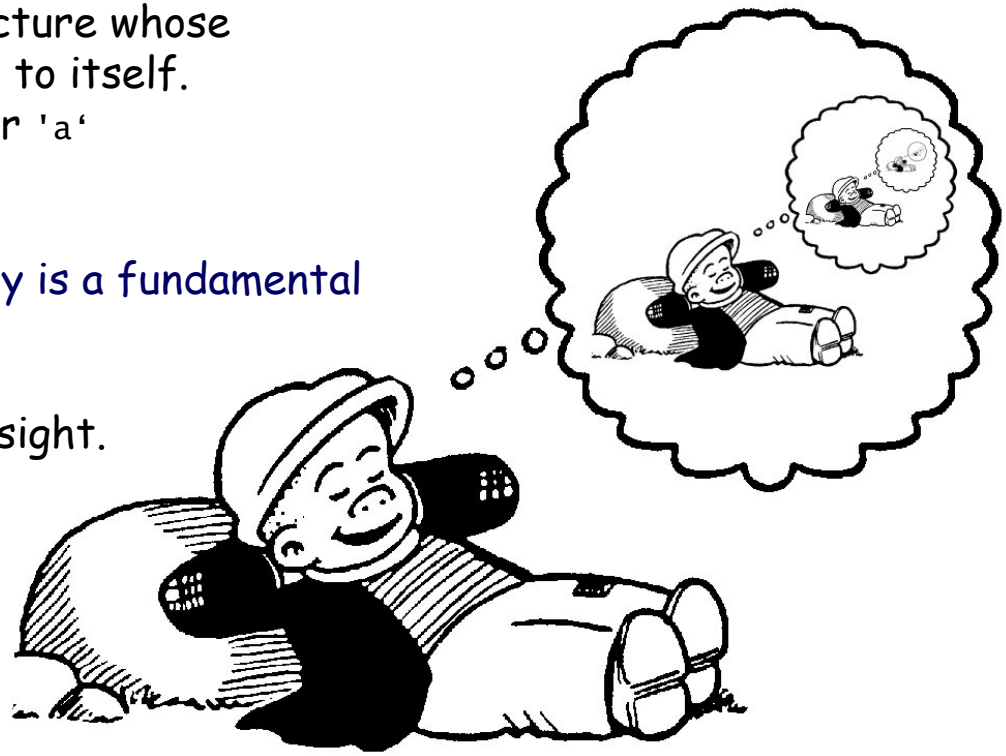
Lecture 12-1

# Recursion

# Recursion

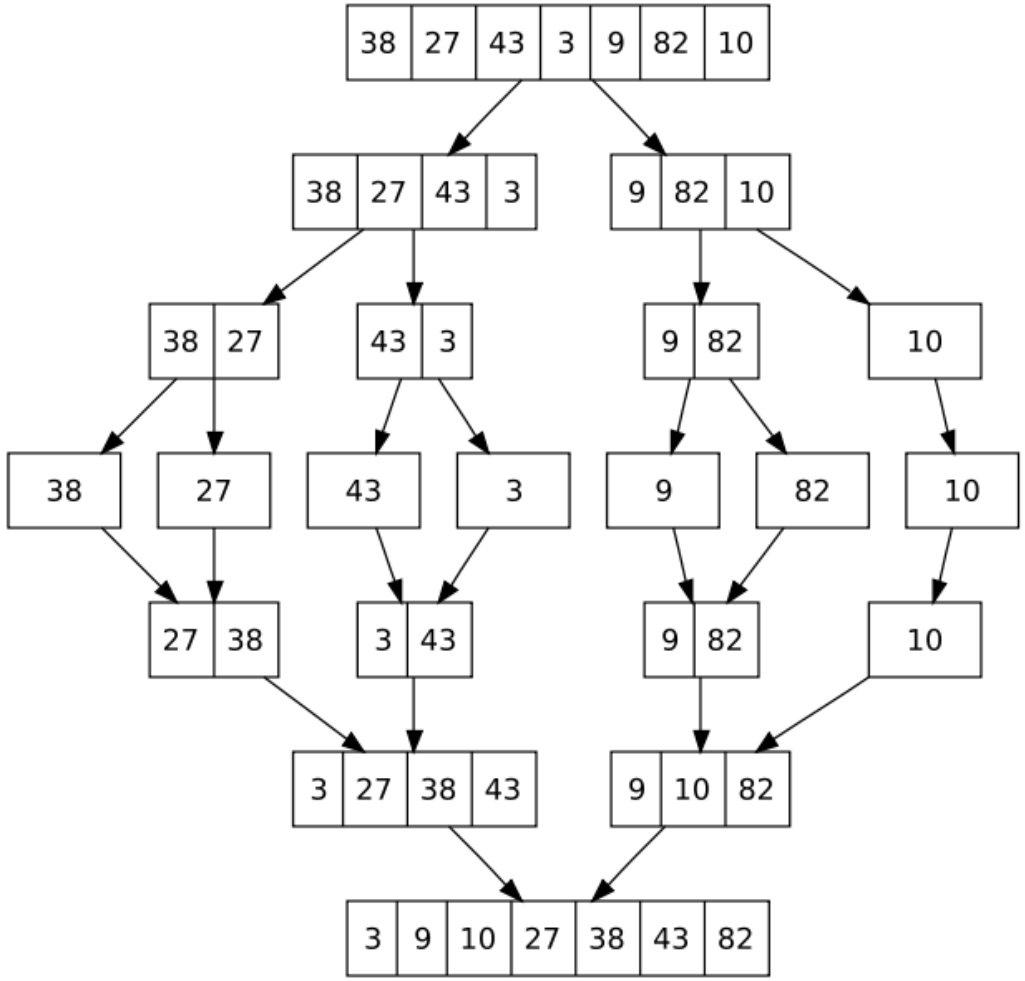
- Recursion: a fundamental algorithmic technique, based on *divide and conquer*
- Recursive function: a mathematical function defined in terms of itself.
  Example: $n! = n*(n-1)!$
- Recursive method: a method that calls itself on smaller subsets of the problem space.
  Example: list all the files in 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 and design recursively is a fundamental programming skill
- But, mastering it takes practice and insight.



# 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

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

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

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

```
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);
}
```

# The elements of the recursive approach

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

**Base case**

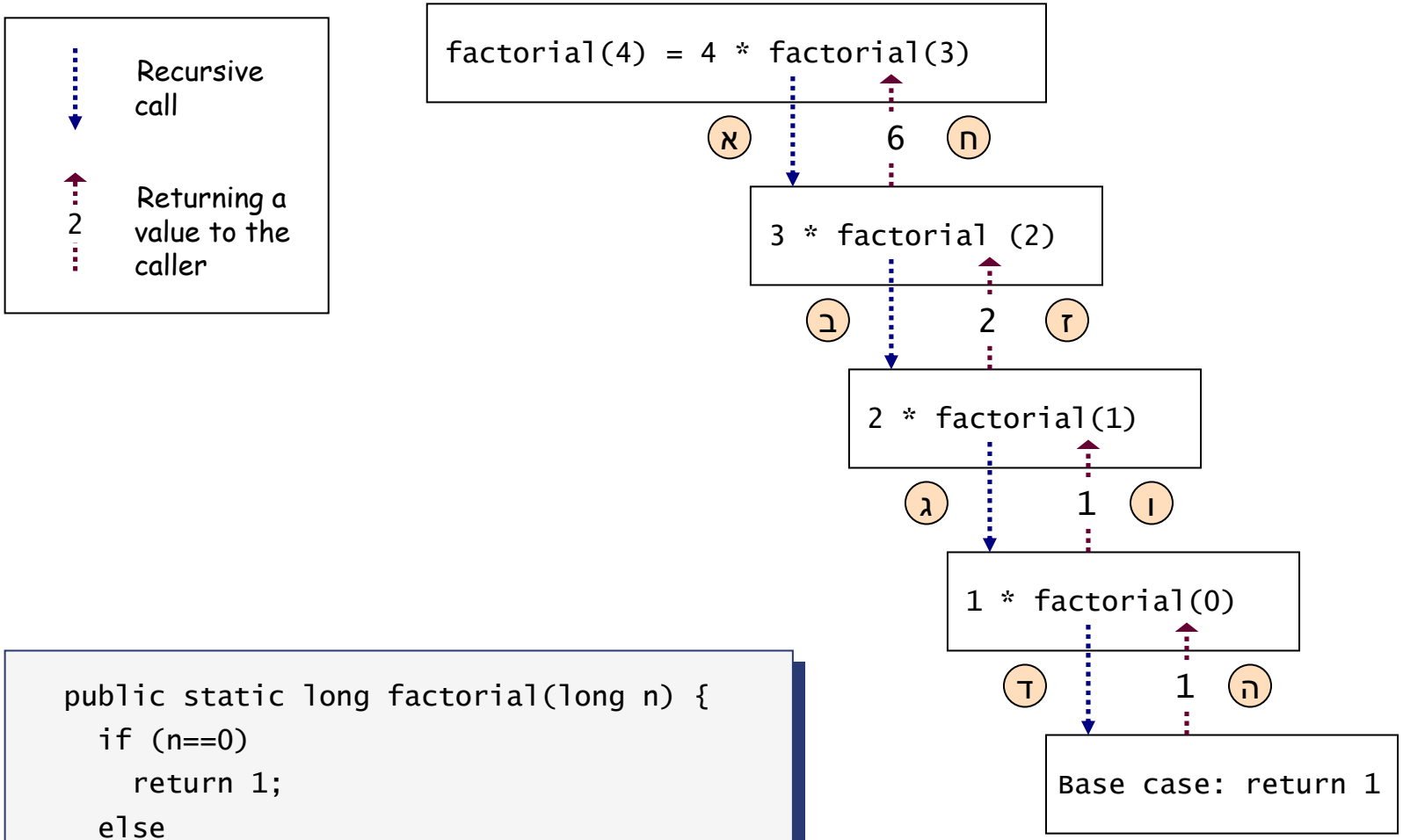
**Assembly**

**Reduction**

A one-liner …

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

# Run-time anatomy




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

# Sum

The problem stated

sum (a , b) = ?

sum (a , o) = a

sum (a , b) = sum(a , (b - 1)) + 1 for b > 0

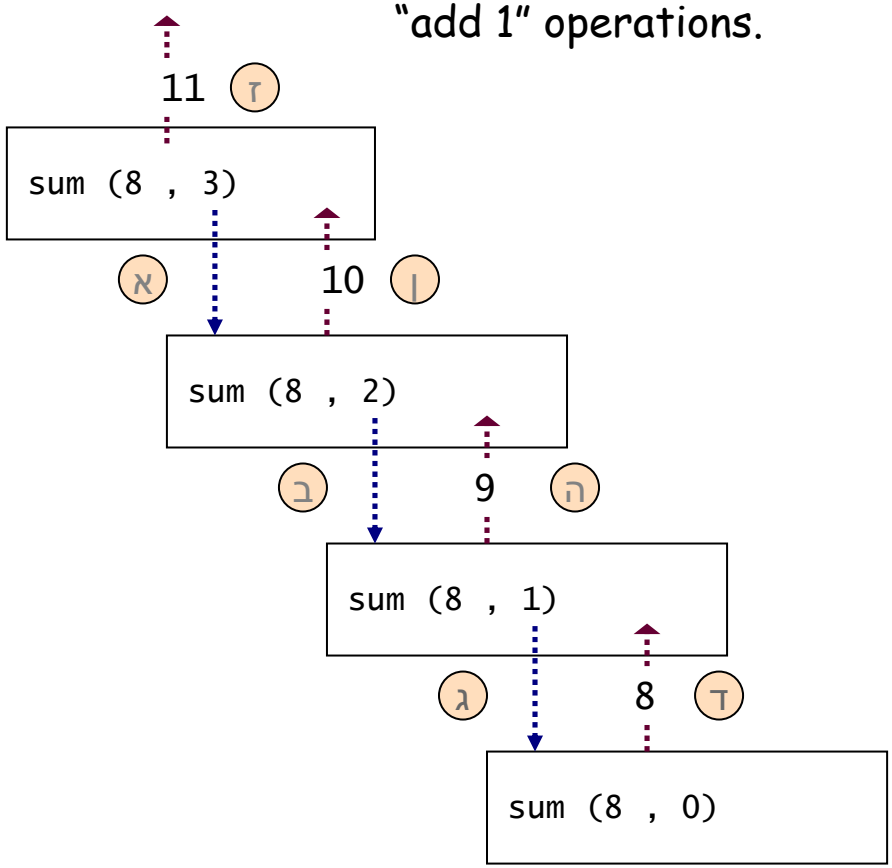
```
sum (a,b) = sum(a,b--)++        // in pseudo code
```

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

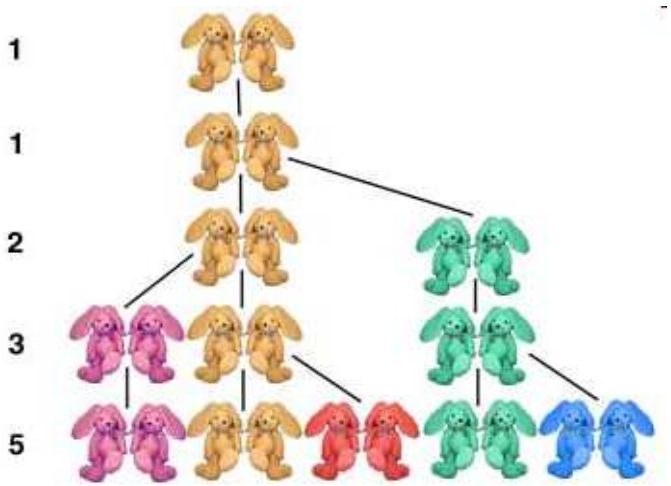
Fibonacci series definition

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

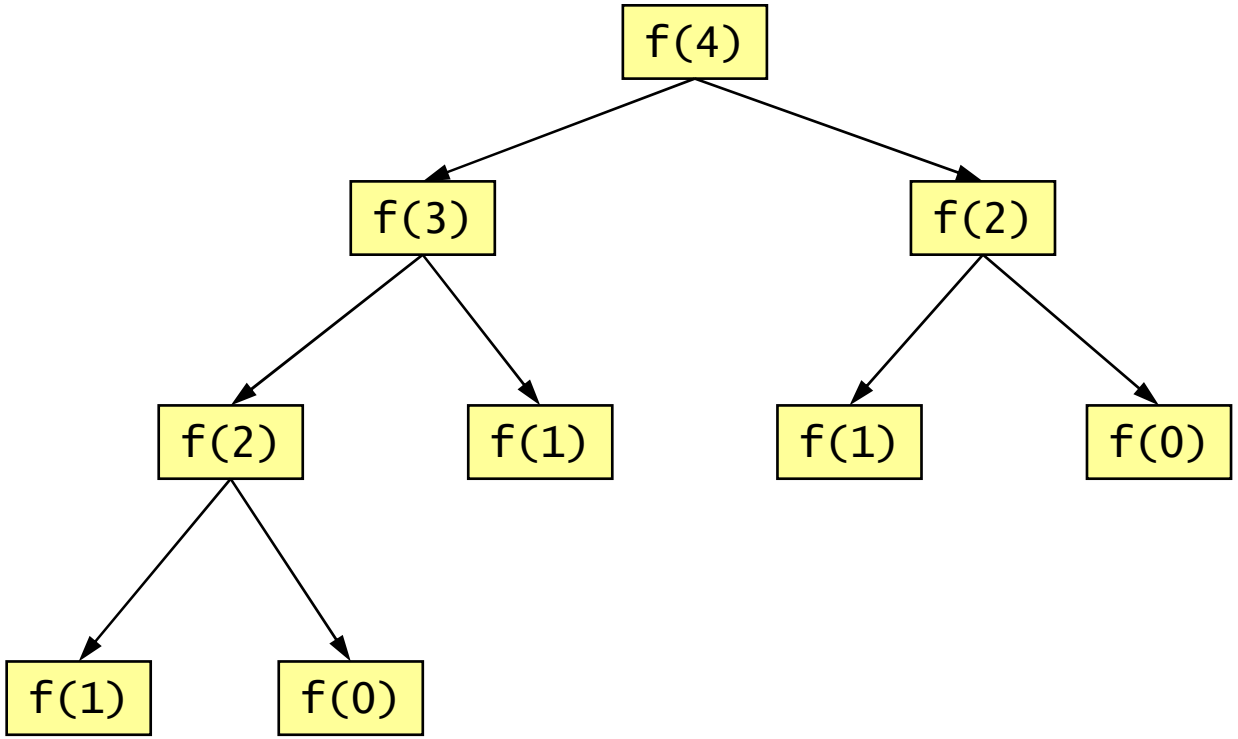
```
public class FibonacciDemo {
  public static void main(String args[]) {
      System.out.println(fibonacci(5));
  }

  // Returns the n'th fibonacci number
  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: Run-time anatomy




Built-in redundancy: the algorithm repeats the same computations (e.g. f(2))

Running time of this algorithm: $O(2^n)$

<u>Q:</u> Can we do better?

<u>A:</u> A bottom-up, iterative solution will run in O(n)

<u>Conclusion:</u> Naïve recursive solutions can be expensive!

# Power

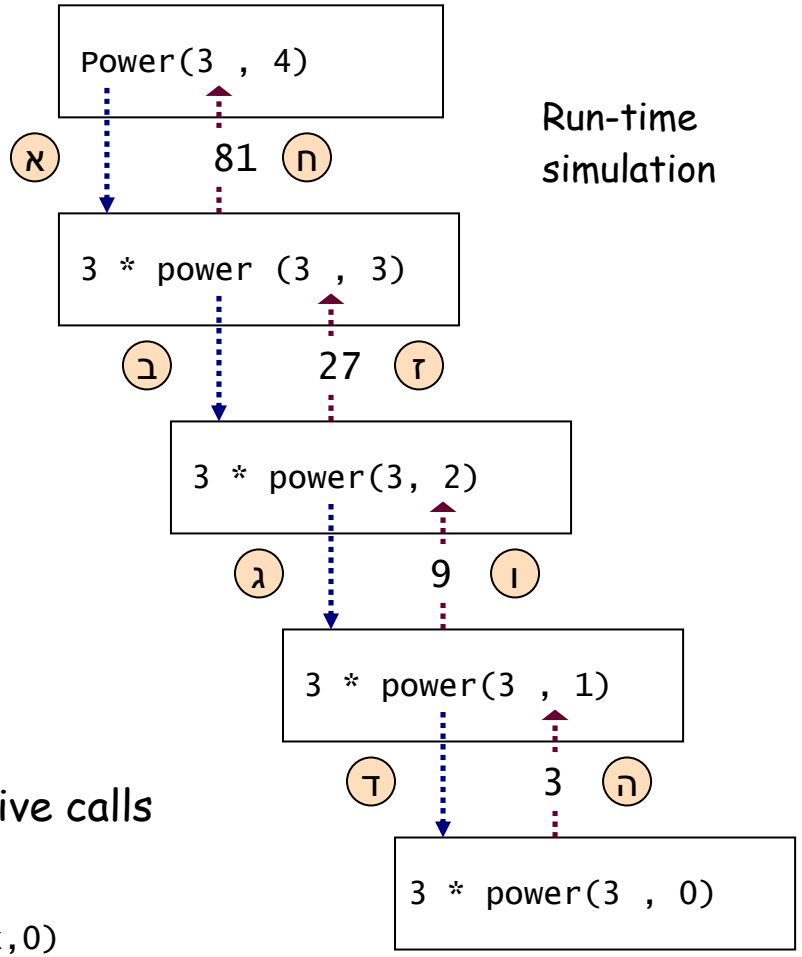
Recursive definition:

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)
}
```




Run-time simulation

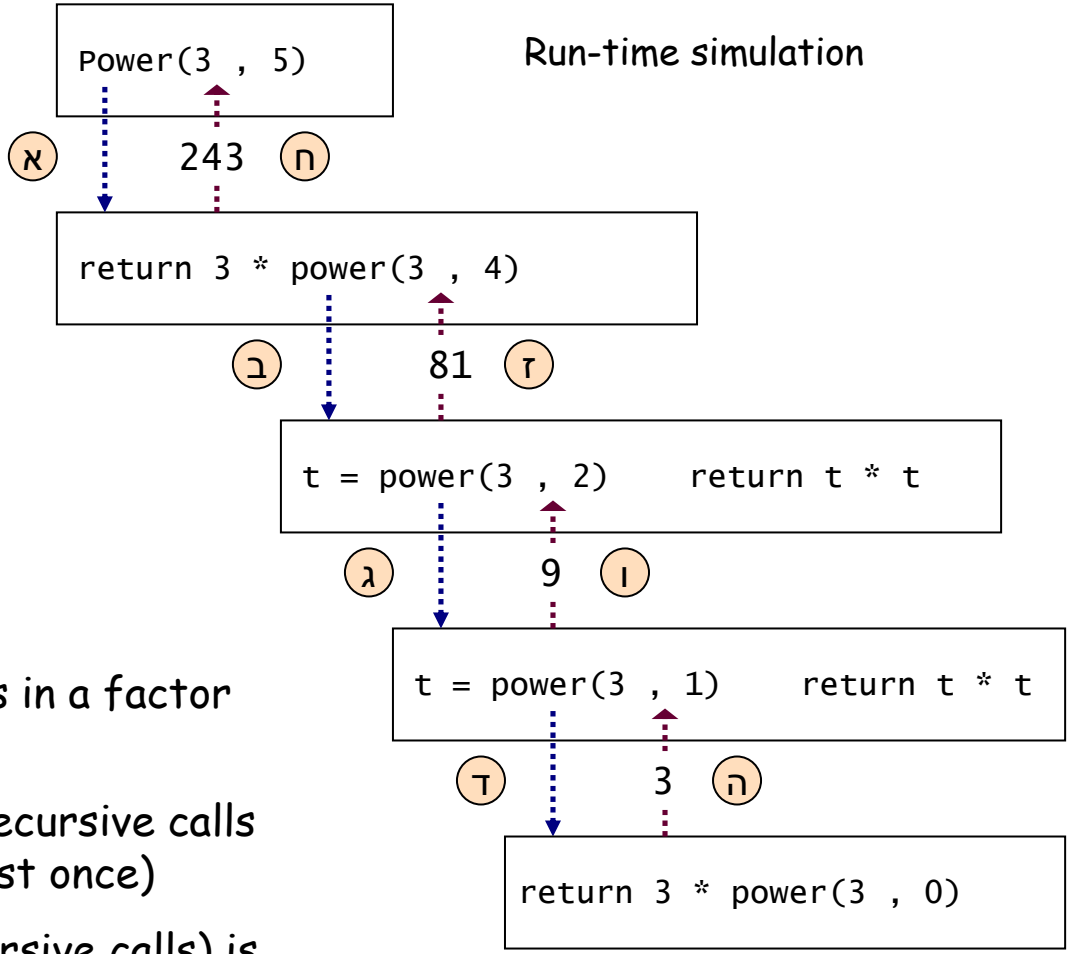
Running time of recursive programs:

- Simply add up the running time of all the recursive calls
- `power(x,n)` requires n+1 recursive calls:
  `power(x,n), power(x,n-1), … , power(x,1), power(x,0)`
- Total running time = (n+1) $O(1)$ = $O(n)$

# Power

```
// 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).

# Power

```
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);
}
```

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

<u>Proof:</u> 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 power(x,n/2) * power(x,n/2)

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

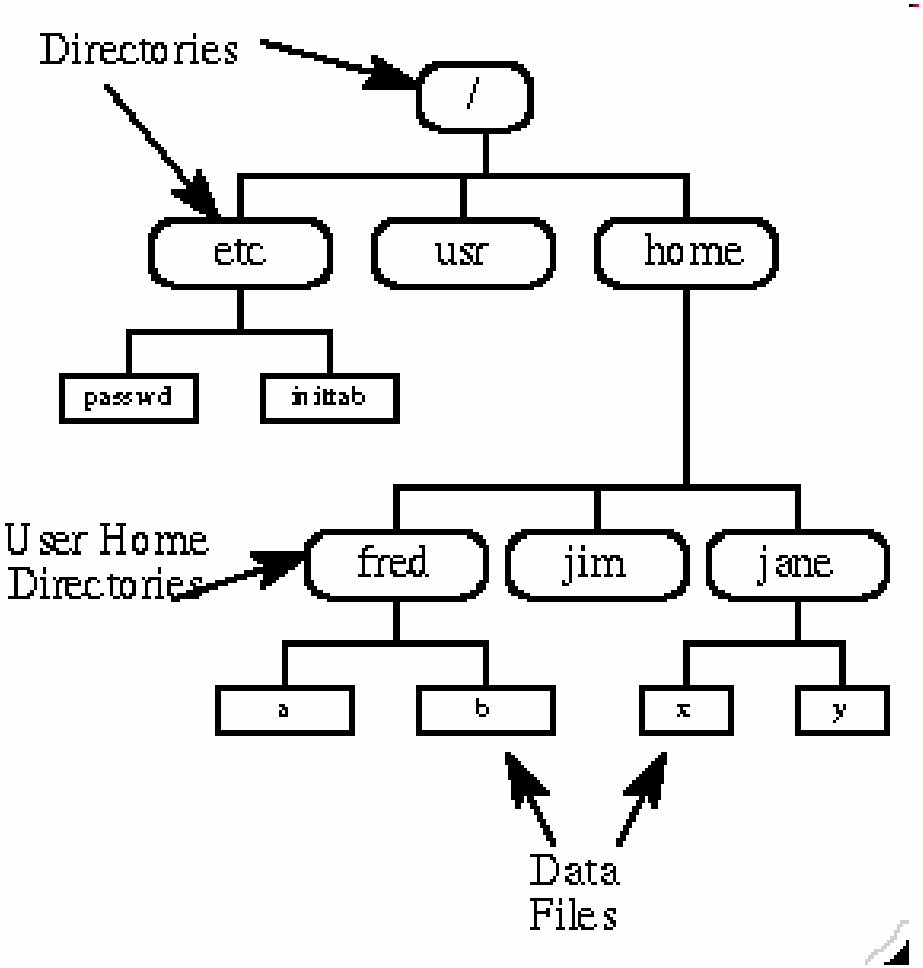
If n is odd, the algorithm returns x * power(x,n-1)

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

# 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 recursively
- Example: files listing utility.

# File listing

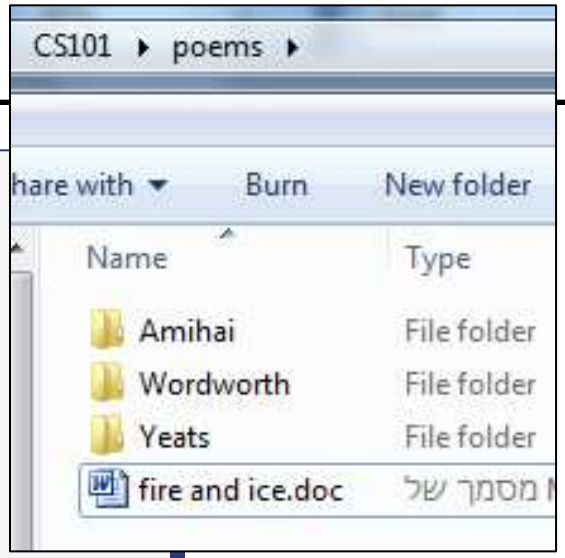


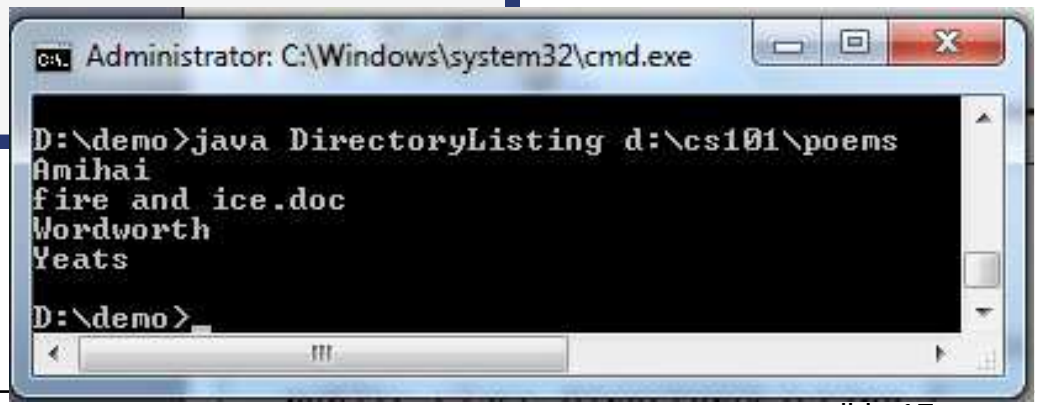
# File listing

```
import java.io.File;

public class DirectoryListing {
  public static void main(String[] args){
    listFiles(new File(args[0]));
  }

  private static void listFiles (File dirName) {
    String[] fileNames = dirName.list();
    if (fileNames == null) {
      System.out.println("Specified directory does not exist.");
      System.exit(0);
    }
    else
      for (String fileName : fileNames)
        System.out.println(fileName);
  }
}
```



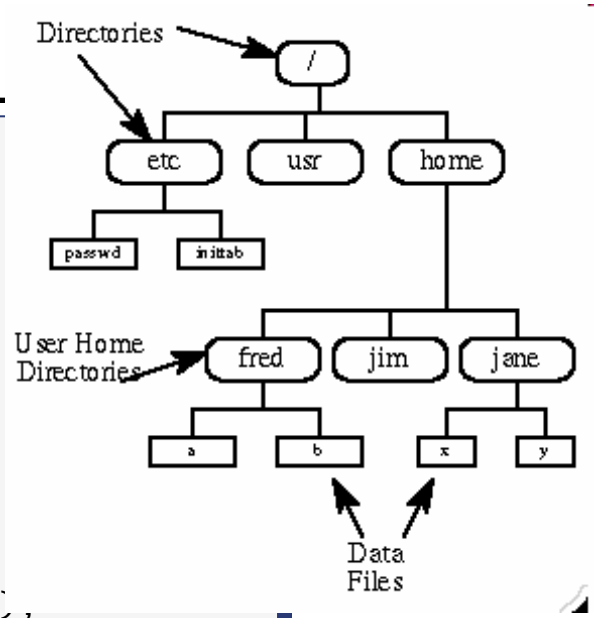



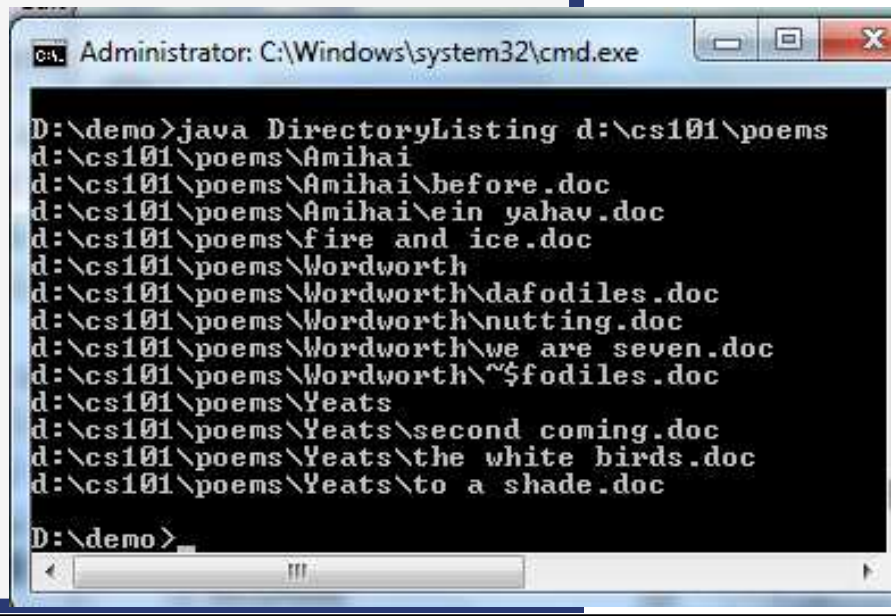

(Read the `java.util.File` API)

# File listing, recursive

```
import java.io.File;
public class DirectoryListing {
  public static void main(String[] args){
    listFiles(new File(args[0]));
  }

  private static void listFiles (File dirName) {
    String[] fileNames = dirName.list();
    if (fileNames == null) {
      System.out.println("Specified directory does not exist.");
      System.exit(0);
    }
    else
      for (String fileName : fileNames) {
        File f = new File(dirName, fileName);
        if (f.isDirectory()) {
          System.out.println(f);
          listFiles(f);
        }
        else
          System.out.println(f);
      }
  }
}
```



# Reverse

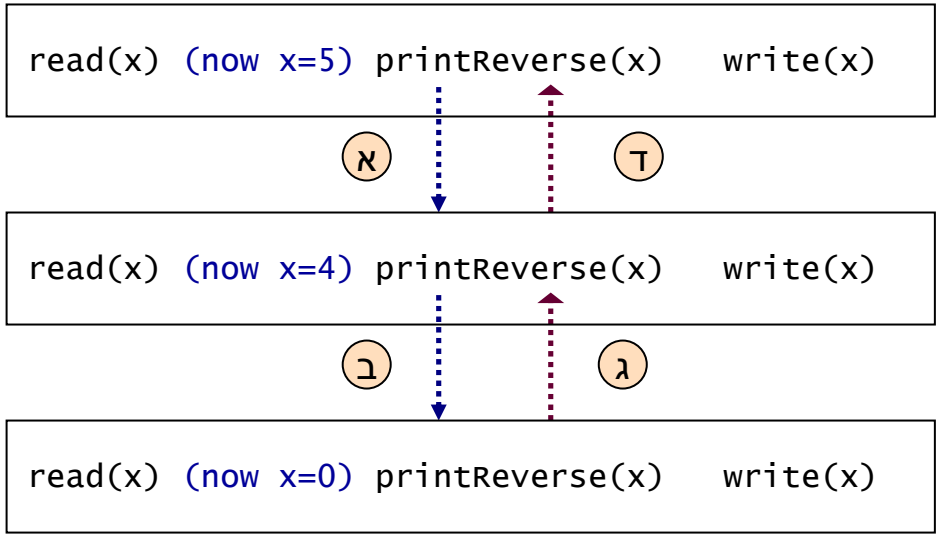
```
printReverse () {
    print("Enter a number, or 0 to end: ")
    read(x)
    if (x != 0) {
        printReverse();
        write(x);
    }
}
```

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
```

Run-time simulation (user inputs: 5, 4, 0)




- When method f calls method g, f's variables are frozen
- When g returns, f's variables are re-instantiated and f's execution resumes
- As far as f is concerned, everything is back to normal
- Java manages this process behind the scene, using a *stack* to save the variables of called methods.