Recursion

One of the basic rules for defining new objects or concepts is that the definition should contain only such terms that have already been defined or that are obvious. On the other hand, there are many programming concepts that are defined in terms of themselves. Such definitions are called recursive definitions.
A recursive definition consists of two parts.

- **Anchor or Ground case**: The basic elements that are building blocks of the other elements of the set are listed in anchor case.
- **Recursive or Inductive case**: In the recursive case, rules are given that allow for the construction of new object.

For instance, the factorial function,!, can be defined in the following manner

n! = 1 if n = 0 (anchor)

n*(n-1)! If n>0 (inductive step)

Using this definition, we can generate the sequence of numbers 1, 1, 2,6,24,120, 720, 5040…….

Recursion can be regarded as the ability of a function defining an object in terms of a simpler case of itself. It is a process by which a function calls itself repeatedly until some condition has been satisfied. For problems to be solved recursively two conditions must be satisfied.

a. The problem must be expressed in recursive form.
b. The problem statement must include a terminating or stopping condition.

Note: At least one statement inside a function must be of type non recursive type

Recursion versus Iteration

Every recursive program can be written in iterative form. Iterative form eliminates the overhead of passing arguments and returning value. Also redundant computations are also eliminated. An iterative calculation seems to be clear and simple.
If the problem is itself recursive in nature, it is wiser to use recursive solution than iterative one. Therefore, in some cases iterative is better solution while in other cases recursive is better solution. Programmer should analyze the problem and select the best solution.

Recursion Types: Direct and Indirect

A function may directly or indirectly call itself in course of its execution. If the function call is within its body then the recursion is direct. If the function calls another function which in turn calls itself, then such recursion is indirect.
The direct recursive function can be explained by following example.

```
int factorial(int n)
{
        If(n==0)
                return 1;
        else
                return(n*factorial(n-1));
}
```

The function factorial () calls itself repeatedly. This type of recursive function is called **direct** recursive function.
A recursive function need not call itself directly. Rather, it may call itself indirectly, as in the following example.

```
fun1(formal parameter)
{
        ……..
        ……..
        fun2 (arguments)
}
```

```
fun2(formal arguments)
{
                ……..
        ……
        fun1 (arguments)
}
```

In this example, method fun1 calls fun2, which may in turn call fun1, which may again call fun2. Thus both fun1 and fun2 are recursive, since they indirectly call on themselves.

Advantages of recursive function

i. Although at most of the times a problem can be solved without recursion, but in some situations in programming, it is a must to use recursion. For example, a program to display a list of all files of the system cannot be solved without recursion.
ii. The recursion is very flexible in data structure like stacks, queues, linked list and quick sort.
iii. Using recursion, the length of the program can be reduced.

Disadvantages

i. It requires extra storage space. The recursive calls and automatic variables are stored on the stack. For every recursive calls separate memory is allocated to automatic variables with the same name.
ii. If the programmer forgets to specify the exit condition in the recursive function, the program will execute out of memory. In such a situation user has to press Ctrl+ break to pause and stop the function.
iii. The recursion function is not efficient in execution speed and time.
iv. It is very difficult to trace and debug recursive function.

Fibonacci Sequence

The Fibonacci sequence is the sequence of integers :0,1,1,2,3,5,8,13,21,34…….
Each element in this sequence is the sum of the two preceding elements. If we let fib(0)=0, fib(1)=1, and so on, then we may define the Fibonacci sequence by the following recursive definition:
fib(n)=n if n==0 or n==1.
fib(n)=fib(n-1)+fib(n-2) if n>2.
A method in Java to compute the sequence of Fibonacci number up to nth term is as shown below:

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

Multiplication of Natural Numbers

The multiplication of two numbers can also be performed recursively. The product a*b where a and b are positive integers, may be defined as a added itself b times. This is iterative definition. An equivalent recursive definition is:
a*b=a if b==1.
a*b=a+a*(b-1) if b>1.
To evaluate 6*3 by this definition, we first evaluate 6*2 and then add 6. To evaluate 6*2, we first evaluate 6*1 and add 6. But 6*1 equals 6 by the first part of the definition. Thus
6*3=6*2+6=6*1+6+6=6+6+6=18.

The recursive method in Java for multiplication of two numbers is as shown below:

```
public static int product(int x, int y)
{
   if(y==1)
   {
         return x;
   }
   else
          return(x+product(x,y-1));
}
```

Method Calls and Recursion Implementation

When method is called, the must know where to resume execution of the program after the method has finished. The information indicating where it has been called from has to be remembered by the system. For a method call, more information has to be stored than just a return a return address. Therefore, a dynamic memory allocation using the run time stack is a much better solution. The run time stack is maintained by a particular operating system.

What information should be preserved when a method is called?

First, local variables must be stored. If a method **f1()** declares variable **x**, calls method **f2()**, which locally declares the variable **x**. The system has to make a distinction between these two variables. This is more important when function call is recursive where **f1 ()** and **f2 ()** are same.

The state of the each method, including main(), is characterized by the contents of all local variables, the values of the method's parameters and the by the return address indicating where to restart its caller. The data area containing all this information is called activation record and is allocated on run time stack. It has short lifespan. The activation record usually contains the following information:

- Values for all the parameters to the method.
- The return address to resume control by the caller.
- The dynamic link, which is a pointer to the caller's activation record.
- The returned value for a method not declared as void.

When a method is called either by main () or by another method, then its activation record is created on the run time stack. The run time stack always reflects the current state of the method. For example, suppose that main() calls method **f1()**, **f1()** calls method **f2()**, **f2()** in turn calls **f3()**. If **f3 ()** is being executed, then the state of the run time stack is shown as below. By nature of stack, if the activation record of f3 is popped by moving the stack pointer right below the return value of **f3()**, then **f2()** removes execution and now has free access to the private pool of information necessary for reactivation of its execution.

Creating an activation record whenever a method is called allows the system to handle recursion properly. Recursion is calling a method that happens to have the same name as the caller. These invocations are represented by different activation records and are thus differentiated by the system.

Activation Record of f3()	Parameters and local variables
	Dynamic Link
	Return Address
	Return value
Activation Record of f2()	Parameters and local variables
	Dynamic Link
	Return Address
	Return value
Activation Record of f1()	Parameters and local variables
	Dynamic Link
	Return Address
	Return value
Activation Record of main()	

Figure: *Contents of the run time stack when main () calls method f1 (), f1() calls f2(), and f2() calls f3()*.

Anatomy of a Recursive Call

The function that defines raising any number x to a nonnegative integer power n is a good example of a recursive function. The most natural definition of this function is given by:

$x^n = 1$ if n=0

$= x*x^{n-1}$

A java method for computing x^n can be written directly from the definition of a power.

```
double power (double x, int n)
{
        if(n==0)
                return 1.0;
        else
                return x*power(x,n-1);
}
```

Using this definition, the value of x^4 can be computed in the following way:

Call 1 $x^4 = x.x^3 = x.x.x.x$

Call 2 $x.x^2 = x.x.x$

Call 3 $x.x^1 = x.x.$

Call 4 $x.x^0 = x.1 = x$

Call 5 1

Or alternatively, as

```
Call 1                  power(x,4)
Call 2                      power(x,3)
Call 3                          power(x,2)
Call 4                              power(x,1)
Call 5                                  power(x,0)
Call 5                              1
Call 4                          x
Call 3                      x.x
Call 2                  x.x.x
Call 1          x.x.x.x
```

The repetitive application of the inductive step eventually leads to the anchor, which is the last step in the chain of recursive calls. The anchor produces 1 as a result of raising x to the power of zero; the result is passed back to the previous recursive call. The process continues to the last call.

What does the system do as the method is being executed?

The system keeps track of all calls on its run time stack. Each line of code is assigned a number by the system, and if a line is a method call, then its number is a return address. The address is used by the system to remember where to resume execution after the method has completed. For this example assume that the line in the method power() are assigned the number 102 through 105 and that it called in main() from the statement

```
                Static public void main())
                {
                ……
/* 130   */     y = power(5.6,2);
                ……..
                }
```

A trace of the recursive calls is shown as follows:

```
Call 1                  power(5.6,2)
Call 2                      power(5.6,1)
Call 3                          power(5.6,0);
Call 3                          1
Call 2                      5.6
Call 3                  31.36
```

When the method is invoked for the first time, four items are pushed onto the run time stack: the return address 136, the actual parameters 5.6 and 2 one location reserved for the value returned by the power(). This information shown in the following figure

Third call to power()			**0←SP 5.6 (105) ?**	**0←SP 5.6 (105) 1.0**	**0 5.6 (105) 1.0**			
Second call to power()		**1←SP 5.6 (105) ?**	**1 5.6 (105) ?**	**1 5.6 (105) ?**	**1←SP 5.6 (105) 1.0**	**1←SP 5.6 (105) 5.6**	**1 5.6 (105) 5.6**	
First call to power()	**2←SP 5.6 (136) ?**	**2 5.6 (136) ?**	**2 5.6 (136) ?**	**2 5.6 (136) ?**	**2 5.6 (136) ?**	**2 5.6 (136) ?**	**2←SP 5.6 (136) ?**	**2←SP 5.6 (136) ?**
AR for main()	**…….. …….. y …….. ……..**	**…….. …….. y …….. ……..**	**…….. …….. y …….. ……..**	**…….. …….. y …….. ……..**	**…….. …….. y …….. ……..**	**…….. …….. y …….. ……..**	**…….. …….. y …….. ……..**	**…….. …….. y …….. ……..**

Tail Recursion and Nontail recursion

Recursive methods are two types.

Tail recursion:

Tail recursion has recursive call as the last statement in the method. In other words, when the call is made, there are no statements left to be executed by the method. Recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect. For example, the following method ***tail()*** defined as

```
void tail(int i)
{
        if(i>0)
        {
                System.out.print(i+ " ");
                tail(i-1);
        }
}
```

is an example of a method with tail recursion.

Nontail recursion:
Recursive methods that are not tail recursive are called non-tail recursive. Another example that can be implemented in recursion is printing an input line in reverse order. Here is a simple recursive implementation:

```
void reverse ()
{
        char ch = getChar();
        if(ch!='\n')
        {
                reverse();
                System.out.print(ch);
        }
}
```

Is the following factorial method a tail recursive method?

```
int fact(int x)
{
        if (x==0)
                return 1;
        else
                return x*fact(x-1);
}
```

- When returning back from a recursive call, there is still one pending operation, multiplication.
- Therefore, factorial is a non-tail recursive method.

Is the following method tail recursive?

```
void tail(int i)
{
        if (i>0)
        {
                System.out.print (i+"")
                tail(i-1);
        }
}
```

It is tail recursive!

Is the following program tail recursive?

```
void  prog(int i)
{
        if (i>0)
        {
                prog(i-1);
                System.out.print(i+" ");
                prog(i-1);
        }
}
```

- No, because there is an earlier recursive call, other than the last one,

- In tail recursion, the recursive call should be the last statement, and there should be no earlier recursive calls whether direct or indirect.

Advantage of Tail Recursive Method

- Tail Recursive methods are easy to convert to iterative.
- Smart compilers can detect tail recursion and convert it to iterative to optimize code
- Used to implement loops in languages that do not support loop structures explicitly (e.g. Prolog).

A Java method for finding factorial of any number using tail recursion.

```
int factorial(int n, int result)
{
   if(n==1)
   return result;
   return(factorial(n-1,n*result));
}
```

Write a tail recursion method in Java for finding product of two numbers.

```
int product(int a, int b, int result)
{
   if(b==1)
   return result+a;
   else
   return(product(a,b-1,a+result));
}
```

Write a tail recursive method in Java that finds sum of natural numbers.

```
int sum(int n,int result)
{
   if(n==1)
   return result+1;
   else
   return(sum(n-1, n+result));
}
```

Write a method in Java that finds nth Fibonacci number using tail recursion.

```
int fib(int n)
{
    if(n<=1)
    return n;
    return(fib(n-2)+fib(n-1));
}
```

Nested Recursion

A more complicated case of recursion is found in definitions in which a function is not only defined in terms of itself, but also is used as one of the parameters. The following definition is an example of such a nesting:

h(n) = 0 if n = 0
= n if n>4
= h(2+h(2n) if n≤4

Function h has a solution for all n≥0. This fact is obvious for all n>4 and n=0, but it has to be proven for n=1, 2, 3 and 4. Thus, h (2) = h (2+h (4)) = h (2+h (8))) = 12.

Excessive Recursion

Logical simplicity and readability are used as an argument supporting the use of recursion. The price for using recursion is slowing down execution time and storing on run time stack more things than required in a non-recursive approach. If recursion is too deep, then we can run out of space on the stack and our program terminates abnormally by raising an unrecoverable **StackOverflowError**. But usually, the number of recursive call is much smaller, so danger of overflowing the stack may not be imminent.

Consider Fibonacci numbers. A sequence of Fibonacci numbers is defined as follows:

Fib (n) = n if n<2

Fib (n-2) +Fib (n-1) otherwise

The definition states that if the first two numbers are 0 and 1, then any numbers in the sequence is the sum of its two predecessors. But these predecessors are in turn sums of their predecessors, and so on, to the beginning of the sequence. The sequence produced by the definition is

0,1,1,2,3,5,8,13,21,34,55,89,……

The definition can be implemented in Java as follows:

```
long Fib(long n){
        if(n<2)
                return n;
        else
                return Fib(n-2)+Fib(n-1);
}
```

The method is simple and easy to understand but extremely inefficient. Let us compute Fib(4).

```
                                        Fib(4)
=       Fib(2)                  +       Fib(3)
=       Fib(0)+Fib(1)                   +       Fib(3)
=       0       +       1               +       Fib(1)  +       Fib(2)
=       0       +       1               +       1       +       Fib(0) + Fib(1)
=       0       +       1               +       1       +       0       + 1
= 3
```

It takes almost a quarter of a million calls to find the twenty –sixth Fibonacci number, and nearly 3 million calls to determine the thirty first. This is too heavy a price for the simplicity of the recursive algorithm.

We can prove that the number of additions required to find Fib(n) using a recursive definition is equal to Fib(n+1)-1. Counting two calls per one addition plus the very first call means that Fib() is called 2.Fib(n+1)-1 times to compute Fib(n). This number can exceedingly large for fairly small ns, as in the following table.

N	Fin(N+1)	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1973
20	10946	10945	21891
25	121393	121392	242785
30	1346269	1346268	2692537

Figure: Number of addition operations and number of recursive calls to calculate Fibonacci numbers.

Backtracking

In solving some problems, a situation arises where there are different ways of leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ensure that such a return is possible and that all paths can be tried. This technique is called backtracking, and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. Using backtracking, we can always return to a position that offers other possibilities for successfully solving the problem. This technique is used in artificial intelligence, and one of the problems in which backtracking is very useful is the eight queens problem.

The eight queen problem attempts to place eight queens on a chessboard in such a way that no queen is attacking any other. The rules of the chess say that a queen can take another piece if it lies on the same row, on the same column, or on the same diagonal as the queen.

To solve this problem, we try to put the first queen on the board, then second so that it is not conflict with the first, then the third so that it is not in conflict with the two already placed, and so on, until all of the queens are placed.

What happens if, for instance, the sixth queen cannot be placed in a no conflicting position? We choose another position for the fifth queen and try again with the sixth. If this does not work the fifth queen is moved again. If all the possible positions for the fifth queen have been tried, the fourth queen is moved and then the process restarts. This process requires a great deal of effort, most of which is spent backtracking to the first crossroads offering some untried avenues. In terms of code, however, the process is rather simple due to the power of recursion, which is a natural implementation of backtracking. Pseudocode for this backtracking algorithm is as follows:

```
putQueen (row)
for every position col on the same row
        if position col is available
        place the next queen in position col
        if(row<8)
          PutQueen (row+1);
        else success;
        remove the queen from position col;
```

This algorithm finds all possible solutions without regard to the fact that some of them are symmetrical.