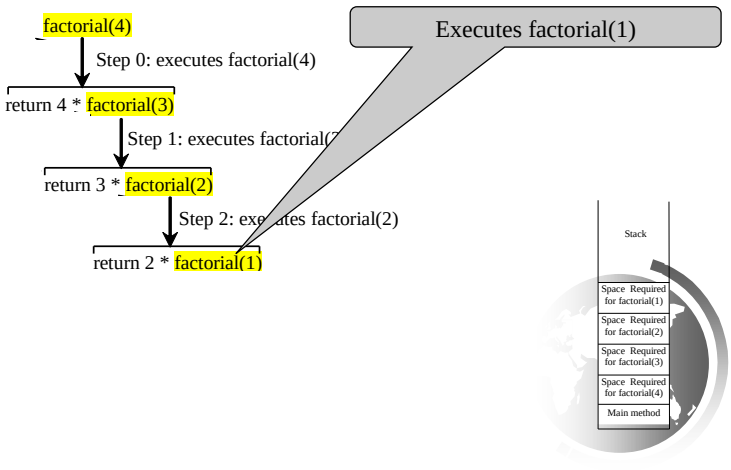
animation

Trace Recursive factorial



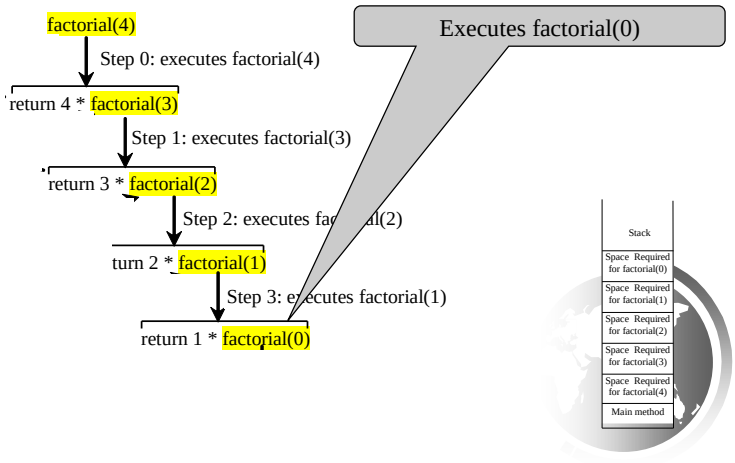
animation

Trace Recursive factorial



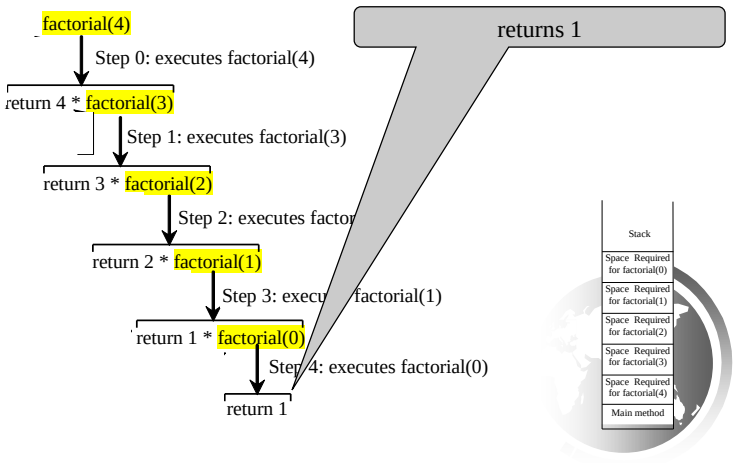
animation

Trace Recursive factorial



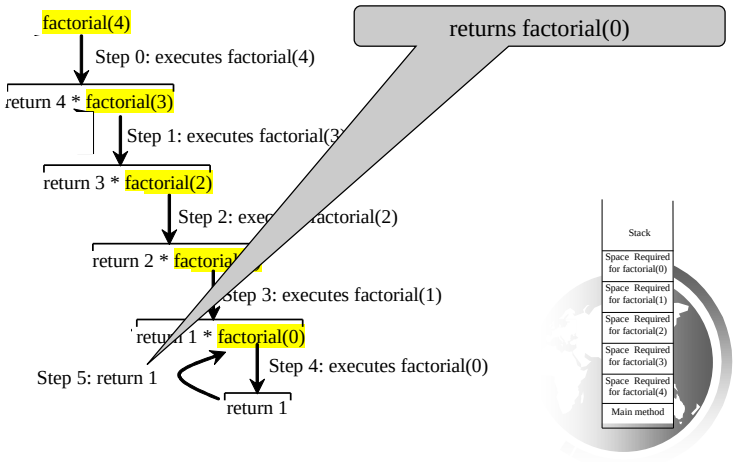
animation

Trace Recursive factorial



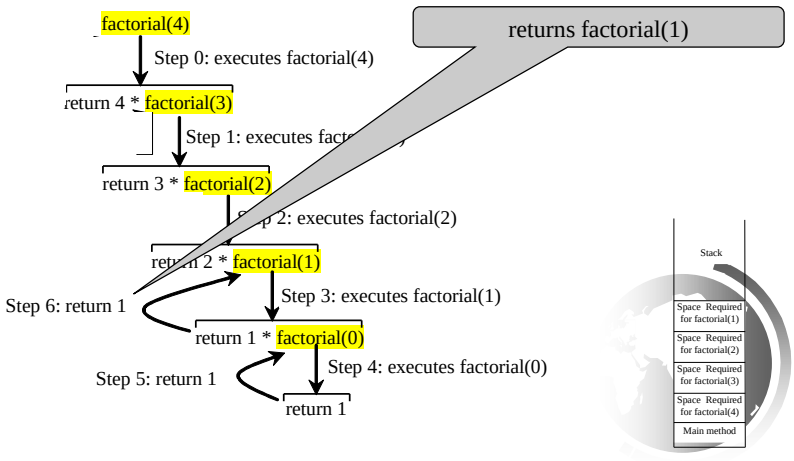
animation

Trace Recursive factorial



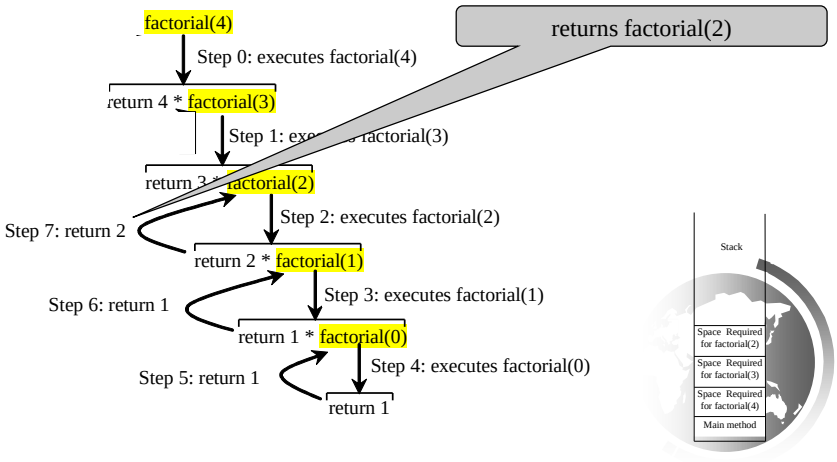
animation

Trace Recursive factorial



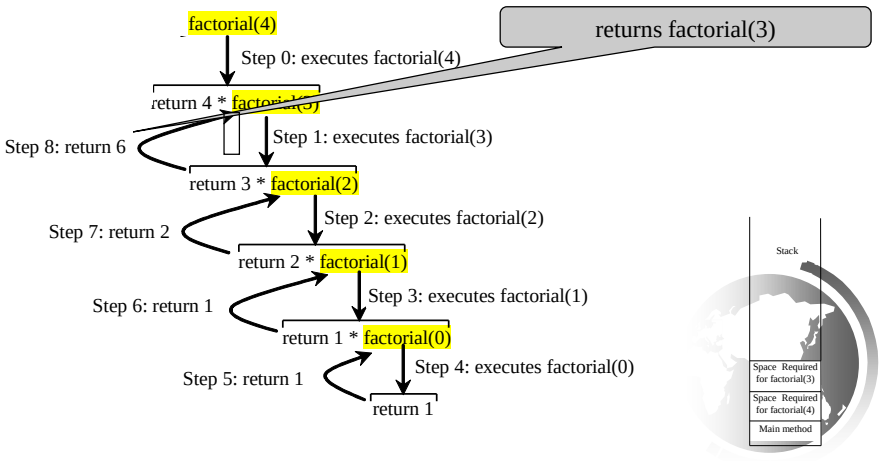
animation

Trace Recursive factorial



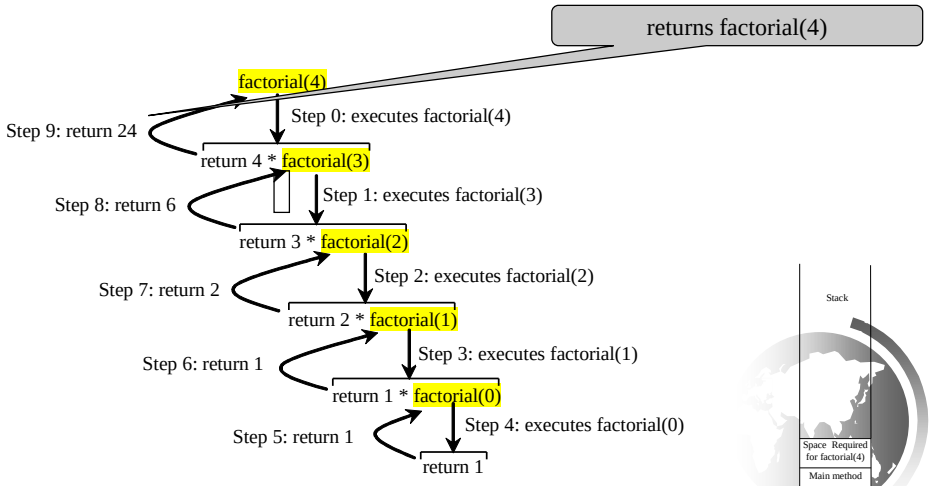
animation

Trace Recursive factorial

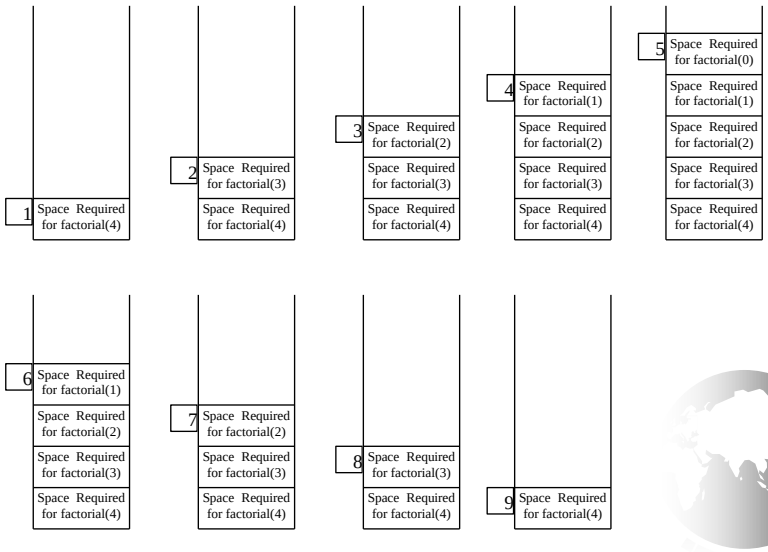


animation

Trace Recursive factorial



factorial(4) Stack Trace



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;

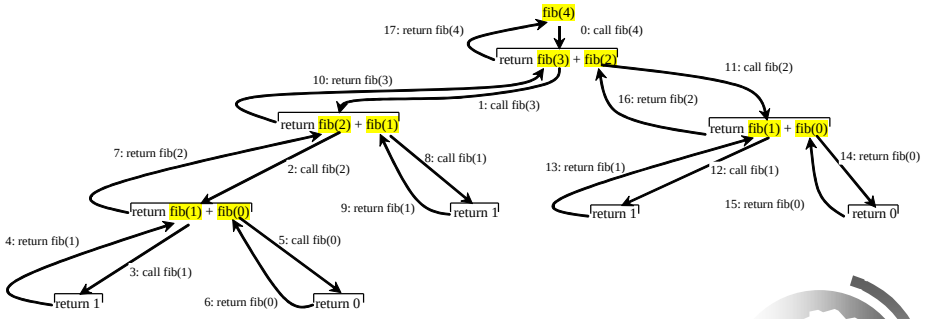
fib(1) = 1;

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.



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.

On to peer instruction

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
}
```

Exercise

- Let's write a method `reverseLines(Scanner scan)` that reads lines using the scanner and prints them in reverse order.
  - Use recursion without using loops.
  - Example input:

```
this
is
fun
no?
```

→

Expected output:

```
no?
fun
is
this
```

  - What are the cases to consider?
    - How can we solve a small part of the problem at a time?
    - What is a file that is very easy to reverse?

Reversal pseudocode

- Reversing the lines of a file:
  - Read a line L from the file.
  - Print the rest of the lines in reverse order.
  - Print the line L.

- If only we had a way to reverse the rest of the lines of the file....

Reversal solution

```
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case
        String line = input.nextLine();
        reverseLines(input);
        System.out.println(line);
    }
}
```

– Where is the base case?

Tracing our algorithm

- **Call stack**: The method invocations active

```
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {    // false
        ...
    }
}
}
```

output:

```
no?
fun
is
this
```

input file:

```
this
is
fun
no?
```

Recursive Helper Methods

This reverseString method is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, use a helper method:

```
public String reverseString(String s){
   if (s.length() == 0)
      return s;
   return reverseString(s.substring(1)) + s.charAt(0);
}
```

Recursive Helper Methods

reverseString method with a helper method:

```
public String reverseString2(String s){
   if (s.length() == 0)
      return s;
   return reverseString2(s,0);
}
public String reverseString2(String s, int index){
   if (index == s.length())
      return "";
   return reverseString2(s,index+1) + s.charAt(index);
}
```

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

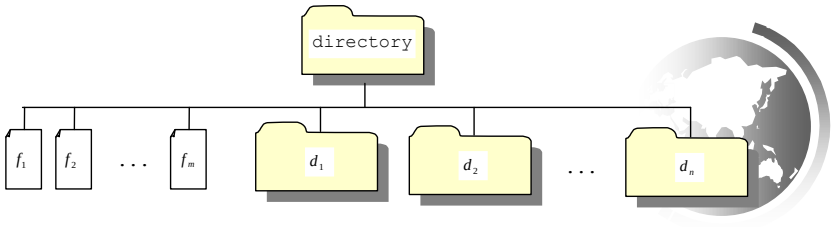
```
/** Use binary search to find the key in the list */
public static int recursiveBinarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;
  return recursiveBinarySearch(list, key, low, high);
}

/** 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

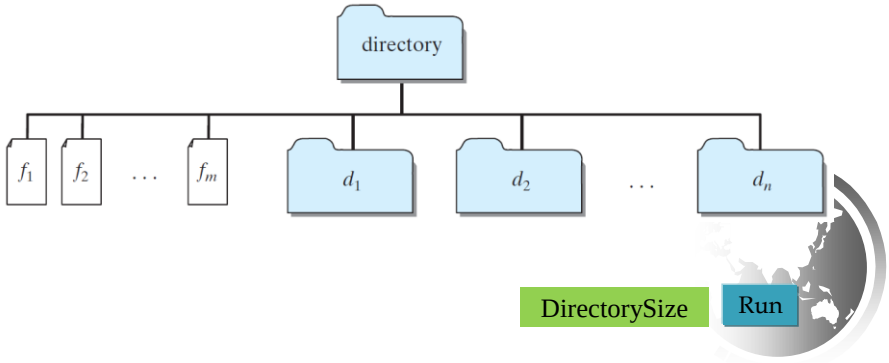
The preceding examples can easily be solved without using recursion. This section presents a problem that is difficult to solve without using recursion. The problem is to find the size of a directory. The size of a directory is the sum of the sizes of all files in the directory. A directory may contain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



Directory Size

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

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



Spock's dilemma

- Entering a star system for the first time, Spock has a limited time before he has to go pick up Kirk.
  - There are n number of planets
  - Spock has time to visit k planets
- How many different combinations of planets can Spock visit?

Spock's dilemma

```
public long combRec(long n, long k){
if (n==k || k==0)
        return 1;
else
        return combRec(n-1,k-1) +
                combRec(n-1,k);
```

mkchange (int n)

Precondition: n >= 0

Your method must return the number of ways amount n can be paid with quarters (25c), dimes (10c), nickels(5c), and penneys (1c).

For example, 10 cent can be paid in four ways:

1. ten penneys
2. a nickel and five penneys
3. two nickels
4. one dime

mkchange (int n)

```
public static final int[] coins = {1, 5, 10, 25};

public int mkChange(int n){
    return mkChange(coins.length-1,n);
}
```

mkchange (int k, int n)

```
public int mkChange(int k, int n){
        if (n < 0 || k < 0)
                return 0;
        if (n == 0)
                return 1;
        return mkChange(k-1,n) +
                mkChange(k, n-coins[k]);
}
```

pentagonPark (int n)

- pentagonPark computes in how many different ways a parking lot of size n can be filled with three kinds of vehicles:
  - Civics, size 1
  - Explorers, size 2
  - Tanks, size 3
- Here are some examples:
  - A parking lot of size 1 can have 1 Civic (C), so the answer is 1.
  - A parking lot of size 2 can have 1 Explorer (E) or two Civics (CC), so the answer is 2.
  - A parking lot of size 3 can have one Tank (T), a Civic and an Explorer (CE), or an Explorer and a Civic (EC), or 3 Civics (CCC), so the answer is 4.

pentagonPark (int n)

```
public static long pentagonPark (int n)
  {
    if (n == 1) return 1;  // a Civic
    else if (n == 2) return 2;  // an Excursion or two Civics
    else if (n == 3) return 4;   //CCC; CE; EC; T
    else return pentagonPark(n-3)  // tank in last position
           + pentagonPark(n-2)  // Excursion in last position
           + pentagonPark(n-1); // Civic in last position
  }
```

Memoization

- Problems like Fibonacci and Pentagon Park create “bushy” trees.
- These trees are full of repeated calls
- Tremendous speedup by saving intermediate results

Fast Fib

```
private long[] memo = new long[100];
public long fastFibo(int n){
    if(n<2) return n;
    if (memo[n]==0)
        memo[n] = fastFibo(n-1) +
                  fastFibo(n-2);
    return memo[n];
}
```

Fast Spock

```
public static long spockDilemma (int n, int k, long [][] A)
  if (A[n][k] == 0)
    {
     if (k == 0 ||  n == k)   // pick nobody or pick everybody
       A[n][k] = 1;
     else
       A[n][k] = spockDilemma(n-1,k,A)  // pick a committee without you
            + spockDilemma(n-1,k-1,A); // pick a committee with you
    }
    return A[n][k];
  }
```

Fast pentagonPark

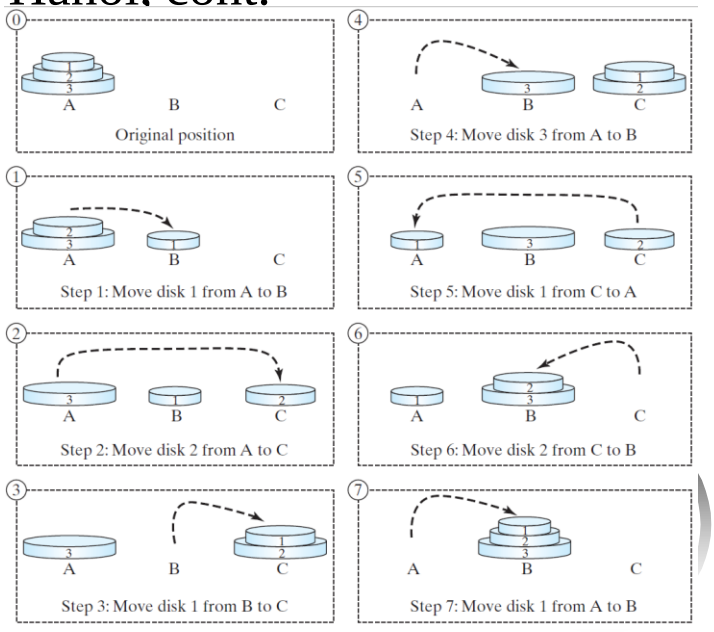
```
public static long pentagonPark (int n, long [] A)
    if (A[n] == 0)  // you haven't already solved this subproblem
    {
      if (n == 1) A[n] = 1;  // a Civic
      else if (n == 2) A[n] = 2;  // an Excursion or two Civics
      else if (n == 3) A[n] = 4;   //CCC; CE; EC; T
      else
         A[n] = pentagonPark(n-3,A)  // tank in last position
              + pentagonPark(n-2,A)  // Excursion in last position
              + pentagonPark(n-1,A); // Civic in last position
    }
    return A[n];
  }
```

On to peer instruction

Tower of Hanoi

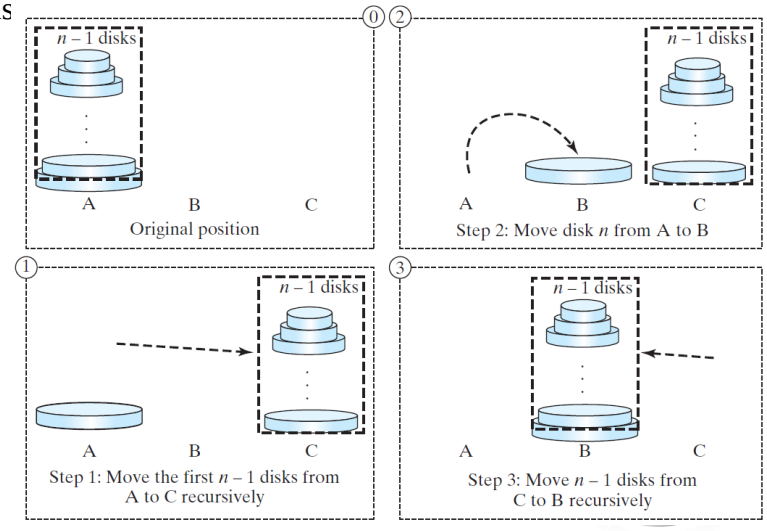
- There are *n* disks labeled 1, 2, 3, . . ., *n*, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

Tower of Hanoi, cont.



Solution to Tower of Hanoi

The Tower of Hanoi problem can be decomposed into three subproblems



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`

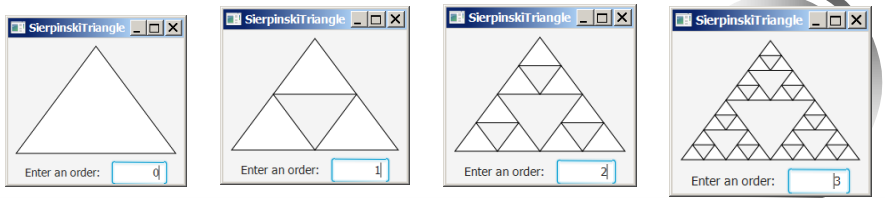
gcd(m, n) = n if m % n = 0;
gcd(m, n) = gcd(n, m % n); otherwise;

Fractals?

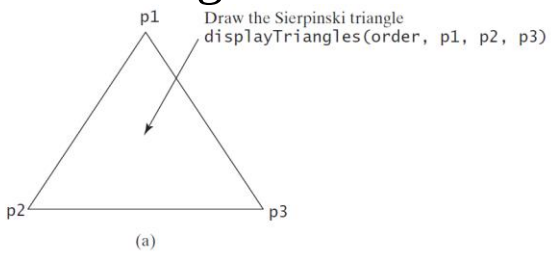
A fractal is a geometrical figure just like triangles, circles, and rectangles, but fractals can be divided into parts, each of which is a reduced-size copy of the whole. There are many interesting examples of fractals. This section introduces a simple fractal, called *Sierpinski triangle*, named after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

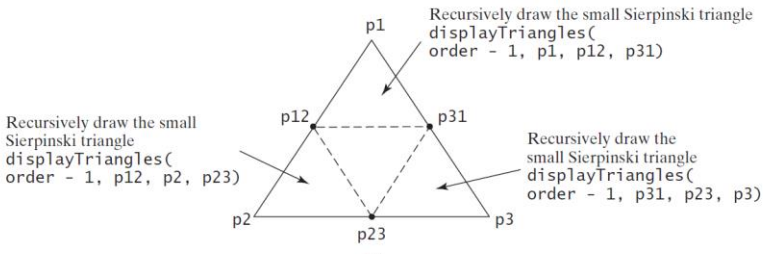
Sierpinski Triangle

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





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