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!
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
    \[ I \text{ will answer } N + 1. \]

  – If there is nobody behind me,
    \[ I \text{ will answer } 0. \]
How do you look up a name in the phone book?

**Search:**

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

Go to middle page

If \( \text{name is on middle page} \)

\[
\text{done; this is the base case}
\]

else

\[
\text{if (name is alphabetically before middle page)}
\]

last page = middle page re define search area to front half

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

else name must be after middle page

first page = middle page re define 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 (הרכבת)

3. What is the smallest problem which can be solved easily (without further decomposition)?
   - Base/stopping case (מקרה בסיסי)
A **base case** does not execute a recursive call!
Factorial \((N!\)\)

- \(N! = (N-1)! \times N\) \(\quad\text{// for } N > 1\)
- \(1! = 1\)
- \(3! = \)
  \[
  = 2! \times 3 = \\
  = (1! \times 2) \times 3 \\
  = 1 \times 2 \times 3
  \]
- Recursive design:
  - Decomposition: \((N-1)!\)
  - Composition: \(* N\)
  - Base case: \(1!\)

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

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

public static int factorial(int 1)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return fact;
}
public static int factorial(int n) {
    int fact;
    if (n > 1) {
        fact = factorial(n - 1) * n;
    } else {
        fact = 1;
    }
    return fact;
}
public static int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return factorial(n - 1) * n;
    }
} // factorial
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 Last In First Out (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.

```
How recursion works

Factorial(1)
n = 1  fact = 1
Factorial(2)
n = 2
Factorial(3)
n = 3
res = factorial(3)

Factorial(2)
n = 2  fact = 2
Factorial(2)
n = 2
Factorial(3)
n = 3  fact = 6
Factorial(3)
n = 3
Factorial(3)
n = 3
res = factorial(3)
res = factorial(3)
res = factorial(3)
res = factorial(3)
```

Stack memory (LIFO)
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

Calculation arithmetic series (sigma) recursive Sum:
If we call the `sigma` method with the Java statement:

```java
int sum = sigma(5);
```

the `else` portion of the first method’s calls the method `sigma` again

```java
return 5 + sigma(4);
```

```latex
\begin{align*}
\text{push} & \quad \text{at the end of the recursive calls the steps are reversed for} \\
& \quad \text{assigning the values.} \\
\text{pop} & \quad \text{return } 1 \\
\text{push} & \quad \text{return } 2 + 1 \quad (=3) \\
\text{push} & \quad \text{return } 3 + 3 \quad (=6) \\
\text{push} & \quad \text{return } 4 + 6 \quad (=10) \\
\text{push} & \quad \text{return } 5 + 10 \quad (=15)
\end{align*}
```
**Calculation**    power

\[ X^y = x \times x \times \ldots \times x \]

Recursive definitions (assume non-negative \( y \)):

\[ X^y = X \times (X^{y-1}) \]

Base case: \( x^0 = 1 \)

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

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

The product \( a \times b \) is actually \( a + a + \ldots + a \) (\( b \) times).

\[
x \times y = x + x + x + x + \ldots + x
\]
y times

Recursive definitions:
\[
x \times y = y \times x = y + (x - 1) \times y
\]

Base case: \( 0 \times Y = 0 \)

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

public static void main(String[ ] args)
{
    int x = reader.nextInt(); // 5
    int y = reader.nextInt(); // 2
    System.out.println(add(x,y));
} // main
public static int sub(int a, int b)
{
    if(b == 0) // Base case
        return a;
    else
        if(b < 0)
            return sub(a + 1, b + 1);
        else
            return sub(a - 1, b - 1);
} // sub

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

public static void main(String[ ] args) {
    int a = reader.nextInt(); //5
    int b = reader.nextInt(); //2
    System.out.println(divide(a, b));
} // main
This recursive method calculates the \textbf{n} member of arithmetic sequence, beginning from \textbf{start} value with \textbf{d} value sequences difference.

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

<table>
<thead>
<tr>
<th>start</th>
<th>d</th>
<th>n</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>100</td>
</tr>
</tbody>
</table>
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
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

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

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
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 ...?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>831</td>
<td>0</td>
</tr>
<tr>
<td>235</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>5101</td>
<td>101</td>
<td>3</td>
</tr>
</tbody>
</table>
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

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:
  
  $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

- Recursive Design:
  - Decomposition & Composition
    - $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$
  - Base case:
    - $\text{fibonacci}(0) = 0$
    - $\text{fibonacci}(1) = 1$

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

**Binary Recursion** is a process where function is called twice at a time instead of once at a time.
Fibonacci series $n = 5$

Binary recursion (רקורסיה כפולה)
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

// main

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

<table>
<thead>
<tr>
<th>input(x)</th>
<th>output(what)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
</tbody>
</table>
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);
}

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

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

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

First output : *

Second output : **

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

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

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

```java
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

Reverse an array

The method gets an array and its length as parameters and returns the array after reversing its values.

```java
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
```

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

Method returns the array after reversing its values:

```
5 4 3 2 1
```
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
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 \(n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1\), which is commonly denoted as "\(n\) factorial" and written "\(n!\)".
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 abc and acb.

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

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.
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
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);
        }
    }
}

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

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
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.

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

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

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.

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

Invoke recSum
recSum(a,0)
recSum(a,1)
recSum(a,2)
recSum(a,3)
recSum(a,4)
recSum(a,5)

Base case
The method gets an array of int value and it's length. It will return the **max value** in the array.

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

index | 0 | 1 | 2 |
value | 2 | 7 | 5 |
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.

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

```java
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 = binSearch(a, target, mid + 1, last );
    }
    return location;
} // binSearch
```
target is **33**
The array \( a \) looks like this:

<table>
<thead>
<tr>
<th>Indexes</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>13</td>
<td>32</td>
<td>33</td>
<td>42</td>
<td>54</td>
<td>56</td>
<td>88</td>
</tr>
</tbody>
</table>

\[
\text{mid} = (0 + 9) / 2 \text{ (which is 4)}
\]

33 > a[mid] (that is, 33 > a[4])
So, if 33 is in the array, then 33 is one of:

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>42</td>
<td>54</td>
<td>56</td>
<td>88</td>
</tr>
</tbody>
</table>

Eliminated half of the remaining elements from consideration because array elements are sorted.
target is 33

The array `a` looks like this:

<table>
<thead>
<tr>
<th>Indexes</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>13</td>
<td>32</td>
<td>33</td>
<td>42</td>
<td>54</td>
<td>56</td>
<td>88</td>
</tr>
</tbody>
</table>

```
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:
```

```
5
6
```

Eliminate half of the remaining elements

```
mid = (5 + 6) / 2 (which is 5)
33 == a[mid]
So we found 33 at index 5:
```

```
5
```
Bubble Sort - example

5 | 1 | 12 | -5 | 16
---|---|----|----|----
5 | 1 | 12 | -5 | 16  
 1 | 5 | 12 | -5 | 16  
 1 | 5 | 12 | -5 | 16  
 1 | 5 | 12 | -5 | 16  
 1 | 5 | -5 | 12 | 16  
 1 | 5 | -5 | 12 | 16  
 1 | 5 | -5 | 12 | 16  
 1 | 5 | -5 | 12 | 16  
 1 | -5 | 5 | 12 | 16  
 1 | -5 | 5 | 12 | 16  
 1 | -5 | 5 | 12 | 16  
 1 | -5 | 1 | 5 | 12  
 -5 | 1 | 5 | 12 | 16  
 -5 | 1 | 5 | 12 | 16  
 -5 | 1 | 5 | 12 | 16  
 -5 | 1 | 5 | 12 | 16  

unsorted

5 > 1, swap
5 < 12, ok
12 > -5, swap
12 < 16, ok
1 < 5, ok
5 > -5, swap
5 < 12, ok
1 > -5, swap
1 < 5, ok
-5 < 1, ok

sorted
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.

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

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
```

Help method will swap the elements at i and i+1 indexes of array a.
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!
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**.
To move \( n \) disks from peg \( a \) to peg \( c \):

1. move \( n-1 \) disks from \( a \) to \( b \). This leaves disk \( n \) alone on peg \( a \)
2. move disk \( n \) from \( a \) to \( c \)
3. move \( n-1 \) disks from \( b \) to \( c \) so they sit on disk \( n \)
Recursive Algorithm - pseudo code

1. If the number of disks is 1, move disk from origin to destination.

2. Otherwise:
   2.1 Move the disks except the last one from origin to auxiliary
   2.2 Move the last disk from origin to destination
   2.3 Move the disks except the last one from auxiliary to destination
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
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
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.