PROGRAMMING

FOR PROBLEM SOLVING

{ C LANGUAGE }

Module :- 7

Recursion

Start

Module 7 :- Recursion

- Recursion, as a different way of solving problem
- Example program , such as
- Finding factorial, Fibonacci series, reverse a string using recursion, and GCD of two numbers, ackerman functions etc..
- Quick sort
- Merge sort

What is Recursion?

- The process of calling a function by itself is called Recursion and the function that calls itself is called Recursive Function
- A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.

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.
- 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.

Steps For Implementing Recursion In A Function

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

2. **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.

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

4. **Combine the solutions :** Combine the solutions of the subproblems to solve the original problem.


```
if ( n is small )
{
    we can answer directly
    easy to solve
    no recursion is needed
}
else
{
    input is large
    hard case
    we can not answer directly
    recursion is needed
}
```

Examples

1. Factorial
2. Fibonacci series
3. Reverse a string
4. GCD of two number
5. Quick sort
6. Merge sort

Recursion VS Iteration

Recursion

- Terminates when the base case becomes true.
- Used with functions.
- Every recursive call needs extra space in the stack memory.
- Smaller code size.

Iteration

- Terminates when the condition becomes false.
- Used with loops.
- Every iteration does not require any extra space.
- Larger code size.

Types of Recursion in C

1. Direct Recursion
2. Indirect Recursion
3. Tail Recursion
4. No Tail/ Head Recursion
5. Linear recursion
6. Tree Recursion

Direct Recursion

- A function calls itself within the definition of the function through direct recursion.

- It is a simple and typical type of recursion in C.
- The size of the problem normally decreases with each recursive iteration until a base case is reached to end the recursion.

Indirect Recursion

- At least two functions that call each other repeatedly in a cycle constitute indirect recursion.
- By transferring control back and forth between each other until a termination condition is satisfied, these functions cooperate to solve a problem.
- Even though it is less common, this kind of recursion has its uses.

Tail Recursion

- When a recursive function calls itself in a loop and that looping statement is the final one the function performs, the function is said to be "tail-recursive“.
- There are no more functions or statements that can call the recursive function after that.

Non-Tail / Head Recursion

- The non-tail or head recursion of a function The initial statement in a function will be the recursive call if it does one on its own. It implies that no statement or operation should be called prior to the recursive calls. Additionally, when a recursive call is made, the head recursive accomplishes nothing. Instead, every action is finished at the return time.

Linear Recursion

- If a function only makes one call to itself each time it is executed and expands linearly as a function of the size of the problem, the function is said to be linear recursive.

Tree Recursion

- When a recursive function in C calls itself more than once, the result is a branching structure that resembles a tree. This is known as "tree recursion." Recursive calls are frequently employed to solve issues involving hierarchical or connected data structures because each call can result in other calls. For issues with numerous recursive subproblems, this kind of recursion is especially helpful.

The advantages of recursion are as follows:

- Writing code may be simpler.
- To resolve issues like the Hanoi Tower that are inherently recursive.
- Lessen the frequency of pointless function calls.
- Exceptionally practical when using the same solution.
- Recursion cuts down on code length.
- It helps a lot in resolving the data structure issue.
- Evaluations of infix, prefix, and postfix stacks, among other things.

The disadvantages of recursion are as follows:

- In general, recursive functions are slower than non-recursive ones.
- To store intermediate results on the system stacks, a significant amount of memory may be needed.
- The code is difficult to decipher or comprehend.
- It is not more effective in terms of complexity over time and space.
- If the recursive calls are not adequately checked, the machine can run out of memory.

Ackerman function

- In computability theory, the Ackermann function, named after **Wilhelm Ackermann**, is one of the simplest and earliest-discovered examples of a total computable function that is not primitive recursive. All primitive recursive functions are total and computable, but the Ackermann function illustrates that not all total computable functions are primitive recursive. Refer this for more.
  It's a function with two arguments each of which can be assigned any non-negative integer.

Ackermann function is defined as:

$$A(m,n) = \begin{cases} n + 1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

where m and n are non-negative integers

Solve A(1, 2)?

A(1, 2)
Here m = 1, n = 2 , m > 0 and n > 0
Hence applying third condition of Ackermann function
A(1, 2) = A(0, A(1, 1)) ———- (1)
Now, Let's find A(1, 1) by applying third condition of Ackermann function
A(1, 1) = A(0, A(1, 0)) ———- (2)
Now, Let's find A(1, 0) by applying second condition of Ackermann function
A(1, 0) = A(0, 1) ———- (3)
Now, Let's find A(0, 1) by applying first condition of Ackermann function

A(0, 1) = 1 + 1 = 2
Now put this value in equation 3
Hence A(1, 0) = 2
Now put this value in equation 2
A(1, 1) = A(0, 2) ———- (4)
Now, Let's find A(0, 2) by applying first condition of Ackermann function
A(0, 2) = 2 + 1 = 3
Now put this value in equation 4
Hence A(1, 1) = 3
Now put this value in equation 1
A(1, 2) = A(0, 3) ———- (5)
Now, Let's find A(0, 3) by applying first condition of Ackermann function
A(0, 3) = 3 + 1 = 4
Now put this value in equation 5
Hence A(1, 2) = 4
So, A (1, 2) = 4

Thank You !!

Dhanybad !!

Shukriya !!