Recursion in Python is a technique where a function calls itself to solve a problem. It is commonly used for problems that can be broken down into smaller, similar subproblems, such as tree traversals, factorial calculation, Fibonacci sequences, and more.

**How Recursion Works**

A recursive function consists of:

1. **Base Case**: The condition where recursion stops.
2. **Recursive Case**: The function calls itself with a modified argument.

**Example 1: Factorial using Recursion**

The factorial of a number n is defined as:

n!=n×(n−1)!n! = n \times (n-1)!

def factorial(n):

if n == 0 or n == 1: # Base case

return 1

else:

return n \* factorial(n - 1) # Recursive case

print(factorial(5)) # Output: 120

**Example 2: Fibonacci Sequence**

The Fibonacci sequence is defined as:

F(n)=F(n−1)+F(n−2)F(n) = F(n-1) + F(n-2)

def fibonacci(n):

if n == 0: # Base case

return 0

elif n == 1:

return 1

else:

return fibonacci(n - 1) + fibonacci(n - 2) # Recursive case

print(fibonacci(6)) # Output: 8

**Example 3: Sum of Digits Using Recursion**

def sum\_of\_digits(n):

if n == 0:

return 0

return (n % 10) + sum\_of\_digits(n // 10)

print(sum\_of\_digits(1234)) # Output: 10

**Example 4: Recursive Binary Search**

Binary search is an efficient algorithm to find an element in a sorted array.

def binary\_search(arr, left, right, target):

if left > right:

return -1 # Base case: not found

mid = (left + right) // 2

if arr[mid] == target:

return mid

elif arr[mid] > target:

return binary\_search(arr, left, mid - 1, target) # Search left half

else:

return binary\_search(arr, mid + 1, right, target) # Search right half

arr = [1, 3, 5, 7, 9, 11]

print(binary\_search(arr, 0, len(arr) - 1, 7)) # Output: 3

**Pros and Cons of Recursion**

**✅ Advantages**

* Simplifies code for problems that have repetitive substructure (like trees, graphs, and mathematical computations).
* Easier to read and understand compared to iterative solutions in some cases.

**❌ Disadvantages**

* Can be **less efficient** due to multiple function calls and increased memory usage (stack space).
* Risk of **stack overflow** if recursion depth is too high.

**Optimizing Recursion: Memoization**

Using **memoization** (caching results) prevents repeated computations.

**Optimized Fibonacci using Memoization**

def fibonacci\_memo(n, memo={}):

if n in memo:

return memo[n]

if n == 0:

return 0

elif n == 1:

return 1

memo[n] = fibonacci\_memo(n - 1, memo) + fibonacci\_memo(n - 2, memo)

return memo[n]

print(fibonacci\_memo(50)) # Fast execution

**When to Use Recursion?**

* When the problem can be broken into **smaller subproblems**.
* When recursion provides a **cleaner solution** than iteration.
* When working with **tree/graph traversal**.

**When to Avoid Recursion?**

* If an **iterative solution** is more efficient (e.g., loops for Fibonacci instead of naive recursion).
* When there is a risk of **deep recursion** leading to stack overflow.

**Conclusion**

Recursion is a powerful concept in Python, useful for breaking down problems into smaller subproblems. However, it should be used wisely to avoid performance issues. When needed, **memoization** or **iteration** can help optimize recursive solutions.