Recursion in Java

Recursion: **Recursion is the process of defining something in terms of itself.**

Leonardo da Vinci
1452 –1519

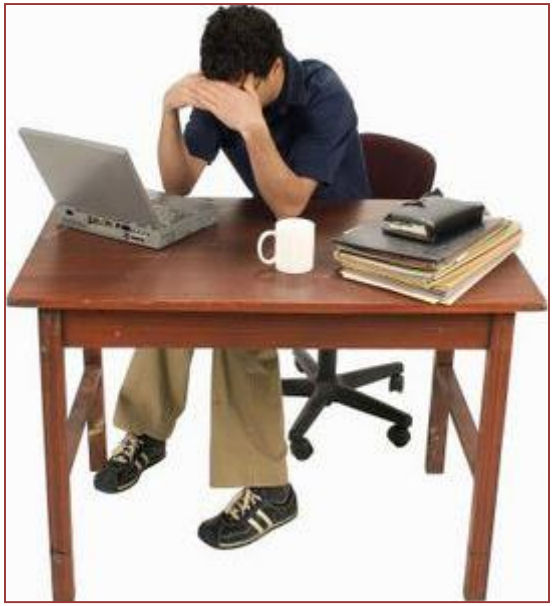
La Giaconda (Mona Lisa)
Louvre, Paris.

Why learn recursion ?

- "cultural experience“ - A different way of thinking on problems.
- Can solve *some kinds of problems* better than iteration (loops).
- Leads to elegant, simplistic, short Java code (*when used well*).
- Many programming languages ("functional" languages such as Scheme, ML, and Haskell) *use recursion exclusively* (no loops).

Why learn recursion ?

Recursion is a key component of the our course **assignments number 3 !**



Exercise 1

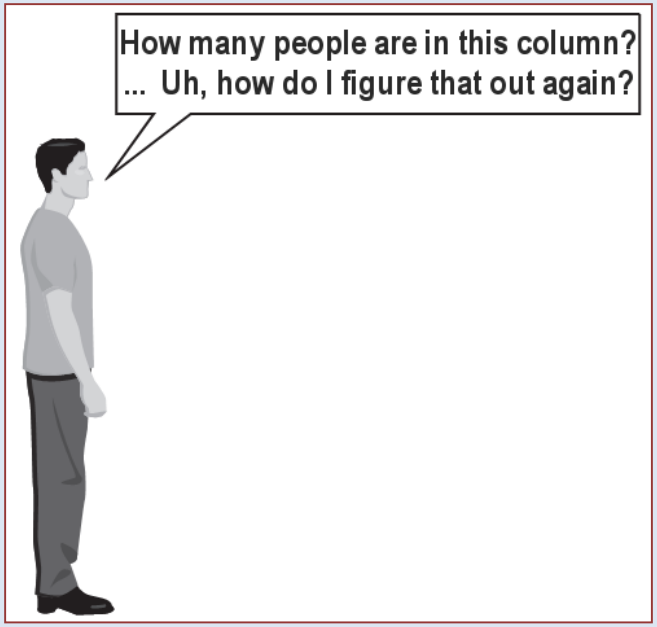
(**To a student in the front row only**)

How many students **total** are directly behind you in your "column" of the classroom?

- You have poor vision, so you can see only the people right next to you. So, you can't just look back and count.

- But you are allowed to ask questions of the *person next to you*.

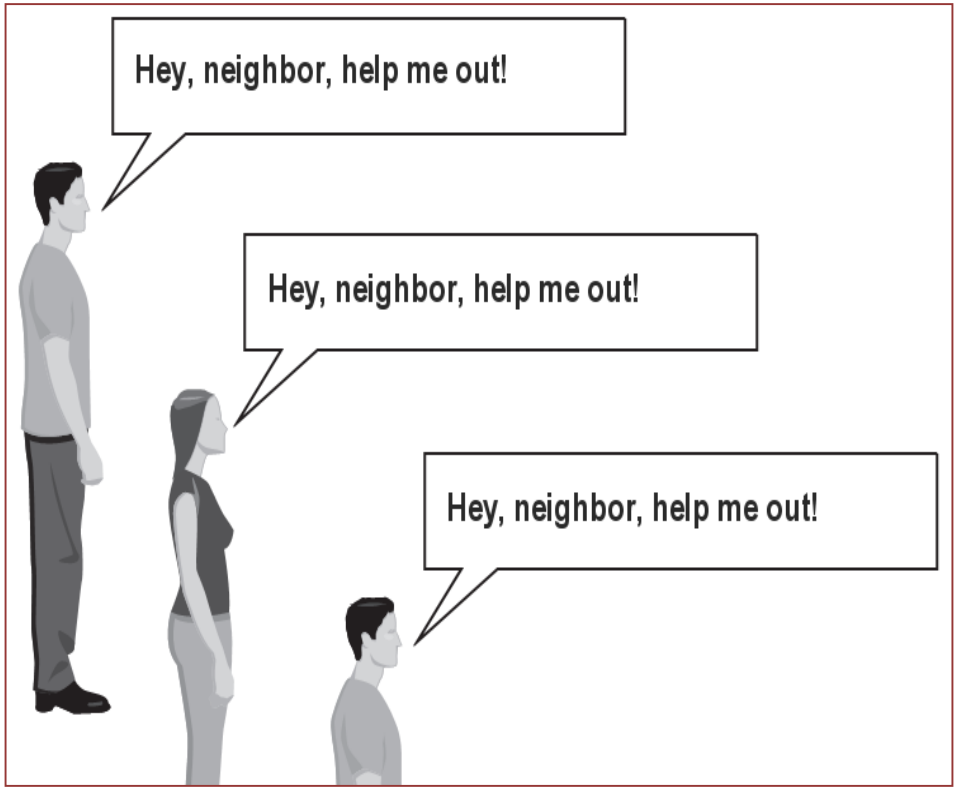
How can we solve this problem ?
(*recursively !*)



The recursion idea

Recursion is all about breaking a big problem into *smaller occurrences* of that same problem.

- Each person can solve a small part of the problem.
- What is a small version of the problem that would be easy to answer?
- What information from a neighbor might help me?

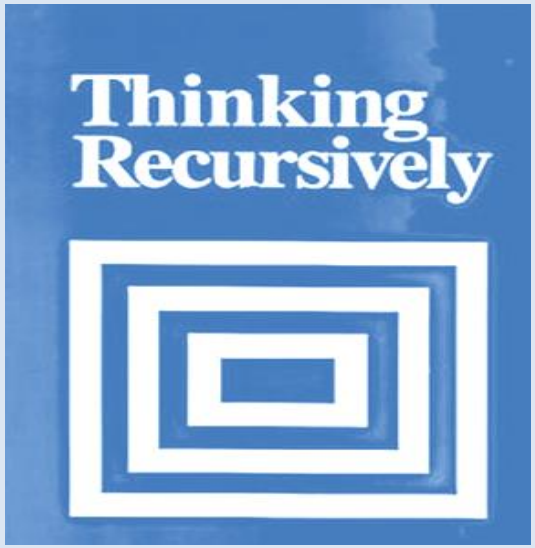


Recursive algorithm

- **Number of people behind me:**
  - If there is someone behind me,
    ask him/her how many people are behind him/her.

    - When they respond with a value **N**, then
      I will answer **N + 1**.

  - **If there is nobody behind me,**
    I will answer **0**.



Exercise 2 - recursive algorithm

How do you look up a name in the phone book?

Search :

middle page = (first page + last page) / 2

Go to middle page

If (*name is on middle page*)

done;  this is the **base case**

else

if (name is alphabetically *before* middle page)

last page = middle page redefine search area to front half

Search same process on **reduced number of pages**

else name must be after middle page

first page = middle page redefine search area to back half

Search same process on **reduced number of pages**

Recursive algorithm - Overview

- Natural approach to ***some*** (**not all**) problems.
- A *recursive algorithm* uses itself to solve one or more smaller identical problems.
- Each successive call to itself must be a " *smaller version of itself* ".
- *A recursive algorithm* must eventually terminate.

  A recursive algorithm must have at least one base, or stopping, case.

  A base case does not execute a recursive call.

Recursive algorithm - Design

Three key Components of a Recursive Algorithm Design:

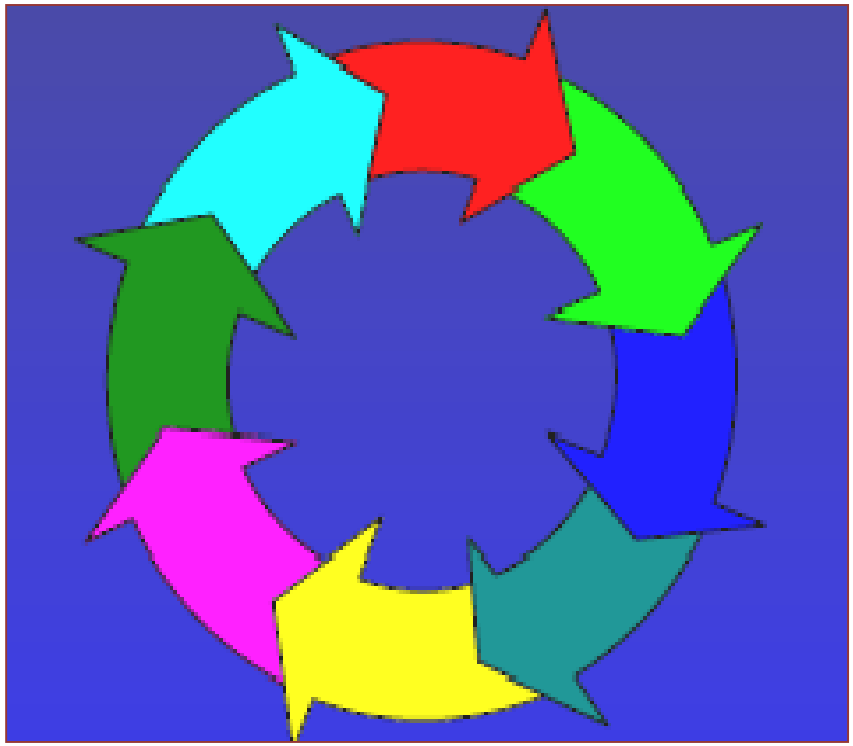
1. What is a smaller identical problem(s)?
   - Decomposition (פירוק)

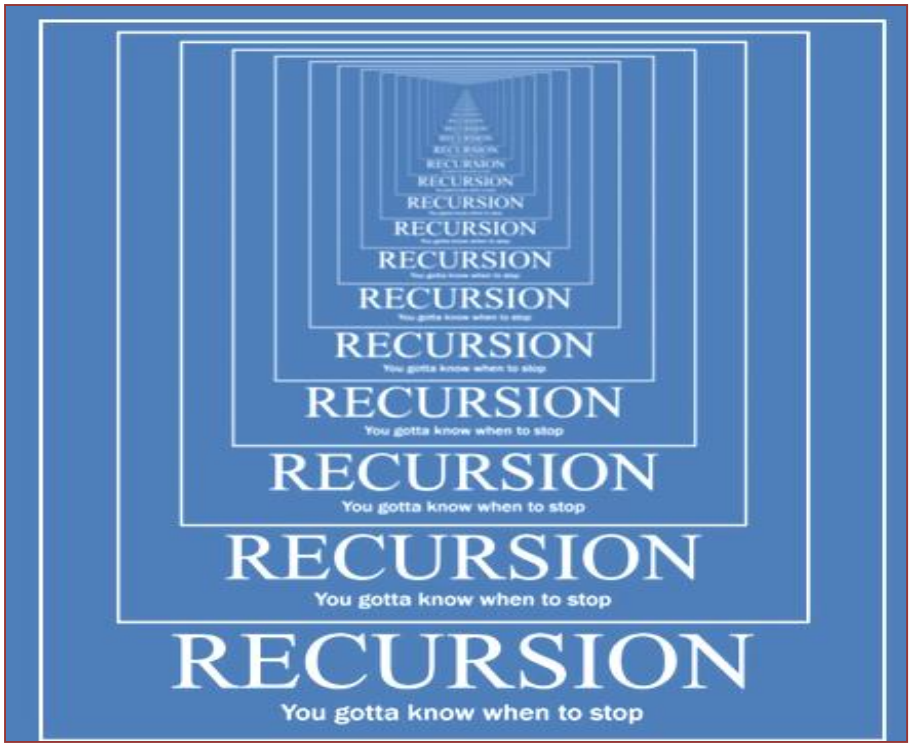
2. How are the answers to smaller problems combined to form the answer for a larger problem?
   - Composition (הרכבה)

3. What is the smallest problem which can be solved easily (without further decomposition)?
   - Base/stopping case (מקרה בסיסי)

Base (stopping) case

A **base case** does not execute a recursive call !





Factorial (*N*!)

- *N*! = (*N*-1)! * N // for *N* > 1
- 1! = 1
- 3! =

  = 2! * 3 =

  = (1! * 2) * 3

  = 1 * 2 * 3
- Recursive design:
  - Decomposition: (*N*-1)!
  - Composition: * *N*
  - Base case: 1!

```
public static int factorial(int n)
{
   int fact;
   if (n > 1) // recursive case (decomposition)
        fact = factorial(n – 1) * n; // composition
   else
        fact = 1; // base case
   return fact;
} // factorial
```

```
public static int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

Each ***recursive call*** to itself must be a " smaller version of itself ".

```
public static int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
public static int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return fact;
}
```

A ***base case*** does not execute a recursive call and stops the recursion.

```
public static int factorial(int 3)
{
  int fact;
  if (n    1)
    fact    actorial(2) * 3;
  el
    fact    1;
  return fact;
}
```

return 6

```
public static int factorial(int 2)
{
  int fact;
  if (    > 1)
    f       factorial(1) * 2;
  else
    fac   = 1;
  return fact;
}
```

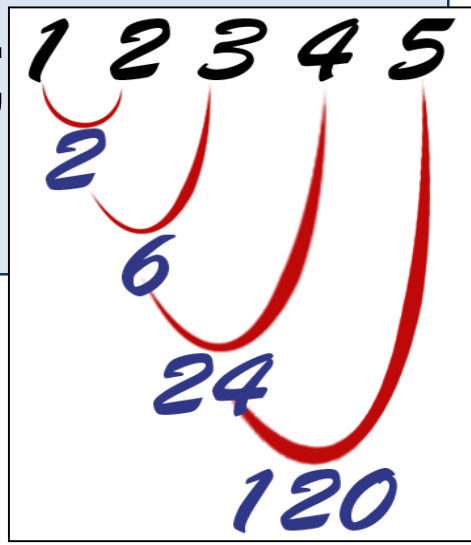
return 2

```
public static int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return fact;
}
```

return 1

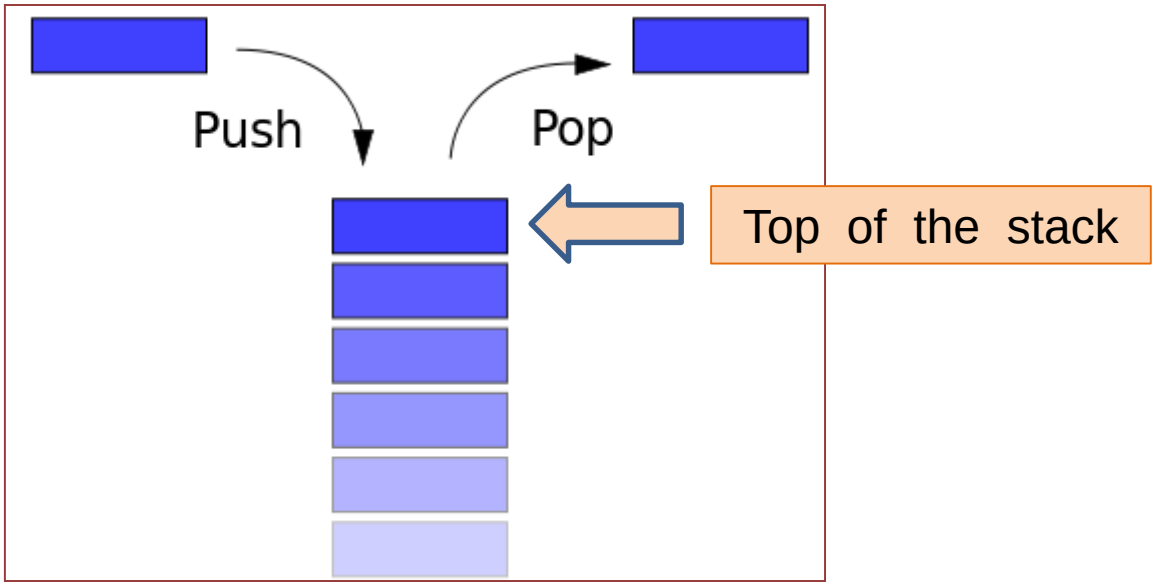
Improved factorial method

```
public static int factorial(int n)
{
   if (n ==1)
         return 1;
   else
         return factorial(n – 1) * n;
} // factorial
```

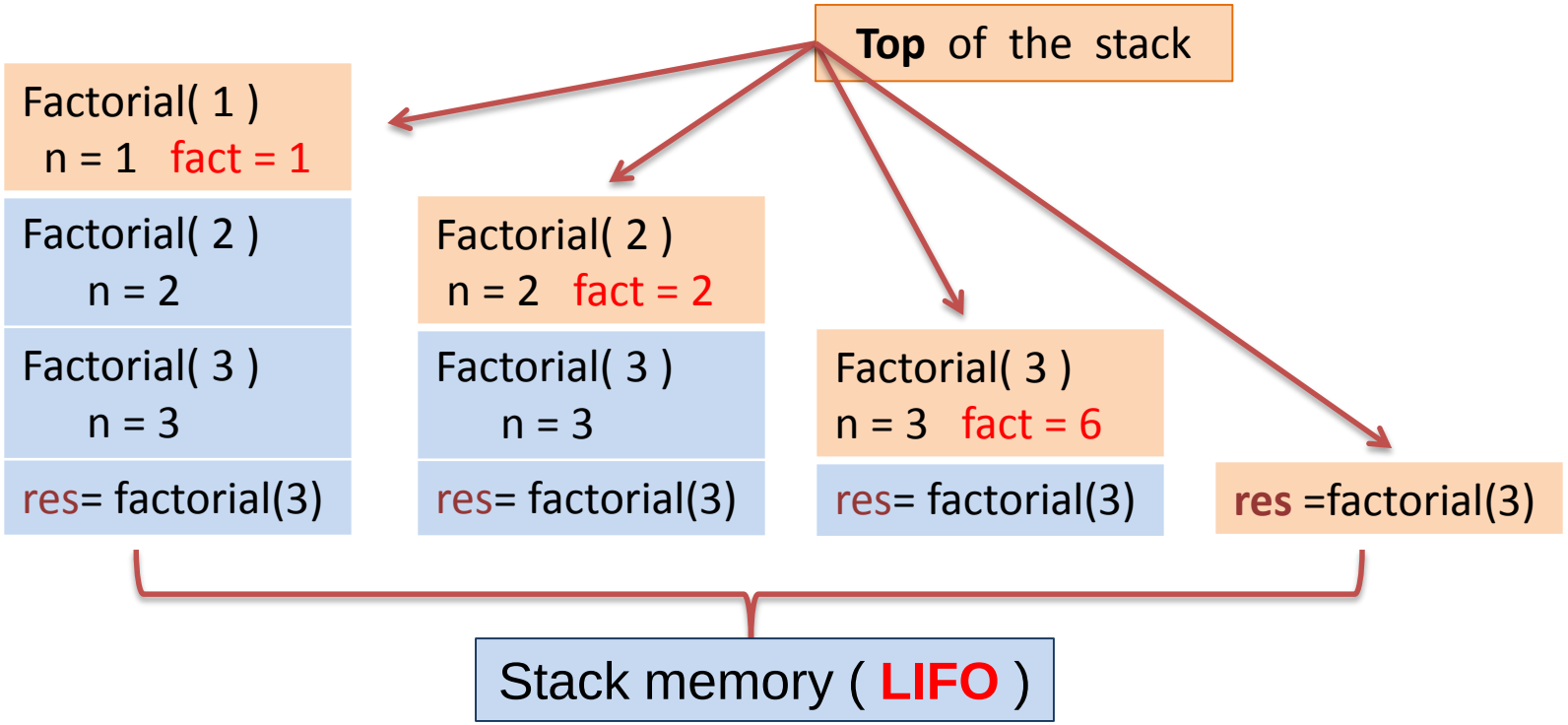


STACK (מחסנית)

- A stack is a special area of memory where access can be made only from one end. You can only access and work with the top of the stack.
- This is called a **L**ast **I**n **F**irst **O**ut (**LIFO**) structure.
- A stack has two fundamental operations - **Push** and **Pop**.
  The **Push** operation stores something on the top of the stack and the **Pop** operation retrieves something from the top of the stack.
- **Calling a method** often means putting the parameter values on to the stack, **returning from a method** is essentially the reverse process.



How recursion works

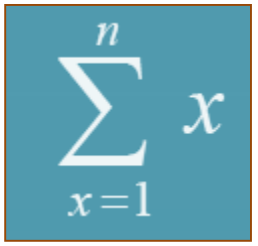


In Java, when a method encounters another method, it gets pushed to the ***top*** *of the stack* and temporarily stops the original method until the new method is completed.

Once the method being called finished, the program picks up from where it left off.

Arithmetic Series

Calculation arithmetic series (sigma) recursive Sum :




```
public static int sigma( int n)
{
     if(n <= 1)
          return n;
    else
          return n + sigma(n-1);
} // sigma
```

Recursive design:

Decomposition: (*n*-1)

Composition: n + sigma(n-1)

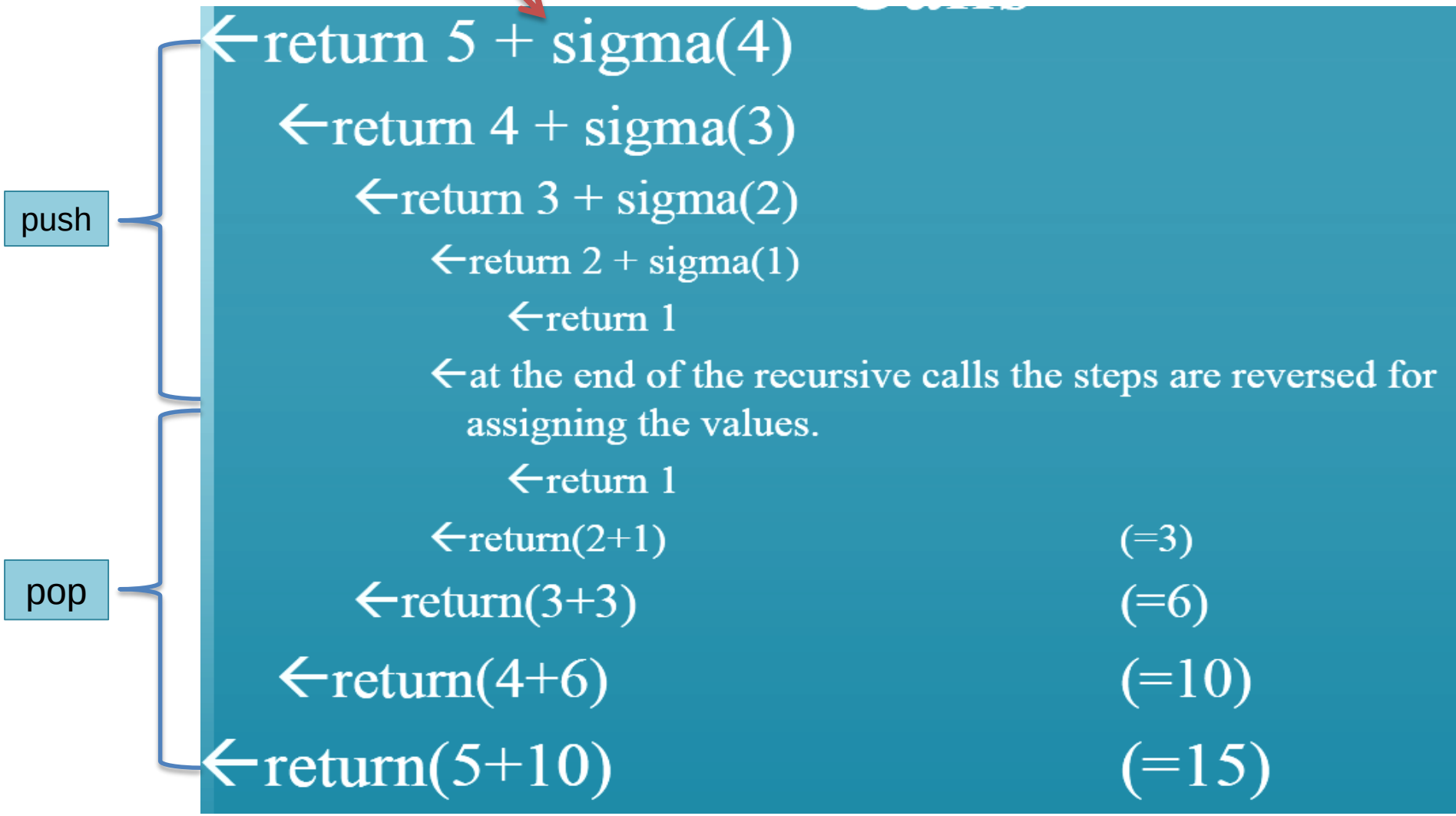
Base case: n=1

If we call the **sigma** method with the Java statement :

int sum = sigma(5) ;

the else portion of the first method's calls the method **sigma** again

return 5 + sigma(4);



Calculation power

$$X^y = \underbrace{x*x*\ldots*x}_{y \text{ times}}$$

```
public static int power(int x, int y)
{
    if (y == 0)
        return 1;
    else
        return x*power(x, y– 1);
} // power
```

Recursive definitions (assume non-negative y):

$X^y = X*(X^{y-1})$

Base case: $x^0=1$

Calculation product

Calculation product of two non - negative integers without multiplication operator '*' (we can use only '+' and '-' operators)

The product a*b is actually a + a +…+ a (b times).

x * y = x + x + x + x + …+ x

y times

Recursive definitions :

x * y = y * x = y + (x – 1) * y

Base case: 0 * Y = 0

```
public static int  recMult( int x, int y )
{
        if( x == 0)
                return 0;
        return   y  +  recMult( x-1,y);
} // recMult
```

Recursive addition

```
public static int add(int a, int b)
{
    if(b == 0)
        return a;
    else
        if(b < 0)
            return add(a -1,b + 1);
        else
            return add(a + 1,b - 1);
} // add
```

Base case

a	b
5	2
6	1
7	0

```
public static void main(String[ ] args)
{
    int x = reader.nextInt(); // 5
    int y = reader.nextInt(); // 2
    System.out.println(add(x,y));
} // main
```

a	b
5	- 2
4	- 1
3	0

Recursive subtraction

```
public static int sub(int a, int b)
{
    if(b == 0)
        return a;
    else
        if(b < 0)
            return sub(a + 1,b + 1);
        else
            return sub(a -1,b -1);
} // sub
```

Base case

a	b
5	2
4	1
3	0

```
public static void main(String[ ] args)
{
    int x = reader.nextInt(); // 5
    int y = reader.nextInt(); // 2
    System.out.println(sub(x,y));
} //main
```

a	b
5	- 2
6	- 1
7	0

Recursive integer division

```
public static int divide(int a, int b)
{
    if(b == 0)
        return 0;
    else
        if(a < b)
            return 0;
        else
            if(b == 1)
                return a;
            else
                return add(1, divide(sub(a,b),b));
} // divide
```

Base cases

a	b
5	2
?	?

```
public static void main(String[ ] args) {
    int a = reader.nextInt(); //5
    int b = reader.nextInt(); //2
    System.out.println(divide(a,b));
} // main
```

Member of arithmetic sequence

This recursive method calculates the **n member** of arithmetic sequence, beginning from **start** value with **d** value sequences difference.

```
public static int memSec(int start, int d, int n)
{
     if(n == 1)
               return start;
    else
               return(d + memSec(start,d,n-1));
} // sumSec
```

start	d	n	sum
5	4	3	13
0	5	5	20
10	10	10	100

What1 recursive method

```
public static int What1 (int a, int b)
{
      int ans;
      if (b == 0)
            ans = 0;
      else
            if (b % 2 == 0)
                  ans = What1(a + a, b/2);
            else
                  ans = What1(a + a, b/2) + a;
     return ans;
 } // What1

 public static void  main(String[ ] args)
 {
      int a = reader.nextInt();
      int b = reader.nextInt();
      int ans = What1(a,b);
     System.out.println( "ans = “ + ans);
 } // main
```

This program reads two integers and **…?**

What1 recursive method trace

```
public static int What1 (2, 3)
{
    int ans;
    if (b == 0)  F
        ans = 0;
    else
        if (b % 2 == 0)  F
            ans = What1(a+a, b/2);
        else
            ans = What1(4, 1) + 2;
    return ans;
 } // What1
```

6

```
public static int What1 (a, b)
{
    int ans;
    if (b == 0)
        ans = 0;
    else
        if (b % 2 == 0)
            ans = What1(a + a, b/2);
        else
            ans = What1(a + a, b/2) + a;
    return ans;
 } // What1
```

```
public static int What1 (4, 1)
{
    int ans;
    if (b == 0)  F
        ans = 0;
    else
        if (b % 2 == 0)  F
            ans = What1(a+a, b/2);
        else
            ans = What1(8, 0) + 4;
    return ans;
 } // What1
```

4

```
public static int What1 (8, 0)
{
    int ans;
    if (b == 0)  T
        ans = 0;
    else
        if (b % 2 == 0)
            ans = What1(a+a, b/2);
        else
            ans = What1(8, 0) + 4;
    return ans;
 } // What1
```

0

What2 recursive method

```
public static int What2(int x,int y)
{
    if((x == 0) || (y == 0))
            return 0;
    else
            if((x % 10 ) == (y%10))
                    return 1+ What2(x/10,y/10);
            else
                    return(What2(x/10, y/10));
} // What2
public static void  main(String[ ] args)
{
    int a = reader.nextInt();
    int b = reader.nextInt();
    int ans = What2(a,b);
   System.out.println( "ans = “ + ans);
} // main
```

This program reads two integers and **…?**

a	b	output
27	831	0
235	15	1
5101	101	3

What3 recursive method

```
public static int What3 (int a, int b)
{
    int ans;
    if (b == 0)
            ans = 1;
    else
            if (b % 2 == 0)
                    ans = What3(a*a, b/2);
            else
                    ans = What3(a*a, b/2) * a;
    return ans;
 } // What3
public static void  main(String[ ] args)
{
    int a = reader.nextInt();
    int b = reader.nextInt();
    int ans = What3(a,b);
   System.out.println( "ans = “ + ans);
} // main
```

This program reads two integers and **…?**

Fibonacci series

- Fibonacci was born on 1170 in Pisa, Italy and died on 1250.
  His real name is **Leonardo Pisano**.
- The ***N*** - th Fibonacci number is the sum of the previous two Fibonacci numbers :

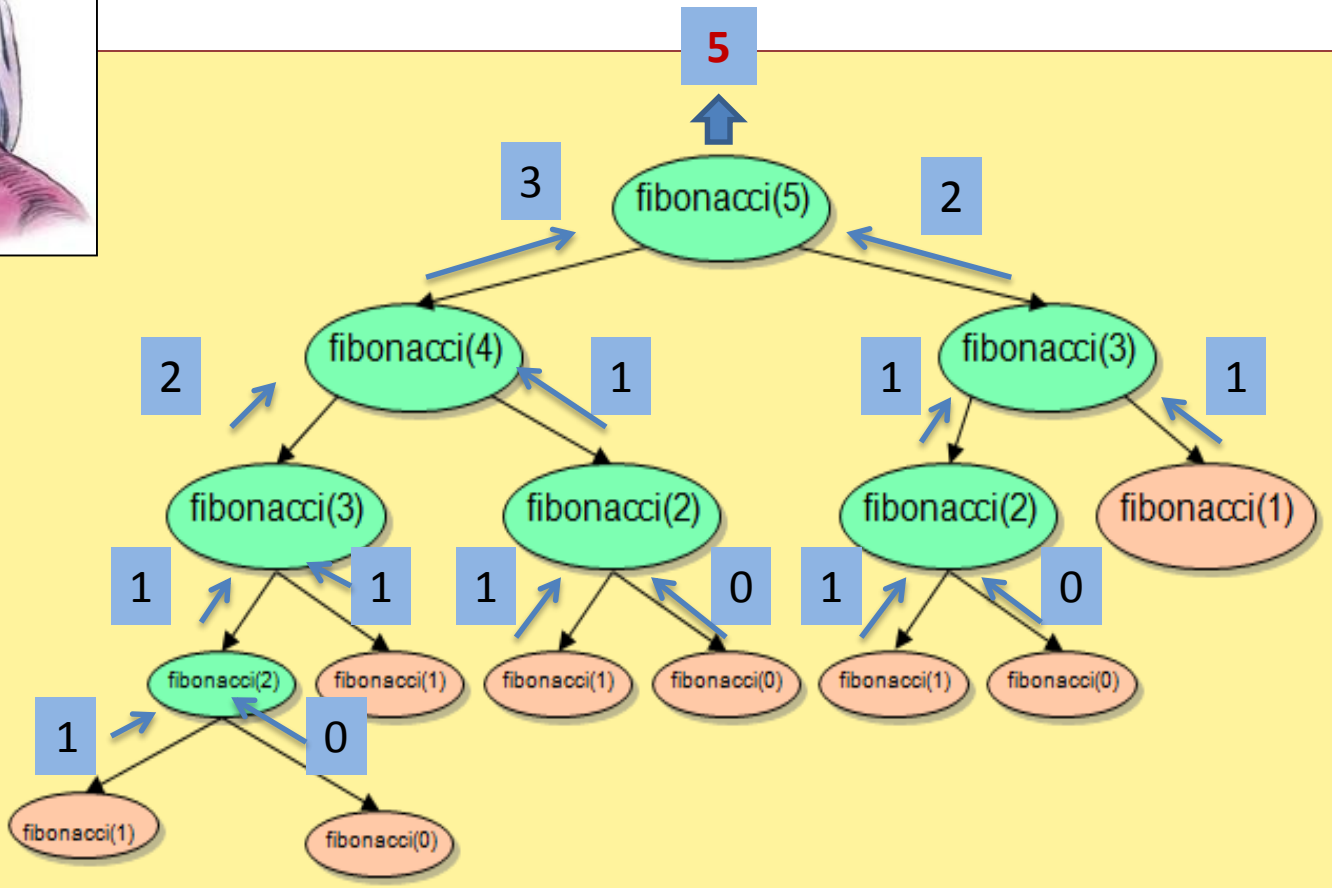
0, 1, 1, 2, 3, 5, 8, 13, …

- Recursive Design:
  - Decomposition & Composition
    - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
  - Base case:
    - fibonacci(0) = 0
    - fibonacci(1) = 1

Binary Recursion is a process where function is called twice at a time instead of once at a time.

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

Fibonacci series n = 5

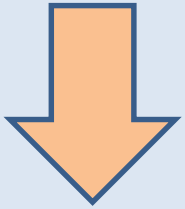


Binary recursion (רקורסיה כפולה)

Recursion and loops

```
public static int recLoop(int num)
{
    if(num == 1)
        return 1;
    int sum = num;
    for(int i = 1; i < num; i++)
        sum = sum + recLoop(i);
    return sum;
} // recLoop
```

```
public static void main(String[ ] args)
{
    System.out.println("enter an integer ->  ");
    int x = reader.nextInt();
    System.out.println("what = “ + recLoop(x));
}  // main
```



input(x)	output(what)
1	1
2	3
3	7
4	15

Printing stars

```
public static void stars1(int n)
{
    if (n < 1)
        return;
    System.out.print(" * ");
    stars1(n-1);
} //stars1
```

```
public static void stars2(int n)
{
    if (n > 1)
        stars2(n-1);
    System.out.print(" * ");

 } // stars2
```

If the method stars1 is called with the value **3**, is it equivalent to the method stars2 ?

Explain !

Invoke stars1

```
static void stars1( 3 )
{
    if (n<1)   FALSE
        return;
    System.out.print(" * ");
    stars1(2);
}
```

First output : *

```
static void stars1( 2 )
{
    if (n<1)    FALSE
        return;
    System.out.print(" * ");
    stars1(1);
}
```

Second output : **

```
static void stars1( 1 )
{
    if (n<1)    FALSE
        return;
    System.out.print(" * ");
    stars1(0);
}
```

Third output : ***

```
static void stars1( 0 )
{
    if (n<1)   TRUE
        return;
    System.out.print(" * ");
    stars1(n-1);
}
```

Invoke stars2

```
static void stars2( 3 )
{
    if (n>1)   TRUE
       stars2(2);
    System.out.print(" * ");
}
```

Third output : ***

```
static void stars2( 2 )
{
    if (n>1)    TRUE
       stars2(1);
    System.out.print(" * ");
}
```

Second output : **

```
static void stars2( 1 )
{
    if (n>1)    FALSE
       stars1(i-1);
    System.out.print(" * ");
}
```

First output : *

Mutual recursion

- *Mutual Recursion* (*רקורסיה הדדית*) is the kind of recursion where recursive methods calling each other.

For example: **YES** method calling **NO** method and **NO** method calling **YES** method recursively.

```
public static void YES (int  num)
{
      System.out.println("yes");
          if (num > 1)
              NO(num-1);
} //YES
```

```
public static void NO (int  num)
{
      System.out.println("no");
      if  (num > 1)
          YES(num-1);
 } // NO
```

invoke YES(5) would produce next output:

yes
no
yes
no
yes

Reverse an array

The method gets an array and it's length as parameters and returns the array after reversing it's values.

```
public static int[ ] reverseArr(int[ ] arr, int length)
{
    if(length <= arr.length / 2 )
                        return arr;
    else // swapping the values
    {
        int tempVal = arr[length - 1];
        arr[length - 1] = arr[arr.length - length];
        arr[arr.length - length] = tempVal;
    }
    return reverseArr(arr, length - 1);
}  // reverseArr
```

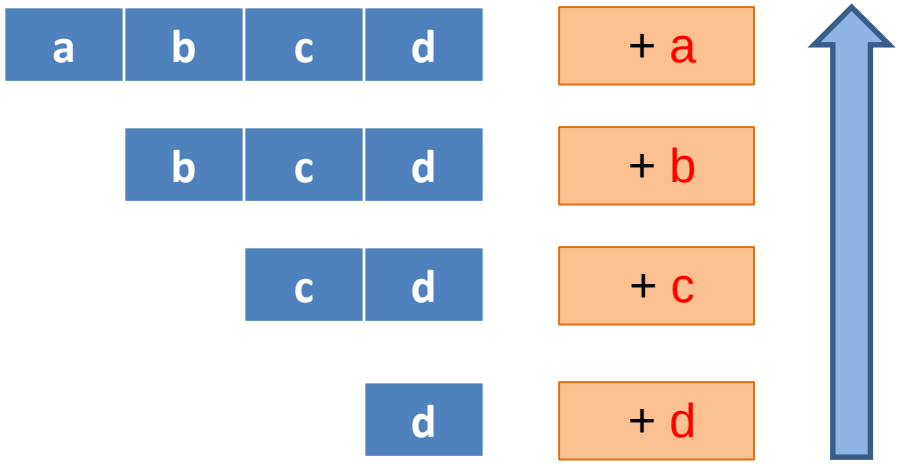
```
int[ ] a = { 1,2,3,4,5 } ;
a = reverseArr( a,a.length);
```

Method returns the array after reversing it's values:

5	4	3	2	1

Reverse a string

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




```
System.out.print("Enter  the string:");
String str = reader.next(); // abcd
System.out.print("The reverse string is"
                        + reverseString(str));
```

would produce next output:

dcba

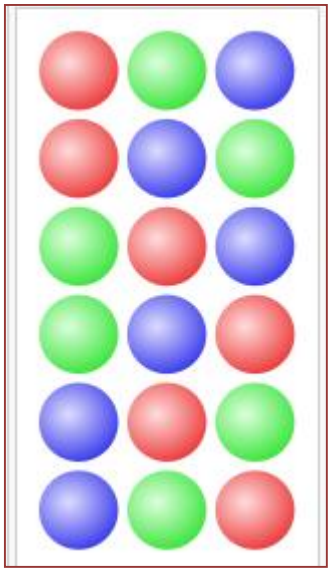
Permutation (שינוי סדר)

The notion of **permutation** is used with several slightly different meanings, all related to the act of **permuting** (rearranging) objects or values.

For example 1 :

There are six permutations of the set {1,2,3}, namely (1,2,3) (1,3,2), (2,1,3), (2,3,1), (3,1,2), and (3,2,1).

For example 2 :



The 6 permutations of 3 balls.

The number of permutations of ***n*** distinct objects is $n\times(n-1)\times(n-2)\times\cdots\times2\times1$, which is commonly denoted as "*n* factorial " and written " ***n!*** ".

Anagram

An **anagram** (**אנגרמה**) of a word is a permutation of its letters :

The result of rearranging the letters of a word to produce a new word, using all the original letters exactly once.

The original word is known as the *subject* of the anagram.

For example :

If the subject word is "**abc**" , then all anagrams of a word is the next set:

abc
acb
bac
bca
cab
cba

$3! = 1 \times 2 \times 3 = 6$

Anagram and recursion

We can use recursion to generate all the anagrams of a word of any length.

The algorithm works as follows:

put every character in the string as *first letter*, and recursively find all anagrams of the remaining letters.

Given *abc*, we would place ***a*** in front of all two permutations of bc - bc and cb to arrive at **a**bc and **a**cb.

Then we would place ***b*** in front of all two permutations of ac – ac and ca to arrive **b**ac and **b**ca and finally ***c*** in front of two permutations of ab – ab and ba.

abc
acb
bac
bca
cab
cba

Thus, there will be three recursive calls to display all permutations of a three-letter word.

The ***base case*** of our recursion would be when we reach a word with just one letter.

Anagram solution

```
public static void printAnagrams( String  prefix, String  word )  {
    if(word.length()  <= 1)
            System.out.println(prefix + word);
    else  {
            for(int i = 0; i < word.length(); i++)   {
                    String cur = word.substring(i, i + 1);
                    String before = word.substring(0, i); // letters before cur
                    String after = word.substring(i + 1); // letters after cur
                    printAnagrams(prefix + cur, before + after);
            } // for
    } // else
} // printAnagrams

public static void  main(String[ ] args)
{
    printAnagrams("","abc");
} // main
```

Anagram debugging

```
public static void printAnagrams( String  prefix, String  word )  {
    if(word.length()  <= 1)
            System.out.println(prefix + word);
    else  {
              for(int i = 0; i < word.length(); i++)   {
                    System.out.println("i = " + i);
                    String cur = word.substring(i, i + 1);
                    System.out.println("cur = " + cur);
                    String before = word.substring(0, i); // letters before cur
                    System.out.println("before = " + before);
                    String after = word.substring(i + 1); // letters after cur
                    System.out.println("after = " + after);
                    printAnagrams(prefix + cur, before + after);
              } // for
    } // else
} // printAnagrams
public static void  main(String[ ] args)
{
    printAnagrams("","abc");
} // main
```

Anagram execution

("", "abc")

i = 0
cur = a
before = ""
after = bc

("a", " bc")

i = 0
cur = b
before = ""
after = c

("ab", " c")

abc

("a", " bc")

i = 1
cur = c
before = b
after = ""

("ac", " b")

acb

```
public static void printAnagrams( String  prefix, String  word )  {
    if(word.length()  <= 1)
            System.out.println(prefix + word);
    else  {
            for(int i = 0; i < word.length(); i++)   {
                System.out.println("i = " + i);
                String cur = word.substring(i, i + 1);
                System.out.println("cur = " + cur);
                String before = word.substring(0, i); // letters before cur
                System.out.println("before = " + before);
                String after = word.substring(i + 1); // letters after cur
                System.out.println("after = " + after);
                printAnagrams(prefix + cur, before + after);
            } // for
    } // else
} // printAnagrams
```

```
public static void  main(String[ ] args)
{
      printAnagrams("","abc");
} // main
```

Printing array recursively

```
public static void recPrint(int[ ] Arr, int index)
{
        System.out.print(Arr[index]);
        if (index < Arr.length - 1 )
        {
            System.out.print(", ");
            recPrint(Arr, index+1);
        } // if
       else
            System.out.println( );
}  // recPrint
 public static void  main(String[ ] args)
 {
     int a[ ] = { 1,2,3,4,5 };
     recPrint(a,0);
 } // main
```

Multiply a range of array elements

The method rangeMult() takes three arguments :

- an int array that contains the range of elements to be multiplied
- an int specifying the *starting* element of the range
- an int specifying the *ending* element of the range.

The method will return the product of the array elements that are within the range.

```
public static int rangeMult(int arr[ ], int start, int end)   {
        if (start > end)
                return 1;
        else
                return arr[start] * rangeMult(arr, start + 1, end);
 } //rangeMult

public static void  main(String[ ] args)   {
    int a[ ] = { 1,2,3,4,5 };
   System.out.println(rangeMult(a, 0, 2));
} // main
```

This will produce:

6

Sum numbers in an array

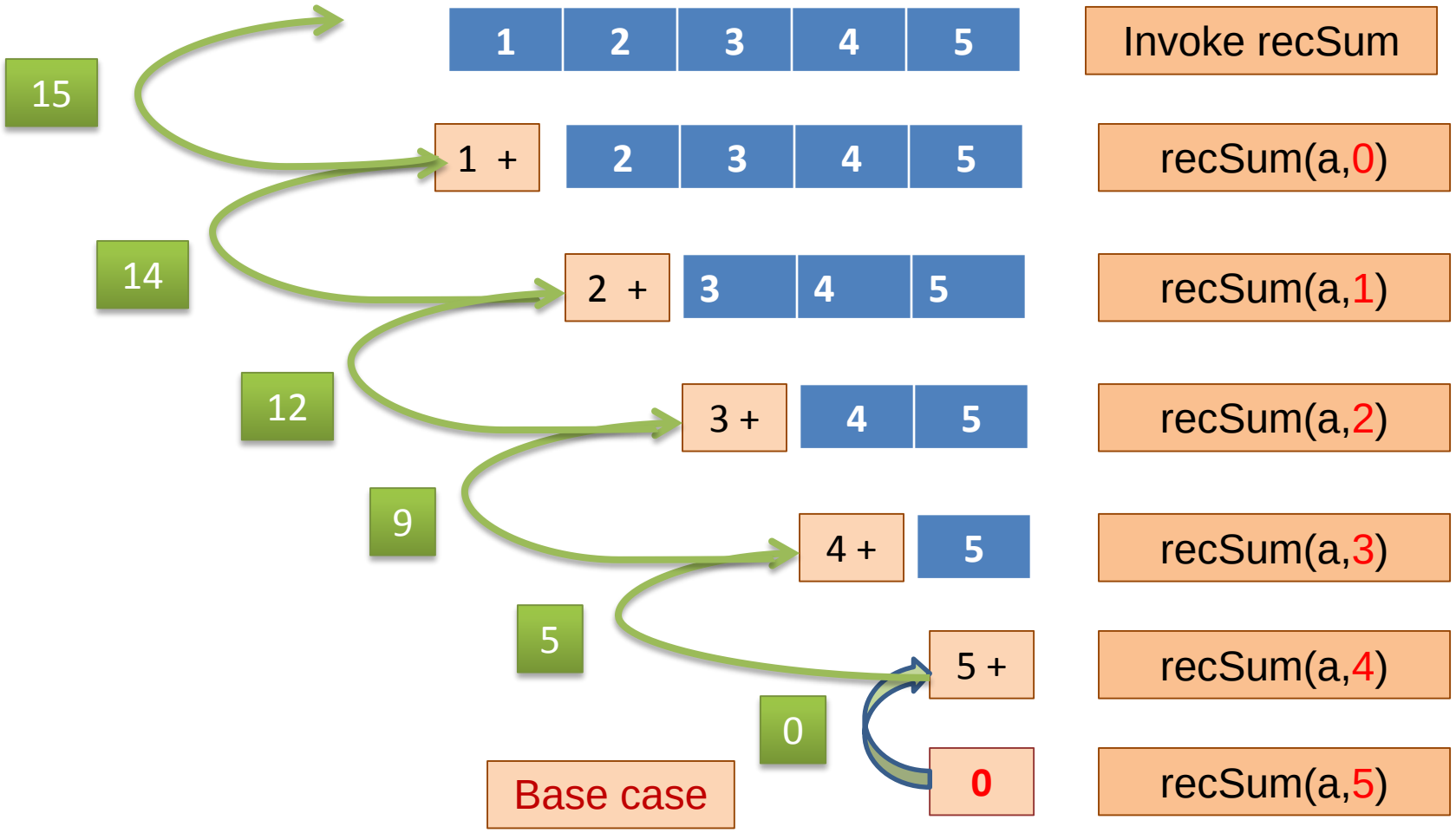
- This method calls itself with an index argument that is higher than the current index. The result of the method call is added to the array element at the index.

- Base case: If the index is equal to the length of the array, we know that all of the array elements have been processed.

```
public static int recSum(int[ ] array, int index)
{
        if (index == array.length)
                return 0;
        else
                return array[index] + recSum(array, index + 1);
} // recSum
```

Sum numbers in array - example

```
public static int recSum(int[ ] array, int index)
{
      if (index == array.length)
            return 0;
      else
          return array[index] + recSum(array, index + 1);
}
```



Find max value in an array

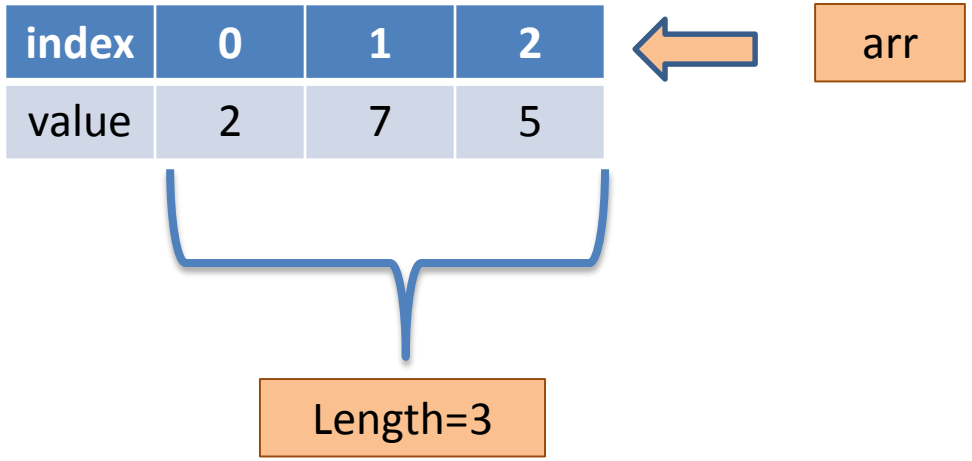
The method gets an array of int value and it's length.

It will return the **max value** in the array.

```
public static int findMax(int array[ ], int length)
{
	if (length == 1)
		return array[0]; // base case
	int result = findMax(array, length - 1);
	if (array[length - 1] > result)
			return array[length - 1];
	else
			return result;
} // findMax
```

Find max value in an array - example

```
public static int findMax(int array[ ], int length)
{
      if (length == 1)
          return array[0]; // base case
      int result = findMax(array, length - 1);
      if (array[length - 1] > result)
            return array[length - 1];
      else
            return result;
}
```




```
public static int findMax(array,3)
{
      if (3 == 1)   F
          return array[0]; // base case
      int result = findMax(array, 2);
      if (array[2] > result)
            return array[2];
      else
            return result;
}
```

7

```
public static int findMax(array,2)
{
      if (2 == 1)   F
          return array[0]; // base case
      int result = findMax(array, 1);
      if (array[1] > result)
            return array[1];
      else
            return result;
}
```

7

```
public static int findMax(array,1)
{
      if (1 == 1)   T
          return array[0]; // base case
      int result = findMax(array, 0);
      if (array[0] > result)
            return array[0];
      else
            return result;
}
```

2

Update student's grades

Next program reads student's grades and calculates their average grades.
If student's grade is less than the average grade, then program adds 5 points factor to this grade.
The program prints all entered grades before and after upgrading.

```
public static void main(String[ ] args)
{
      System.out.print( "Enter number of students : " );
      int num = reader.nextInt();
      int [ ] arrGrades = new int [num];
      inputGrades(arrGrades,0);   // input student's grades
      printGrades(arrGrades,0);   // print grades before update
      int avg = sumGrades(arrGrades,num)/num;   // calculate average
      updGrades(arrGrades,0,avg);   // update student's grades
      printGrades(arrGrades,0);   // print grades after update
}  // main
```

inputGrades recursive method

```
public static void inputGrades(int [ ] grades ,int n)
{
      int grade = 0;
      if(n <= grades.length - 1)
      {
            do
            {
                System.out.print("Enter the " + (n+1) + " student's grade : ");
                grade = reader.nextInt();
            }  while ( (grade < 0) || (grade > 100));
            inputGrades(grades,n+1);
            grades[n] = grade;
      } // if
} // inputGrades
```

printGrades recursive method

```
public static void printGrades(int[ ] d, int m)
{
    System.out.println("The"+(m+1)+"student's grade is " + d[m] );
    if(m < d.length - 1)
            printGrades(d,m+1);
} // printGrades
```

sumGrades recursive method

```
public static int sumGrades(int [ ] b, int k)
{
    if(k == 0)
            return 0;
    else
            return (b[k - 1] + sumGrades(b, k - 1));
} // sumGrades
```

What4 recursive method

```
public static int What4( int[ ] arr, int start, int end, int val )
{
    int len = end - start;
    if( len <= 0 )
            return 0;
    if( len == 1 )
            return arr[ start ] == val ? 1 : 0;
    int len2 = len / 2;
    return What4( arr, start, start + len2, val ) + What4( arr, start + len2, end, val );
} // What4

public static void  main(String[ ] args)
{
      int a[ ] = { 1,2,3,2,1 };
      System.out.println(What4(a,0,5,2));
} // main
```

This method returns… **?**

What5 recursive method

```
public static boolean What5(String s, int l, int r)
{
      if( l >= r)
            return true;
    else
            if(s.charAt(l) != s.charAt(r))
                                    return false;
           else
           {
                 l++;
                 r--;
                 return What5(s, l, r);
            } // else
}  // What2

public static void  main(String[] args)
{
       String str = "abzba";
       if(What5(str, 0, str.length() - 1 ))
                        System.out.println("YES");
      else
                        System.out.println("NO");
} // main
```

This program tests if **...?**

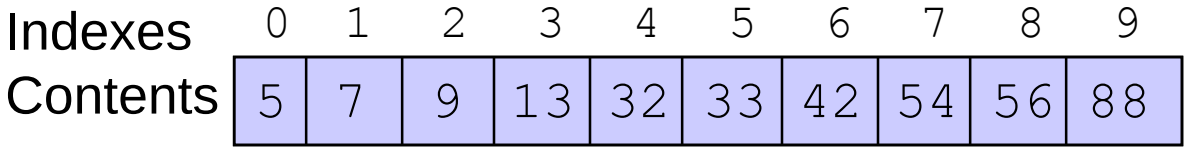
Recursive binary search

This method returns the index of the entry if the target value is found or -1 if it is not found.

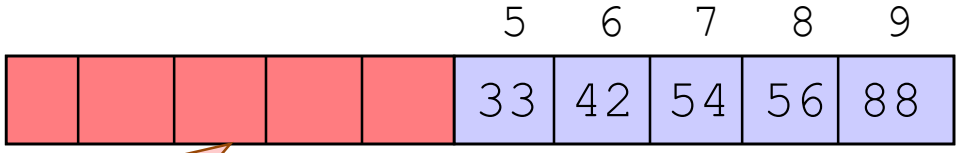
```
public static int  binSearch( int a[ ],int target, int first,int last )
{
    int location = -1; // target not found;
    if(first <= last)
    {
        int mid = (first + last)/2;
        if(target == a[mid])
                location = mid;
        else
                if(target < a[mid])  // first half
                        location = binSearch( a, target, first, mid - 1)
                else // second half
                        location = bianSearch(a, target, mid + 1, last )
    }
    return location;
}  // binSearch
```

Binary search example

target is 33
The array **a** looks like this:



mid = (0 + 9) / 2 (which is 4)
33 > a[mid] (that is, 33 > a[4])
So, if 33 is in the array, then 33 is one of:

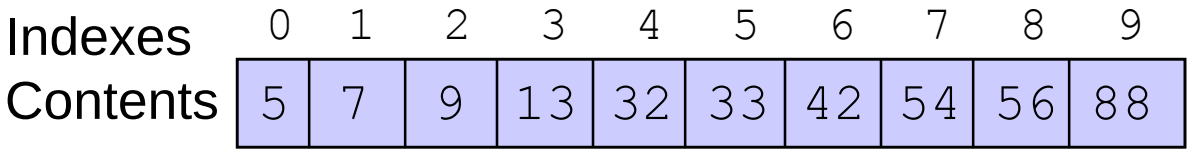


Eliminated half of the remaining elements from consideration because array elements are sorted.

Binary search example, cont.

target is 33

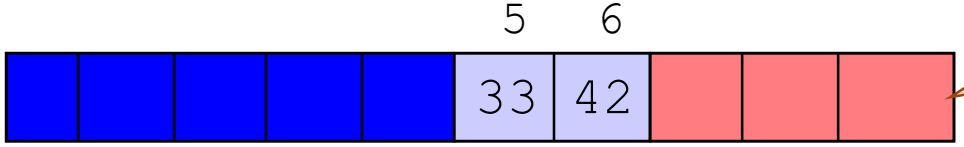
The array **a** looks like this:



mid = (5 + 9) / 2 (which is 7)

33 < a[mid] (that is, 33 < a[7])

So, if 33 is in the array, then 33 is one of:

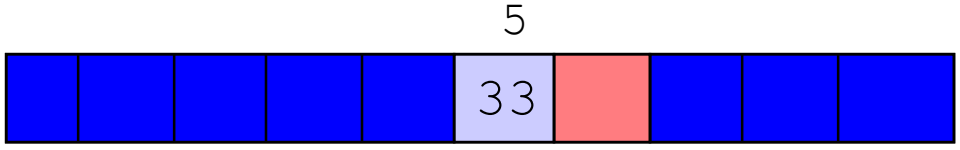


Eliminate half of the remaining elements

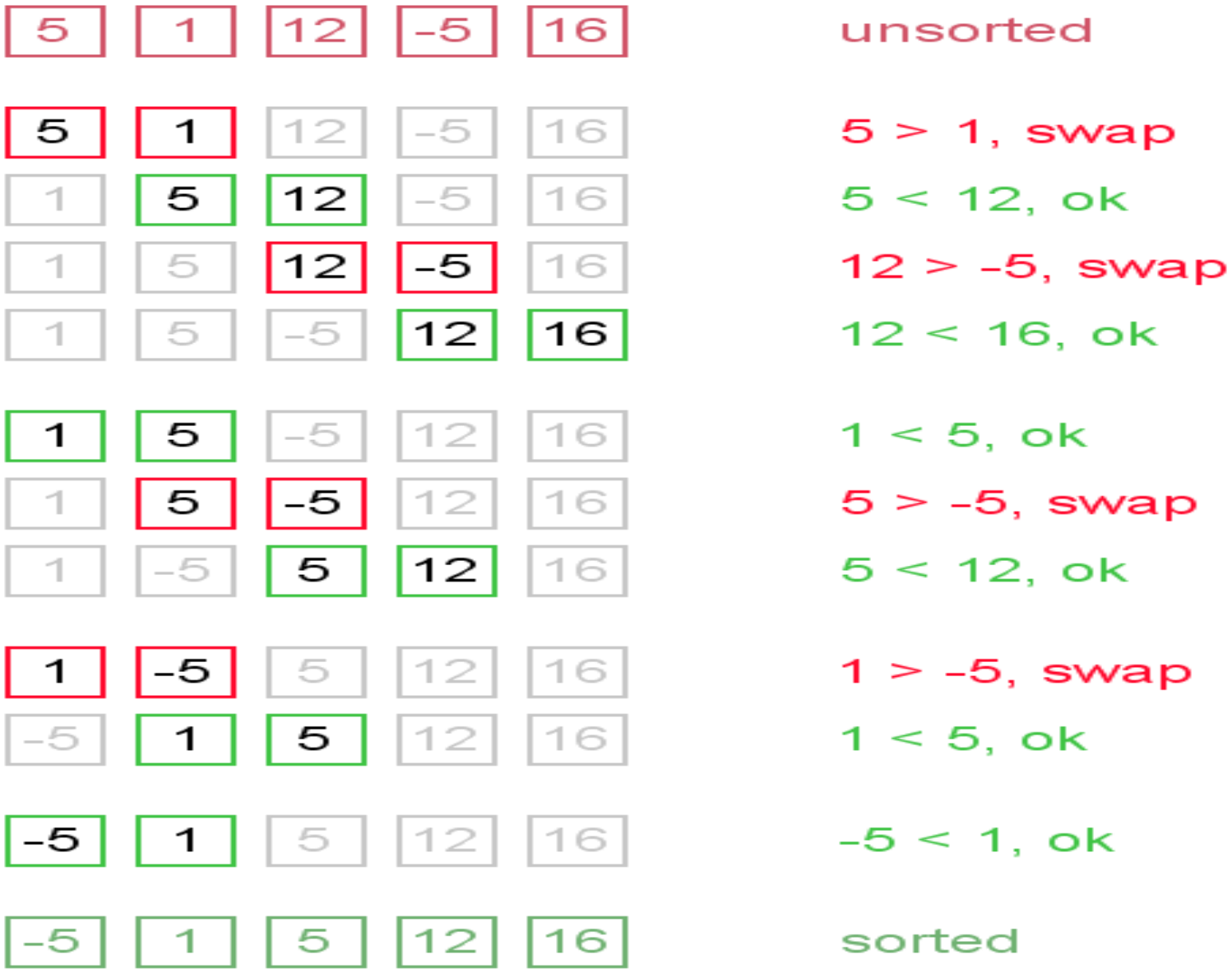
mid = (5 + 6) / 2 (which is 5)

33 == a[mid]

So we found 33 at index 5:



Bubble Sort - example



Bubble Sort - recursive solution

Algorithm :

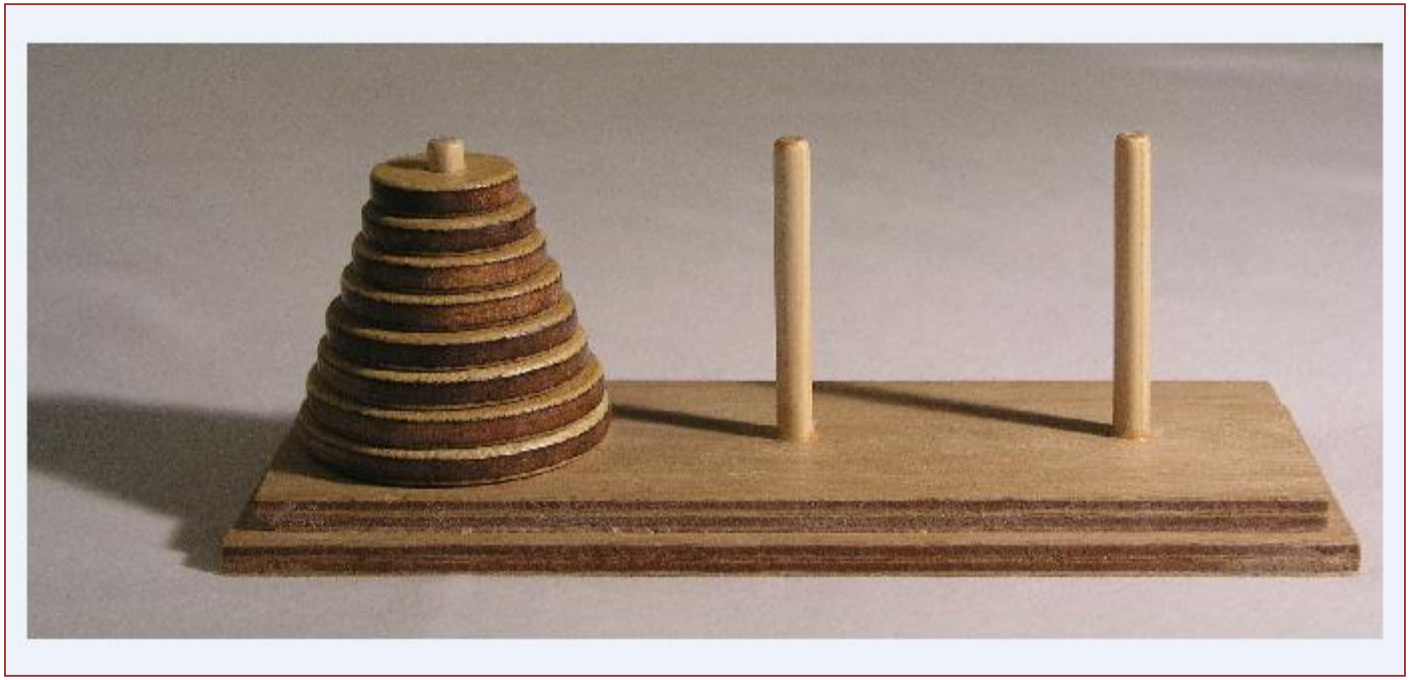
1. Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
2. If at least one swap has been done, repeat step 1.

```
public static void swap(int a[ ],int i)
{
    int t = a[ i ];
    a[ i ] = a[ i+1 ];
    a[ i+1 ] = t;
} // swap
```

Help method will swap the elements at i and i+1 indexes of array a.

```
public static int[ ] recBubSort(int arr[ ], int n)
{
        if(n < 2)
                return arr;
        for(int i = 0; i < n-1; i++)
        {
                if(arr[ i] > arr[i+1])
                        swap( arr, i); // help method
        } // for
        return recBubSort(arr,n-1);
} // recBubSort
```

Towers of Hanoi



The Towers of Hanoi is a mathematical game or puzzle.
It was invented by the French mathematician, Eduard Lucas, in 1883. The Tower of Hanoi puzzle appeared in 1883 under the name of M. Claus. Notice that Claus is an anagram of Lucas!

Towers of Hanoi - rules:

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- *Only one disk* may be moved at a time.
- *Each move* consists of taking the *upper disk* from one of the rods and sliding it onto another rod, on *top of the other disks* that may already be present on that rod.
- *No disk* may be placed on top of a *smaller disk.*

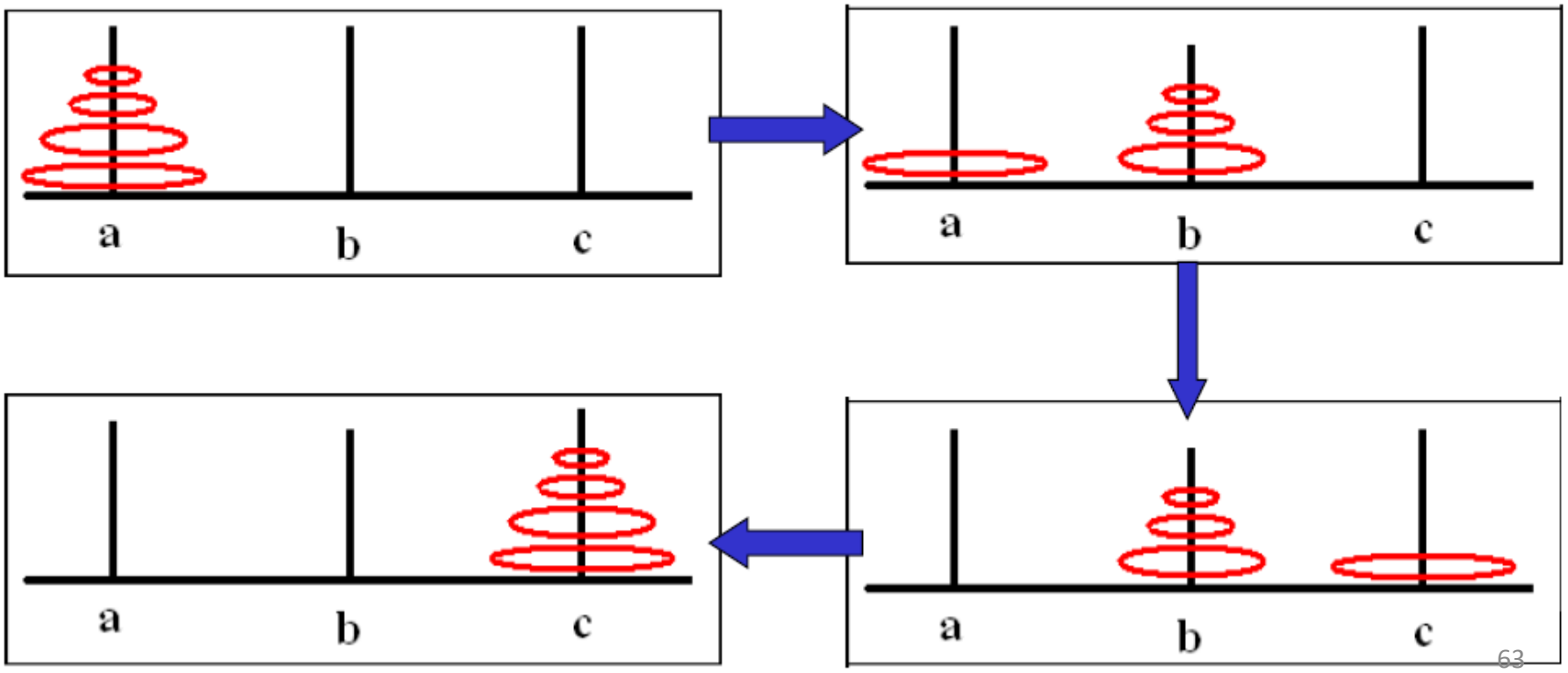
Recursive Algorithm

To move n disks from peg **a** to peg **c**:

move n−1 disks from **a** to **b**. This leaves disk #n alone on peg **a**

move disk #n from **a** to **c**

move n−1 disks from **b** to **c** so they sit on disk #n



Recursive Algorithm - pseudo code

חנוי(מספר_דיסקיות,מוט_מקור,מוט_יעד,מוט_עזר)

1. אם מספר_דיסקיות = **1**

   העבר את הדיסקית מוט_מקור למוט_יעד

2. אחרת

   2.1 חנוי(מספר_דיסקיות -**1**,מוט_מקור,מוט_עזר,מוט_יעד)

   2.2 חנוי(**1**,מוט_מקור,מוט_יעד,מוט_יעזר)

   2.3 חנוי(מספר_דיסקיות -**1** ,מוט_עזר,מוט_יעד,מוט_מקור)

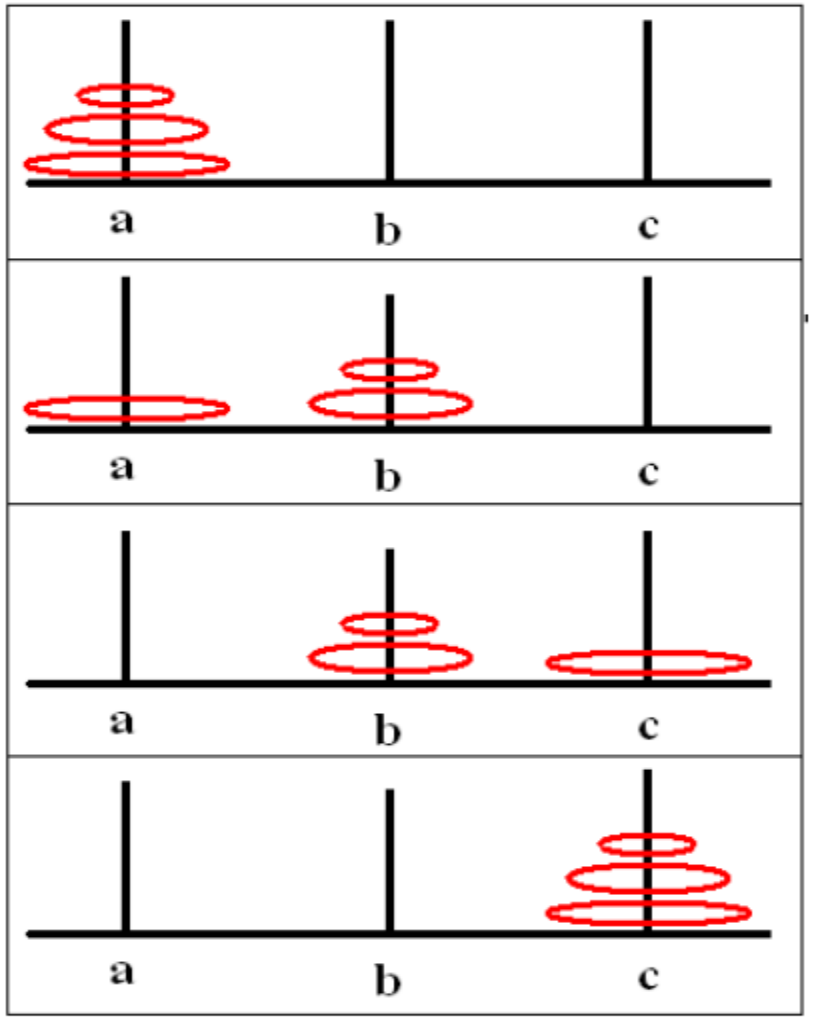
Towers of Hanoi - Java solution

```
public static void hanoi (int  x, char  from, char  to, char  aux )
{
    if (x == 1)
        System.out.println( "Move disk from “  + from + " to “ +  to);
    else
    {
        hanoi(x - 1, from, aux, to);
        System.out.println( "Move disk from “ + from + " to “ +  to);
        hanoi(x - 1,aux,to,from);
     }
} // hanoi
```

Towers of Hanoi num = 3

```
System.out.print( “Enter an integer  “ );
int  num = reader.nextInt();
hanoi( num, 'a' , 'c ' , 'b' );
```

Move disk from a to c
Move disk from a to b
Move disk from c to b
Move disk from a to c
Move disk from b to a
Move disk from b to c
Move disk from a to c



M. C. Escher & Recursion

Maurits Cornelis Escher (17 June 1898 – 27 March 1972) was a Dutch graphic artist.

He is known for his often mathematically inspired , lithographs.

"**Drawing Hands**" lithograph - illustrates the concept of Recursion.