Recursion

Recursion is a programming and mathematical concept where a function calls itself in its own definition.The process of recursion involves breaking down a problem into smaller, more manageable subproblems and solving each subproblem independently. This continues until the problem becomes small enough to be solved directly without further recursion.

Key components of recursion:

1. **Base Case:**
    - A condition that checks whether the problem has become small enough to be solved directly without further recursion. It prevents the recursion from continuing indefinitely.
2. **Recursive Case:**
    - The part of the function where it calls itself with modified parameters, effectively solving a smaller instance of the same problem.
3. **Termination:**
    - The recursive calls eventually lead to the base case, ensuring that the recursion terminates.

Advantages of Recursion:

1.Readability and Simplicity
2.Divide and conquer

3.Solving recursive problems
4.Backtracking
5.Reduction of repeated code
6.Dynamic memory allocation
7.Tree and graph traversals
8.Mathematical simplicity

Difference Between Recursion and Iteration:

Feature	Recursion	Iteration
Definition	A function calls itself to solve smaller instances of the same problem.	A set of instructions or statements is repeatedly executed within a loop structure.
Mechanism	Involves a chain of function calls, each solving a smaller part of the problem.	Achieved through loops (e.g., for, while), where a block of code is executed repeatedly.
Termination	Requires a base case to stop the recursive calls and prevent infinite recursion.	Relies on loop conditions or explicit termination statements to control execution.
Memory Usage	May consume more memory due to the overhead of maintaining multiple function calls on the call stack.	Typically uses less memory as it involves reusing the same set of instructions.
Readability	Can be more intuitive for certain problems, expressing the solution in a more natural way.	May be perceived as more straightforward for some tasks, especially when dealing with simple iterations.
Performance	May have higher overhead due to the function call stack, and some recursive algorithms may be less efficient.	Generally more efficient in terms of memory and performance for many tasks.
Use Cases	Well-suited for problems that exhibit a recursive structure, such as tree traversal or problems that naturally divide into smaller subproblems.	Often used for tasks involving repetition or sequential processing, like array traversal or numerical calculations.

Examples of Recursion:

1. **Factorial:**
   - **Base Case:** factorial(0)=1 (the factorial of 0 is 1)
   - **Recursive Case:** factorial(n)=n×factorial(n−1)
2. **Fibonacci Sequence:**
   - **Base Cases:** fibonacci(0)=0 and fibonacci(1)=1
   - **Recursive Case:** fibonacci(n)=fibonacci(n−1)+fibonacci(n−2)