# Chapter #1: what is the Algorithm?

## I. Introduction

In this unit of work you will learn what is the algorithm, algorithm notations, the developing strategies of algorithm, and analysis of algorithm.

Algorithm is defined as a solution for a problem that comes in well defined collection of instructions (steps, conditions, and repetitions) in which this solution uses given parameters to produce the correct result (achieve defined goal).The study of algorithms is at the heart of computer science.

In our live, we experience algorithms in everything we do. Algorithm, actually, is an answer to "how to make" question. For example, if we like to answer the question "how do you make a cake?" the answer comes as a recipe of making cake which is algorithm.

As a programmer, if you have been asked: "how do you make software?" you will be trying to do four out of five things at once. These are:

1. **Problem definition**
2. **Analyse the problem**
3. **Design a solution/program**
4. **Code/enter the program**
5. **Test the program**
6. **Evaluate the solution**

It is impossible to do six things correctly at the same time, We will therefore concentrate on each step, one at a time; - it is an algorithm for making software !!!

Program design is the most important part in producing a computer program. This is the stage where you decide how your program will work to meet the decisions you made during analysis. Program design does not require the use of a computer - you will carry out your design using pencil and paper. In your design, you will use a method called top down design. By using this method, you simply write down the steps in the correct order, that you are required to perform, to solve the problem. When all these steps are put together you have what is called an Algorithm, which is a step-by-step set of instructions for solving a problem in a limited number of steps. The instructions for each step are exact and precise and can be carried out by a computer.

## II. Algorithm Notations:Methodsfor Representing Algorithms

1. Pseudocode: English language statements with some defined rules of structure and some keywords that make it appear a bit like program code.
2. Flowchart: is diagrammatic method of representing algorithms. They use an intuitive scheme of showing operations in boxes connected by lines and arrows that graphically show the flow of control in an algorithm. Flowcharts are made up of different box types connected by lines with arrowheads indicating the flow. It is common practice only to show arrowheads where the flow is counter to that stated above.

3. Programming languages are essentially a way of expressing algorithms.

## III. Tracing Algorithm

An algorithm trace is a method for hand simulating the execution of your code in order to manually verify that it works correctly before you compile it.

## IV. Analyzing of Algorithm

The analysis of algorithms is the determination of the amount of resources (such as time and storage) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length. Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. ***Big O notation, Big-omega notation and Big-theta notation*** are used to this end. For instance, binary search is said to run in a number of steps proportional to the logarithm of the length of the list being searched, or in

O(log(n)), colloquially "in logarithmic time". Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency. However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called a hidden constant.

Exact (not asymptotic) measures of efficiency can sometimes be computed but they usually require certain assumptions concerning the particular implementation of the algorithm, called model of computation. A model of computation may be defined in terms of an abstract computer, e.g., Turing machine, and/or by postulating that certain operations are executed in unit time. For example, if the sorted list to which we apply binary search has n elements, and we can guarantee that each lookup of an element in the list can be done in unit time, then at most log2 n + 1 time units are needed to return an answer.

## V. Strategies of Developing Algorithm

Three methods used in creating an algorithm;which are:

### a. STEPPING OUT

First of all we will look at a method of creating an algorithm calledSTEPPING. This is where all the instructions needed to solve our problem are set out one after the other.

***EXAMPLES:***

**Problem:** To find the sum of two numbers.

**Algorithm:** 1 add the two numbers together

2 write down the answer.

**Problem:**To change the temperature in Fahrenheit to Centigrade.

**Algorithm:** 1 subtract 32

2 multiply the result in (1) by 5

3 divide the result in (2) by 9

4 write down the answer.

Both of these examples list the algorithm in an ordered set of steps. Now it's your turn to try a few.

**Problem:** Find the volume of a cone given its diameter and height. The formula is:

Volume =( 3.14* radius$^2$* height)/3

**Algorithm:** 1 divides the diameter by 2 to give the radius

2 multiply 3.14 by the radius

3 multiply the result in (2) by the radius

4 multiply the result in (3) by the height

5 divide the result in (4) by 3 to give the volume

6 write down the answer.

***EXERCISE*:** write down an algorithm for each of the following problems:-

1) Find the average of four numbers.
2) Change the time in seconds to minutes.
3) Find the product of two numbers (this means to multiply the two numbers).
4) Find the difference between two numbers
5) Change a volume in pints to litres (there are 2.2 pints in every litre).
6) Find the average speed of a car given the journey time and the distance travelled.
7) Find the volume of a cube given the length of a side.
8) Find the volume of a pyramid, given the length and width of its base and its height.

## b. CHOOSING [IFs, THENs AND ELSEs]

So far we have come across the method of stepping commands. Now we will look at a method for making a choice or a decision. This technique is called CHOOSING, and uses the commands **IF**, **THEN** and sometimes (but not always) **ELSE .**

With this type of command a condition is given with the **IF** command followed by an action to be taken. If the condition is satisfied, we follow it by the **THEN** command. Here are a couple of examples which should make things a little clearer: -

***EXAMPLES:***

**Problem:** To decide if a fire alarm should be sounded

**Algorithm:** 1 **IF***fire is detected* condition

2 **THEN***sound fire alarm* action

**Problem:** To decide whether or not to go to university

**Algorithm:** 1 **IF***it is a weekday* **AND** *it is not a holiday*

2 **THEN***go to university*

3 **ELSE***stay at home*

In this example a connectivity operation, **AND**, was used. This connectivity operation is used to allow for two or more conditions that have to be satisfied. You will notice that we have also used the **ELSE** command in this example where we are stating an action that must be followed if the conditions are not met.

***EXERCISE:***

In your jotters write down an algorithm for each of the following problems (All of these problems can be solved by using the choosing method):-

1) To decide whether or not to wash your hands.
2) To decide whether or not it is time to make lunch.
3) To print the largest of two given numbers.
4) To decide if a number is between 10 and 15, if it is print the number.

## c. GOING LOOPY! [LOOPING]

So far, all of the problems we have solved used an algorithm with the set of instructions in a set order that we refer to as stepping. What if you have to repeat an instruction or a set of instructions a number of times to find your solution? In this case you can use loops. We will look at three different types of loops:-

- **REPEAT UNTIL LOOP**

  This type of loop keeps on carrying out a command or commands UNTIL a given condition is satisfied, the condition is given with the UNTIL command, for example:-

  **Problem:** To wash a car

  **Algorithm:**

  1 **REPEAT**

  2 wash with warm soapy water

  3 **UNTIL** the whole car is clean

- **WHILE LOOP**

  In this type of loop the condition is given along with the WHILE command and the loop keeps on carrying out a command or commands until an END WHILE command is given, for example:-

  **Problem**: To ask for a number more than 10, then stop when such a number is given

  **Algorithm:** 1 **WHILE** number given is less than 10

  2 ask for a number more than 10

  3 **END WHILE**

***EXERCISE:***

In your jotters write down an algorithm for each of the following problems (All of these problems can be solved by using WHILE loops):-

1) To keep asking for a number less than 5 and stop when a number less than 5 is given.
2) To keep asking for a password, and to give the message "accepted" when the correct password is given.
3) To ask for the length of one side of a square, then keep asking for guesses for the area of the square until the correct area is given (think about stepping and looping).

- **FOR LOOP**

A FOR loop keeps on carrying out a command or commands, **FOR** a given number of times. In this type of loop the number of times it is repeated must be known. The commands to be repeated are sandwiched between the FOR and END FOR commands, for example:-

**Problem:** To print the numbers from 10 to 20

**Algorithm:**
1. **FOR** number = 10 to 20
2. Print number
3. END FOR

**NOTE:** In this loop, the value of **number** starts at 10 and goes up to 20. Its value increases by one each time it goes through the loop until it reaches 21. Since

the value of **number** changes or varies, **number** is called a *variable*. A variable can be given any name provided the name remains the same throughout your program. We will use the word variable many times in this work. **VARIABLE**: Aquantityorthing, which throughoutacalculationwillchangein value. Its name should describe what it is representing e.g. Number orliter or weight etc.

**Problem:** To print the 13 times table from 1 x 13 to 10 x 13

**Algorithm:**
1. FOR z = 1 to 10
2. Print z x 13
3. END FOR

In this loop our variable is **z**, whose value starts at one and then increases in value by 1 each time through the loop until **z** has the value of 10. Line (2) of this algorithm prints the value of the variable **z** multiplied by 13.

## d. ALL TOGETHER NOW

When you are programming you will find very few problems that can be solved using just stepping, looping or choosing. In fact most problems will probably need a combination of all three techniques. These examples combine STEPPING, LOOPING and CHOOSING to find a solution to the problems.

**Problem:** To find the biggest number in a large set of numbers.

**Algorithm:**

1. largest = 0
2. **WHILE** still numbers to check
3. Read number
4. **IF** number is bigger than largest
5. **THEN** largest = number
6. **ENDWHILE**
7. Print largest

**Problem:** To count the number of red boxes in a set of red, white and blue boxes

**Algorithm:**

1. number = 0
2. **WHILE** still boxes to check
3. Read color
4. **IF**color is red
5. **THEN** number = number +1
6. **ENDWHILE**
7. **Print** number

Now it's your turn to try out all this new found knowledge in the writing of **ALGORITHMS** using **STEPPING**, **LOOPING** with **VARIABLES** and **CHOOSING**

## VI. Top Down Design

If you are asked to find a solution to a major problem, it can sometimes be very difficult to deal with the complete problem all at the same time. For example building a car is a major problem and no-one knows how to make every single part of a car. A number of different people are involved in building a car, each responsible for their own bit of the car's manufacture.The problem of making the car is thus broken down into smaller manageable tasks. Each task can then be further broken down until we are left with a number of step-by-step sets of instructions in a limited number of steps. The instructions for each step are exact and precise.

Top Down Design uses the same method to break a programming problem down into manageable steps. First of all we break the problem down into smaller steps and then produce a Top Down design for each step. In this way sub-problems are produced which can be refined into manageable steps. An example of this is:-

**Problem:** Ask for the length, breadth and height of a room. Then calculate and display the volume of the room.

**Algorithm:**

1 ask for the dimensions

2 calculate the volume

3 display the volume

**Refinement:**

**Step 1:** 1. Ask for the dimensions

1.1 get the length

1.2 get the breadth

1.3 get the height

**Step 2:** 2. Calculate the volume

2.1 volume = length x breadth x height

## VII. EXERCISE:

1) To print the 9 times table from 1 x 9 to 12 x 9.
2) To print a message of your choice 4 times.
3) To print a table giving the price of petrol from 1 litre to 25 litres, given that the price of petrol is 69.5p per litre.
4) To print the values of a number squared from 2 to 20.
5) To print a table showing the area of circles with radii from 10cm to 20cm (ask if you are not sure how to find the area of a circle).
6) To print the smallest number in a large set of numbers
7) To print the average of a large set of numbers.
8) To check a list of job applicants and reject all those who either smoke or drink.
9) To ask for a number, less than 50, and then keep adding 1 to it until 50 is reached.
10) To ask for the length of all the walls in a room, ask for the height of the room, calculate and then display the total area of the room.
11) To calculate an employee's wages using these conditions: Ask for the employee's

name and how many hours they worked. Up to 40 hours are paid at £4.00 per hour, all hours over 40 hours are paid at "time and a half" Display the employee's name, their basic rate pay, their overtime pay and their total pay.

## VIII. Recursion

Repetitive algorithm is a process wherby a sequence of operations is executed repeatedly until certain condition is achieved. Repetition can be implemented using loop: while, for or do..while. Besides repetition using loop, many programming languages allow programmers to implement recursive methods.

Recursive is a repetitive process in which an algorithm calls itself. Recursion can be used to replace loops. Recursively defined data structures, like lists, are very well-suited to processing by recursive procedures and functions (methods). A recursive procedure is mathematically more elegant than one using loops. Sometimes procedures that would be tricky to write using a loop are straightforward using recursion.

Note that not all problems can be solved using recursive. Problem that can be solved using recursive is a problem that can be solved by breaking the problem into smaller instances of problem, solve & combine

Drawback of recursive is that the execution running time for recursive function is not efficient compared to loop, since every time a recursive function calls itself, it requires multiple memory to store the internal address of the function.

*Steps to Design a Recursive Algorithm:*

There must be at least one case (the base case), for a small value of n, that can be solved directly. A problem of a given size n can be split into one or more smaller versions of the

same problem (recursive case). Recognize the base case and provide a solution to it.

Devise a strategy to split the problem into smaller versions of itself while making progress toward the base case. Combine the solutions of the smaller problems in such a way as to solve the larger problem.

```
if (terminal case is reached)    // base case
<solve the problem>
else                             // general case
< reduce the size of the problem and
  call recursive function >
```

## IX. Examples

1. Multiply 2 numbers using Addition Method: Multiplication of 2 numbers can be achieved by using addition method. For example, to multiply 8 x 3, the result can also be achieved by adding value 8, 3 times as follows: 8 + 8 + 8 = 24

   Implementation of Multiply() using loop

   ```
   int Multiply(int M,int N)
   {
   int result=0;
   for (int i=1;i<=N;i++);
   result += M;
   return result;
   }//end of method Multiply
   ```

   Steps to solve Multiply() problem recursively:

   Problem size is represented by variable N. In this example, problem size is 3.

Recursive function will call Multiply() repeatedly by reducing N by 1 for each respective call.

Terminal case is achieved when the value of N is 1 and recursive call will stop. At this moment, the solution for the terminal case will be computed and the result is returned to the called function.

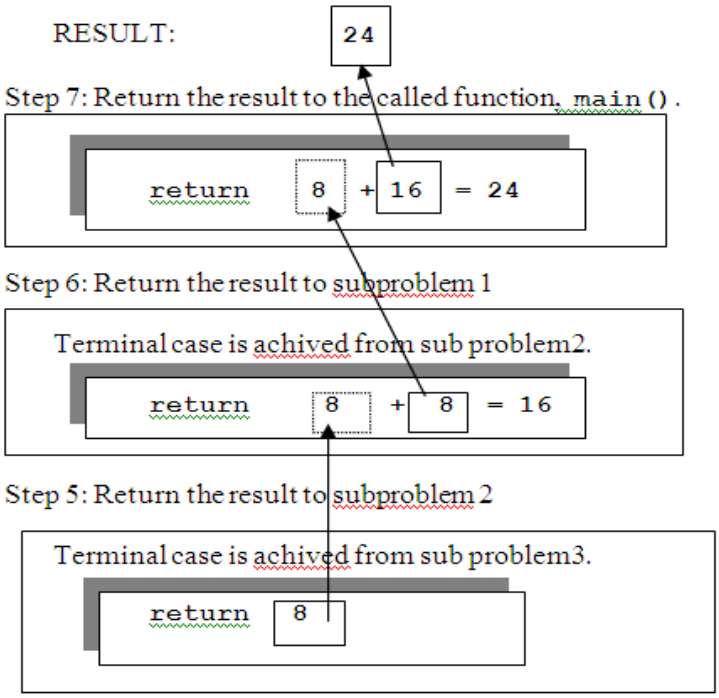
The simple solution for this example is represented by variable M. In this example, the value of M is 8.

Implementation of recursive function: Multiply()

```
int Multiply (int M,int N)
{
 int result;
 if (N==1)
   result=M;
 else
   result= M + Multiply(M,N-1);
 return result;
}//end Multiply()
```

Returning the Multiply() result to the called function

Step 8: Final result after multiply 2 numbers.



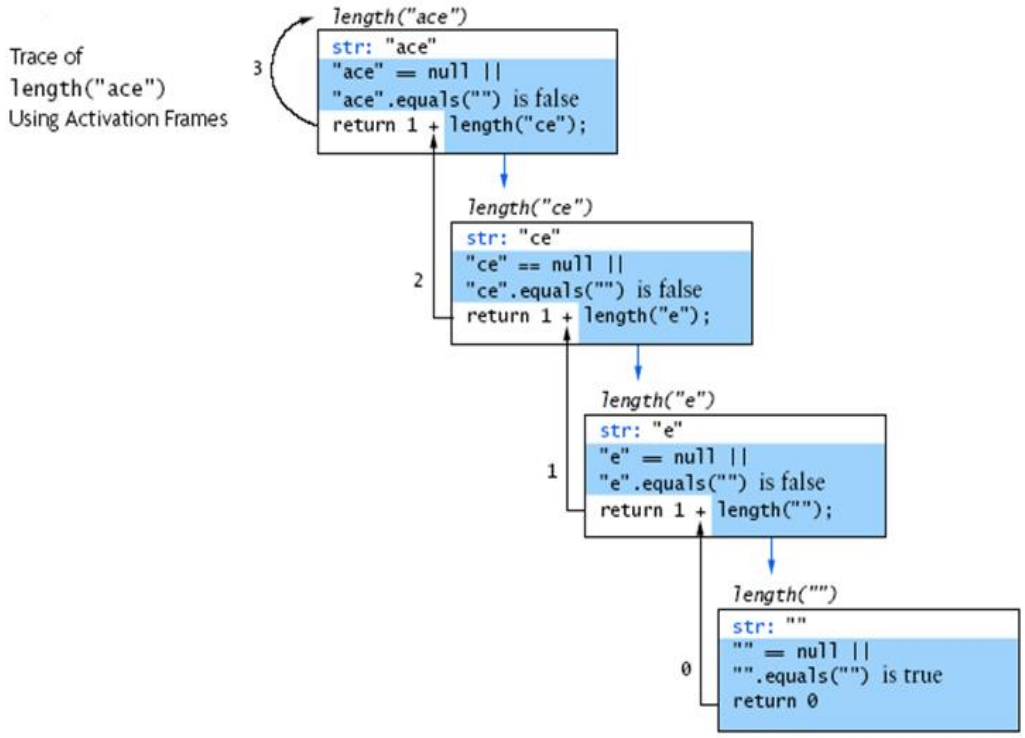
2: String Length Algorithm (Example)

**Recursive Algorithm for Finding the Length of a String**

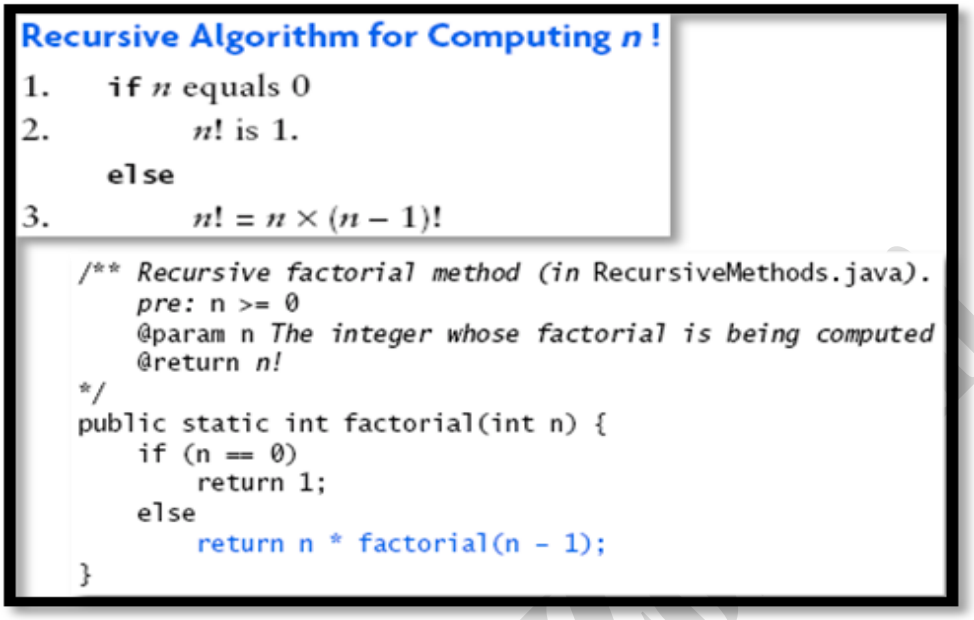
1. `if` the string is empty (has no characters)
2. The length is 0.

`else`

3. The length is 1 plus the length of the string that excludes the first character.



3: Factorial Method




Suppose we call the method to solve fact(4). This will result in four calls of method fact.

Step by step

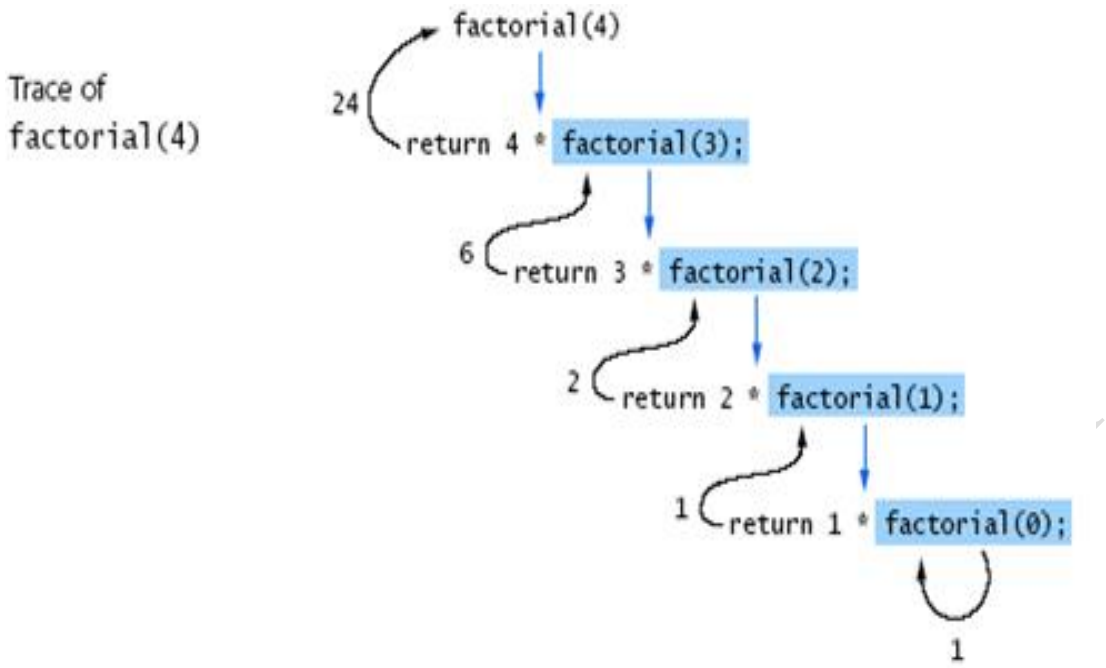
fact(4): This is not the base case (n==1) so we return the result of 4 * fact(3). This multiplication will not be carried out until an answer is found for fact(3). This leads to the second call of fact to solve fact(3).

fact(3): Again, this is not the base case and we return 3 * fact (2).

This leads to another recursive call to solve fact(2).

fact(2): Still, this is not the base case, we solve 2 * fact(1).

fact(1): Finally we reach the base case, which returns the value 1.




4. Fibonacci Problem

Problem : Get Fibonacci series for an integer positive.

Fibonacci Siries : 0, 1, 1, 2, 3, 5, 8, 13, 21,…..

Starting from 0 and have features that every Fibonacci series is the result of adding 2 previous Fibonacci numbers.

Solution: Fibonacci value of a number can be computed as follows:

Fibonacci ( 0) = 0

Fibonacci ( 1) = 1

Fibonacci ( 2) = 1

Fibonacci ( 3) = 2

Fibonacci ( N) = Fibonacci (N-1) + Fibonacci (N-2)

Solving Fibonacci Recursively

The simple solution for this example is represented by the Fibonacci value equal to 1.

N represents the series in the Fibonacci number. The recursive process will integrate the call of two Fibonacci () function. Terminal case for Fibonacci problem is when N equal to 0 or N equal to 1. The computed result is returned to the called function.

Fibonacci() function

```
int Fibonacci (int N )
{ /* start Fibonacci*/
if (N<=0)
return 0;
else if (N==1)
return 1;
else
return Fibonacci(N-1) + Fibonacci (N-2);
}
```

Implementation of Fibonacci() : passing and returning value from function.

Infinite Recursive

There's a condition where the function will stop calling itself. (if this condition is not fulfilled, infinite loop will occur). Each recursive function call, must return to the called function. Variable used as condition to stop the recursive call must change towards terminal case. Avoiding infinite recursion to avoid infinite recursion: must have at least 1 base case (to terminate the recursive sequence) each recursive call must get closer to a base case

Infinite Recursive : Example

```
#include <stdio.h>
#include <conio.h>
void printIntegers(int n);
main()
{ int number;
cout<<"\nEnter an integer value :";
cin >> number;
printIntegers(number);
}
void printIntegers (int nom)
{    if (nom >= 1)
cout << "\Value : " << nom;
printIntegers (nom-2);
}
```

4: Recursive Algorithm for Calculating $x^n$

**Recursive Algorithm for Calculating $x^n$ ($n \geq 0$)**

1. if $n$ is 0
2. The result is 1.

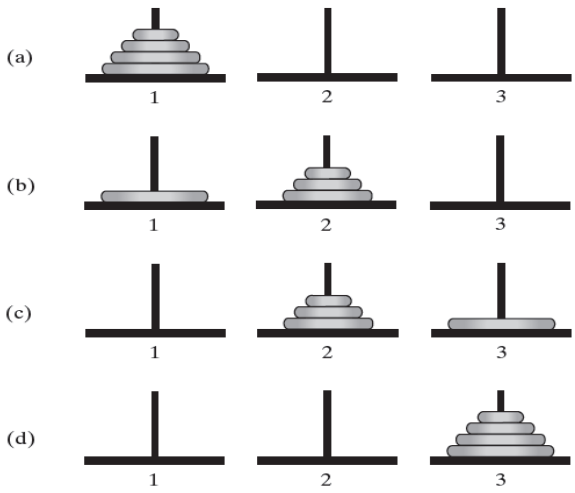
else

3. The result is $x \times x^{n-1}$.

We show the method next.

```
/** Recursive power method (in RecursiveMethods.java).
    pre: n >= 0
    @param x The number being raised to a power
    @param n The exponent
    @return x raised to the power n
*/
public static double power(double x, int n) {
    if (n == 0)
        return 1;
    else
        return x * power(x, n - 1);
}
```


5: Towers of Hanoi




Rules:

Move one disk at a time. Each disk you move must be a topmost disk.

No disk may rest on top of a disk smaller than itself.

You can store disks on the second pole temporarily, as long as you observe the previous two rules.

To solve for n disks …

Ask friend to solve for n – 1 disks

He in turn asks another friend to solve for n – 2

Etc.

Each one lets previous friend know when their simpler task is finished

The smaller problems in a recursive solution for four disks

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
// Version 2
if (numberOfDisks > 0)
{
   solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
   Move disk from startPole to endPole
   solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```

6: Mathematical series examples

(a) Consider the following table:

| identity(num) | returned value |
|---|---|
| identity (0) | 10 |
| identity (2) | 12 |
| identity (4) | 16 |
| identity (6) | 22 |
| identity (8) | 30 |
| identity (10) | 40 |

```
intidentity(intnum)
{
   if (num < 1)
      return 10;
   else
      return
      num + identity(num - 2);
}
```

(b) Consider the following relation table:

| negative(num) | returned value |
|---|---|
| negative(21) | -5 |
| negative(17) | 29 |
| negative(13) | 55 |
| negative(9) | 73 |
| negative(5) | 83 |
| negative(1) | 85 |
| negative(-3) | 79 |

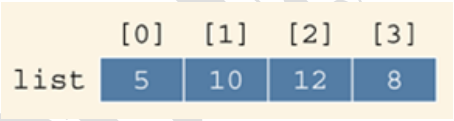
```
intnegative(intnum)
{
if (num >= 20)
return -5;
else
return negative(num + 4) + 2 *
num;
}
```

(c) Consider the following relation table:

| product(num) | returned value |
|---|---|
| product(64) | -1 |
| product(-32) | 32 |
| product(16) | 512 |
| product(-8) | -4096 |
| product(4) | -16384 |
| product(-2) | 32768 |
| product(1) | 32768 |

```
intproduct(intnum)
{
if (num > 20)
return -1;
else
return num * product(-2 * num);
}
```

7: example of finding the largest element of an array




To find the largest element in list[a]...list[b], First find the largest element in list[a+1]...list[b] , Then compare this largest element with list[a].

list[a]...list[b] stands for the array elements list[a], list[a + 1], ..., list[b].

list[0]...list[5] represents the array elements list[0], list[1], list[2], list[3], list[4], and list[5].

If list is of length 1, then list has only one element, which is the largest element.

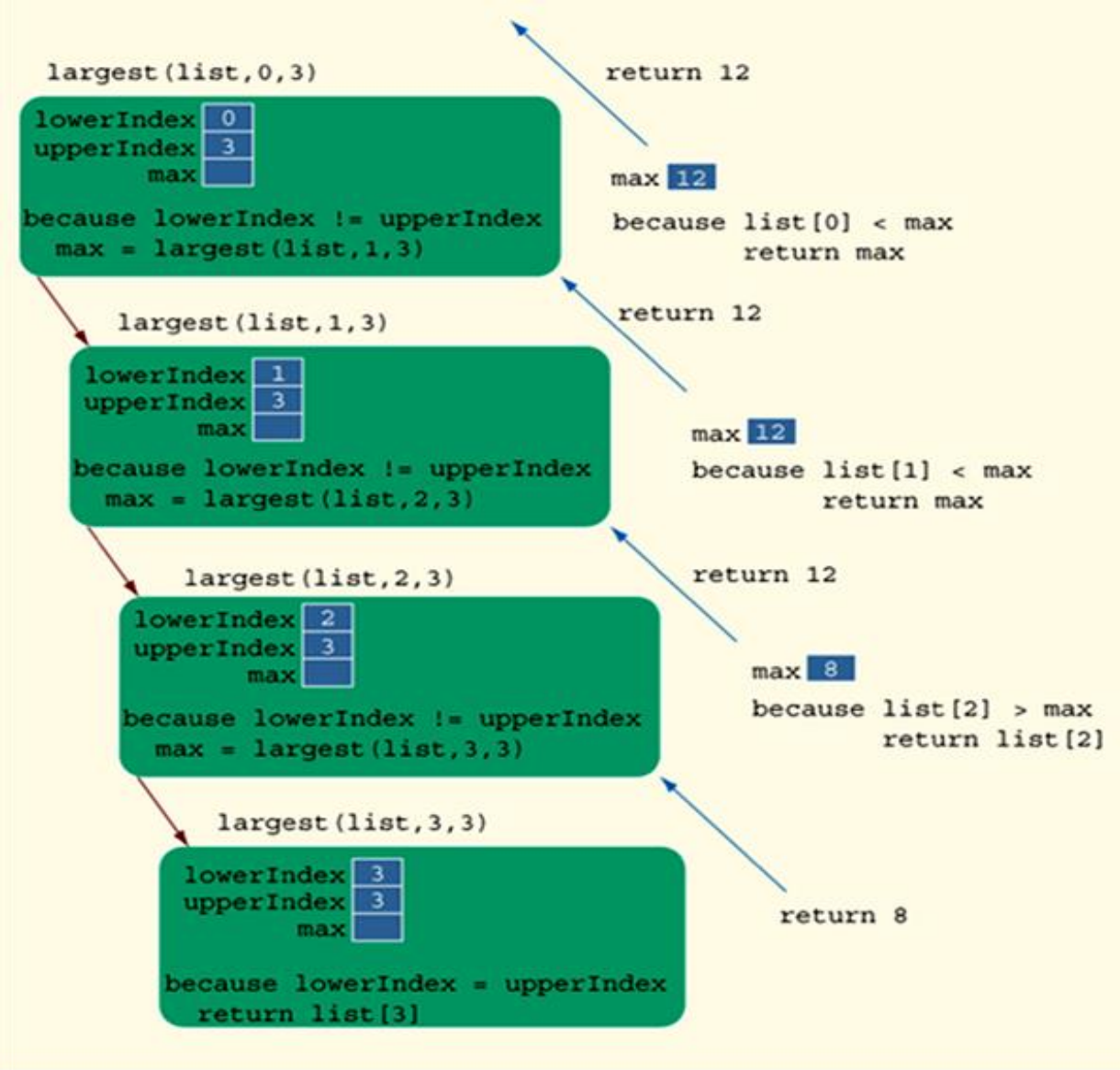
Suppose the length of list is greater than 1.

To find the largest element in list[a]...list[b], we first find the largest element in list[a + 1]...list[b] and then compare this largest element with list[a].

The largest element in list[a]...list[b] is given by: maximum(list[a], largest(list[a + 1]...list[b]))

```
Base Case: The size of the list is 1
           The only element in the list is the largest element

General Case: The size of the list is greater than 1
      To find the largest element in list[a]...list[b]

      a. Find the largest element in list[a + 1]...list[b]
         and call it max
      b. Compare the elements list[a] and max
         if (list[a] >= max)
            the largest element in list[a]...list[b] is list[a]
         otherwise
            the largest element in list[a]...list[b] is max
```

```
int largest(int list[],int lowerIndex,int upperIndex)
{
 int max;

 if (lowerIndex == upperIndex)
     return lowerIndex ;
 else
 {
     max = largest(list, lowerIndex + 1, upperIndex);

     if (list[lowerIndex] >= max)
         return lowerIndex ;
     else
         return max;
 }
```




Execution of `largest(list, 0, 3)`