RECURSION

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. Examples of Recursive algorithms: Merge Sort, Quick Sort, Tower of Hanoi, Fibonacci Series, Factorial Problem, etc

PROPERTIES OF RECURSION

- Performing the same operations multiple times with different inputs
- In every step, we try smaller inputs to make the problem smaller
- Base condition is needed to stop the recursion otherwise infinite loop will occur.

Base case is used to terminate the recursive function when the case turns out to be true.

A Mathematical Interpretation

Let us consider a problem that a programmer has to determine the sum of first n natural numbers, there are several ways of doing that but the simplest approach is simply to add the numbers starting from 1 to n. So the function simply looks like this,

approach(1) – Simply adding one by one

$f(n) = 1 + 2 + 3 + \ldots\ldots + n$

But there is another approach to solving this

approach(2) – Recursive adding

$f(n) = 1 \qquad n=1$

$f(n) = n + f(n-1) \quad n>1$

There is a simple difference between the approach (1) and approach(2) and that is in approach(2) the function " f( ) " itself is being called inside the function, so this phenomenon is named recursion, and the function containing recursion is called recursive function.

How are Recursive functions stored in memory

Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished.

What is base condition in recursion

In the recursive program, the solution to the base case is provided and the solution to the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
    if (n < = 1) // base case
        return 1;
    else
        return n*fact(n-1);
}
```

In the above example, the base case for n < = 1 is defined and the larger value of a number can be solved by converting to a smaller one till the base case is reached.

How a particular problem is solved using recursion

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial

n if we know the factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

What is the difference between direct and indirect recursion?

A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function, say fun_new and fun_new calls fun directly or indirectly. Example:

```
// An example of direct recursion
void directRecFun()
{
    // Some code....

    directRecFun();

    // Some code...
}

// An example of indirect recursion
void indirectRecFun1()
{
    // Some code...

    indirectRecFun2();

    // Some code...
}
void indirectRecFun2()
{
    // Some code...

    indirectRecFun1();

    // Some code...
}
```

What is the difference between tail and non-tail recursion?

A recursive function is tail recursive when a recursive call is the last thing executed by the function.

How memory is allocated to different function calls in recursion

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is deallocated and the process continues.

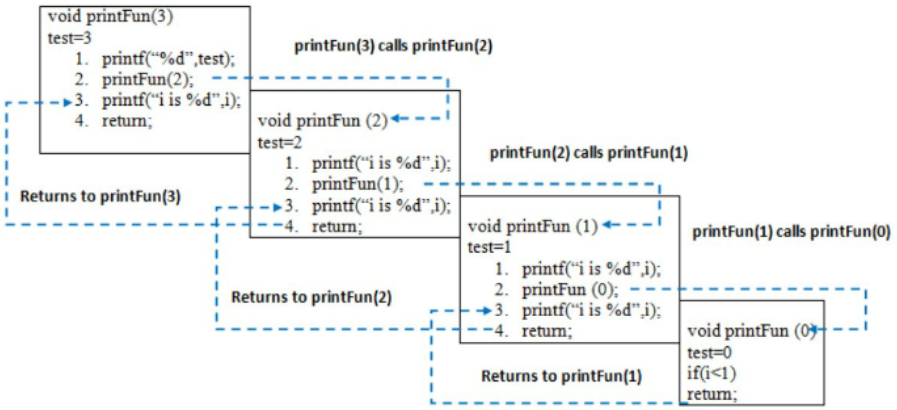
Let us take the example of how recursion works by taking a simple function.

```
using System;

class GFG {

    // function to demonstrate
    // working of recursion
    static void printFun(int test)
    {
        if (test < 1)
            return;
        else {
            Console.Write(test + " ");

            // statement 2
            printFun(test - 1);

            Console.Write(test + " ");
            return;
        }
    }

    // Driver Code
    public static void Main(String[] args)
    {
        int test = 3;
        printFun(test);
    }
}
```

Output:

3 2 1 1 2 3

When printFun(3) is called from main(), memory is allocated to printFun(3) and a local variable test is initialized to 3 and statements 1 to 4 are pushed on the stack as shown in the diagram below. It first prints '3'. In statement 2, printFun(2) is called and memory is allocated to printFun(2) and a local variable test is initialized to 2 and statements 1 to 4 are pushed into the stack. Similarly, printFun(2) calls printFun(1) and printFun(1) calls printFun(0). printFun(0) goes to the if statement and it returns to printFun(1). The remaining statements of printFun(1) are executed and it returns to printFun(2) and so on. In the output, values from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in the diagram below.



Recursion vs Iteration

RECURSION	ITERATION
Terminates when the base becomes true	Terminates when the condition becomes false
Used with functions	Used with loops
Every recursive call needs extra space in the stack memory	Every iteration does not require any extra space
Smaller code size	Larger code size

PRACTICAL PROBLEMS ON RECURSION

Problem 1: Write a program and recurrence relation to find the Factorial of n where n>2 .

Mathematical Equation:

```
1 if n == 0 or n == 1;
f(n) = n*f(n-1) if n> 1;
```

Recurrence Relation:

```
T(n) = 1 for n = 0
T(n) = 1 + T(n-1) for n > 0
```

Recursive Program: …..**Input:** n = 5 …. **Output:** factorial of 5 is: 120

Implementation:

```
// C# code to implement factorial
using System;
class GFG {

  // Factorial function
  static int f(int n)
  {
    // Stop condition
    if (n == 0 || n == 1)
      return 1;

    // Recursive condition
    else
      return n * f(n - 1);
  }

  // Driver code
  static void Main()
  {
    int n = 5;
    Console.WriteLine("factorial of " + n + " is: " + f(n));
  }
}

// This code is contributed by divyeshrabadiya07.
```

Output

Factorial of 5 is: 120

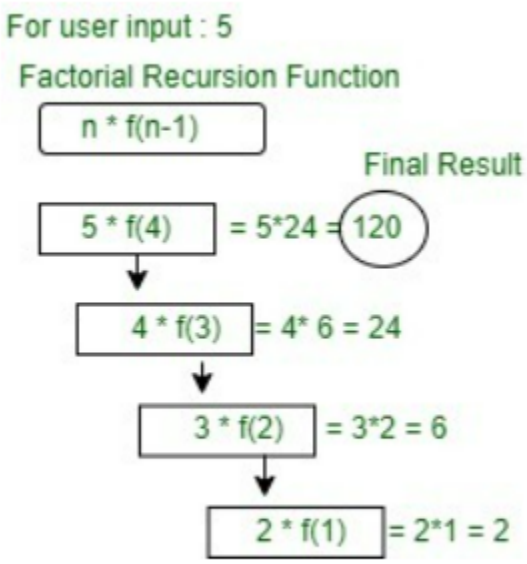


Diagram of factorial Recursion function for user input 5.

Problem 2: Printing numbers from n to 1 in decreasing order

Pseudocode:

```
void countdown(int n)
{
    if(n < 1 ) // Base Case
    {
        return
    }
    print(n)
    countdown(n-1) // Recursive
}
```

Problem 3: Explain the following function:

```
void fun2(int n)
{
if(n == 0)
    return;

fun2(n/2);
Console.Write(n%2);
}
```

Answer: The function fun2() prints the binary equivalent of n. For example, if n is 21 then fun2() prints 10101.

Problem 4: Predict the output of the following program:

```
using System;

class GFG{

    static void fun(int x)
    {
        if(x > 0)
        {
            fun(--x);
            Console.Write(x + " ");
            fun(--x);
        }
    }

    static public void Main ()
    {
        int a = 4;
        fun(a);
    }
}
// This code is contributed by SHUBHAMSINGH10
```

Output:

0 1 2 0 3 0 1

Problem 4: Write a program to calculate the sum of numbers from 1 to n using recursion.

```c
#include<stdio.h>
int  numPrint(int);
int main()
{
    int n = 1;
    printf("\n\n Recursion : print first 50 natural numbers :\n");
    printf("-------------------------------------------------\n");
    printf(" The natural numbers are :");
    numPrint(n);
    printf("\n\n");
    return 0;
}
int numPrint(int n)
{
    if(n<=50)
    {
         printf(" %d ",n);
         numPrint(n+1);
    }
}
```

Sample Output

```
Recursion : print first 50 natural numbers :
-------------------------------------------------
 The natural numbers are : 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23
24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
```

Problem 5: Write a program in C to count the digits of a given number using recursion.

Test Data :

Input a number : 50

Expected Output :

The number of digits in the number is : 2

Solution:

```c
#include<stdio.h>

int noOfDigits(int n1);
int main()
{
  int n1,ctr;
    printf("\n\n count the digits of a given number :\n");
    printf("-----------------------------------------\n");
    printf(" Input  a number : ");
    scanf("%d",&n1);

    ctr = noOfDigits(n1);

    printf(" The number of digits in the number is :  %d \n\n",ctr);
    return 0;
}

int noOfDigits(int n1)
{
    static int ctr=0;

     if(n1!=0)
     {
          ctr++;
         noOfDigits(n1/10);
    }

    return ctr;
}
```

FIBONACCI SERIES

In simple terms, Fibonacci series is a series of numbers in which each number (*Fibonacci number*) is the sum of the two preceding numbers. The simplest is the series 1, 1, 2, 3, 5, 8, etc.

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

With seed values

$$F_0 = 0 \text{ and } F_1 = 1.$$

So a Fibonacci series can look like this −

F8 = 0 1 1 2 3 5 8 13

or, this −

F8 = 1 1 2 3 5 8 13 21

Algorithm

Algorithm of this program is very easy:

```
START
   Step 1 → Take integer variable A, B, C
   Step 2 → Set A = 0, B = 0
   Step 3 → DISPLAY A, B
   Step 4 → C = A + B
   Step 5 → DISPLAY C
   Step 6 → Set A = B, B = C
   Step 7 → REPEAT from 4 - 6, for n times
STOP
```

Pseudocode

```
procedure fibonacci : fib_num

   IF fib_num less than 1
      DISPLAY 0

   IF fib_num equals to 1
      DISPLAY 1

   IF fib_num equals to 2
      DISPLAY 1, 1

   IF fib_num greater than 2
      Pre = 1,
      Post = 1,

      DISPLAY Pre, Post
      FOR 0 to fib_num-2
         Fib = Pre + Post
         DISPLAY Fib
         Pre = Post
         Post = Fib
      END FOR
   END IF

end procedure
```

Implementation

Implementation of this algorithm in c is given below:

```c
#include <stdio.h>

int main() {
   int a, b, c, i, n;

   n = 4;

   a = b = 1;

   printf("%d %d ",a,b);

   for(i = 1; i <= n-2; i++) {
      c = a + b;
      printf("%d ", c);

      a = b;
      b = c;
   }

   return 0;
}
```

Other Test Cases:

```
Input  : n = 2
Output : 1

Input  : n = 9
Output : 34
```

Implementation of example above :

Write a function int fib(int n) that returns Fn. For example, if n = 0, then fib() should return 0. If n = 1, then it should return 1. For n > 1, it should return Fn-1 + Fn-2

Test data:

```
For n = 9
Output:34
```

The following are different methods to get the nth Fibonacci number.

Method 1 (Use recursion)

A simple method that is a direct recursive implementation of mathematical recurrence relation is given above.

```
// C# program for Fibonacci Series
// using Recursion
using System;

public class GFG
{
    public static int Fib(int n)
    {
        if (n <= 1)
        {
            return n;
        }
        else
        {
            return Fib(n - 1) + Fib(n - 2);
        }
    }

    // driver code
    public static void Main(string[] args)
    {
        int n = 9;
        Console.Write(Fib(n));
    }
}

// This code is contributed by Sam007
```

Output:

34

Method 2: (Use Dynamic Programming)

We can avoid the repeated work done in method 1 by storing the Fibonacci numbers calculated so far.

```
<script>

// Fibonacci Series using Dynamic Programming

    function  fib(n)
    {
        /* Declare an array to store Fibonacci numbers. */
        let f = new Array(n+2); // 1 extra to handle case, n = 0
        let i;
        /* 0th and 1st number of the series are 0 and 1*/
        f[0] = 0;
        f[1] = 1;
        for (i = 2; i <= n; i++)
        {
            /* Add the previous 2 numbers in the series
            and store it */
            f[i] = f[i-1] + f[i-2];
        }
        return f[n];
    }
    let n=9;
    document.write(fib(n));

    // This code is contributed by avanitrachhadiya2155

</script>
```

Output: 34

Method 3: (Space Optimized Method 2)

We can optimize the space used in method 2 by storing the previous two numbers only because that is all we need to get the next Fibonacci number in series.

```
using System;

namespace Fib
{
    public class GFG
    {
        static int Fib(int n)
        {
            int a = 0, b = 1, c = 0;

            // To return the first Fibonacci number
            if (n == 0) return a;

            for (int i = 2; i <= n; i++)
            {
                c = a + b;
                a = b;
                b = c;
            }

            return b;
        }

    // Driver function
    public static void Main(string[] args)
        {

            int n = 9;
            Console.Write("{0} ", Fib(n));
        }
    }
}
```

Output: 34

To display fibonacci series using for loop:

```
class Main {
  public static void main(String[] args) {

    int n = 10, firstTerm = 0, secondTerm = 1;
    System.out.println("Fibonacci Series till " + n + " terms:");

    for (int i = 1; i <= n; ++i) {
      System.out.print(firstTerm + ", ");

      // compute the next term
      int nextTerm = firstTerm + secondTerm;
      firstTerm = secondTerm;
      secondTerm = nextTerm;
    }
  }
}
```

Output:

```
Fibonacci Series till 10 terms:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
```

In the above program, firstTerm and secondTerm are initialized with 0 and 1 respectively (first two digits of Fibonacci series).

Here, we have used the for loop to

- print the firstTerm of the series
- compute nextTerm by adding firstTerm and secondTerm
- assign value of secondTerm to firstTerm and nextTerm to secondTerm

To display fibonacci series using while loop:

```
class Main {
  public static void main(String[] args) {

    int i = 1, n = 10, firstTerm = 0, secondTerm = 1;
    System.out.println("Fibonacci Series till " + n + " terms:");

    while (i <= n) {
      System.out.print(firstTerm + ", ");

      int nextTerm = firstTerm + secondTerm;
      firstTerm = secondTerm;
      secondTerm = nextTerm;

      i++;
    }
  }
}
```

Output:

```
Fibonacci Series till 10 terms:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
```

The working of this program is the same as the previous program.

And, though both programs are technically correct, it is better to use a for loop in this case. It's because the number of iterations (from 1 to n) is known.

To display fibonacci series upto a given number:

```
class Fibonacci {
  public static void main(String[] args) {

    int n = 100, firstTerm = 0, secondTerm = 1;

    System.out.println("Fibonacci Series Upto " + n + ": ");

    while (firstTerm <= n) {
      System.out.print(firstTerm + ", ");

      int nextTerm = firstTerm + secondTerm;
      firstTerm = secondTerm;
      secondTerm = nextTerm;

    }
  }
}
```

Output:

```
Fibonacci Series Upto 100:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,
```

In this example, instead of displaying the Fibonacci series of a certain number, we are displaying the series up to the given number (100).

For this, we just need to compare the firstTerm with n. And, if firstTerm is less than n, it is printed in the series. Else, the series is completed.