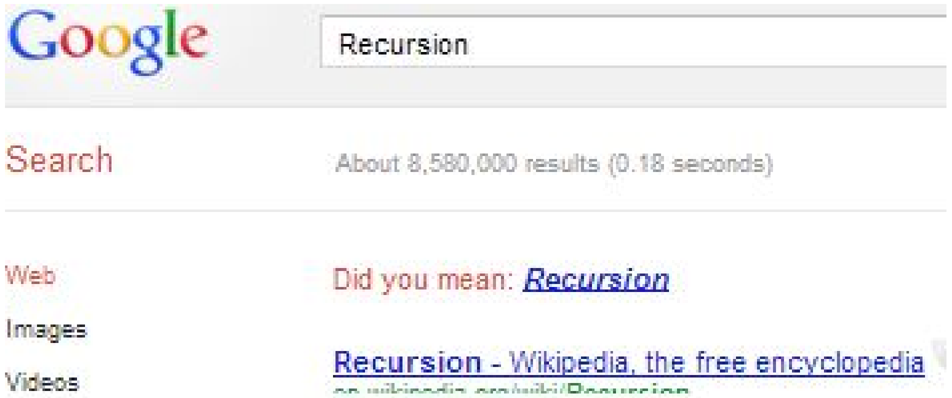
# Algorithms

## 3. Recursion

Frank Walsh, Robert O’Connor



Recursion

Search

About 8,580,000 results (0.18 seconds)

Web

Images

Videos

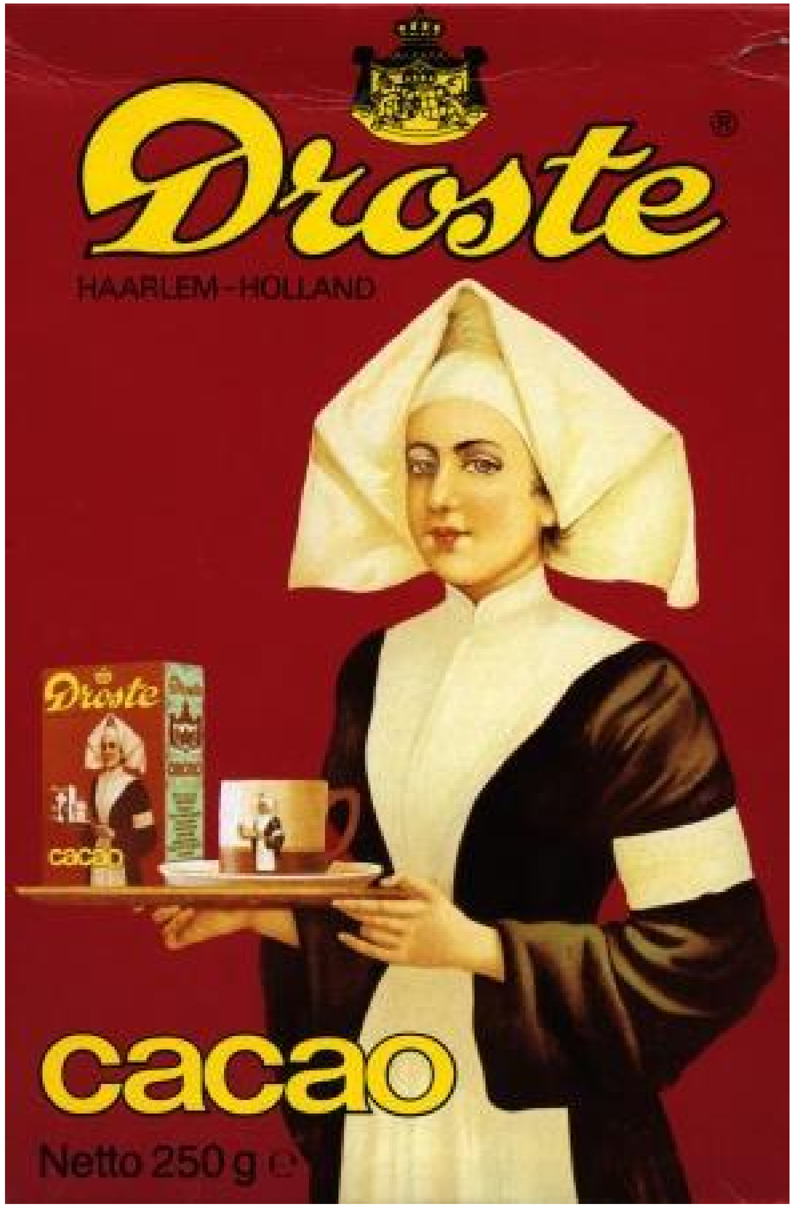
Did you mean: *Recursion*

Recursion - Wikipedia, the free encyclopedia

*Recursion, see Recursion.*

*PHP: "PHP Hypertext Preprocessor"*

*GNU: "GNU's not Unix".*



# Objectives

- What Is Recursion?
- Tracing a Recursive Method
- Recursive Methods That Return a Value
- Recursively Processing an Array
- The Time Efficiency of Recursive Methods
    - Time Efficiency of **countDown**
    - Time Efficiency of computing $x^n$
- A Simple Solution to a Difficult Problem
- A Poor Solution to a Simple Problem

## What is Recursion?

- It is a problem-solving process
- Breaks a problem into identical but smaller problems
- Eventually you reach a smallest problem
  - Answer is obvious or trivial
- Using that solution enables you to solve the previous problems
- Eventually the original problem is solved

# What is Recursion?

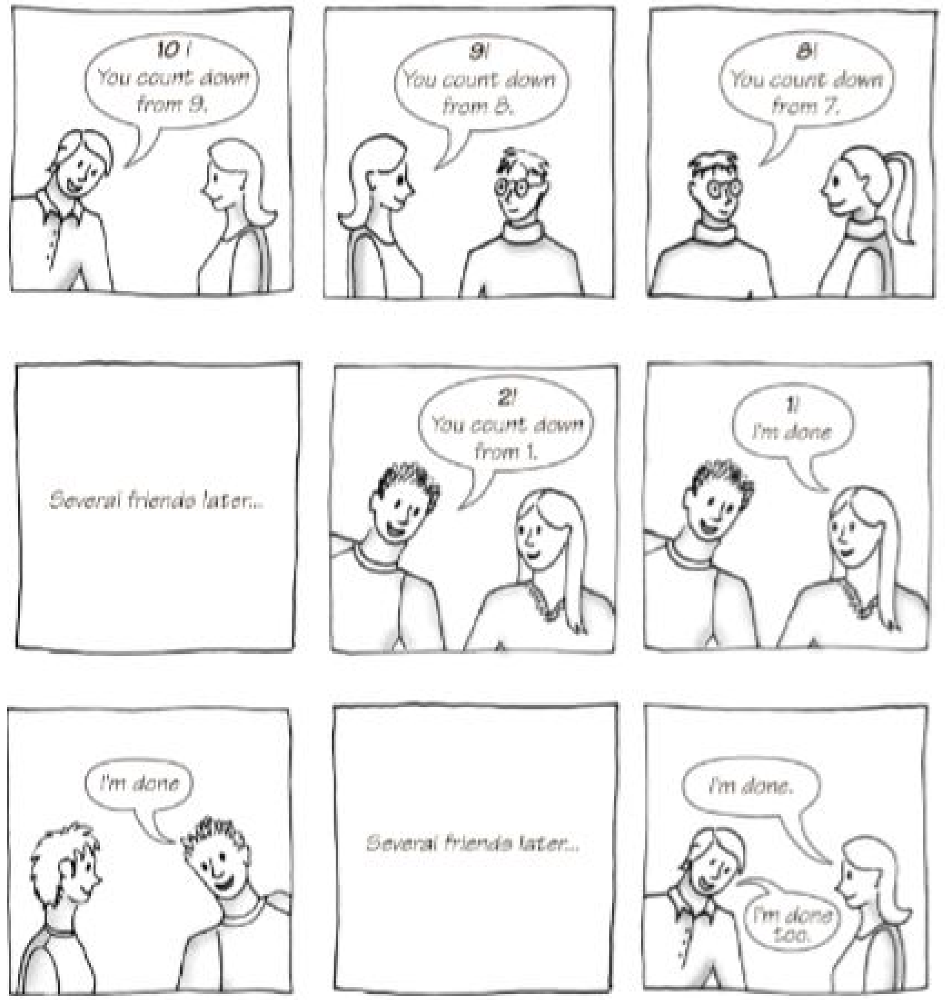

10! You count down from 9.
9! You count down from 8.
8! You count down from 7.
Several friends later...
2! You count down from 1.
1! I'm done
I'm done
Several friends later...
I'm done.
I'm done too.

Fig. 4-1 Counting down from 10.

# What is Recursion?

- A method that calls itself is a recursive method

```
/** Task: Counts down from a given positive
integer.
* @param integer an integer > 0 */
public static void countDown(int integer)
{ System.out.println(integer);
if (integer > 1)
countDown(integer - 1);
} // end countDown
```

- The invocation is a
  - Recursive call
  - Recursive invocation

# When Designing Recursive Solution

- What part of the solution can you contribute directly?
- What smaller but identical problem has a solution that …
  - When taken with your contribution provides solution to the original problem
- When does the process end?
  - What smaller but identical problem has known solution
  - Have you reached the base case

# When Designing Recursive Solution

- Method definition must provide parameter
  - Leads to different cases
  - Typically includes an **if** or a **switch** statement
- One or more of these cases should provide a non recursive solution
  - The base or stopping case
- One or more cases includes recursive invocation
  - Takes a step towards the base case

## Tracing a Recursive Method

- Given:

```
public static void countDown(int integer)
{ System.out.println(integer);
if (integer > 1)
    countDown(integer - 1);
} // end countDown
```

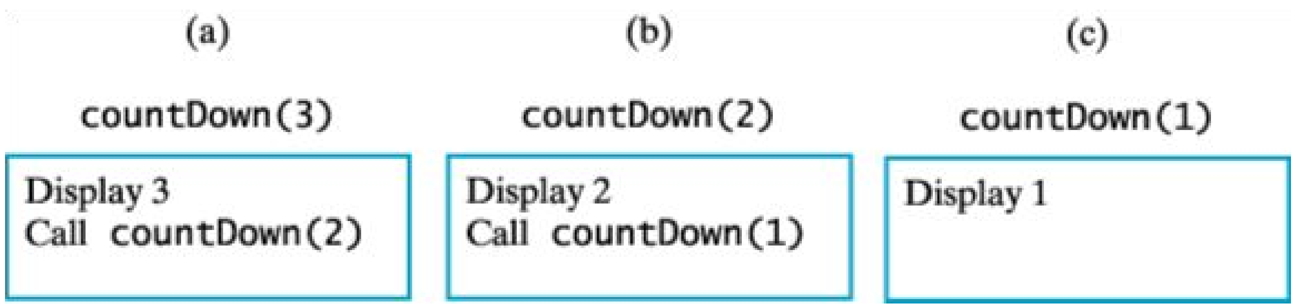
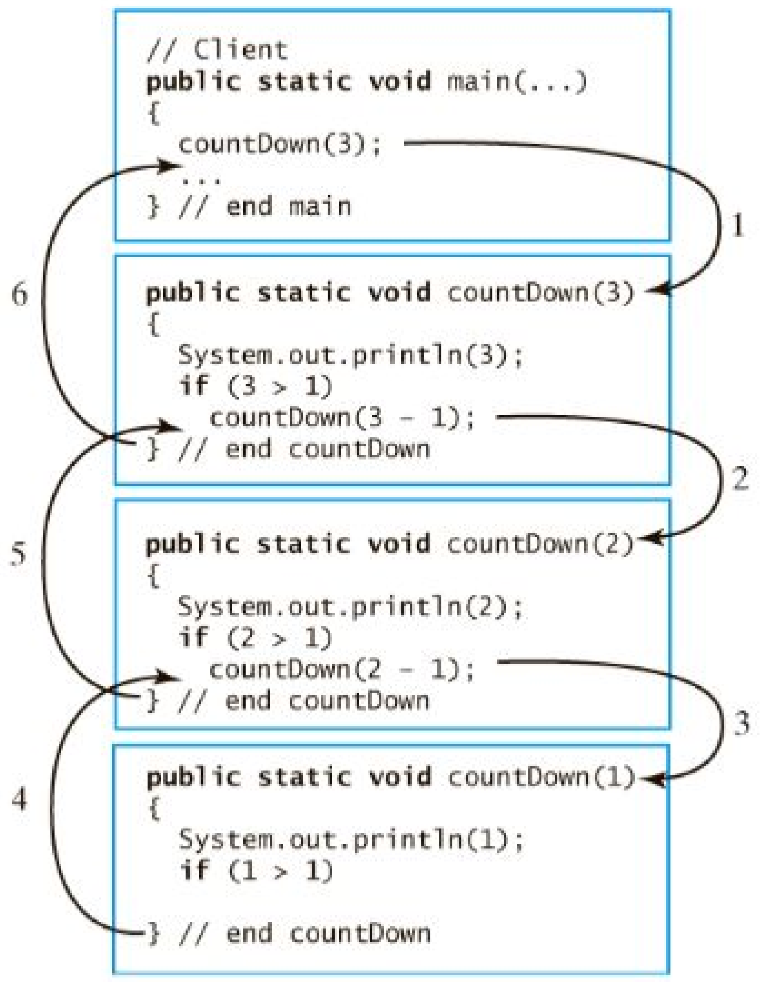


Fig.4-2 The effect of method call **countDown(3)**

# Tracing a Recursive Method



```
// Client
public static void main(...)
{
  countDown(3);
  ...
} // end main
```

```
public static void countDown(3)
{
  System.out.println(3);
  if (3 > 1)
    countDown(3 - 1);
} // end countDown
```

```
public static void countDown(2)
{
  System.out.println(2);
  if (2 > 1)
    countDown(2 - 1);
} // end countDown
```

```
public static void countDown(1)
{
  System.out.println(1);
  if (1 > 1)

} // end countDown
```

Fig. 4-3 Tracing the recursive call **countDown(3)**

# Tracing a Recursive Method

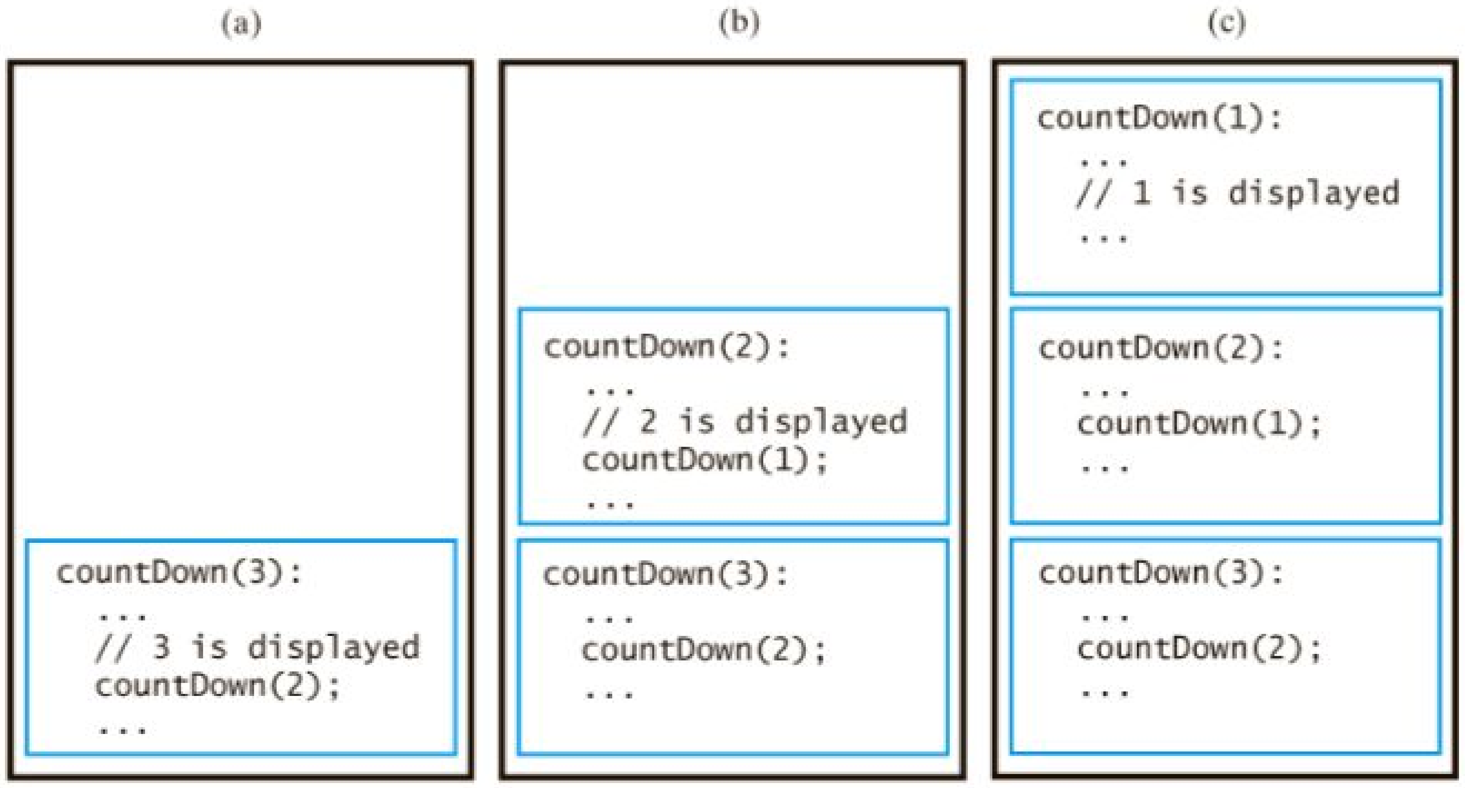
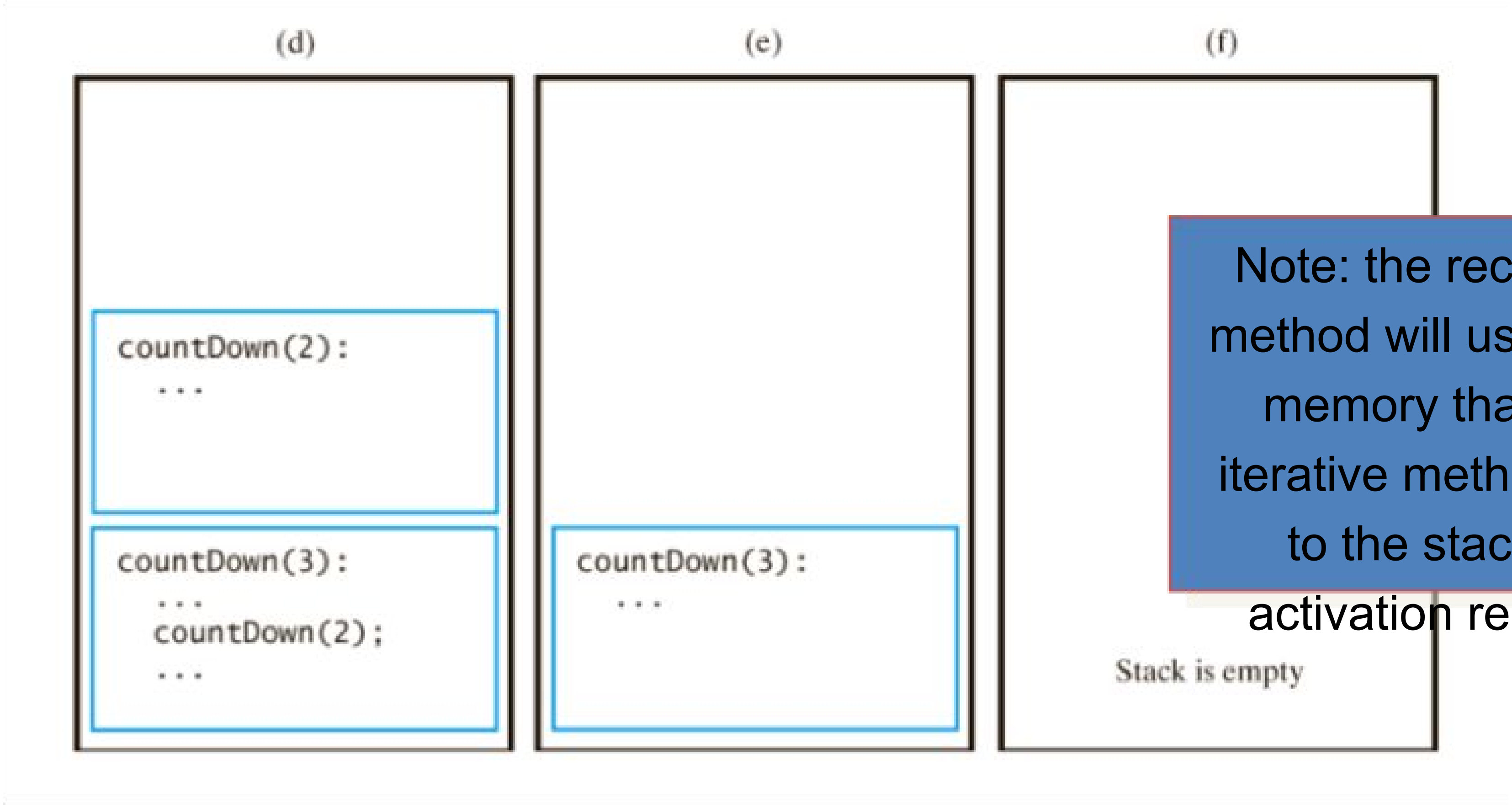


Fig. 4-4 The stack of activation records during the execution of a call to **countDown(3)…** continued →

# Tracing a Recursive Method



Note: the recursive method will use more memory than an iterative method due to the stack of activation records

Fig. 4-4 ctd. The stack of activation records during the execution of a call to **countDown(3)**

## Recursive Methods that Return a Value

- Task: Compute the sum

  1 + 2 + 3 + … + n for an integer n > 0

```
public static int sumOf(int n)
{ int sum;
if (n = = 1)
    sum = 1; // base case
else
    sum = sumOf(n - 1) + n; // recursive call
return sum;
} // end sumOf
```

# Recursive Methods that Return a Value

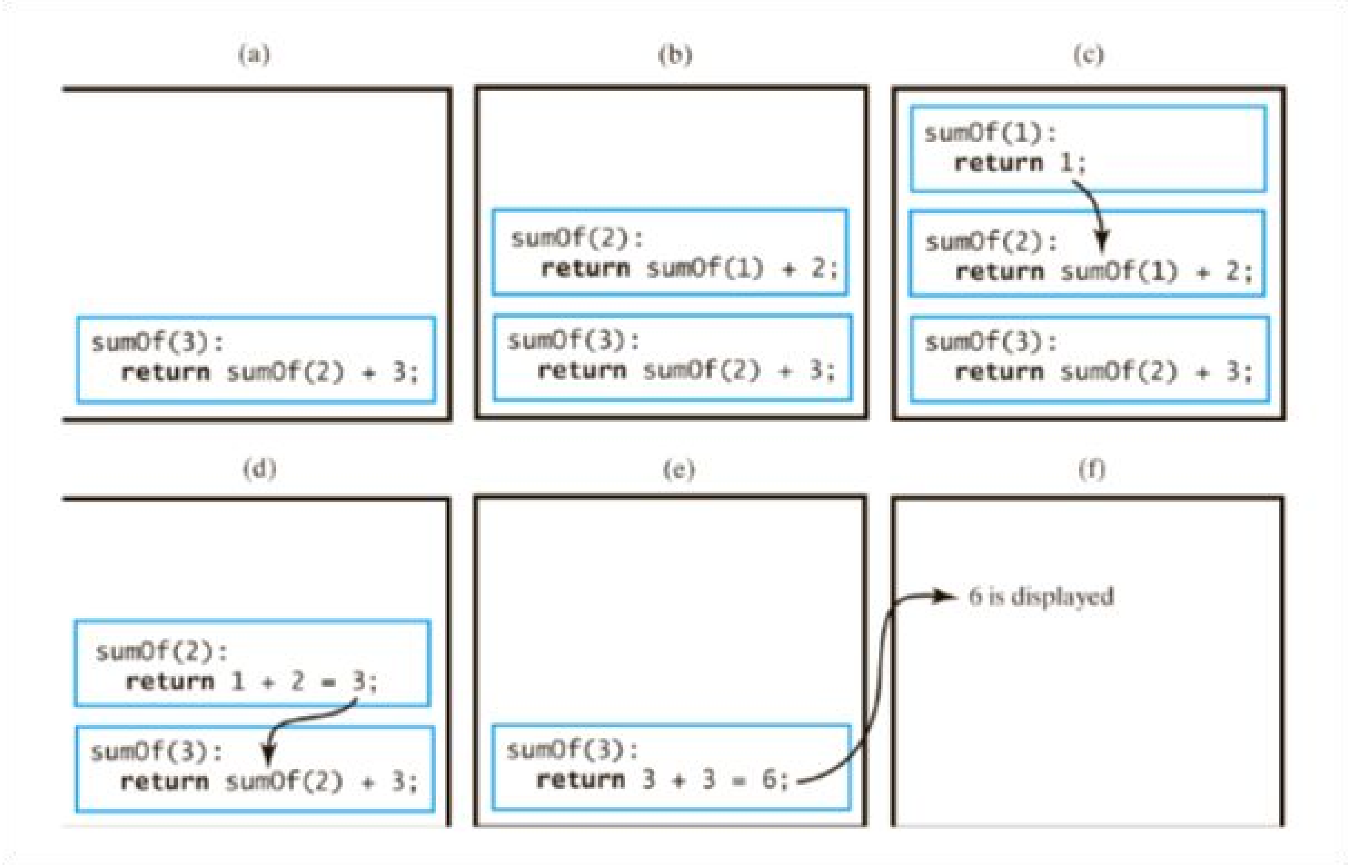


Fig. 4-5 The stack of activation records during the execution of a call to **sumOf(3)**

## Recursively Processing an Array

- When processing array recursively, divide it into two pieces
  - Last element one piece, rest of array another
  - First element one piece, rest of array another
  - Divide array into two halves

# Recursively Processing an Array

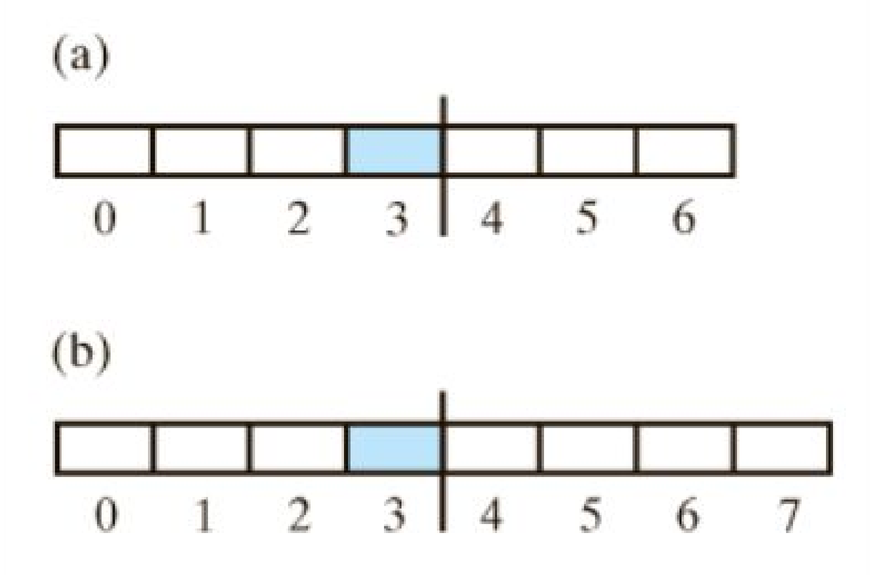


Fig. 4-6 Two arrays with middle elements within left halves

## Recursively Processing an Array

- A method that processes an array recursively

```
public static void displayArray(int array[], int first, int last){
  if(first == last){
      System.out.println(array[first] + " ");
  }else{
       int mid = (first + last)/2;
       displayArray(array, first, mid);
       displayArray(array, mid+1, last);
   }
} // end displayArray
```

## Time Efficiency of Recursive Methods

- For the **countDown** method

```
public static void countDown(int integer)
{ System.out.println(integer);
if (integer > 1)
countDown(integer - 1);
} // end countDown
```

  - The efficiency is O(n)

## A Simple Solution to a Difficult Problem

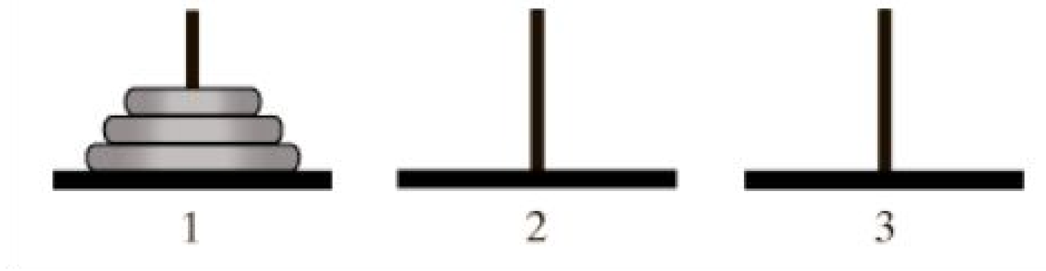


Fig. 4-7 The initial configuration of the Towers of Hanoi for three disks

## A Simple Solution to a Difficult Problem

Rules for the Towers of Hanoi game

1. Move one disk at a time. Each disk you move must be a topmost disk.
2. No disk may rest on top of a disk smaller than itself.
3. You can store disks on the second pole temporarily, as long as you observe the previous two rules.

## A Simple Solution to a Difficult Problem

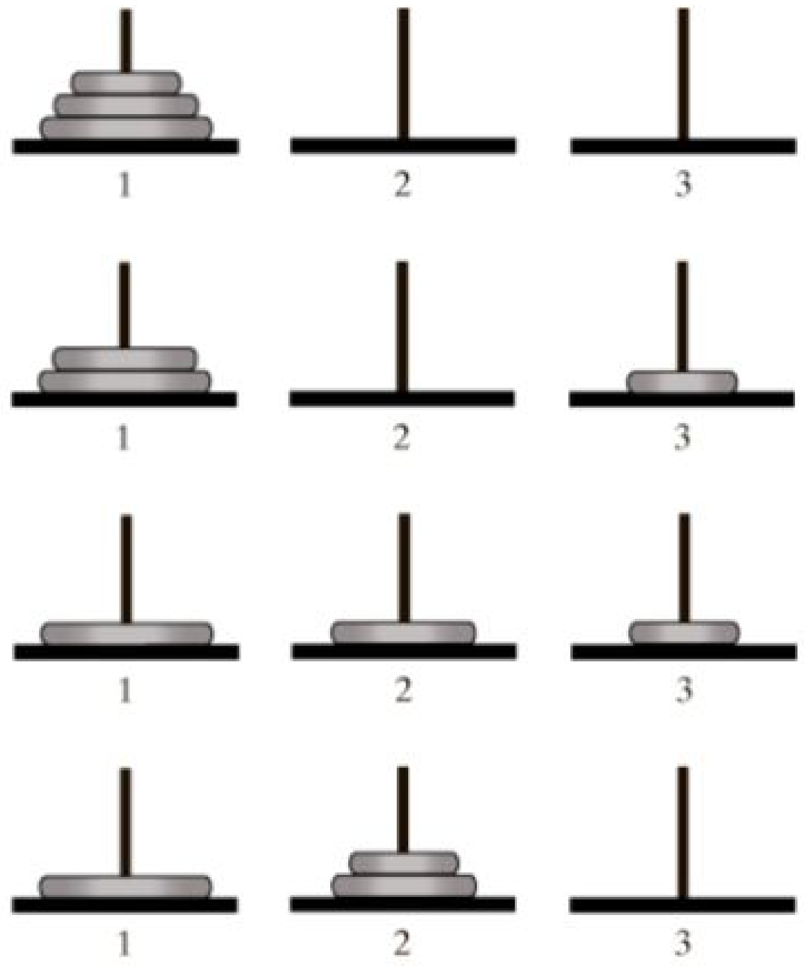


Fig. 4-8 The sequence of moves for solving the Towers of Hanoi problem with three disks.

Continued →

# A Simple Solution to a Difficult Problem

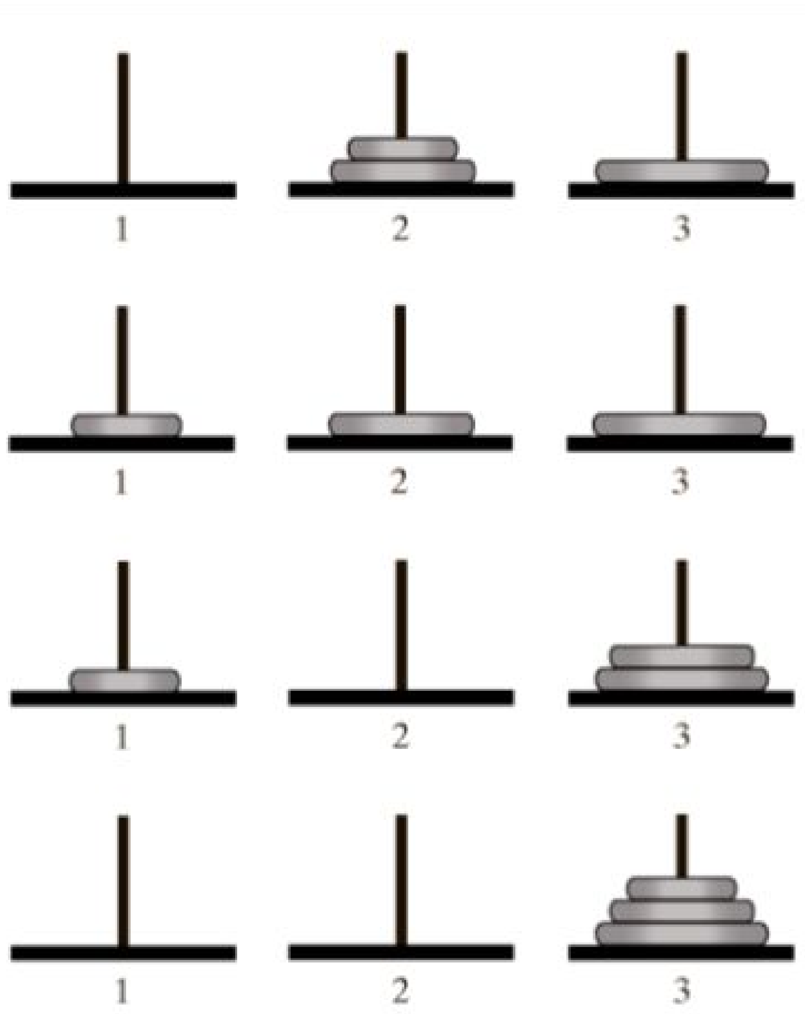


Fig. 4-8 (ctd) The sequence of moves for solving the Towers of Hanoi problem with three disks

## A Simple Solution to a Difficult Problem

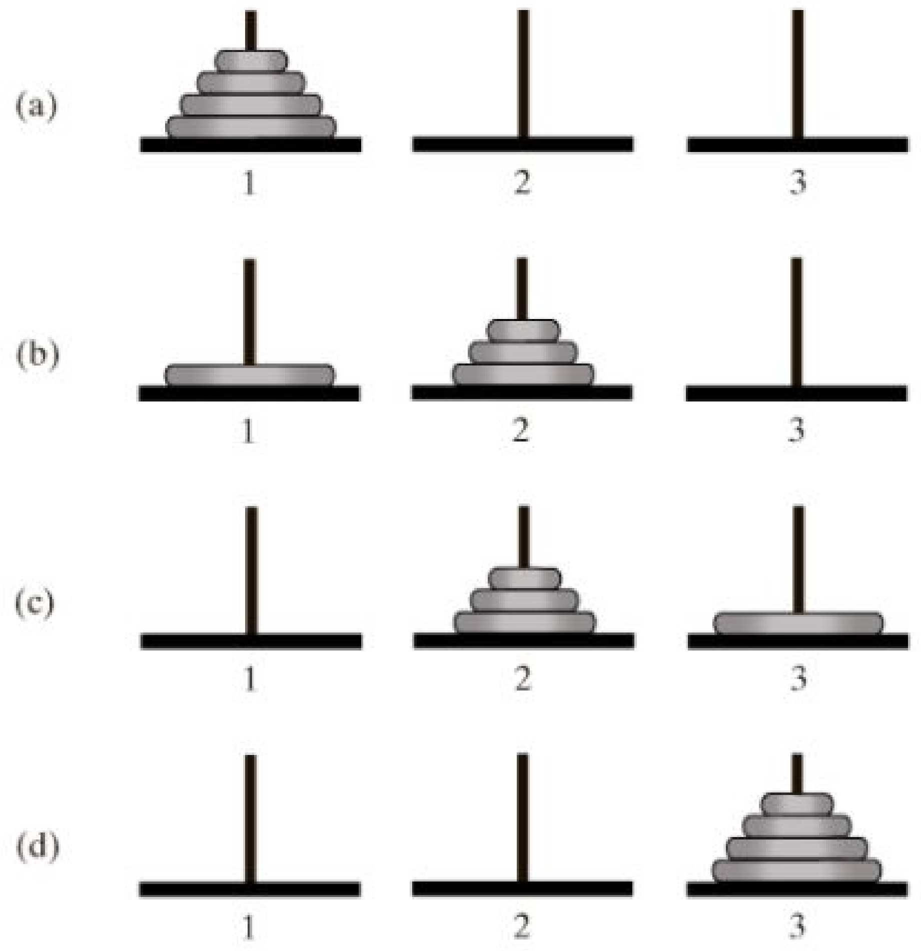


Fig. 4-9 The smaller problems in a recursive solution for four disks

## A Simple Solution to a Difficult Problem

- Algorithm for solution with 1 disk as the base case

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
Move disk from startPole to endPole
else
{
solveTowers(numberOfDisks-1, startPole, endPole, tempPole)
Move disk from startPole to endPole
solveTowers(numberOfDisks-1, tempPole, startPole, endPole)
}
```

## A Simple Solution to a Difficult Problem

- Algorithm for solution with 0 disks as the base case

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks > 0)
{
    solveTowers(numberOfDisks-1, startPole, endPole, tempPole)
    Move disk from startPole to endPole
    solveTowers(numberOfDisks-1, tempPole, startPole, endPole)
}
```

## A Poor Solution to a Simple Problem

- Fibonacci numbers
  - First two numbers of sequence are 1 and 1
  - Successive numbers are the sum of the previous two
  - 1, 1, 2, 3, 5, 8, 13, …
- This has a natural looking recursive solution
  - Turns out to be a poor (inefficient) solution

## A Poor Solution to a Simple Problem

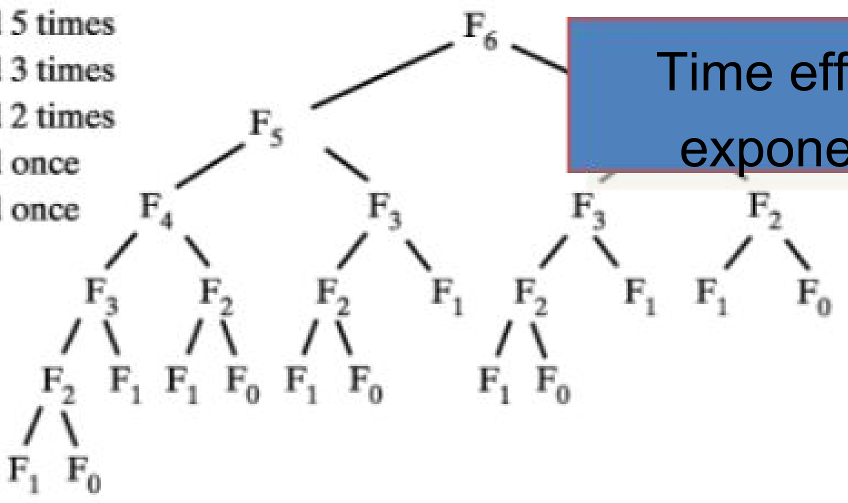
- The recursive algorithm

```
Algorithm Fibonacci(n)
if (n <= 1)
return 1
else
return Fibonacci(n-1) + Fibonacci(n-2)
```

# A Poor Solution to a Simple Problem

(a)

$F_2$ is computed 5 times
$F_3$ is computed 3 times
$F_4$ is computed 2 times
$F_5$ is computed once
$F_6$ is computed once


$F_6$
$F_5$
$F_4$ $F_3$ $F_3$ $F_2$
$F_3$ $F_2$ $F_2$ $F_1$ $F_2$ $F_1$ $F_1$ $F_0$
$F_2$ $F_1$ $F_1$ $F_0$ $F_1$ $F_0$ $F_1$ $F_0$
$F_1$ $F_0$

Time efficiency grows exponentially with n

(b)

$F_0 = 1$
$F_1 = 1$
$F_2 = F_1 + F_0 = 2$
$F_3 = F_2 + F_1 = 3$
$F_4 = F_3 + F_2 = 5$
$F_5 = F_4 + F_3 = 8$
$F_6 = F_5 + F_4 = 13$

Iterative solution is O(n)

Fig. 4-10 The computation of the Fibonacci number $F_6$
(a) recursively; (b) iteratively

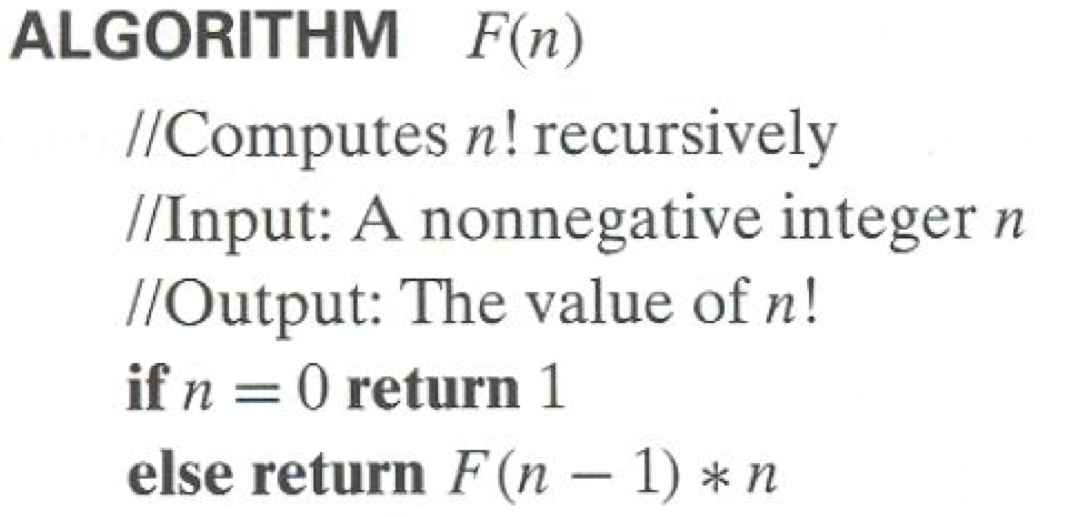
## Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input’s size.
- Identify the algorithm’s basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution’s order of growth) by backward substitutions or another method.

# Example 1: Recursive evaluation of *n*!

Definition: *n* ! = 1 x 2 x … x(*n*-1) x *n* for *n* ≥ 1 and 0! = 1

Recursive definition of *n*!: *F*(*n*) = *F*(*n*-1) x *n* for *n* ≥ 1 and *F*(0) = 1



```
ALGORITHM  $F(n)$
    //Computes $n!$ recursively
    //Input: A nonnegative integer $n$
    //Output: The value of $n!$
    if $n = 0$ return 1
    else return $F(n-1) * n$
```

Size: n

Basic operation: Multiplication

Recurrence relation: F(n) = F(n-1)n ; F(0) = 1 (values of n!)

M(n) =M(n-1) + 1 M(0) = 0(Number of Multiplications)

# Example 1 - Recursive Evaluation of n!

**1. Input size**: n

**2. Basic Operation**: Multiplication
**# of Operations different for different inputs of same size**:No

**3. Recurrence relation for # times basic operation executed**:
M(n)=M(n-1) + 1;    M(0)=0;

4. **Solving using backward substitution**:
= M(n-2) +2 = M(n-3) + 3...= M(n-i) + i
= M(0) + n = n

5. Algorithm is O(M(n-1) + 1) = O(n)

## Example 2 - Towers of Hanoi

1. **Input size**: n disks (n)

2. **Basic Operation**: Moving a disk
**# of Operations different for different inputs of same size**:No

3. **Recurrence relation for # times basic operation executed**:
$M(n)=M(n-1) + 1 + M(n-1); \quad M(1)=1;$

4. **Solving using backward substitution**:
$M(n)= 2M(n-1) + 1= 2[2M(n-2)+1) + 1=2^2M(n-2) + 2 + 1$
$= 2^2[2M(n-3)+1) + 2 + 1 = 2^3M(n-3) +2^2 + 2 + 1$
$= 2^iM(n-i) +2^{i-1} + 2^{i-2} +... + 2 + 1 = 2^iM(n-i) + 2^i - 1$
let $i = n-1$
$M(n) = 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n- 1$
5. Algorithm is $O(2M(n-1) + 1) = O(2^n)$ (exponential)

## Example 2 - Towers of Hanoi

Say every move takes 1millisecond($1x10^{-6}$)

$T(n) = C_{op}M(n);$

$C_{op}=1x10^{-6}$

$M(n) = 2^n$

To solve puzzle with 5 disks:

$T(5) = 1x10^{-6}(2^5)$ = 32 milliseconds

To solve puzzle with 15 disks:

T(15) = 0.033 seconds

To solve puzzle with 30 disks:

T(30) = 1074 seconds = 17 minutes

To solve puzzle with 50 disks:

$T(50) = 1.125 \times 10^9$ seconds = 36 years!

Exponential algorithms are not advised for large input sizes!

## Summary

- Recursion is a problem-solving process that breaks a problem into identical but smaller problems
- The definition of a recursive method is that is must contain logic that involves a parameter to the method and leads to different cases. One or more of these cases are base (stopping) cases, as they provide a solution that does not require further recursion.

## Summary

- For each call to a method, the values of the method’s parameters and local variables are stored in a Stack, that organises them chronologically. The most recent at the top. In this way, execution of a recursive method can be suspended and invoked again when new values are obtained.

## Summary

- A recursive method that process an array often divides the array into portions. Recursive calls to the method work on each of these array portions
- A recursive method that processes a linked list needs a reference to its first node as a parameter
- Any solution to the Towers of Hanoi problem with *n* disks requires at least $2^n -1$ moves. A recursive solution is clear and efficient

## Summary

- Each number is the Fibonacci sequence (after the first two) is the sum of the previous two numbers. Computing a Fibonacci sequence recursively is quite inefficient, as the required previous numbers are computed several times each

# Bibliography

- Frank M. Carrano & Walter Savitch, *“Data Structures and Abstractions with Java”*, Prentice Hall/Pearson Education, 2003
- David J. Barnes & Michael Kölling, “*Objects First with Java: A Practical Introduction using BlueJ”,* Prentice Hall / Pearson Education, 2006
- Eclipse Software Development Kit [www.eclipse.org](http://www.eclipse.org)