# Recursion

- References: Text book Chapter 7

## 1. Introduction

- Recursion is a powerful problem solving technique
    - It breaks a problem into smaller identical problems
    - Then breaks new problem into even smaller problems
    - Eventually, new problem is small and can be solve easily (base case)
    - The solutions to small problems can lead to the solution of original problem

- Recursive methods are methods that call themselves

- The invocation is a
  Recursive call
  Recursive invocation

- Simple example 1:

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

- When Designing Recursive Solution

  Note: Explain using above example

  - What part of the solution can you contribute directly? // display number

  - What smaller but identical problem has a solution that, when taken with your contribution provides solution to the original problem // integer-1

  - When does the process end?
    What smaller but identical problem has known solution?
    Have you reached the base case?
    // when integer =1, <1 do not call again

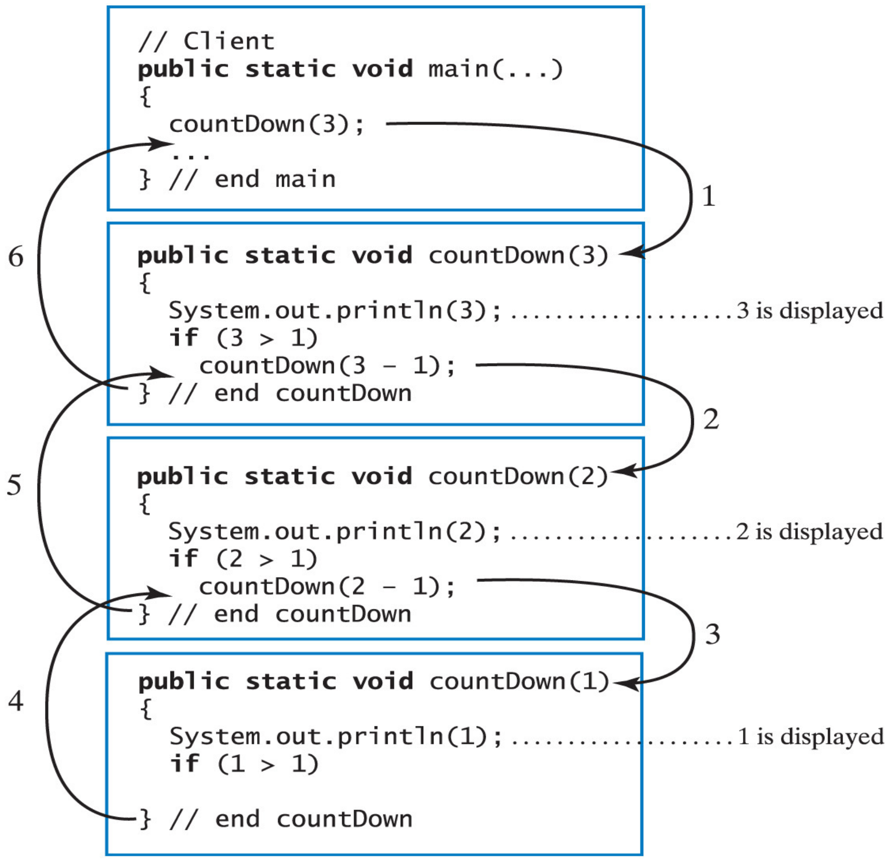
- General Guidelines:

  - 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: call countdown(3);

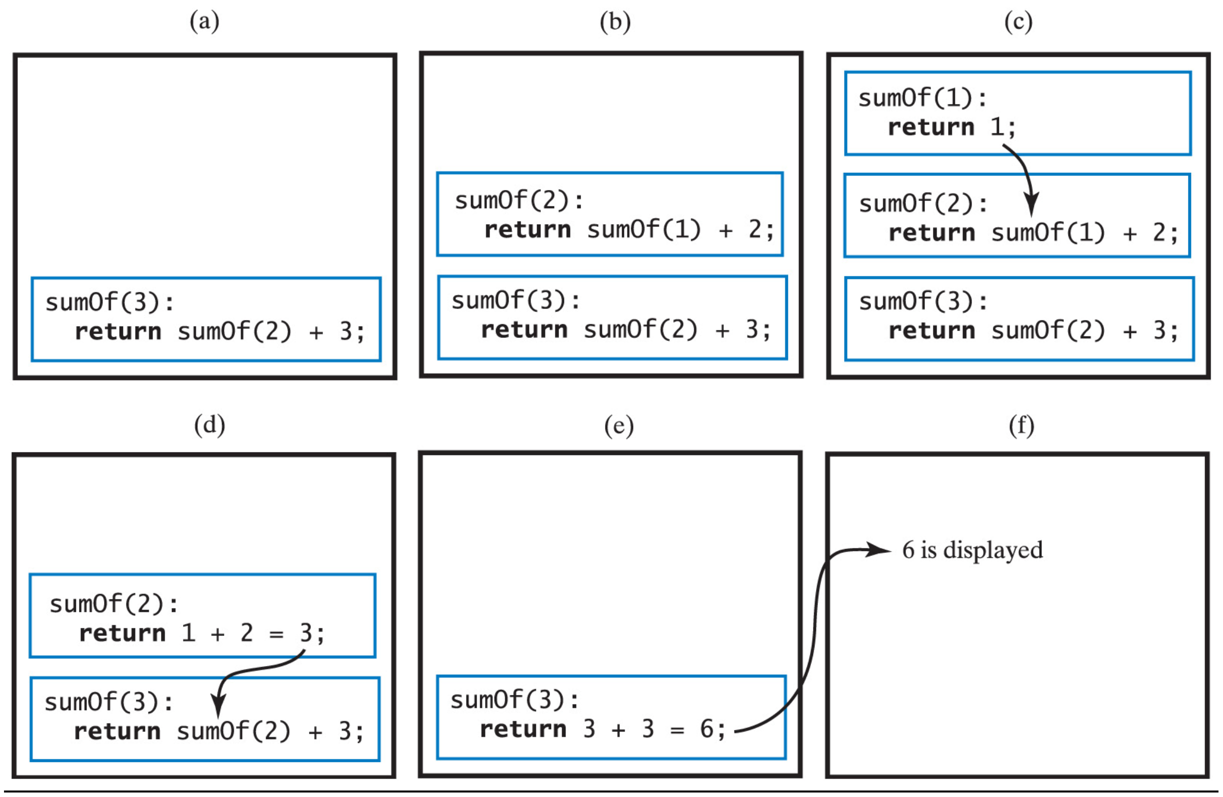


```
// Client
public static void main(...)
{
  countDown(3);
  ...
} // end main

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
```

Note : Uncompleted method calls (activation record) are saved in the process stack! (cover more in future…)

- Simple example 2: Recursive Methods That Return a Value

```
/** @param n  an integer > 0
 *  @return the sum 1 + 2 + ... + n */
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
```



- Simple example 3: Factorial function

```
int myFactorial( int integer)
{

        if( integer == 1)
                return 1;
        else
        {
              return( integer * myFactorial(integer-1) );
         }

}
```

## 2. More Examples

- Example 1 : [Binary Search] An array list[n] consists of n ≥ 1 distinct integers such that list[0] ≤ list[1] ≤ ...≤ list[n-1].

  Goal : We want to figure out if an integer "searchNum" is in the list. If it is, we should return an index, i, such that list[i] = searchNum; otherwise return -1.

  Algorithm Outline (note: this is not a Java program, just pseudo code):

  let "left" and "right" denote the left and right ends of the list to be searched.

  initially, left = 0 and right = n-1

```
int search(left, right, searchNum)

    if  (left > right)
        index = -1
    else
        let middle = (left + right)/2
        if (list[middle] = searchNum) index = middle
        else if (list[middle] > searchNum)
            index = search(left, middle-1, searchNum)
        else // list[middle] < searchNum
            index = search(middle+1, right, searchNum)
        return index
```

```
Example :       1  5  9  12  15  21  29  31     searchNum = 9
         index: 0  1  2   3   4   5   6   7

    [0,7] → [0,2] → [2,2]   // found list[2] = 9
                               return 2 → return 2 → return 2  // exit function!
```

**Note: See Textbook, Chapter 18, Page 457 for Java code and Java API in Arrays**

- Example 2: Tower of Hanoi

  There are 3 poles and n disks of different diameter placed on the 1st pole. The disks are in order of decreasing diameter as one scan up the pole. Monks were repeatedly supposed to move disks from pole 1 to pole 3 using spare pole 2 and obeying the following rules:

  (i) only one disk can be moved at any time,
  (ii) no disk can be placed on the top of a disk with smaller diameter.

  Major steps : // poles : source, spare & destination

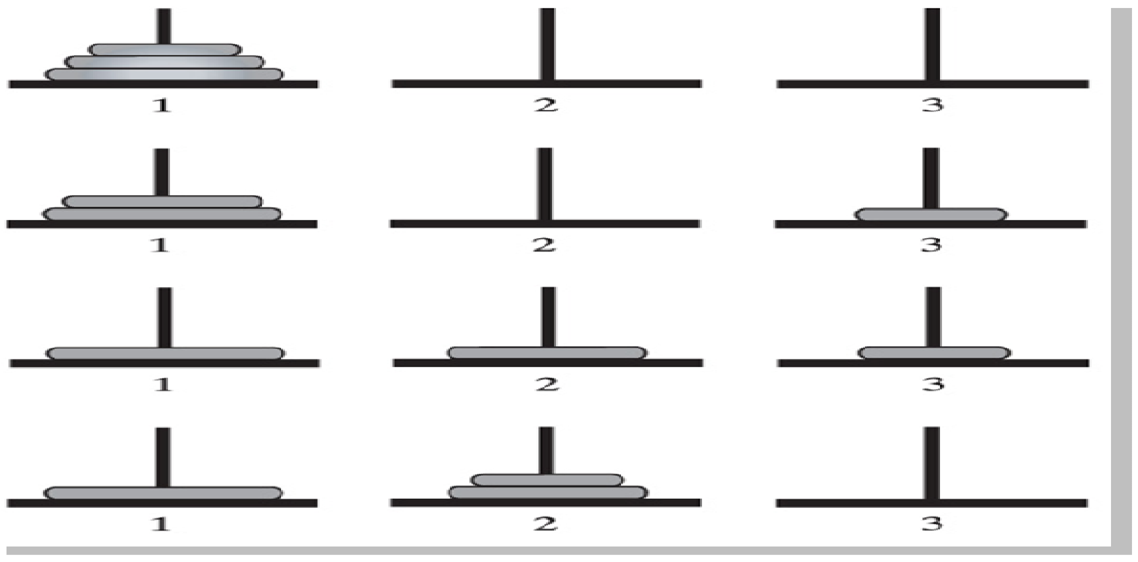
  Base case : n=1

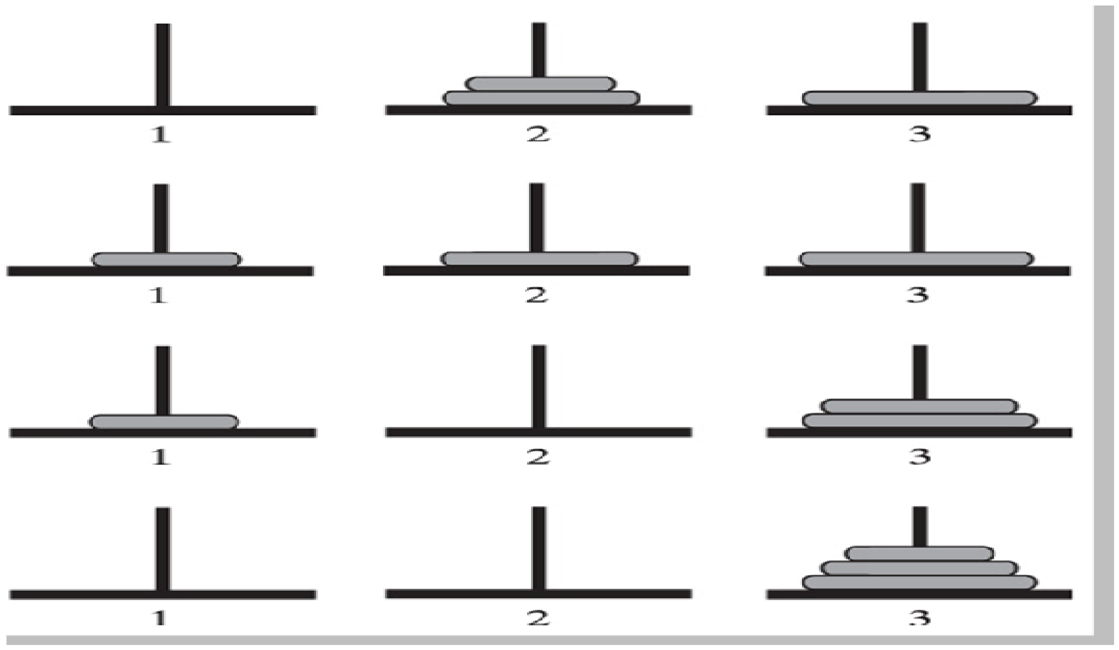
  move the disk from source to destination directly

  Recursion : n > 1

  move (n-1) disks from source to spare
  move n th disk from source to destination
  move (n-1) disk from spare to destination

  Note : n = 1 or 2 are easy! Next, n=3 disks





Main : Move 3 disks from #1 to #3, spare #2

(a) move 2 disks from #1 to #2
(b) move 1 disk from #1 to #3
(c) move 2 disks from #2 to #3

- From (a), source #1, destination #2, spare #3

(a.1) move 1 disk from #1 to #3
(a.2) move 1 disk from #1 to #2
(a.3) move 1 disk from #3 to #2

- From (b), source #1, destination #3 (base case)

- From (c), source #2, destination #3, spare #1

(c.1) move 1 disk from #2 to #1
(c.2) move 1 disk from #2 to #3
(c.3) move 1 disk from #1 to #3

n = 4 disks

- Main : Move 4 disks from #1 to #3, spare #2

  (a) move 3 disks from #1 to #2 // same case as n=3 disks
  (b) move 1 disk from #1 to #3 // base case
  (c) move 3 disks from #2 to #3 // same case as n=3 disks

  Towers of Hanoi Program (C++ program, see textbook for Java program)

```
/* n = # of disks, i = source, j = destination, k = spare */

void tower(int n, int i, int j, int k);

main()
{
    int n;
    cout << "enter a number n : " << endl;
    cin >> n;
    tower(n,1,3,2);
}

void tower(int n, int i,  int j, int k)
{
    if (n==1) {         // base case
            cout  << "move top disk from pole " << i  << "to pole "
                    << j << endl;
    } else {            // for n > 1

            tower(n-1,i,k,j);
            tower(1,i,j,k);
            tower(n-1,k,j,i);
    }
}
```

## 3. More on recursion…

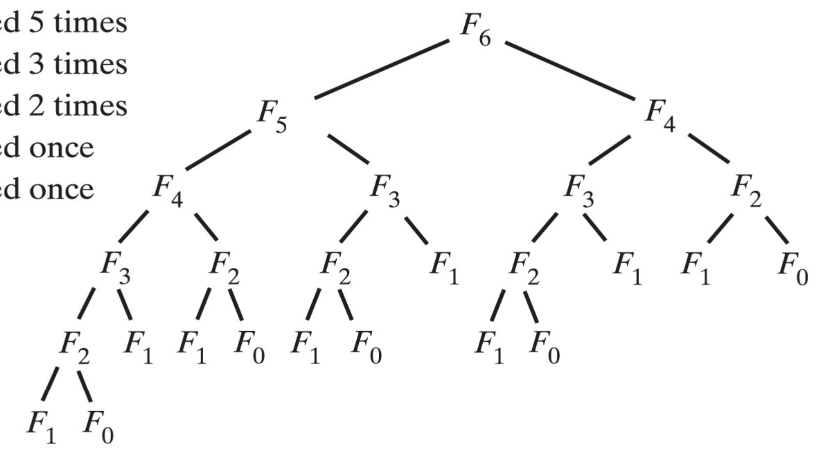
- A Poor Solution to a Simple Problem

  Compute Fibonacci numbers
    First two numbers of sequences 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

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

(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


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

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

(a) is recursive solution: exponential time
(b) can be done iteratively: O(n)

- Tail Recursion: Occurs when the last action performed by a recursive method is a recursive call

  - countdown() and factorial number methods
  - This performs a repetition that can be done more efficiently with iteration
  - Conversion to iterative method is usually straightforward

- Mutual Recursion (or indirect recursion)

  - Happens when Method A calls Method B
    Which calls Method B
    Which calls Method A
    etc.

  - Difficult to understand and trace
    Happens naturally in certain situations

  - Example: A simple recursive way to determine if a number N is either even or odd. Given that

    0 is even
    N is even if n-1 is odd
    N is odd if n-1 is even

```
Start with even(n) // to check even number
Start with odd(n)  // to check odd number

odd(n)

    if n = 0 then return FALSE
    return even(n-1)

even(n)

    if n = 0 then return TRUE
    else return odd(n-1)
```