Recursion-Class Notes

By Dr GC Jana

Introduction to Recursion

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

Using a recursive algorithm, certain problems can be solved quite easily.

Examples of such problems are **Towers of Hanoi (TOH)**, **Inorder/Preorder/Postorder Tree Traversals**, **DFS of Graph**, etc.

Need of Recursion:

- 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.
- It has certain advantages over the iteration technique.
- A task that can be defined with its similar subtask, recursion is one of the best solutions for it. For example; The Factorial of a number.

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.

Principles of Recursion

- Recursion is a programming technique where a function calls itself to solve a problem. It breaks down a problem into smaller subproblems, solving each recursively until a base case is reached. The key principles of recursion are:

➔ **Base Case:** The condition that stops the recursion. Without a base case, the function would call itself indefinitely.

➔ **Recursive Case:** The part where the function calls itself, breaking down the problem into smaller instances of the same problem.

➔ **Termination:** Every recursive function must ensure it eventually reaches the base case, otherwise, it leads to infinite recursion and program failure.

➔ **Stack Usage:** Each recursive call adds a frame to the call stack. When a base case is reached, the stack begins to unwind as each recursive call completes.

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.
- ➔ The recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure.

Algorithm: Steps

The algorithmic steps for implementing recursion in a function are as follows:

Step1 - Define a base case: Identify the simplest case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.

Step2 - Define a recursive case: Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.

Step3 - Ensure the recursion terminates: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.

step4 - Combine the solutions: Combine the solutions of the subproblems to solve the original problem.

A Mathematical Interpretation

Let us consider a problem that a programmer has to determine the sum of first n natural numbers.

Approach-1: – Simply adding one by one

f(n) = 1 + 2 + 3 +……..+ n

Approach-2: - Recursive adding

f(n) = 1 n=1

f(n) = n + f(n-1) n>1

What is the base condition in recursion?

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

```
        return n*fact(n-1);
    }
```

Why Stack Overflow error occurs in recursion?

If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
   // wrong base case (it may cause
   // stack overflow).
   if (n == 100)
      return 1;

   else
      return n*fact(n-1);
}
```

If **fact(10)** is called, it will call **fact(9), fact(8), fact(7)**, and so on but the number will **never reach 100**. So, the **base case is not reached**. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

What is the difference between direct and indirect recursion?

```
// 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...
}
```

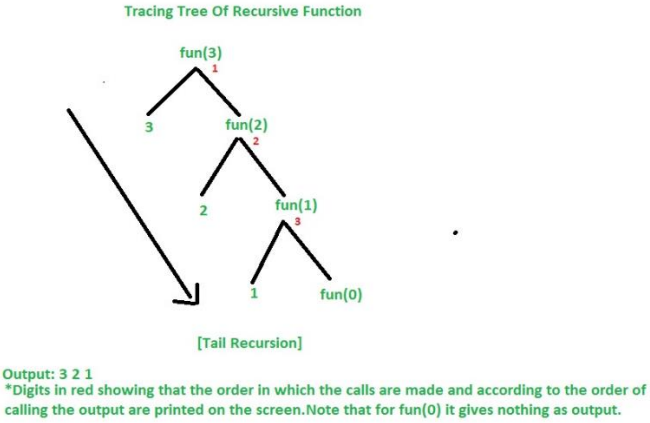
1. Direct Recursion: These can be further categorized into **four types**:

➔ **Tail Recursion:** If a recursive function calling itself and that recursive call is the last statement in the function then it's known as Tail Recursion. After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling and it does nothing at returning time

// Java code Showing Tail Recursion

```
class GFG {
// Recursion function
static void fun(int n)
{
        if (n > 0)
        {
        System.out.print(n + " ");
        // Last statement in the function
        fun(n - 1);
        }
}
// Driver Code
public static void main(String[] args)
{
        int x = 3;
        fun(x);
}
}
```

Output: 3 2 1

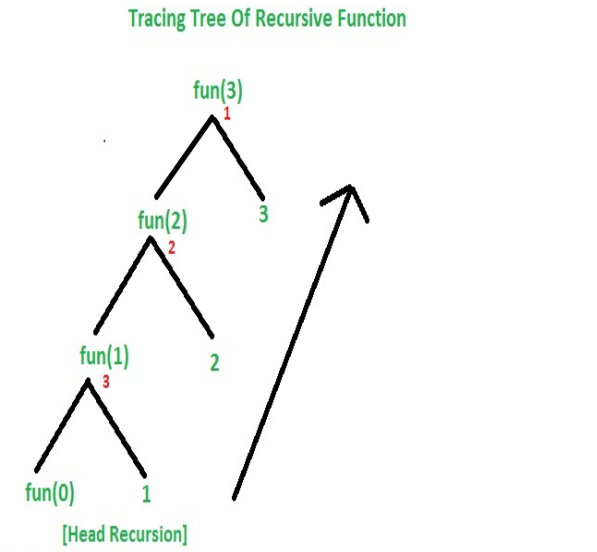


➔ **Head Recursion:** If a recursive function calling itself and that recursive call is the first statement in the function then it's known as Head Recursion. There's no statement, no operation before the call. The function doesn't have to process or perform any operation at the time of calling and all operations are done at returning time.

// Java program showing Head Recursion

```
import java.io.*;

class GFG{

// Recursive function
static void fun(int n)
{
        if (n > 0) {

                // First statement in the function
                fun(n - 1);

                System.out.print(" "+ n);
        }
}

// Driver code
public static void main(String[] args)
{
        int x = 3;
        fun(x);

}
}
```

Output: 1 2 3



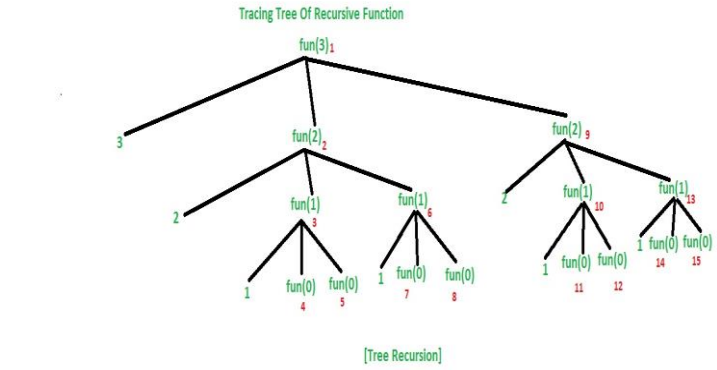
Output: 1 2 3

*Digits in red showing that the order in which the calls are made and note that printing done at returning time. And it does nothing at calling time.

➔ **Tree Recursion:** To understand Tree Recursion let's first understand Linear Recursion. If a recursive function calling itself for one time then it's known as Linear Recursion. Otherwise if a recursive function calling itself for more than one time then it's known as Tree Recursion.

// Java program to show Tree Recursion

```
class TRE
{

// Recursive function
static void fun(int n)
{
        if (n > 0) {
                System.out.print(" "+ n);

                // Calling once
                fun(n - 1);

                // Calling twice
                fun(n - 1);
        }
}
// Driver code
public static void main(String[] args)
{

        fun(3);
        }
}
```



[Tree Recursion]

Output: 3 2 1 1 2 1 1

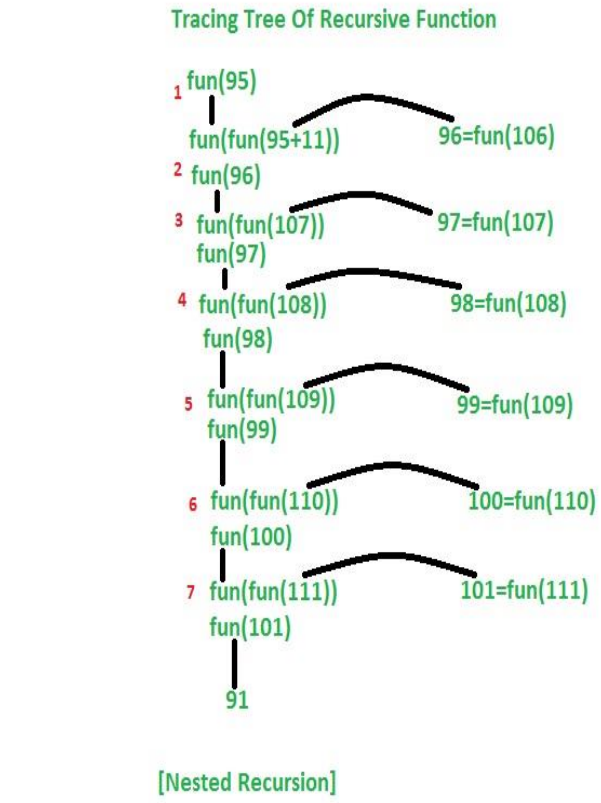
*Digits in red showing that the order in which the calls are made and according to the order of calling the output are printed on the screen.Note that for fun(0) it gives nothing as output.

Output: 3 2 1 1 2 1 1

➔ **Nested Recursion:** In this recursion, a recursive function will pass the parameter as a recursive call. That means “recursion inside recursion”. Let see the example to understand this recursion.

```
// Java program to show Nested Recursion
import java.util.*;
class NERE {
static int fun(int n)
{
        if (n > 100)
                return n - 10;
        // A recursive function passing parameter
        // as a recursive call or recursion
        // inside the recursion
        return fun(fun(n + 11));
}
// Driver code
public static void main(String args[])
{
        int r;
        r = fun(95);
        System.out.print(" "+ r);

}
}
```

Output: 91



Output: 91

*Digits in red showing that the order in which the calls are made

Fibonacci Numbers using Recursion in Java and C++

Java:

```
public class Fibonacci {
   // Recursive method to find the nth Fibonacci number
   public static int fibonacci(int n) {
      if (n == 0) {
         return 0;  // Base case: F(0) = 0
      } else if (n == 1) {
```

```
        return 1;  // Base case: F(1) = 1
    } else {
        // Recursive call for F(n-1) + F(n-2)
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
  }
  public static void main(String[] args) {
    int n = 10;  // Example: Find the 10th Fibonacci number
    System.out.println("Fibonacci of " + n + " is " + fibonacci(n));
  }
}
```

CPP:

```
#include <iostream>
using namespace std;
// Recursive function to find the nth Fibonacci number
int fibonacci(int n) {
  if (n == 0)
    return 0;  // Base case: F(0) = 0
  else if (n == 1)
    return 1;  // Base case: F(1) = 1
  else
    // Recursive call for F(n-1) + F(n-2)
    return fibonacci(n - 1) + fibonacci(n - 2);
}
int main() {
```

```
    int n = 10;  // Example: Find the 10th Fibonacci number
    cout << "Fibonacci of " << n << " is " << fibonacci(n) << endl;
    return 0;
}
```

Computing a^n (Power of a Raised to n) Using Recursion in Java and C++

Java

```
public class Power {
    // Recursive method to calculate a^n
    public static int power(int a, int n) {
        if (n == 0) {
            return 1;  // Base case: a^0 = 1
        } else {
            // Recursive call for a * a^(n-1)
            return a * power(a, n - 1);
        }
    }
    public static void main(String[] args) {
        int a = 2, n = 5;  // Example: 2^5
        System.out.println(a + " raised to the power " + n + " is " + power(a, n));
    }
}
```

CPP

```
#include <iostream>
using namespace std;
// Recursive function to calculate a^n
int power(int a, int n) {
    if (n == 0)
        return 1;  // Base case: a^0 = 1
    else
```

```
    // Recursive call for a * a^(n-1)
    return a * power(a, n - 1);
}
int main() {
  int a = 2, n = 5;  // Example: 2^5
  cout << a << " raised to the power " << n << " is " << power(a, n) << endl;
  return 0;
}
```

Reverse of a String using Recursion in Java and C++

Java

```
public class ReverseString {
  // Recursive method to reverse a string
  public static String reverse(String str) {
    if (str.isEmpty()) {
      return str;  // Base case: return empty string
    } else {
      // Recursive call: reverse the substring and append the first character
      return reverse(str.substring(1)) + str.charAt(0);
    }
  }
  public static void main(String[] args) {
    String str = "hello";  // Example string
    System.out.println("Reversed string: " + reverse(str));
  }
}
```

CPP

```
#include <iostream>
using namespace std;
// Recursive function to reverse a string
string reverse(string str) {
```

```
    if (str.length() == 0) {
        return str;  // Base case: return empty string
    } else {
        // Recursive call: reverse the substring and append the first character
        return reverse(str.substr(1)) + str[0];
    }
}
int main() {
    string str = "hello";  // Example string
    cout << "Reversed string: " << reverse(str) << endl;
    return 0;
}
```

References:

[1] https://www.geeksforgeeks.org/introduction-to-recursion-2/

[2] Introduction to Algorithms, by Corman