# MODULE 3

# RECURSIVE

## DATA STRUCTURE AND ALGORITHMS

FACULTY OF COMPUTING
UNIVERSITI TEKNOLOGI MALAYSIA

# MODULE 3: RECURSIVE

## OBJECTIVES FOR STUDENTS

1. Identify problem solving characterestics to be solved using recursive.
2. Trace the implementation of a recursive function.
3. Write recursive function to solve problems.

## KEY CONCEPT

### 1.0 INTRODUCTION TO RECURSION

1.1 **Repetitive** algorithm is a process whereby a sequence of operations is executed repeatedly until certain condition is achieved. Repetition can be implemented using loop : **while**, **for** or **do**.. **while**.

1.2 Besides repetition using loop, C++ allow programmers to implement recursive. Not all programming language allow recursive implement, e.g. Basic language.

1.3 **Recursive** is a repetitive process in which an algorithm calls itself. Recursion can be used to replace loops. Recursively defined data structures, like lists, are very well-suited to processing by recursive procedures and functions.

1.4 A recursive procedure is mathematically more elegant than one using loops. Sometimes procedures can become straightforward and simple using recursion as compared to loop solution procedure.

1.5 **Advantages of recursive** - A recursive procedure is mathematically more elegant than one using loops. Sometimes procedures that would be tricky to write using a loop are straightforward using recursion. Recursive is a powerful problem solving approach, since problem solving can be expressed in an easier and neat approach.

1.6 **Drawback of recursive** - Execution running time for recursive function is not efficient compared to loop, since every time a recursive function calls itself, it requires multiple *memories* to store the internal address of the function.

## 2.0 DESIGNING RECURSIVE ALGORITHM

2.1 **Recursive solution** - Not all problems can be solved using recursive. Problem that can be solved using recursive is a problem that can be solved by breaking the problem into smaller instances of problem, solve and combine. Every recursive definition has two parts:

- BASE CASE(S): case(s) so simple that they can be solved directly
- RECURSIVE CASE(S): more complex – make use of recursion to solve smaller sub-problems and combine into a solution to the larger problem

2.2 **Rules for designing recursive algorithm**:

- Determine the **base case** - There are one or more terminal cases whereby the problem will be solved without calling the recursive function again.
- Determine the **general case** – recursive call by reducing the size of the problem.
- Combine the base case and general case into an algorithm.

**2.3 Recursive algorithm**

```
if (terminal case is reached)  // base case
        <solve the problem>
else   // general case
        < reduce the size of the problem and
        call recursive function >
```

## 3.0 IMPLEMENTATION OF THE RECURSIVE ALGORITHMS

3.1 Classic examples of recursive algorithms:

- Multiplying numbers
- Find Factorial value.
- Fibonacci numbers

3.2 **Multiplication** of 2 numbers can be achieved by using addition method.

- Example : To multiply **8 x 3**, the result can also be achieved by adding value 8, 3 times as follows:

  **8 + 8 + 8 = 24**

- Program 3.1 shows the implementation of multiply using loop.

```
// Program 3.1
int Multiply(int M,int N)
{ for (int i=1,i<=N,i++)
    result += M;
  return result;
}//end Multiply()
```

- **Steps to solve Multiply()** problem recursively:
  - Problem size is represented by variable ***N***. In this example, problem size is 3. Recursive function will call **Multiply()** repeatedly by reducing ***N*** by 1 for each respective call.
  - Terminal case is achieved when the value of ***N*** is 1 and recursive call will stop. At this moment, the solution for the terminal case will be compted and the result is returned to the called function.
  - The simple solution for this example is represented by variable **M**. In this example, the value of ***M*** is 8.
- **Implementation** of recursive function: **Multiply()**, refer toProgram 3.2.

```
// Program 3.2
int Multiply (int M,int N)
{
 if (N==1)
  return M;
 else
  return M + Multiply(M,N-1);
}
```

3.3 Three important factors for **recursive implementation**:

- There is a condition where the function will stop calling itself. (if this condition is not fulfilled, infinite loop will occur)
- Each recursive function call, must return to the called function.
- Variable used as condition to stop the recursive call must change towards terminal case.

3.4 **Tracing** Recursive Implementation for **Multiply(8,3)**. Figure 3.1 illustrates the calling recursive function steps. **Returning** the **Multiply(8,3)** result to the called function, shown in steps in Figure 3.2.

3.5 **Factorial Problem**

- **Problem** : Get Factorial value for a positive integer number.
- **Solution** : The factorial value can be achieved as follows:

  0! is equal to 1
  1! is equal to 1 x 0! = 1 x 1 = 1
  2! is equal to 2 x 1! = 2 x 1 x 1 = 2
  3! is equal to 3 x 2! = 3 x 2 x 1 x 1 = 6
  4! is equal to 4 x 3! = 4 x 3 x 2 x 1 x 1 = 24
  N! is equal to ***N*** x (***N***-1)! For every ***N***>0

- **Solving Factorial Recursively**
  - The simple solution for this example is represented by the factorial value equal to 1.

- ***N*** represent the factorial size. The recursive process will call **factorial()** function recursively by reducing ***N*** by 1.
- Terminal case for factorial problem is when ***N*** equal to 0. The computed result is returned to called function.

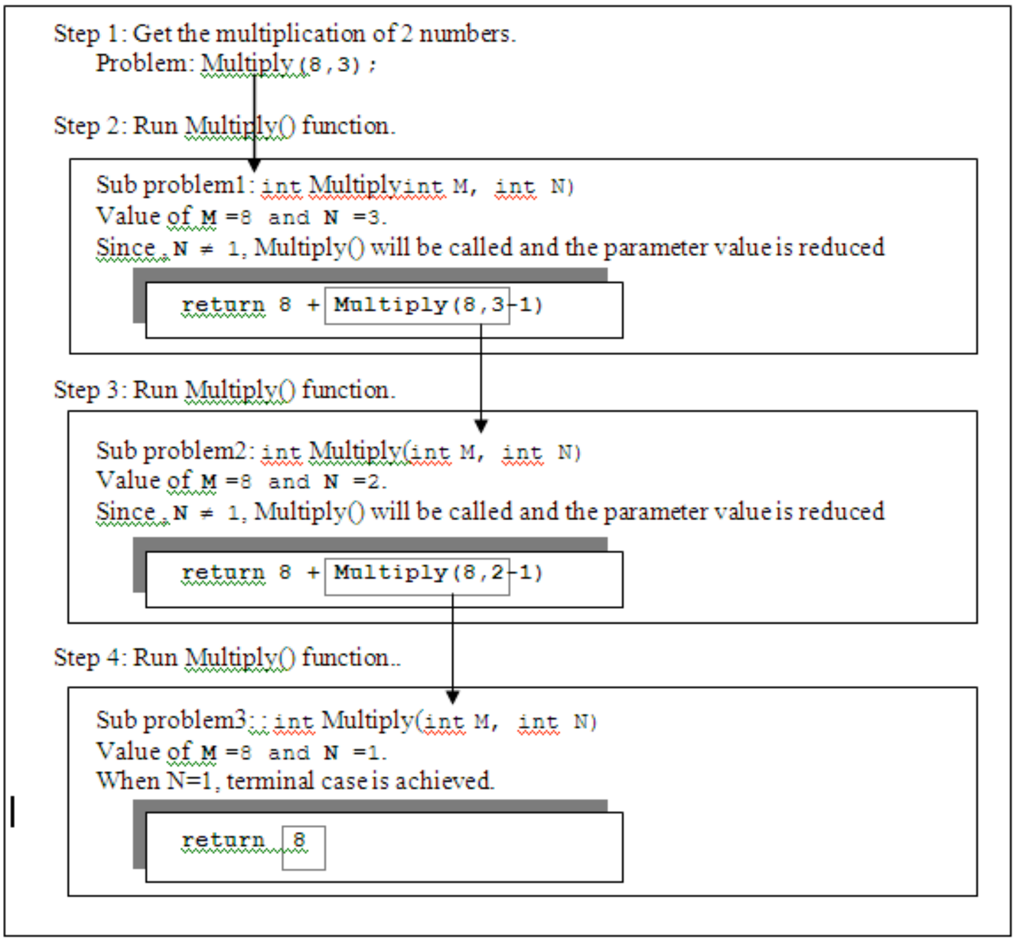


Figure 3.1 Calling recursive function steps for **Multiply(8,3)**

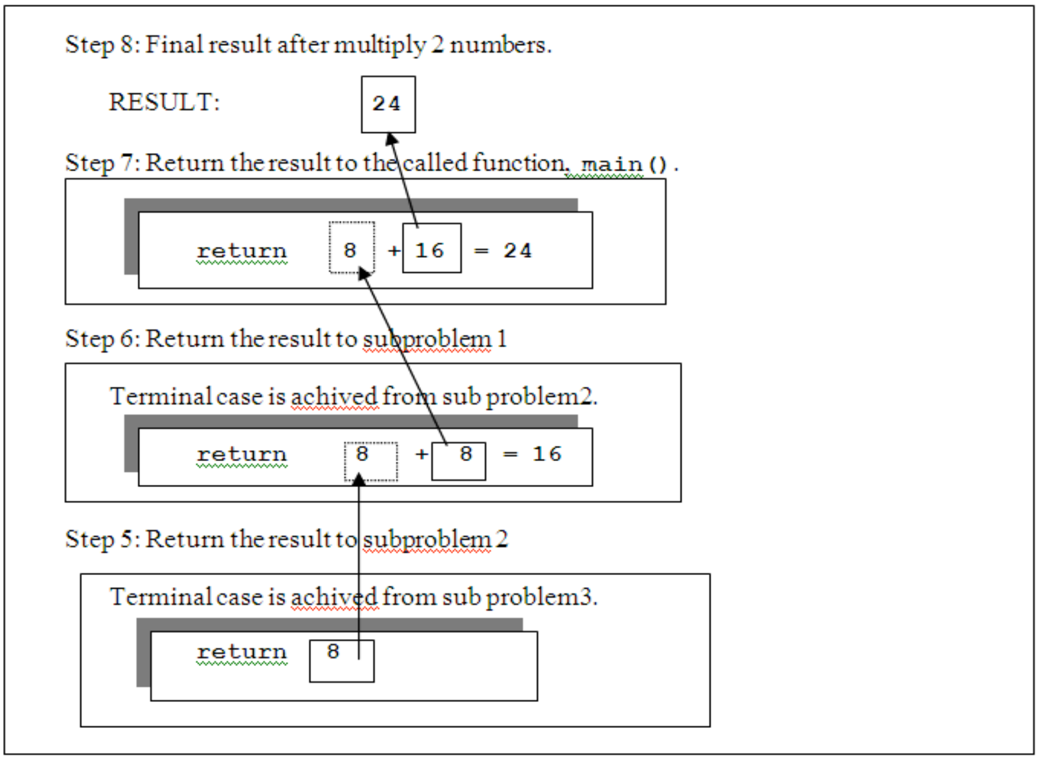


Figure 3.2 **Multiply(8,3)**returning recursive function steps

- Factorial function - Here is a function that computes the factorial of a number N without using a loop.
  - It checks whether ***N*** is equal 0. If so, the function just returns 1.
  - Otherwise, it computes the factorial of (***N*** – 1) and multiplies it by ***N***.

```
// Program 3.3
int Factorial (int N )
{ /*start Factorial*/
if (N==0)
        return 1;
else
        return N * Factorial (N-1);
} /*end Factorial
```

- Figure 3.3 shows the calling execution of **Factorial(3)**

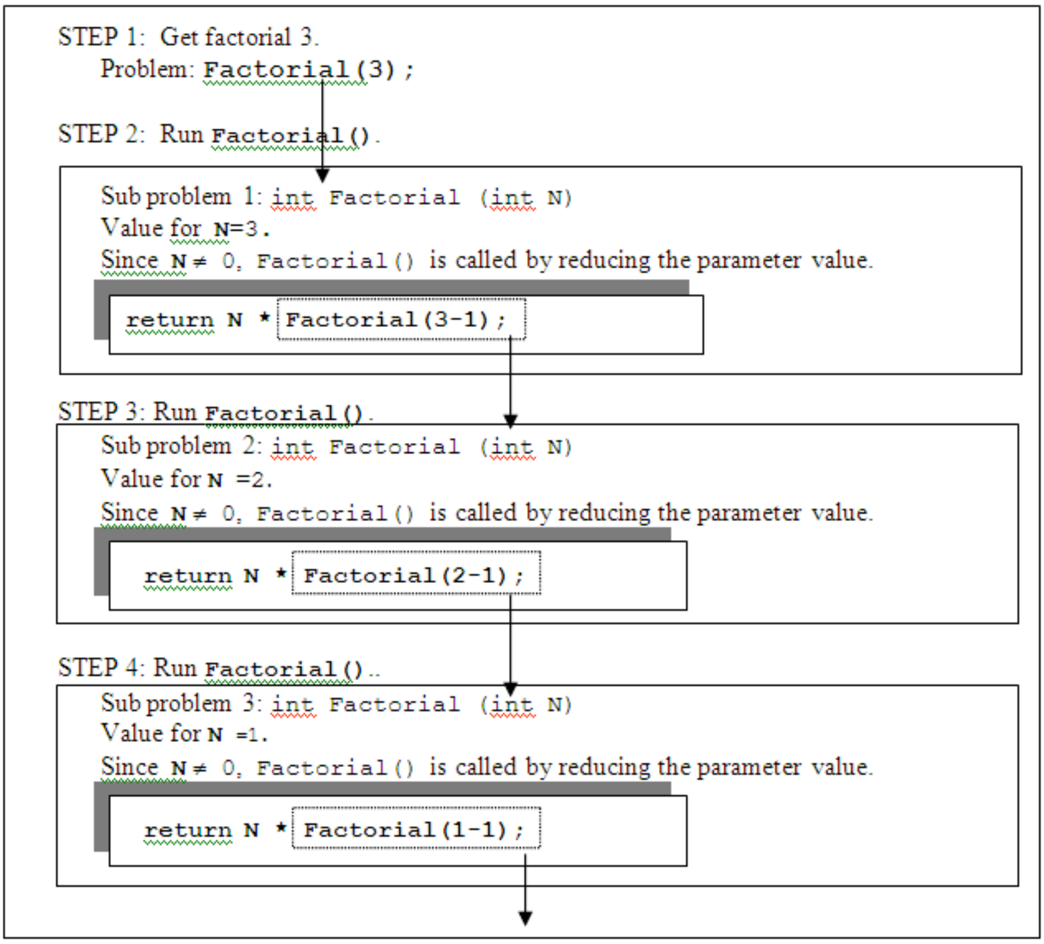


Figure 3.3 **Factorial(3)**calling steps

- Terminal case for **Factorial(3)** is achieved in Figure 3.4.

Step 5: Run **Factorial()**

Sub problem 4: **int Factorial (int N )**
Value for **N=0**
Since **N=0**, terminal case is achieved.

```
return 1
```

Figure 3.4 **Factorial(3)**terminal case

- Figure 3.5 shows the steps for return value for **Factorial(3)**

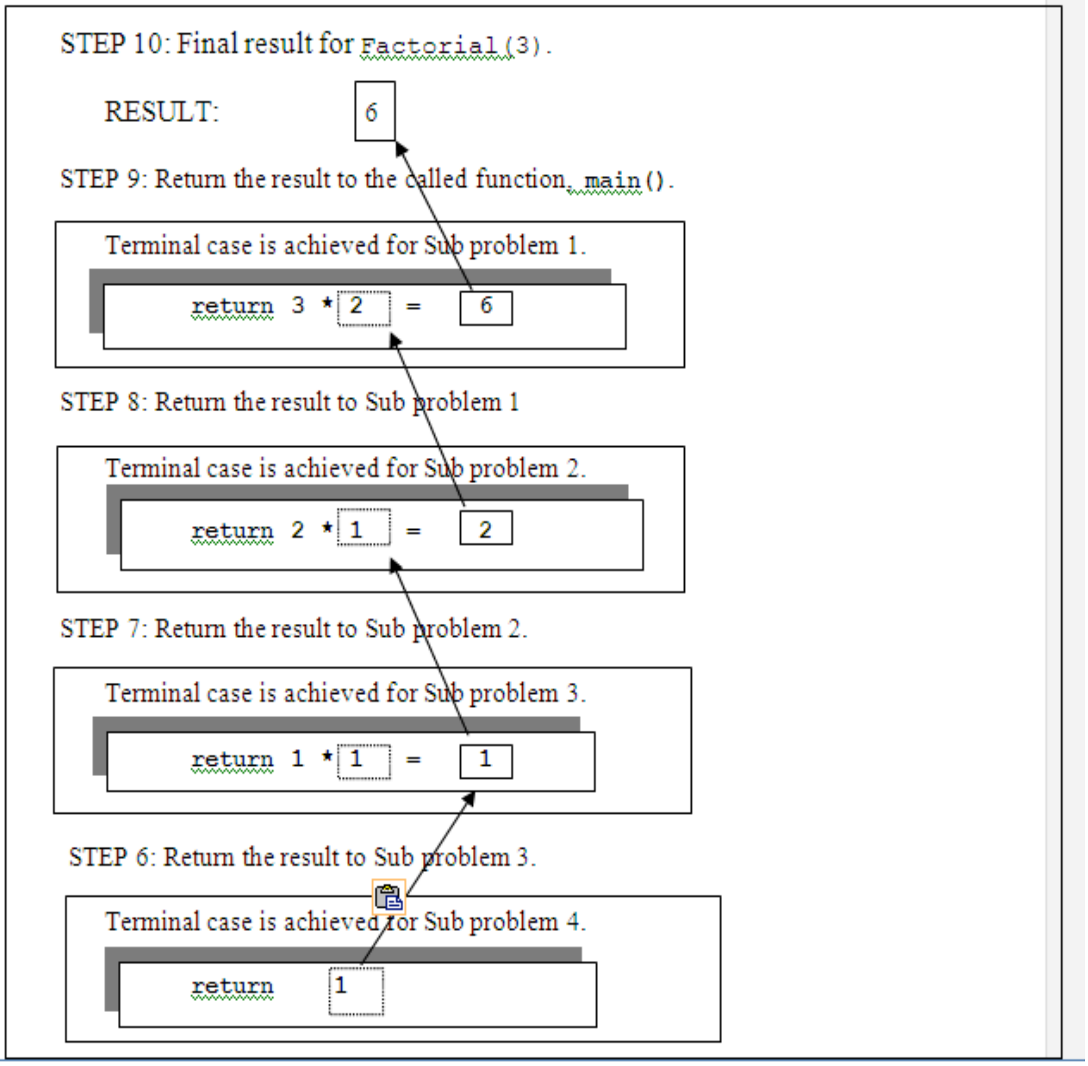


Figure 3.5 **Factorial(3)**returning steps

## 3.6 Fibonacci Problem

- **Problem**: Get Fibonacci series for an integer positive.
- Fibonacci Siries : 0, 1, 1, 2, 3, 5, 8, 13, 21,…..
- Starting from 0 and have features that every Fibonacci series is the result of adding 2 previous Fibonacci numbers.
- **Solution**: Fibonacci value of a number can be computed as follows:

**Fibonacci(0)** = 0
**Fibonacci(1)** = 1
**Fibonacci(2)** = 1
**Fibonacci(3)** = 2
**Fibonacci(N)** = **Fibonacci(N-1)** + **Fibonacci(N-2)**

- Solving Fibonacci Recursively
  - The simple solution for this example is represented by the Fibonacci value equal to 1.
  - ***N*** represents the series in the Fibonacci number. The recursive process will integrate the call of two **Fibonacci()** function.
  - Terminal case for Fibonacci problem is when ***N*** equal to 0 or ***N*** equal to 1. The computed result is returned to the called function.

- **Fibonacci()** function

```
// Program 3.4
int Fibonacci (int N)
{ if (N<=0)
          return 0;
  else if (N==1)
          return 1;
  else
          return Fibonacci(N-1) + Fibonacci (N-2);
}
```

- Figure 3.6 shows the recursive trace for **Fibonacci()** function . Each step calling and returning is labeled from L1 to L10.

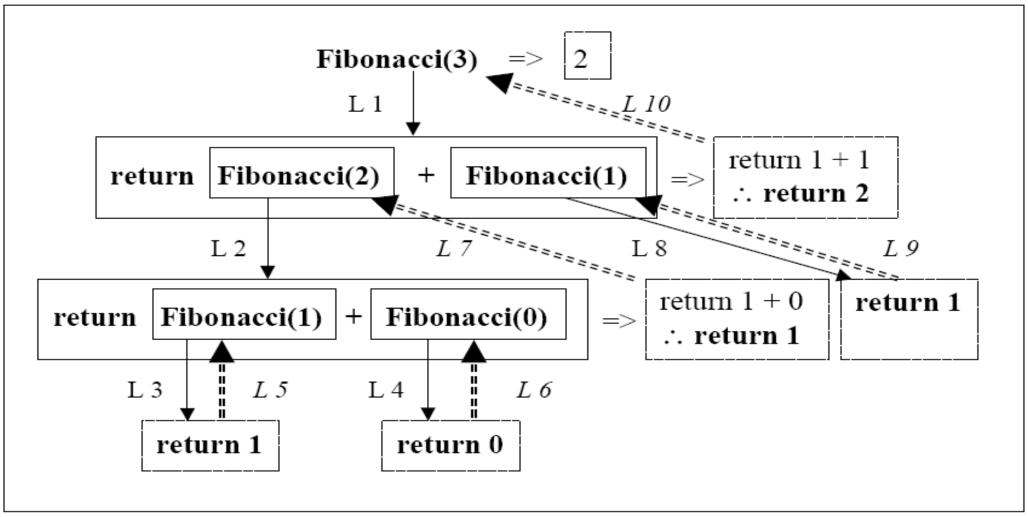


Figure 3.6 **Fibonacci(3)**execution steps

3.7 Infinite Recursive

- It is a state whereby the recursive functions run indefinitely and must be avoided in a programming discipline.
- Characteristics of a recursive function to avoid infinite recursion:
  - must have at least 1 base case (to terminate the recursive sequence)
  - each recursive call must get closer to a base case

- Example of infinite recursive is shown in Program 3.5.

```
// Program 3.5
void printIntegers(int n);
main()
{       int number;
        cout<<"\nEnter an integer value :";
        cin >> number;
        printIntegers(number);
}
void printIntegers (int nom)
{       cout << "\Value : " << nom;
        printIntegers(nom);
}
```

1. No condition satatement to stop the recursive call.
2. Terminal case variable does not change.

- The correct recursive function is shown in Program 3.6.

```
// Program 3.6
void printIntegers(int n);

main()
{ int number;
  cout<<"\nEnter an integer value :";
  cin >> number;
  printIntegers(number);
}

void printIntegers(int nom)
{  if (nom >= 1)
   {  cout << "\Value : " << nom;
      printIntegers (nom-2);
   }
}
```

condition statement to stop the recursive call and changes the terminal case during recursive call