Chapter

11 RECURSIVE FUNCTION

- *Recursive function and base case*
- *Use of memory stack*
- *Function calling itself*
- *Features of Recursive Function and its comparison with Iteration*
- *List of programs on Recursive Functions (worked out & exercise)*

1. Sum of first n natural numbers S = 1 + 2 + 3 + ... + n
2. Sum of first n natural numbers (another method)
3. p * q , without using * operator but by using repetitive addition
4. *p ^ q , by using repetitive multiplication*
5. Print the numbers from p to q and back from q to p
6. *Print the even numbers from p to q. (assuming p is even and p<=q)*
7. *Print the even numbers from p to q. (where p may be even or odd, and p<=q)*
8. Print the factors of a number from 1 till its half (forward and backward).
9. Print the Fibonacci series up to n. [1 1 2 3 5 8 13 21 ...][sum of prev 2 values]
10. *Print the Tribonacci series up to n. [0 1 1 2 4 7 13 24 ..] [sum of prev 3 values]*
11. Print the HCF (or GCD) of two numbers.
12. *Print the LCM of two numbers.*
13. Division of a number by another. Method fnDivide(int n, int f) returns the quotient of n/f and prints the remainder.
14. Print the HCF (or GCD) of two numbers. [Another method]
15. Convert a Binary number to its equivalent Decimal value
16. Recursive Sorting Algorithms – Merge Sort and Quick Sort
17. *Codes to identify the output*

Recursive Function and its Base Case

A programming technique in which a reference is made to itself is called recursion. A recursive function is a special kind of function that makes a call to itself.

A **base case** is the condition where recursion ends. Every recursive function should have a base case. Upon reaching the base case, the recursion stops and calculation begins. Without a base case, a recursive function repeats infinite times.

Use of Memory Stack in a Recursive Function

A recursive function uses the *memory stack* to store its intermediate values (which are calculated after the control returns from the base case). If a recursive function enters infinite (an error situation) then a runtime error occurs, which states "Stack Overflow".

Functions Calling Other Functions

Example Code :

main()
{
...
...
fn Test1();
...
...
}
void fn Test1()
{
...
...
}
	EXPLANATION
	
	main() is the super function and fnTest1() is the sub function.
	
	The control goes to the sub function when it is called and when its code is complete, it goes back to its super function. The super function carries on with the rest of its code after the function call is over.

Functions Calling Itself

In case of recursive functions, the function becomes its own super and sub function.

Example Code :

fnRecl (argument list a)
{
...
...
fnRecl (argument list b);
...
...
}
	EXPLANATION
	
	In the given example, fnRecl, fnRecl (..) makes a fn call to itself.
	It is a recursive function.
	While making a fnction call to itself, the value of arguments identify super and sub function.

Features of Recursive Function

- It makes a call to itself with a change in the arguments.
- It terminates when it reaches its base case.
- It does not use a variable to store the intermediate value instead uses the memory stack of the computer.
- In case of memory shortage, it gives an error "Stack Overflow".
- Each recursion performs a fresh memory allocation for its arguments/variables.
- Memory requirement of recursive function is high, hence the system cost also becomes high.
- Certain kind of programs cannot be done without using recursion [like Tower of Hanoi and more of the same kind].

Comparison between Recursion and Iteration

- Recursion using a recursive function and iteration using a loop are meant for doing repetitive tasks.
- Example

```
// Iteration
int fnFactorialLoop(int n)
{
  int f = 1;
  for(int j = 1; j<=n; j++)
  {
     f = f * j;
  }
}
```

```
// Recursion
  int fnFactorialRecur(int n)
  {
     if (n<=1);
     {
        return 1;
     }
     return ( n * fnFactorialRecur(n–1));
  }
```

- In iteration, first the calculation is done and then the control moves ahead but in recursion, the calculation is done after the base case is reached.
- In general, the space requirement (in terms of memory stack) is more compared to iteration. [in iteration, same memory block is used for calculation].
- Space and time complexity of recursion is generally more than iteration.

In the next section, programs using recursive functions are shown.

[Note : Every recursive function must contain a base case.]

Programs on Recursive Functions

Example 1 : Sum of first n natural numbers. S = 1 + 2 + 3 + ... + n

```
import java.io.*;
public class clRecursive
{
   public void main()
   {
      int r1 = 0;
      r1 = fnAdd(5);
      System.out.print("\n Result of fnAdd(5) = " + r1);
```

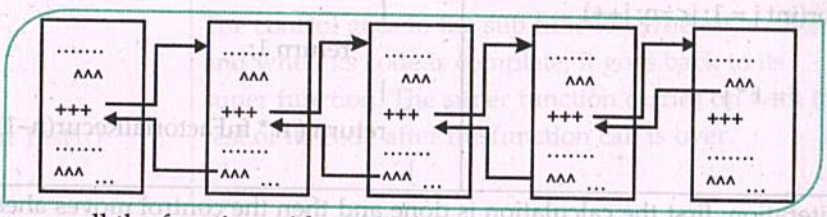
```
    }
    public int fnAdd(int n)
    {
        if( n <= 0)  // base case
        {
            return 0;
        }
        System.out.print("\n in fnAdd .. n = " + n);
        return (n + fnAdd(n–1));
    }
}
```

Output

```
infnAdd .. n = 5
infnAdd .. n = 4
infnAdd .. n = 3
infnAdd .. n = 2
infnAdd .. n = 1
Result of fnAdd(5) = 15
```

WORKING (shown in 2 ways)

^^^ indicates a return statement.
+++ indicates recursive fn call
..... indicates other executable statements.



Assume we call the function with $n = 5$, then fnAdd(5) will work as follows :

```
5 + fnAdd(4)
-------------------
        |
        | 4 + fnAdd(3)
        -------------------
                |
                | 3 + fnAdd(2)
                ---------------
                        |
                        | 2 + fnAdd(1)
                        -------------
                                |
                                | 1 + fnAdd(0)
                                ---------------
                                        |
                                        | 0
                                        ==========
```

When the recursion ends, the calculation takes place back up.

So the calculation here goes as

fnAdd(5)

= 5 + (fnAdd (4))

= 5 + (4 + fnAdd(3))

= 5 + (4 + (3 + fnAdd (2)))

= 5 + (4 + (3 + (2 + fnAdd (1))))

= 5 + (4 + (3 + (2 + (1 + fnAdd (0)))))

= 5 + (4 + (3 + (2 + (1 + (0)))))

We can also say that fnAdd(4) is a sub function of fnAdd(5). Again fnAdd(3) is a sub function of fnAdd(4). The base case takes place when n = 0. fnAdd(0) does not call the function but returns 0 directly. After the base case, the calculation gets completed.

Example 2 : Sum of first n natural numbers (another method). S2 = 1 + 2 + 3 + ... + *n*

```
public int fnRec1(int n, int limit)
{
    if( n > limit)
    {
        return 0;
    }
    System.out.print("\n in fnRec1 .. n = " + n);
    return (n + fnRec1(n+1, limit) );
}
```

WORKING : Assume we call the function with n = 1 and limit = 5.

```
fnRec1 (1, 5)
  1+ fnRec1 (2, 5)
       2 + fnRec1 (3, 5)
            3 + fnRec1 (4, 5)
                 4 + fnRec1 (5, 5)
                      5 + fnRec1 (6, 5)
                           0
```

When the recursion ends, the calculation takes place back up. So the calculation here is

(1 + (2 + (3 + (4 + (5 + (0))))))

This function makes n start from 1 and increase till the limit.

Example 3 : p * q , without using * operator but by using repetitive addition.

If p = 5, q = 4 then, p*q = 5*4 = 5 + 5 + 5 + 5

```
public int fnRec3(int p, int q)
{
    if( q < 1)
    {
        return 0;
    {
    System.out.print("\n in fnRec3 .. p = " + p +  " q = " + q) ;
    return (p + fnRec3(p, q–1) );
}
```

OUTPUT

```
in fnRec3 .. p = 5  q = 4
in fnRec3 .. p = 5  q = 3
in fnRec3 .. p = 5  q = 2
in fnRec3 .. p = 5  q = 1
```

[The function finally returns 20.]

WORKING

Assume we call the function with p = 5 and q = 4.

```
fnRec3 (5, 4)
5  +  fnRec2 (5, 3)
     5  +  fnRec3 (5, 2)
          5  +  fnRec3 (5, 1)
               5  +  fnRec3 (5, 0)
                    0
```

When the recursion ends, the calculation takes place back up. So the calculation here is

(5 + (5 + (5 + (5 + (0)))))

Example 4 : ***p ^ q , by using repetitive multiplication.***

*[Hint : return (p * fnRec4(p, q–1));]*

Example 5 : Print the numbers from p to q and back q to p.

```
public void fnRec6(int p, int q)
{
    if ( p > q )
    {
        return ;
    }
    System.out.print("\n  ---  " + p) ;
    fnRec6 (p+1, q ) ;
    System.out.print("\n  +++  " + p) ;
}
```

OUTPUT of the above code

fnRec6 result

--- 5
--- 6
--- 7
--- 8
--- 9
+++ 9
+++ 8
+++ 7
+++ 6
+++ 5

WORKING

Assume we call the function with p = 5 and q = 9.

```
fnRec6 (5, 9)                    5
    fnRec6 (6, 9)                6
        fnRec6 (7, 9)            7
            fnRec6 (8, 9)        8
                fnRec6 (9, 9)    9
                    fnRec6 (10, 9)
                fnRec6 (9, 9)    9
            fnRec6 (8, 9)        8
        fnRec6 (7, 9)            7
    fnRec6 (6, 9)                6
fnRec6 (5, 9)                    5
```

This function shows that a recursive function goes in step by step. After the recursion is over, the control comes back step by step.

Example 6 : *Print the even numbers from p to q. (assuming p is even and p<=q)*

[Hint : System.out.print("\n --- " + p) ; fnRec7 (p+2, q) ;]

Example 7 : *Print the even numbers from p to q. (where p may be even or odd, and p<=q).*

[Hint : If p is odd, increment it by 1, then similar as above]

Example 8 : Print the factors of a number from 1 till its half (forward and backward).

```
public void fnRec10 (int num, int f)
{
    if (f > num/2)
    {
        return;
    }
```

```
    if (num % f == 0)
    {
        System.out.print("\n Before " + f + " is a factor of  " + num);
    }
    fnRec10(num, f+1);    //recursive function call
    if (num % f == 0)
    {
        System.out.print("\n After " + f + " is a factor of  " + num);
    }
}
```

OUTPUT of the above function with num = 12 and f = 1.

fnRec10 result :

Before 1 is a factor of 12
Before 2 is a factor of 12
Before 3 is a factor of 12
Before 4 is a factor of 12
Before 6 is a factor of 12
After 6 is a factor of 12
After 4 is a factor of 12
After 3 is a factor of 12
After 2 is a factor of 12
After 1 is a factor of 12

WORKING

Assume we call the function with num = 12 and f = 2.

Each time the recursive function is called, the present level values are stored in memory stack. The values are restored when the control returns to the previous level.

Assume we call fnRec10(12, 2)

Example 9 : Print the Fibonacci series up to n.

```
public void fnRec11(int a, int b, int n)
{
    if (n<=0)
    {
        return;
    }
    System.out.print(" .. " + a  );
    fnRec11( b, (a+b), n–1);
}
```

OUTPUT with a = 1, b = 1, n = 9

.. 1 .. 1 .. 2 .. 3 .. 5 .. 8 .. 13 .. 21 .. 34

WORKING

Assume we call the function with a = 1, b = 1, n = 9.

```
fnRec11 (1, 1, 9)                                          1
    fnRec11 (1, 2, 8)                                      1
        fnRec11 (2, 3, 7)                                  2
            fnRec11 (3, 5, 6)                              3
                fnRec11 (5, 8, 5)                          5
                    fnRec11 (8, 13, 4)                     8
                        fnRec11 (13, 21, 3)                13
                            fnRec11 (21, 34, 2 )           21
                                fnRec11 (34, 55, 1)        34
                                    fnRec11 (55, 89, 0) ....no work for this step
```

Example 10 : ***Print the Tribonacci series up to n.***

[0 1 1 2 4 7 13 24..] [sum of prev 3 values]

Example 11 : Print the HCF (or GCD) of two numbers.

```
public int fnRec13(int a, int b, int hc )
{
    if (a % hc == 0 && b % hc == 0 )  // test if both are factors
    {
        return hc;
    }
    return ( fnRec13(a, b, hc–1));
}
public void main()
{
    System.out.print("\n fnRec13 result = " + fnRec13(9, 6,Math.min(9, 6 ) ) ) ;
}
```

OUTPUT with a = 9, b = 6, hc = 6.

fnRec13 result = 3

WORKING

Assume we call the function with a = 9, b = 6, hc = 6.

```
fnRec13 (9, 6, 6)    Cond false, so fn call again
fnRec13 (9, 6, 5)    Cond false, so fn call again
fnRec13 (9, 6, 4)    Cond false, so fn call again
fnRec13 (9, 6, 3)    Cond is true, so fn returns hc(3)
```

Example 12 : ***Print the LCM of two numbers.***

Example 13 : Division of a number by another. Method fnDivide(int n, int f) returns the quotient when n is divided by f and prints the remainder.

```
//code to return the quo and to print the remainder
public int fnDivide(int n, int f)
{
    if( n == 0 || (n–f) < 0 )
    {
        System.out.print("\n Remainder is " + n);
        return 0;
    }
    return (1 + fnDivide(n–f, f) );
}
```

OUTPUT ANALYSIS

What will happen if the given code is executed with :

(i) n = 38 and f = 7

(ii) n = 100 and f = 35

(iii) n = 25 and f = 9

(iv) n = 150 and f = 50

Example 14 : Print the HCF (or GCD) of two numbers. [Another method]

```
public int fnHCF2(int m, int n)
{
    if (m == n)
    {
        return n;
    }
    else if (m > n)
    {
        fnHCF2 (m–n , n);
    }
    else
    {
        fnHCF2 (m, n–m);
    }
}
public void main()
{
    System.out.print("\n fnHCF result = " + fnHCF(9,6) );
}
```

OUTPUT with m = 9, n = 6

fnHCF result = 3

WORKING

m = 9, n = 6 ; m==n → false

m>n → true so,

m = 3, n = 6 ; m == n → false

m>n → false

else → true so,

m = 3, n = 3 ; m == n → true so

return n → 3 which is the HCF

Example 15 : Convert a Binary number to its equivalent Decimal value.

```
int fnBinToDeci(int d, int p)
{
    if(d<=0)
    {
        return 0;
    }
    return ( (int)(d%10 * Math.pow(2, p) ) + fnBinToDeci ( d/10 , p+1 ) ) ;
}
```

OUTPUT with d = 1110 , p = 0

fnBinToDeci result = 14

WORKING

d = 1110 , p = 0 ;

fnBinToDeci (1110, 0)

$0*2^0$ + fnBinToDeci (111, 1)

$1*2^1$ + fnBinToDeci (11, 2)

$1*2^2$ + fnBinToDeci (1, 3)

$1*2^3$ + fnBinToDeci (0, 4) [d <=0 is true]

0

Example 16 : Given the following code, analyse the output.

```
int Pilot ( int n )
{
    System.out.print(" * " + n ) ; // to show the sequence of call
    if (n == 1)
        return 2 ;
    else if (n == 2)
        return 3 ;
    else
        return Pilot (n–2) + Pilot (n–1) ;
}
```

WORKING

Assume we call the function with n = 5 , the result is 13.

```
Pilot (5)
   Pilot (3) + Pilot (4)
         Pilot (1) + Pilot (2) + Pilot (2) + Pilot (3)
            2    +    3    +    3    + Pilot (1)  + Pilot(2)
            2    +    3    +    3    +    2    +    3
```

Example 17 : Given the following code, analyse the output.

```
void Plane ( int n )
{
   System.out.print (" _ " + n ) ;
   if (n <= 0)
      System.out.println();
   else if (n%2 == 0 )
      Plane (n–2 );
   else
      Plane (n–1 );
}
```

WORKING

Assume we call the function with n = 7, the output is _ 7 _ 6 _ 4 _ 2 _ 0

```
Plane (7)                  _ 7
n <= 0 is false
n%2 == 0 is false
so  Plane (6)
Plane (6)                  _ 6
n <= 0 is false
n%2  == 0 is true
so        Plane (4)
Plane (4)                  _ 4
n <= 0 is false
n%2  == 0 is true
so        Plane (2)
Plane (2)                  _ 2
n <= 0 is false
n%2  == 0 is true
so        Plane (0)
Plane (0)                  _ 0
n <= 0 is true             base case, no further recursion
```

Example 18 : Given the following code, analyze the output

```
void fnFan (int n, int ka, int kb)
{
    System.out.print (" : " + ka ) ;
    if (n <= 1)
    {
        return ;
    }
    else
    {
        fnFan( n–1 , kb, ka+kb );
    }
}
```

WORKING

Assume we call the function with n = 10, ka = 1, kb = 1 ,
the output is : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : 55
[Student's task is to explain the working.]

Example 19 : Given the following code, analyse the output.

```
int fnLite( int n )
{
    System.out.print("  " + n ) ;
    if( n <= 1)
    {
        System.out.println() ;
        return n;
    }
    else
    {
        return (fnLite( n–1) + fnLite(n–2) );
    }
}
```

WORKING

Assume we call the function with n = 5
The output is

```
5  4  3  2  1
0
1
2  1
0
3  2  1
0
1
```

```
fnLit (5)
fnLit (4)                                   +  fnLit (3)
fnLit (3)                                   +  fnLit(2)              +  fnLit(2)              + fnLit(1)
fnLit (2)              +  fnLit (1)  +  fnLit(1)  +  fnLit(0)  +  fnLit(1)  +  fnLit(0)  + 1
fnLit (1)  +  fnLit (0)  +  1        +  1         +  0         +  1         +  0         + 1
1          +  0          +  1        +  1         +  0         +  1         +  0         + 1
= 5
```

Example 20 : Given the following code, analyse the output.

```
void fnLake (String k, int p)
{
    if ( p <= 0 )
    {
        return;
    }
    if (p%2 == 0 )
    {
        System.out.println ("2  " + k.charAt(p) ) ;
    }
    else
    {
        System.out.println ("3 " + k.charAt(p–1) ) ;
    }
    fnLake(k, p–1);
}
```

[Student's task is to explain the working.]

Example 21 : Program to convert a number from Binary To Decimal (using Recursion)

```
import java.util.Scanner;
public class base_Change
{
    public static void main(String [] args)
    {
        int input, size;  Scanner scan = new Scanner(System.in);
        System.out.print("Enter any decimal integer: ");
        input = scan.nextInt();
        convert(input);
        System.out.println();
```

```
        System.out.print("Enter binary integer and size : ");
        input = scan.nextInt();
        size = scan.nextInt();
        System.out.println (BTD (input, size) ) ;
    }
    public static void convert(int num)
    {
        if (num > 0)
        {
            convert (num / 2);
            System.out.print (num % 2 + " ");
        }
    }
    public static int BTD(int binary, int size)
    {
        if (binary == 0)
        {
            return 0 ;
        }
        return (binary % 10 * (int) Math.pow (2, size–1) + BTD ((int) binary / 10, size – 1) ) ;
    }
}
```

Recursive Sorting Algorithms

A recursive algorithm is considerably more efficient than quadratic algorithm. The quadratic algorithms are faster in execution.

Examples of recursive algorithms that use a divide and conquer paradigm are Merge Sort and Quick Sort.

QuickSort

Algorithm

- Select an element that belongs in the middle of the array (called the pivot), and place it at the midpoint
- Place all values less than the pivot before the pivot, and all elements greater than the pivot after the pivot

The following partial code shows the application -

```
void quicksort (int data[ ], size_t n)
{
        size_tpivot_index;    // array index of pivot
        size_t n1, n2;        // number of elements before & after pivot
        if (n > 1)
        {
            partition (data, n, pivot_index);
            n1 = pivot_index;
            n2 = n – n1 – 1;
            quicksort (data, n1);
            quicksort ((data+pivot_index+1), n2);
        }
}
```

MERGESORT

Algorithm

- divide the array at midpoint
- sort half-arrays via recursive calls
- merge the two halves array

The following **partial code** shows the application -

```
void mergesort (item data)[ ], size_t n)
{
        size_t n1, n2; // sizes of the two subarrays
        if (n > 1)
        {
            n1 = n / 2;
            n2 = n – n1;
            mergesort (data, n1);
            mergesort ((data + n1), n2);
            merge (data, n1, n2);
        }
}
```

Program of Merge Sort using Recursive Technique

```
import java.util.*;
public class MergeSort
{
```

```
void fnDisplay(int B[ ])
{
    for ( int j = 0; j<=B.length-1; j++ )
        System.out.print(" " + B[j]) ;
}
void main()
{
    int n = 10 ;
    int A[ ] = new int [n] ;
    for ( int j = 0; j<=n–1 ; j++ )
    A[j] = (int)(Math.random() * 100 ) ;
    System.out.print(" The Array is.\n") ;
    fnDisplay(A) ;
    fnMergeSort(A);
    System.out.print("\n\n\tThe Array after Merge Sort is \n") ;
    fnDisplay(A) ;
}
void fnMergeSort(int B[ ])
{
    if(B.length<=1 )
        return;
    int A1[ ] = new int [B.length / 2];
    int A2[ ] = new int [B.length – A1.length];
    int j = 0, k = 0;
    for(j = 0; j<= B.length/2 –1 ; j++)
    {
        A1[j] = B[j];
    }
    System.out.print("\n       The Array A1 \n") ;
    fnDisplay(A1) ;
    for(k = 0; k<= A2.length–1 ; k++)
    {
        A2[k] = B[j++];
    }
    System.out.print("\n    The Array A2 \n") ;
    fnDisplay(A2) ;
    fnMergeSort(A1) ;
    fnMergeSort(A2) ;
    Merge(A1, A2, B);
    System.out.print("\n              The Merged Array \n") ;
    fnDisplay(B) ;
}
```

```
    void Merge ( int A1[], int A2[] , int B[])
    {
        int cA1=0, cA2=0, cB = 0;
        while( cA1 < A1.length && cA2 < A2.length )
        {
            if( A1[cA1] < A2[cA2])
            {
                B[cB++] = A1[cA1++];
            }
            else
            }
                B[cB++] = A2[cA2++];
            }
        }
        if( cA1 == A1.length )
        {
            while(cA2 < A2.length)
            {
                B[cB++] = A2[cA2++];
            }
        }
        if( cA2 == A2.length )
        {
            while(cA1 < A1.length)
            {
                B[cB++] = A1[cA1++];
            }
        }
    }
}
/*
Output

 The Array is

9  47  15  60  95  28  60  55  68  92

The Array after Merge Sort is

9  15  28  47  55  60  60  68  92  95

*/
```

Exercise

1. The following codes use recursive functions. Choose the correct answer of the question given below the code.

Code I.

```
public void Recur(int n)
{
    if ( n>1 )
    {
        System.out.print ( n + "  " ) ;
        if ( n%2 != 0 )
        {
            n = 3 * n  + 1 ;
            System.out.print ( n + "  " ) ;
        }
        Recur ( n/2 ) ;
    }
}
```

- What will be the output if n = 10 ?

(a) 10 31 15 7 3

(b) 5 16 8 4 2

(c) 10 5 16 8 4 2

Code II.

```
public void wit(String n, int p)
{
    if ( p < 0 )
        System.out.println();
    else
    {
        System.out.print ( n.charAt(p) + " . " ) ;
        wit(n,p–1);
        System.out.print ( n.charAt(p) + " – " ) ;
    }
}
```

- What will be the output if n = "SOLAR", p = 4 ?

(a) S . O . L . A . R .
R – O – L – A – S -

(b) R . A . L . O . S .
S - O - L - A - R -

(c) R - A - L - O - S -
S . O . L . A . R .

Code III.

```
void Focus(int cn)
{
    if (cn ==1 )
        System.out.println ( "\n – – – – – – " ) ;
    else
    {
        System.out.print (" + " + cn ) ;
        Focus (– – cn ) ;
        System.out.print( " = " + cn );
    }
}
```

- What will be the output if cn = 5 ?
  (a) + 1 + 2 + 3 + 4
  – – – – – –
  = 5 = 4 = 3 = 2
  (b) + 5 + 4 + 3 + 2
  – – – – – –
  = 1 = 2 = 3 = 4
  (c) + 5 = 4 + 3 = 2
  – – – – – –
  = 1 + 2 = 3 + 4

Code IV.

```
public void Number1 ( int n )
{
    if ( n>0 )
    {
        System.out.print ( n + " " ) ;
        Number1 ( n–2 ) ;
        System.out.print ( n + " " ) ;
    }
    System.out.println() ;
}
```

- What will be the output if n = 5 ?
  (a) 5
  3
  1
  1 3 5
  (b) 5 3 1
  1
  3
  5
  (c) 5 3 1
  1 3 5

Code V.

```
public String Number2 ( int n )
{
   if ( n<=0 )
      return " K ";
   return( Number2 (n–1) + n + " _ " );
}
```

- What will be the return if n = 5 ?
  (a) K 5_4_3_2_1_
  (b) 5_4_3_2_1_ K
  (c) K 1_2_3_4_5_

Code VI.

```
void Plane ( int n )
{
   System.out.print(" _ " + n ) ;
   if (n <= 0)
      System.out.println();
   else if (n%2 != 0)
      Plane (n–2 );
   else
      Plane (n–1 );
}
```

- What will be the output if n = 7 ?
  (a) z 7 z 6 z 5 z 4 z 3 z 2 z 1 z 0
  (b) z 7 z 5 z 3 z 1 z –1
  (c) z 6 z 4 z 2 z 0

Code VII.

```
void call_Verify(int n)
{
   System.out.print (" call_Verify for " + n + " is " + Verify (n,2) ) ;
}
boolean Verify(int n, int f)
{
   if (n == f )
      return true ;
   if (n%f == 0 || n == 1 )
      return false ;
   else
      return ( Verify ( n, f+1 ) ) ;
}
```

- What will happen with
  (a) call_Verify (19)
  (b) Verify (19, 1)
  (c) call_Verify (100)
  (d) Verify (29, 2)
- What is the Aim of the combined methods?