Recursion

Recursion

The term Recursion can be defined as the process of defining something in terms of itself. In simple words, it is a process in which a function calls itself directly or indirectly.

Recursion

- **Recursion** is a fundamental concept in computer science.
- In all recursive concepts, there are one or more base cases.
- **Recursive [math] functions**: functions that call themselves.
- **Recursive data types**: data types that are defined using references to themselves.
- **Recursive algorithms:** algorithms that solve a problem by solving one or more smaller instances of the same problem.

Syntax:

```
def func(): <--
              |
              | (recursive call)
              |
    func() ----
```

Recursion

Recursive Definition

- int factorial(int N) {
  - if (N == 0) return 1;
    - return N*factorial(N-1);
- }

Different Types of Recursion

- **Direct Recursion**
  - A function is called direct recursive if it calls itself in its function body repeatedly. To better understand this definition, look at the structure of a direct recursive program.
- **Indirect Recursion**
  - The recursion in which the function calls itself via another function is called indirect recursion. Now, look at the indirect recursive program structure.

Different Types of Recursion

- Direct Recursion
  - int fun(int z){
    - fun(z-1); //Recursive call
  - }
- Indirect Recursion
  - int fun1(int z){
    - fun2(z-1);
    - }
  - int fun2(int y){
    - fun1(y-2)
    - }

Advantages of using recursion

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- **Reducing code duplication:** Recursive functions can help reduce code duplication by allowing a function to be defined once and called multiple times with different parameters.
- **Solving complex problems:** Recursion can be a powerful technique for solving complex problems, particularly those that involve dividing a problem into smaller subproblems.

Disadvantages of using recursion

- **Performance Overhead:** Recursive algorithms may have a higher performance overhead compared to iterative solutions. This is because each recursive call creates a new stack frame, which takes up additional memory and CPU resources. Recursion may also cause stack overflow errors if the recursion depth becomes too deep.
- **Difficult to Understand and Debug:** Recursive algorithms can be difficult to understand and debug because they rely on multiple function calls, which can make the code more complex and harder to follow.
- **Memory Consumption:** Recursive algorithms may consume a large amount of memory if the recursion depth is very deep.
- **Limited Scalability:** Recursive algorithms may not scale well for very large input sizes because the recursion depth can become too deep and lead to performance and memory issues.

Recursion

- BASE:

  1 is an odd positive integer.

- RECURSION:

  If k is an odd positive integer, then k + 2 is an odd positive integer.

  - Now, 1 is an odd positive integer by the definition base.
  - With k = 1, 1 + 2 = 3, so 3 is an odd positive integer.
  - With k = 3, 3 + 2 = 5, so 5 is an odd positive integer
  - and so, 7, 9, 11, … are odd positive integers.

- The main idea is to “reduce” a problem into smaller problems.

Recursion

$f(0) = 3$

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

- Find $f(1), f(2), f(3)$ and $f(4)$

- From the recursive definition it follows that

$f(1) = 2\,f(0) + 3 = 2(3) + 3 = 6 + 3 = 9$

$f(2) = 2\,f(1) + 3 = 2(9) + 3 = 18 + 3 = 21$

$f(3) = 2\,f(2) + 3 = 2(21) + 3 = 42 + 3 = 45$

$f(4) = 2\,f(3) + 3 = 2(45) + 3 = 90 + 3 = 93$

THE FACTORIAL OF A POSITIVE INTEGER: Example

- For each positive integer n, the factorial of n denoted as n! is defined to be the product of all the integers from 1 to n:

$$n! = n.(n-1).(n-2)\dots 3.2.1$$

- Zero factorial is defined to be 1

$$0! = 1$$

EXAMPLE:

0! = 1	1! = 1
2! = 2.1 = 2	3! = 3.2.1 = 6
4! = 4.3.2.1 = 24	5! = 5.4.3.2.1 = 120
! = 6.5.4.3.2.1= 720	7! = 7.6.5.4.3.2.1= 5040

REMARK:

$5! = 5 .4.3 . 2 . 1 \quad = 5 .(4 . 3 . 2 .1) \quad = 5 . 4!$

In general, **$n! = n(n-1)!$** for each positive integer n.

THE FACTORIAL OF A POSITIVE INTEGER: Example

- Thus, the recursive definition of factorial function F(n) is:

1. $F(0) = 1$
2. $F(n) = n\,F(n-1)$

Example

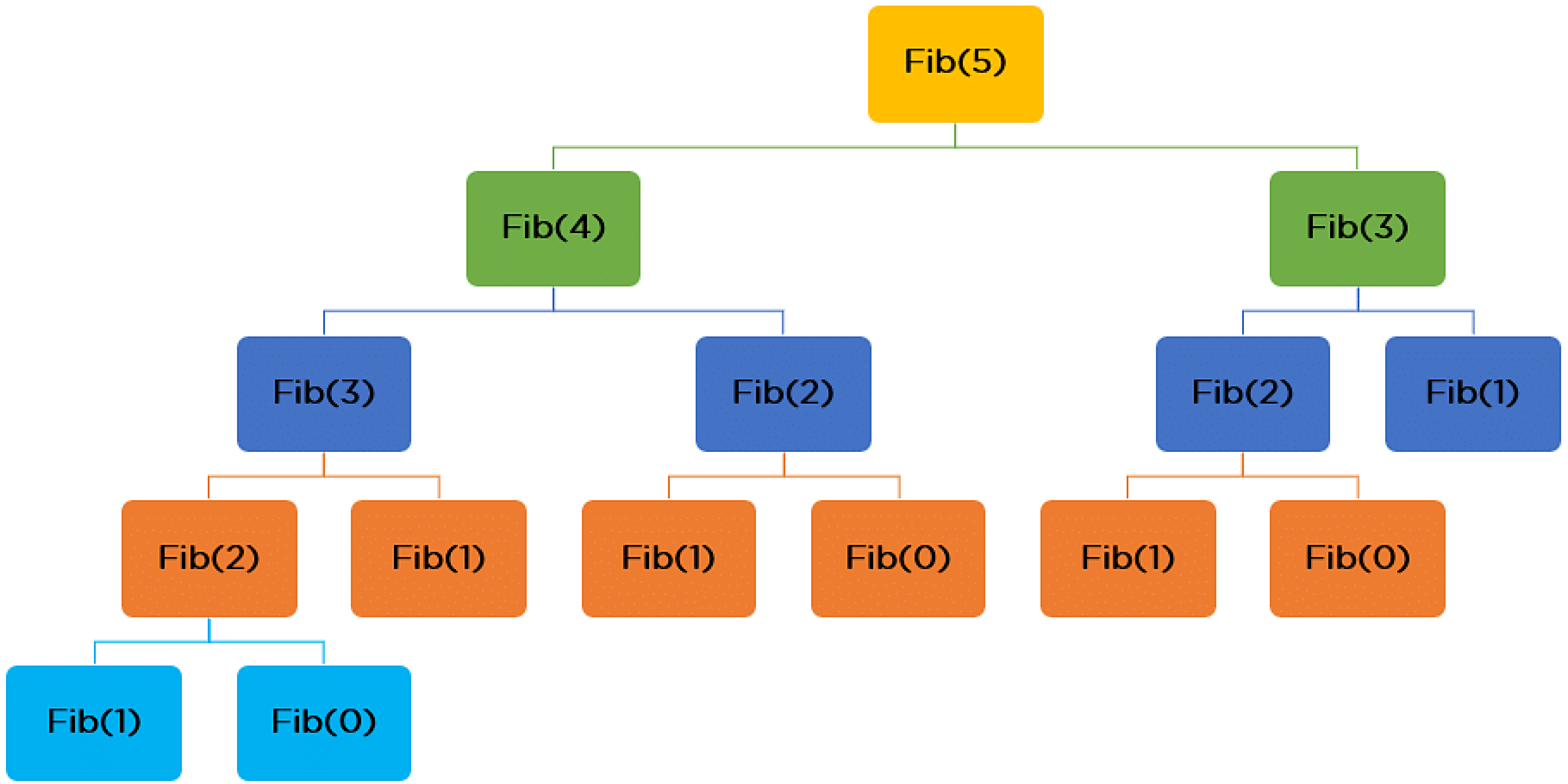
- The factorial of 6 is denoted as 6! = 1*2*3*4*5*6 = 720.

```
# Program to print factorial of a number
# recursively.

# Recursive function
def recursive_factorial(n):
  if n == 1:
      return n
  else:
      return n * recursive_factorial(n-1)

# user input
num = 6

# check if the input is valid or not
if num < 0:
  print("Invalid input ! Please enter a positive number.")
elif num == 0:
  print("Factorial of number 0 is 1")
else:
  print("Factorial of number", num, "=", recursive_factorial(num))
```

Output

```
Factorial of number 6 = 720
```

Call Stack – Computation Flow Chart



The Fibonacci numbers

- $f(0) = 0, f(1) = 1$
- $f(n) = f(n - 1) + f(n - 2)$

$f(0) = 0$

$f(1) = 1$

$f(2) = f(1) + f(0) = 1 + 0 = 1$

$f(3) = f(2) + f(1) = 1 + 1 = 2$

$f(4) = f(3) + f(2) = 2 + 1 = 3$

$f(5) = f(4) + f(3) = 3 + 2 = 5$

$f(6) = f(5) + f(4) = 5 + 3 = 8$

Example

- A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8…

```
# Program to print the fibonacci series upto n_terms

# Recursive function
def recursive_fibonacci(n):
  if n <= 1:
      return n
  else:
      return(recursive_fibonacci(n-1) + recursive_fibonacci(n-2))

n_terms = 10

# check if the number of terms is valid
if n_terms <= 0:
  print("Invalid input ! Please input a positive value")
else:
  print("Fibonacci series:")
for i in range(n_terms):
    print(recursive_fibonacci(i))
```

Output

```
Fibonacci series:
0
1
1
2
3
5
8
13
21
34
```