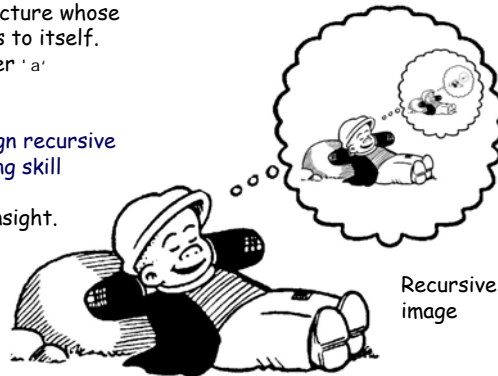


## Lecture 9.1 - 9.2

# Recursion

## Recursion

- **Recursion:** a fundamental algorithmic technique, based on *divide and conquer*
- **Recursive function:** a mathematical function defined in terms of itself.  
Example:  $n! = n \cdot (n-1)!$
- **Recursive method:** a method that calls itself on smaller subsets of the problem space.  
Example: list all the files in a given directory
- **Recursive data structure:** a data structure whose elements are defined using references to itself.  
Example: the list "abcd" is the character 'a' followed by the list "bcd"
- Learning to think recursively and design recursive programs is a fundamental programming skill
- But, mastering it takes practice and insight.



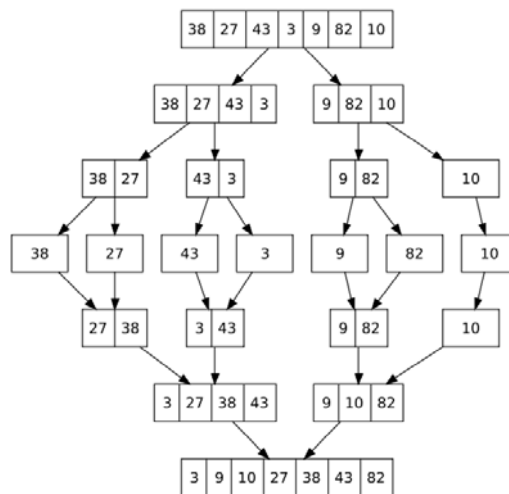
## Outline

- ➔ ■ Recursive algorithms
- Recursive functions
  - Factorial
  - Fibonacci
  - Power
- Recursive procedures
  - File listing
  - Reverse
  - Permutations
  - Integer.toString
  - Tower of Hanoi
  - Permutations
  - Fractals

## The building blocks of a recursive design

Every recursive algorithm is based on three design elements:

- **Reduction:**  
it must be possible to reduce the original problem into sub-problems that are simpler instances of the same problem
- **Base case:**  
At some point of the reduction we must arrive to a sub-problem that can be solved directly
- **Assembly:**  
Once the sub-problems have been solved, it must be possible to combine the sub-solutions into a solution of the original problem.



Merge Sort:  
a recursive algorithm example

## Factorial

### Iterative definition

$$factorial(n) = \begin{cases} 1 & \text{if } n=1 \text{ or } n=0 \\ n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1 & \text{otherwise} \end{cases}$$

Example:  $5! = factorial(5) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

### Iterative implementation

```
public class FactorialDemo1 {
    public static void main(String[] args) {
        System.out.println("5! = " + factorial(5));
    }
    // Returns the factorial (n!) of a given n.
    public static long factorial(long n) {
        long fact = 1;
        for (int i=1; i<=n; i++) {
            fact *= i;
        }
        return fact;
    }
}
```

Example of a  
non-recursive solution

## Factorial: a recursive solution

### Recursive definition

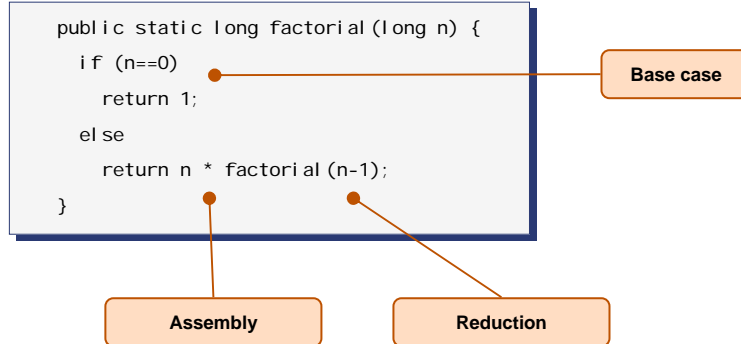
$$factorial(n) = \begin{cases} 1 & \text{if } n=1 \text{ or } n=0 \\ n \cdot factorial(n-1) & \text{otherwise} \end{cases}$$

$factorial(5) = 5 * factorial(4) =$   
 $5 * 4 * factorial(3) =$   
 $5 * 4 * 3 * factorial(2) =$   
 $5 * 4 * 3 * 2 * factorial(1) =$   
 $5 * 4 * 3 * 2 * 1 * factorial(0) =$   
 $5 * 4 * 3 * 2 * 1 * 1 = 120$

### Recursive implementation

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

## Elements of the recursive approach

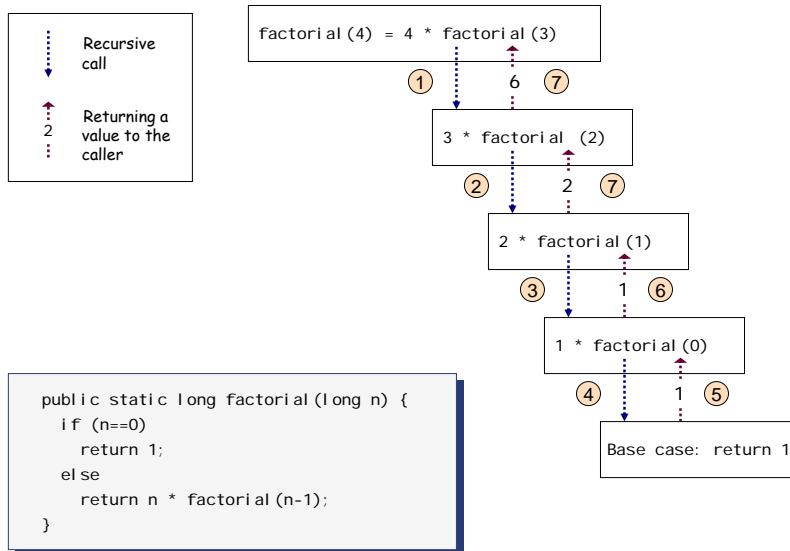


A one-liner ...

```

public static long factorial (long n){return ((n==1) ? 1 : n * factorial (n-1));}
    
```

## Run-time anatomy



## Sum

The problem stated

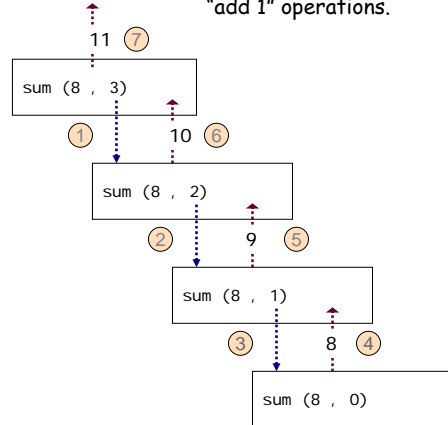
```
sum(a, b) = ?  
sum(a, 0) = a  
sum(a, b) = sum(a, (b - 1)) + 1    for b > 0  
sum(a, b) = sum(a, b--) ++      // in pseudo code
```

The challenge: defining sum without actually summing anything.

Instead, we reduce the problem into a series of "add 1" operations.

Recursive algorithm

```
sum(a, b) {  
  if (b == 0) return a  
  s = sum(a, --b)  
  return ++s  
}
```



## In Java

Algorithm

```
sum(a, b) {  
  if (b == 0) return a  
  s = sum(a, --b)  
  return ++s  
}
```

Implementation

```
public class SumDemo {  
  
  public static void main(String args[]) {  
    System.out.println(sum(5, 3));  
  }  
  
  public static long sum(long a, long b) {  
    if (b == 0)  
      return a;  
    long s = sum(a, --b);  
    return ++s;  
  }  
}
```

## Fibonacci series

### Fibonacci series definition

```
fib(0) = 1
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2) for all n > 1
```

The Fibonacci series: 1, 1, 2, 3, 5, 8, 13, ...

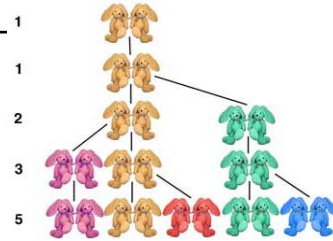
### Recursive implementation

```
public class Fibonacci Demo {
    public static void main(String args[]) {
        System.out.println(fibonacci(5));
    }

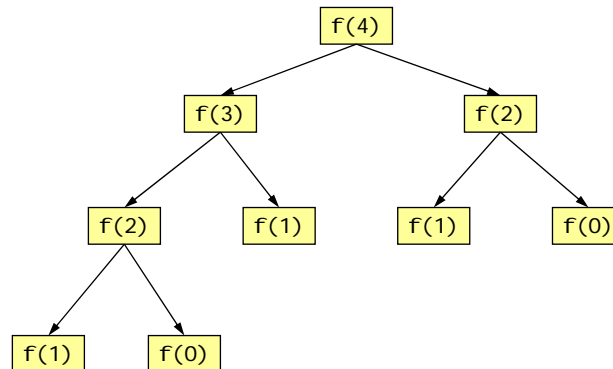
    //Returns the n number in fibonacci series
    public static int fibonacci(int n) {
        if (n <= 1) {
            return 1;
        }
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

The recursive implementation follows directly from the recursive definition:

- **Base case:**  $n=0, n=1$
- **Reduction:** from  $n$  to  $(n-1)$  and  $(n-2)$
- **Assembly:** using +



## Fibonacci series: Run-time anatomy



- **Built-in redundancy:** the algorithm repeats the same computations (e.g.  $f(2)$ )
- **Running time of this algorithm:**  $\sim O(2^n)$
- **Q:** Can we do better?
- **A:** A bottom-up, iterative solution will run in  $O(n)$
- **Conclusion:** Naïve recursive solutions can be expensive!

## Outline

- Recursive algorithms
- Recursive functions
  - Factorial
  - Fibonacci
  - ➔ ● Power
- Recursive procedures
  - File listing
  - Reverse
  - Permutations
  - Integer.toString
  - Tower of Hanoi
  - Permutations
  - Fractals

## Power function: recursive implementation

### Recursive definition:

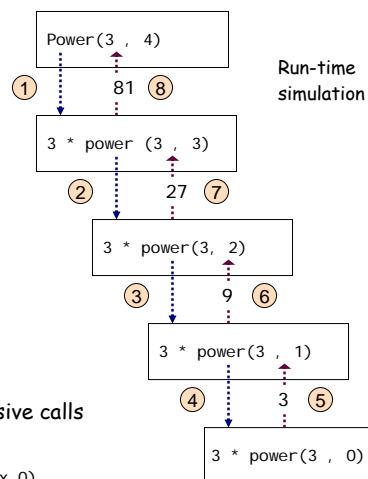
```
power(x, 0) = 1
Power(x, n) = n * power(x, n-1) for all n>0
```

### Recursive implementation:

```
// Computes x raised to the power of n
power(x, n) {
  if n = 0 return 1
  return x * power(x, n-1)
}
```

### Running time of recursive programs:

- Simply add up the running time of all the recursive calls
- $\text{power}(x, n)$  requires  $n+1$  recursive calls:  
 $\text{power}(x, n), \text{power}(x, n-1), \dots, \text{power}(x, 1), \text{power}(x, 0)$
- Total running time =  $(n+1) O(1) = O(n)$



## Power function: recursive implementation, Take 2

```
// Computes x raised
// to the power of n

power(x, n) {
  if (n = 0) return 1
  if (n%2 = 0) {
    t = power(x, n/2)
    return t * t
  }
  return x * power(x, n-1)
}
```

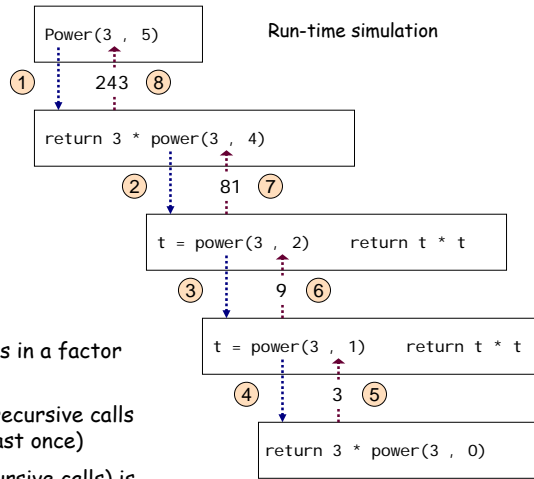
**Running-time:**  $O(\log n)$

After two recursive calls  $n$  decreases in a factor of at least 2

(Either  $n$  or  $n-1$  is even, thus in two recursive calls the algorithm divides  $n$  by 2 at least once)

Thus the total number of steps (recursive calls) is at most  $2 * \log n$

Thus the running time is  $O(\log n)$ .



## Recursive power, take 2

```
power(x, n) {
  if (n = 0) return 1;
  if (n%2 = 0) {
    t = power(x, n/2);
    return t * t;
  }
  return x * power(x, n-1);
}
```

**Theorem:** For any  $x$  and positive integer  $n$ , the algorithm returns  $x^n$

**Proof:** By strong induction on  $n$ .

Base case: if  $n=0$  the algorithm returns 1.

Inductive hypothesis: Assume that for all  $k < n$  the algorithm returns  $x^k$

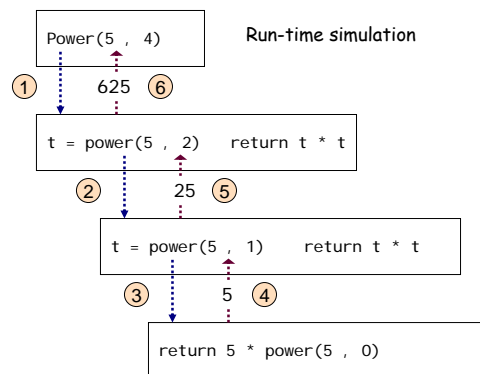
Inductive step:

If  $n$  is even, the algorithm returns  $\text{power}(x, n/2) * \text{power}(x, n/2)$

By the induction hypothesis,  $\text{power}(x, n/2)$  returns  $x^{\frac{1}{2}n}$ , thus the algorithm returns  $(x^{\frac{1}{2}n}) * (x^{\frac{1}{2}n}) = x^n$

If  $n$  is odd, the algorithm returns  $x * \text{power}(x, n-1)$

By the induction hypothesis,  $\text{power}(x, n-1)$  returns  $x^{n-1}$ , thus the algorithm returns  $x * (x^{n-1}) = x^n$

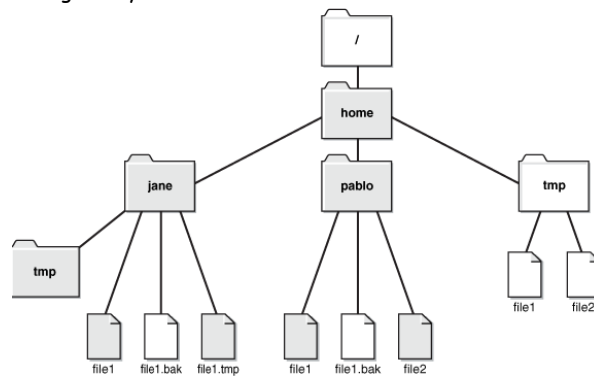


## Outline

- Recursive algorithms
- Recursive functions
  - Factorial
  - Fibonacci
  - Power
- ➔ ■ Recursive procedures
  - File listing
  - Reverse
  - Permutations
  - Integer.toString
  - Tower of Hanoi
  - Permutations
  - Fractals

## Recursive procedures

- Many functions such as factorial, Fibonacci, etc. have inherent recursive definitions. Therefore, implementing them using recursive methods is natural
- Procedures, however, are designed to do various things without returning values (in Java, implemented as void methods)
- In many cases, procedures can also be described in recursive terms
- Example: files listing utility

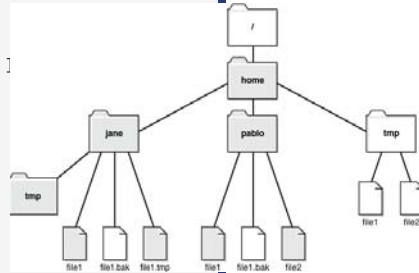


## File listing

```
// List all files under a given directory
private static void listFiles(File directory) {
    // Creates an array of all file names
    String[] itemNames = directory.listFiles();
    for (int i=0 ; i<itemNames.length ; i++) {
        File file = new File(directory, itemNames[i])
        if (file.isDirectory())
            printDirectoryDetails(file);
            listFiles(file);
        else
            printFileDetails(file);
    }
}

// Prints directory name and details
private static void printDirectoryDetails(File directory) {
}

// Prints file name and details
private static void printFileDetails(File file) {
}
```

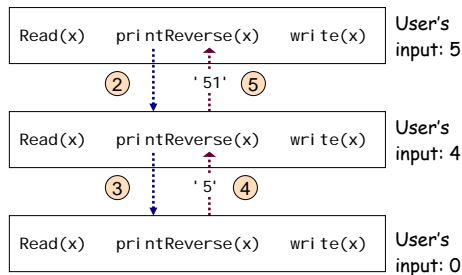


## Reverse

```
printReverse()
    print("Enter a number, or 0 to end: ")
    read(x)
    if (x != 0) {
        printReverse();
        print(x);
    }
}
```

cmd C:\WINDOWS\system32\cmd.exe

```
D:\demo>javac PrintReverseDemo.java
D:\demo>java PrintReverseDemo
Enter a number, or 0 to end: 8
Enter a number, or 0 to end: 7
Enter a number, or 0 to end: 6
Enter a number, or 0 to end: 5
Enter a number, or 0 to end: 4
Enter a number, or 0 to end: 0
4
5
6
7
8
```



- Method calling is a complex process, managed behind the scene
- When a method calls another, the caller's variables and return address are saved
- When the called method returns, the saved values are re-instantiated
- As far as the caller is concerned, everything is back to normal.

## Integer.toString()

The task:

Write a method that takes a non-negative integer and outputs its decimal representation using character output

Example: Given the integer input 513, output the string "513"

The Unicode values of '0', '1', '2', '3', ..., '9' are 48, 49, 50, 51, ..., 57

Therefore, if  $n < 10$ , one way to get its Unicode value is  $(\text{char})(\text{'0'} + n)$

Recursive thinking:

- Base case: If  $n < 10$  we output  $(\text{char})(\text{'0'} + n)$
- Reduction: an integer of the form  $n = \text{xyyy} \dots \text{y}$  (each  $x$  and  $y$  being a single digit) can be reduced using  $x = n/10$  and  $\text{yyy} \dots \text{y} = n\%10$
- Assembly: The concatenation operator  $+$  can be used to combine partial results into the final return value.

## Recursive implementation

The task:

Write a method that takes a non-negative integer and outputs its decimal representation using character output

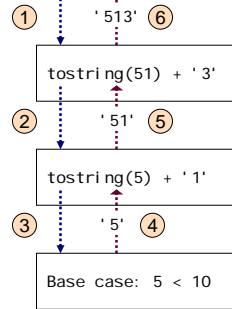
Example: Given the integer input 513, output the string "513"

```
public class PrintIntegerDemo {  
  
    public static void main(String[] args) {  
        System.out.print(toString(513));  
    }  
  
    static String toString(int n) {  
        if (n < 10)  
            return "" + ((char) (48 + n));  
        else  
            return toString(n / 10) + (char) (48 + n % 10);  
    }  
}
```

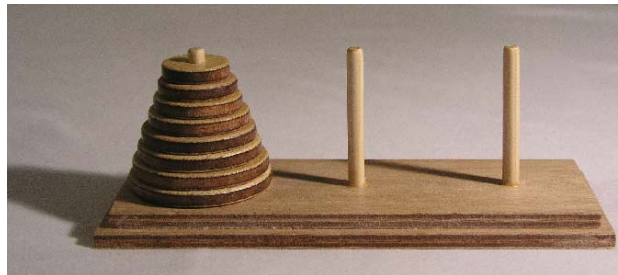
## Run-time anatomy

```
static String toString(int n) {  
    if (n < 10)  
        return "" + (char) (48 + n);  
    else  
        return toString(n / 10) + (char) (48 + n % 10);  
}
```

toString(5137) = toString(513) + '7'



## Tower of Hanoi

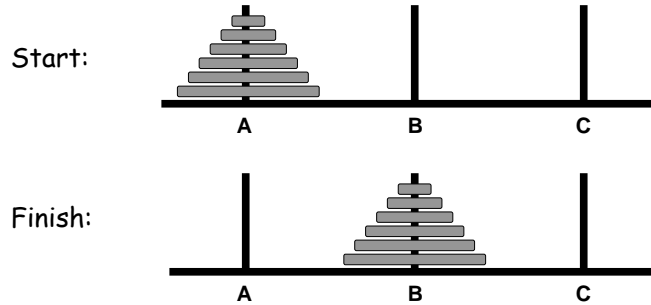


The rules of the game:

Move all the disks to another spindle, so that:

- (1) only one disk moves at a time
- (2) The smaller disks are always at the top

## Tower of Hanoi

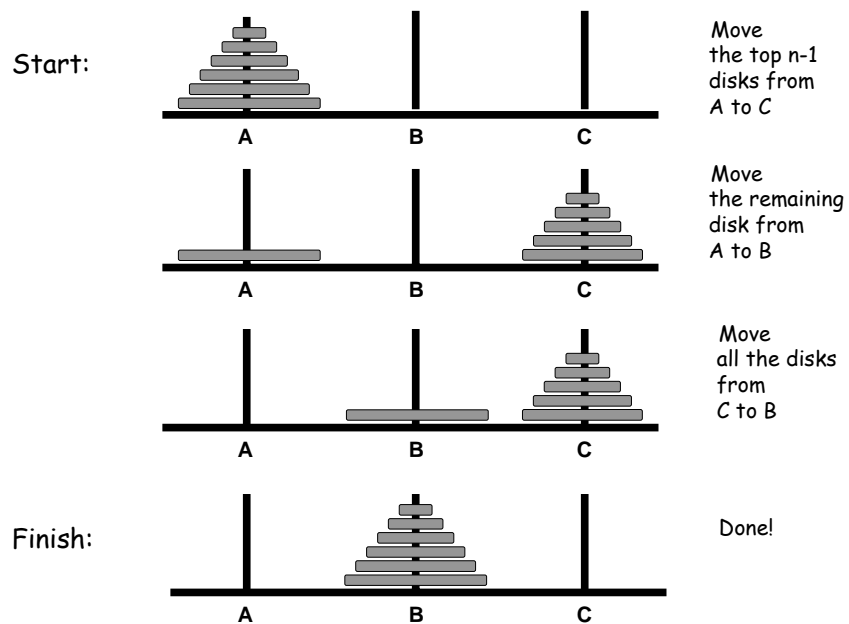


The rules of the game:

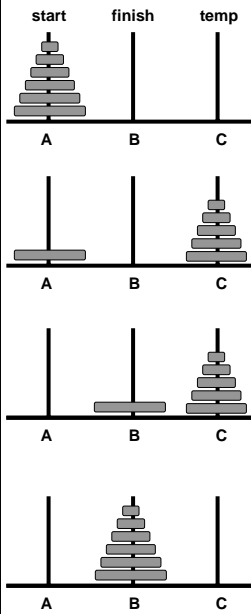
Move all the disks to another spindle, so that:

- (1) only one disk moves at a time
- (2) The smaller disks are always at the top

## The recursive strategy



## Tower of Hanoi implementation

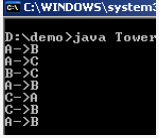


if  $n = 1$ : move this single disk from start to finish  
 if  $n > 1$ :

1. Move the top  $n-1$  disks from start to temp, using finish
2. Move the bottom disk from start to finish
3. Move the top  $n-1$  disks from temp to finish, using start

```
public class TowerOfHanoi {
    public static void main(String[] args) {
        moveTower(3, "A", "B", "C");
    }

    // Moves a tower of disks from A to B using C
    static void moveTower(int n, String start,
        String finish, String temp) {
        if (n==1)
            System.out.println(start + "->" + finish);
        else {
            moveTower(n-1, start, temp, finish);
            System.out.println(start + "->" + finish);
            moveTower(n-1, temp, finish, start);
        }
    }
}
```



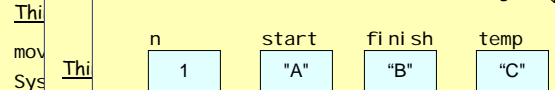
## Simulation

moveTower(3, "A", "B", "C")

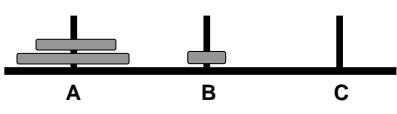
Goal: move a tower of size 3 from A to B using C:

Goal: move a tower of size 2 from A to C using B:

Goal: move a tower of size 1 from A to B using C: ←



Things to do:  
 moveTower(n-1, start, temp, finish);  
 System.out.println(start + "->" + finish);  
 moveTower(n-1, temp, finish, start);



## Simulation

moveTower(3, "A", "B", "C")

Goal: move a tower of size 3 from A to B using C:

Goal: move a tower of size 2 from A to C using B:

n	start	finish	temp
2	"A"	"C"	"B"

This

move

System

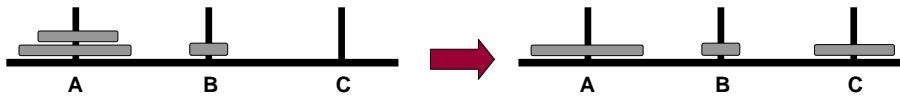
move

Things to do:

~~moveTower(n-1, start, temp, finish);~~

~~System.out.println(start + "->" + finish);~~

moveTower(n-1, temp, finish, start);



## Simulation

moveTower(3, "A", "B", "C")

Goal: move a tower of size 3 from A to B using C:

Goal: move a tower of size 2 from A to C using B:

n	start	finish	temp
2	"A"	"C"	"B"

This

move

System

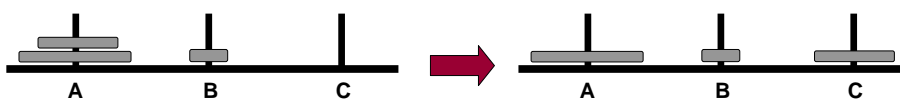
move

Things to do:

~~moveTower(n-1, start, temp, finish);~~

~~System.out.println(start + "->" + finish);~~

moveTower(n-1, temp, finish, start);



## Simulation (TBD)

`moveTower(3, "A", "B", "C")`

Goal: move a tower of size 3 from A to B using C:

Goal: move a tower of size 2 from A to C using B:

n	start	finish	temp
2	"A"	"C"	"B"

Thi

mov

Sys

mov

Things to do:

~~`moveTower(n-1, start, temp, finish);`~~

~~`System.out.println(start + " -> " + finish);`~~

`moveTower(n-1, temp, finish, start);`



## Permutations

The task:

Write a method `listPermutations(String s)` that generates a complete set of all the permutations of the characters in `s`.

For example, `listPermutations("abcd")` will generate:

```
abcd  bacd  cabd  dabc
abdc  badc  cadb  dacb
acbd  bcad  cbad  dbac
acdb  bcda  cbda  dbca
adbc  bdac  cdab  dcab
adcb  bdca  cdba  dcba
```

(one after the other)

Observations:

- If `length(s) = n`, there are  $n * (n-1) * (n-2) * \dots * 1 = n!$  permutations
- the list of permutations starting with "a" contains the prefix "a" followed by all the permutations of "bcd"

## Strategy

Given "abcd" we have to generate:

```
abcd  bacd  cabd  dabc
abdc  badc  cadb  dacb
acbd  bcad  cbad  dbac
acdb  bcda  cbda  dbca
adbc  bdac  cdab  dcab
adcb  bdca  cdba  dcba
```

Print 'a' + all permutations of "abcd" without charAt 0

Print 'b' + all permutations of "abcd" without charAt 1

Print 'c' + all permutations of "abcd" without charAt 2

Print 'd' + all permutations of "abcd" without charAt 3

```
ex C:\WINDOWS\system32\c
D:\demo>java ListPermu
abcd
abdc
acbd
acdb
adbc
adcb
bacd
badc
bcad
bcda
bdac
bdca
cabd
cadb
cbad
cbda
cdab
cdba
dabc
dacb
dbac
dbca
dcab
dcba
```

```
// String without charAt(i):
rest = s.substring(0,i) + s.substring(i+1 , s.length());
```

## Implementation

```
public class ListPermutationsDemo {

    public static void main(String[] args) {
        listPermutations("abcd");
    }

    public static void listPermutations(String s) {
        listPermutations("", s);
    }

    private static void listPermutations(String prefix, String s) {
        if (s.length()==0)
            System.out.println(prefix);
        else
            for (int i=0 ; i<s.length() ; i++) {
                char ch = s.charAt(i);
                String rest = s.substring(0,i) + s.substring(i+1);
                listPermutations(prefix + ch, rest);
            }
    }
}
```

```
ex C:\WINDOWS\system32\c
D:\demo>java ListPermu
abcd
abdc
acbd
acdb
adbc
adcb
bacd
badc
bcad
bcda
bdac
bdca
cabd
cadb
cbad
cbda
cdab
cdba
dabc
dacb
dbac
dbca
dcab
dcba
```

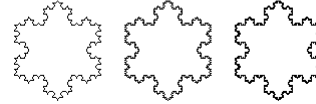
## Fractals

Fractal: A geometric shape that can be split into parts, each of which is (either exactly or approximately) a reduced-size copy of the whole

Some well-known fractal shapes:



Sierpinski triangles



Koch snowflake

Typical generative strategy:



Level 0

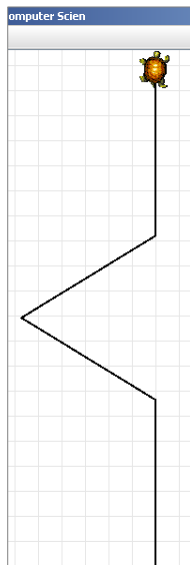


Level 1

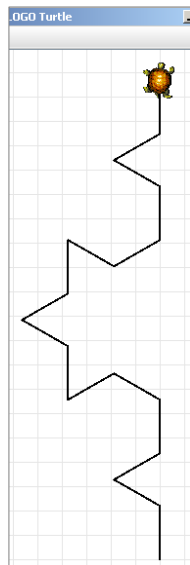


Level 2

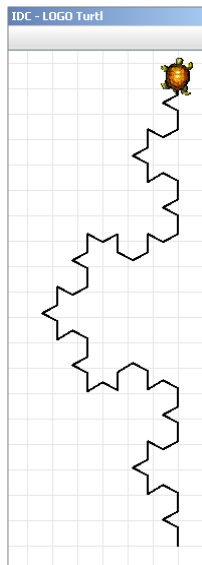
## TurtleFractal



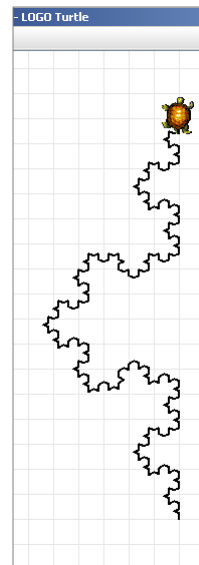
DrawFractal(500, 1)



DrawFractal(500, 2)



DrawFractal(500, 3)

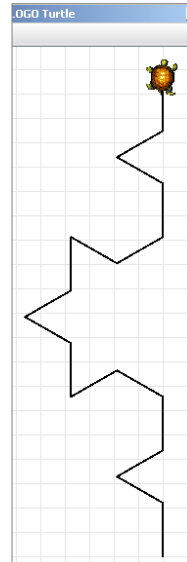


DrawFractal(500, 4)

## Implementation

```
public class TurtleFractal {
    static Turtle turtle = new Turtle();
    public static void main(String[] args) {
        turtle.tailDown();
        drawFractal(500, 2);
    }

    public static void drawFractal(int length, int level) {
        if (level == 0)
            turtle.moveForward(length);
        else {
            drawFractal(length/3, level - 1);
            turtle.turnLeft(60);
            drawFractal(length/3, level - 1);
            turtle.turnRight(120);
            drawFractal(length/3, level - 1);
            turtle.turnLeft(60);
            drawFractal(length/3, level - 1);
        }
    }
}
```



## Fractals

