Tutorial Link https://codequotient.com/tutorials/Recursion - Tail & Non-tail/5a014b34cbb2fe34b7775059

TUTORIAL

Recursion - Tail & Non-tail

Chapter

1. Recursion - Tail & Non-tail

Topics

1.4 Code Implementation

Order of execution will only tell the execution sequence of statements. Now another aspect is weather after calling the function recursively the caller can end its own copy or it have to be alive till the end of recursive call to do some work. This aspect is called tail or non-tail recursion. This is probably the most wrongly interpreted thing in recursion. Consider the factorial function again in consideration. While factorial function is making recursive call, it will wait for the recursive call to end and when the recursive call ends it will use the value returned by the call to make its execution complete. So the first call will wait for subsequent calls to terminate and goes to a kind of sleep mode till their completion. This is called non-tail recursion or augmenting recursion. Its counterpart is tail recursion, like binary search. There we first find the element in whole array, then if not found then we will try to find the element in either half of the array. The first call will not wait for the next call to return. Whatever call in recursion stack will find the element, whole search will terminate at that point. For example, let's compare the factorial function and GCD calculator in recursion.

We already treated factorial function above, so let's study the GCD function now. The eucledian algorithm which computes the greatest

common divisor (GCD) of two integers can be written recursively as below: -

```
Gcd(x, y)       = x if y=0
                    = Gcd(y, remainder(x,y))    if y > 0
```

For example,

```
gcd(111, 259)   = gcd(259, 111% 259)                    =
gcd(259, 111)
                = gcd(111, 259% 111)                = gcd(111, 37)
                = gcd(37, 111% 37)                      = gcd(37,
0)
                = 37
```

So gcd of 111 and 259 is 37. Now following are the recursive implementations of these two functions: -

GCD

```
//INPUT: Integers x, y such that x >= y and y > 0
int gcd(int x, int y) {
    if (y == 0)
        return x;
    else
        return gcd(y, x % y);
}
```

Execution sequence

gcd(18, 4) = gcd(4, 18%4) = gcd(4, 2)
= gcd(2, 4%2) = gcd(2,0)
= 2

Factorial

```
//INPUT: n is an Integer such that n >= 0
int fact(int n) {
    if (n == 0)
        return 1;
    else
        return n * fact(n - 1);
}
```

Execution Sequence

Fact(4) = 4 * fact(3)
= 4 * (3 * fact(2))
= 4 * (3 * (2 * fact(1)))
= 4 * (3 * (2 * (1 * fact(0))))
= 4 * (3 * (2 * (1 * 1)))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24

Tail-recursive functions are functions in which all recursive calls are tail calls and hence do not build up any deferred operations. For example, the gcd function is tail-recursive. In contrast, the factorial function is not tail-recursive; because its recursive call is not in tail position, it builds up deferred multiplication operations that must be performed after the final recursive call completes. From the execution sequence of these two functions, see the recursive call behavior, the gcd function just call itself recursively with new parameters and finished its execution, it will not wait for the recursive call to complete first, where as in case of factorial the recursive calls will return the values to the parent calls, which then use these values and then finally terminates. So if the recursive call is the last thing done by the functions it is called tail recursion (gcd above), and if the recursive calls needs to be present in the stack till the execution of recursion, (factorial above) is called augmenting or non-tailed recursion.

Following program will use both these functions: -

Code Implementation

Javascript

```
function gcd(x,y){
    if(y == 0)
        return x
    return gcd(y,x%y)
}

function fact_recursive(num){
    let result;
    if (num == 0)               // Base case for recursion.
        return 1;     // fact() return 1 if argument is 0
    else{
        result = num * fact_recursive(num-1);      // Call recursively with lesser number.
        return result;
    }
}

function main(){
    let number, fact1;
    number = 5;
    fact1 = fact_recursive(number);        // Call the Recursive version
    console.log(`Number=${number}`);
    console.log(`Recursive_Factorial=${fact1}`);
    let num1 = 111,num2 = 259
    console.log(`GCD of ${num1} & ${num2} = ${gcd(num1,num2)}`)
}

main()
```

C

```
#include<stdio.h>

int gcd(int x, int y) {
  if (y == 0)
    return x;
  else
```

```
    return gcd(y, x % y);
}

int fact_recursive(int num)
{
  if (num == 0)         // Base case for recursion.
    return 1;     // fact() return 1 if argument is 0
  else
    return num * fact_recursive(num-1);      // Call recursively with lesser number.
}
int main()
{
  int number, fact1;
  int num1, num2;
  number = 5;
  fact1 = fact_recursive(number);        // Call the Recursive version
  printf("Number=%d\n", number);
  printf("Recursive_Factorial=%d\n", fact1);

  num1= 111;
  num2 = 259;
  printf("GCD of %d & %d = %d\n", num1, num2, gcd(num1, num2));
  return 0;
}

```

Java

```
class Main{
    static int gcd(int x,int y){
        if(y == 0)
            return x;
        return gcd(y,x%y);
    }

    static int fact_recursive(int num){
        int result;
        if (num == 0)           // Base case for recursion.
            return 1;     // fact() return 1 if argument is 0
        else{
```

```
            result = num * fact_recursive(num-1);
// Call recursively with lesser number.
            return result;
        }
    }

    public static void main(String[] args){
        int number, fact1;
        number = 5;
        fact1 = fact_recursive(number);        // Call
the Recursive version
        System.out.println("Number=" + number);
        System.out.println("Recursive_Factorial=" +
fact1);
        int num1= 111;
        int num2 = 259;
        System.out.printf("GCD of %d & %d = %d\n",
num1, num2, gcd(num1, num2));
    }
}
```

Python 3

```

def gcd(x,y):
    if y == 0:
        return x
    return gcd(y,x%y)

def fact_recursive(num):
  if (num == 0):                # Base case for
recursion.
    return 1                    # fact() return 1 if
argument is 0
  else:
    result = num * fact_recursive(num-1);      # Call
recursively with lesser number.
    return result

if __name__ == '__main__':
    number = 5
    fact1 = fact_recursive(number)        # Call the
Recursive version
    print("Number=",number)
    print("Recursive_Factorial=",str(fact1))

```

```
    num1 = 111;num2 = 259
    print('GCD of',num1,'&',num2,'=',gcd(num1,num2))
```

C++

```
#include<iostream>
using namespace std;
int gcd(int x, int y) {
  if (y == 0)
    return x;
  else
    return gcd(y, x % y);
}

int fact_recursive(int num)
{
  if (num == 0)         // Base case for recursion.
    return 1;     // fact() return 1 if argument is 0
  else
    return num * fact_recursive(num-1);      // Call
recursively with lesser number.
}
int main()
{
  int number, fact1;
  int num1, num2;
  number = 5;
  fact1 = fact_recursive(number);        // Call the
Recursive version
  cout<<"Number="<<number<<endl;
  cout<<"Recursive_Factorial="<<fact1<<endl;

  num1= 111;
  num2 = 259;
  cout<<"GCD of "<<num1<<" & "<<num2 <<" = "
<<gcd(num1,num2)<<endl;
  return 0;
}

```

The significance of tail recursion is that when making a tail-recursive call (or any tail call), the caller's return position need not be saved on the call stack; when the recursive call returns, it will branch directly on the previously saved return position. Therefore, in languages that

recognize this property of tail calls, tail recursion saves both space and time.