

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.

- A recursive algorithm takes one step toward solution and then recursively call itself to further move. The algorithm stops once we reach the solution.
- Since called function may further call itself, this process might continue forever. So it is essential to provide a base case to terminate this recursion process.

Steps to Implement Recursion

Step1 - Define a base case: Identify the simplest (or base) 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.

Example 1 : Sum of Natural Numbers (n=3)

Input : $n = 3$

Output : 6

Explanation : The sum of first 3 natural numbers is $1+2+3 = 6$.

Input : $n = 7$

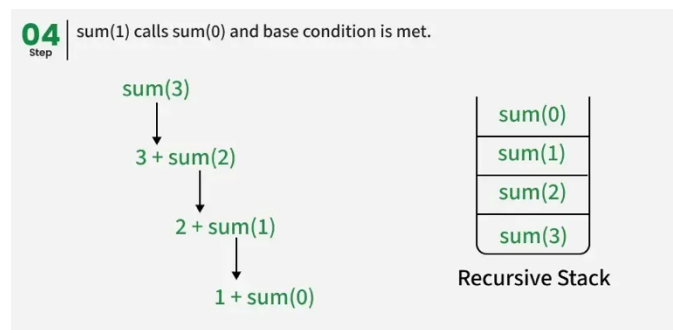
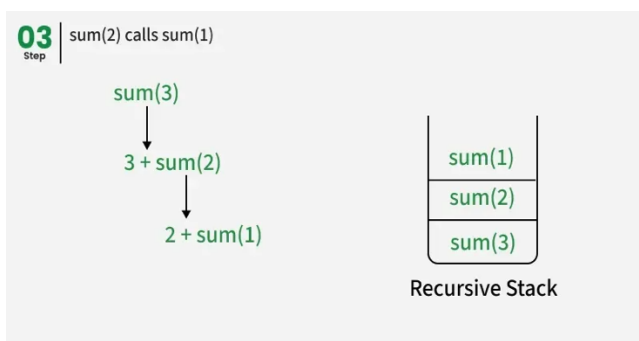
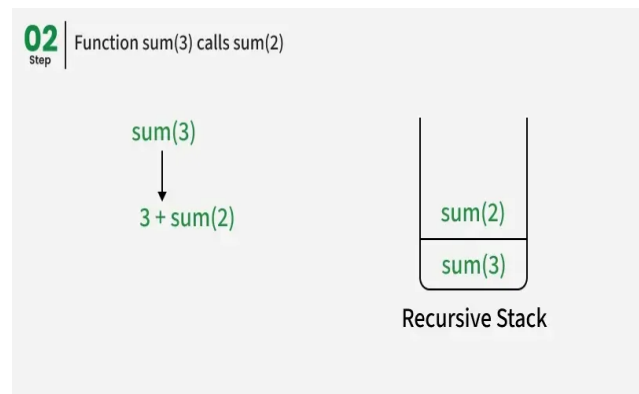
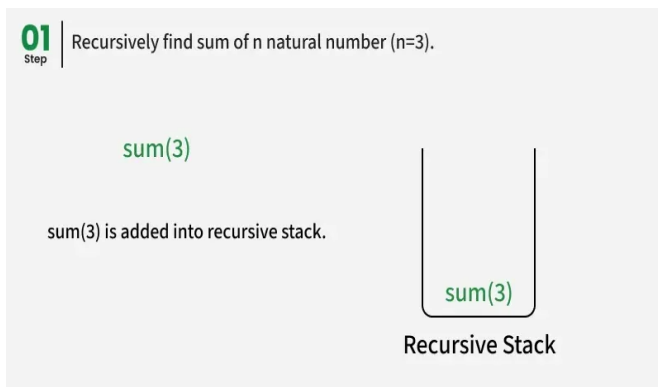
Output : 28

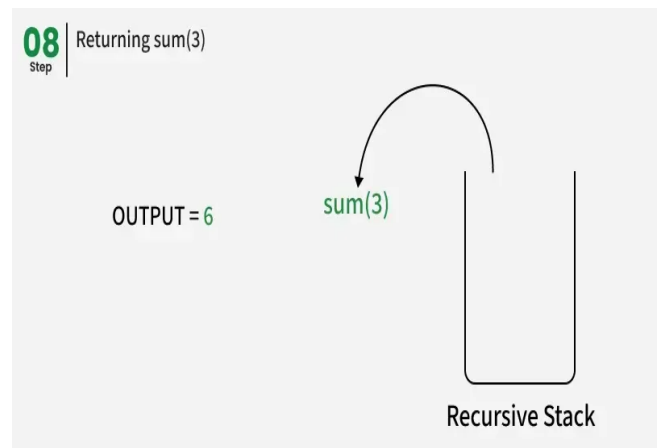
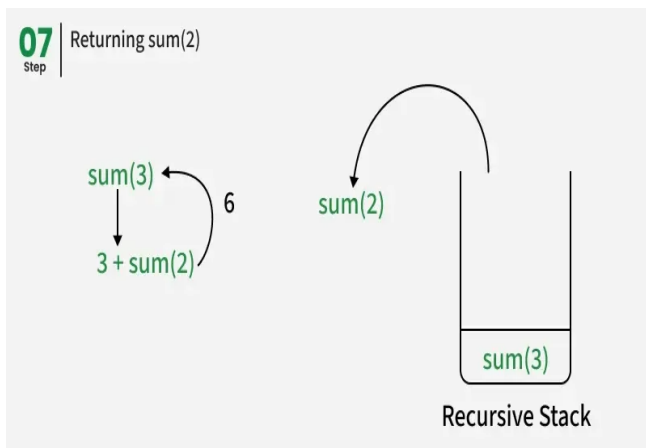
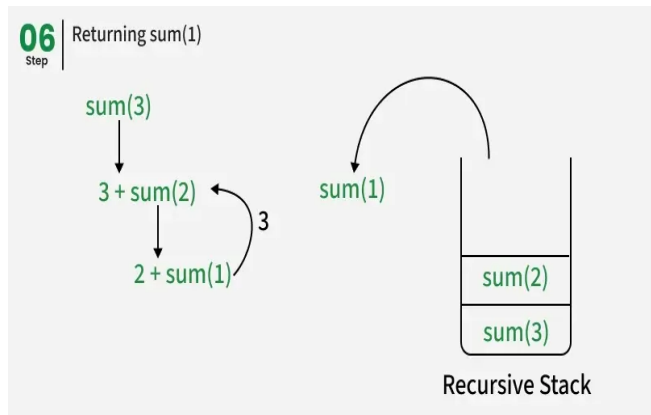
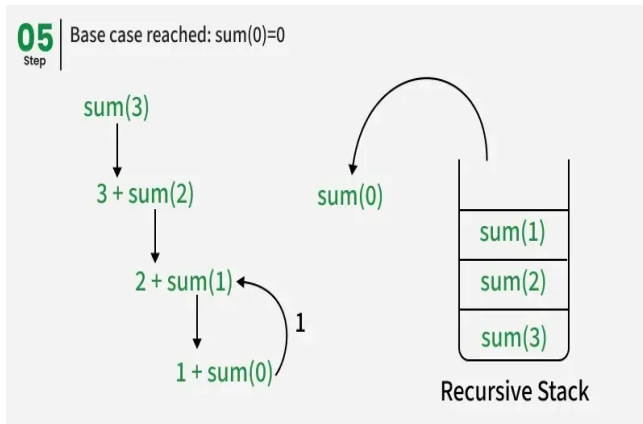
Explanation : The sum of first 7 natural numbers is $1+2+3+4+5+6+7 = 28$.

- **Base Case :** At $n == 1$, it returns 1 for $n = 3$, the recursion reaches this after going through $3 \rightarrow 2 \rightarrow 1$.
- **Recursive Case :** Each call adds n to $\text{sum}(n-1)$, so $\text{sum}(3) = 3 + \text{sum}(2)$, $\text{sum}(2) = 2 + \text{sum}(1)$.

Execution Flow of the Recursive Solution

Calls are stacked as $\text{sum}(3) \rightarrow \text{sum}(2) \rightarrow \text{sum}(1)$ before any addition happens. Results are added back in reverse: $\text{sum}(1) = 1$, $\text{sum}(2) = 3$, $\text{sum}(3) = 6$.





Comparison of **Recursive** and **Iterative** Approaches

A. Need of Recursion?

- Recursion helps in logic building. Recursive thinking helps in solving complex problems by breaking them into smaller subproblems.
- Recursive solutions work as a basis for **Dynamic Programming** and **Divide and Conquer** algorithms.
- Certain problems can be solved quite easily using recursion like Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

B. What is the base condition in recursion?

A recursive program stops at a base condition. There can be more than one base conditions in a recursion. In the above program, the base condition is when $n = 1$.

C. How a particular problem is solved using recursion?

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. Example 2 : Factorial of a Number, the factorial of a number n (where $n \geq 0$) is the product of all positive integers from 1 to n . To compute the factorial recursively, we calculate the factorial of n by using the factorial of $(n-1)$. The base case for the recursive function is when $n = 0$, in which case we return 1.

D. What is the difference between direct and indirect recursion?

A function is called **direct recursive** if it calls itself directly during its execution. In other words, the function makes a recursive call to itself within its own body.

An **indirect recursive function** is one that calls another function, and that other function, in turn, calls the original function either directly or through other functions. This creates a chain of recursive calls involving multiple functions, as opposed to direct recursion, where a function calls itself.

E. Tail recursion is defined as a recursive function in which the recursive call is the last statement that is executed by the function. So basically nothing is left to execute after the recursion call.

Need for Tail Recursion:

- Tail-recursive functions are better than non-tail-recursive ones because they can be optimized by the compiler.
- The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame can be avoided by simply sending the control back to the beginning of the function either using a loop or goto statement.

Common Applications of Recursion

1. **Tree and Graph Traversal:** Used for systematically exploring nodes/vertices in data structures like trees and graphs.
2. **Sorting Algorithms:** Algorithms like quicksort and merge sort divide data into subarrays, sort them recursively, and merge them.
3. **Divide-and-Conquer Algorithms:** Algorithms like binary search break problems into smaller subproblems using recursion.
4. **Fractal Generation:** Recursion helps generate fractal patterns, such as the Mandelbrot set, by repeatedly applying a recursive formula.
5. **Backtracking Algorithms:** Used for problems requiring a sequence of decisions, where recursion explores all possible paths and backtracks when needed.
6. **Memoization:** Involves caching results of recursive function calls to avoid recomputing expensive subproblems.