**1. What is Recursion? Explain with an Example.**

Recursion is a technique in programming where a function **calls itself** to solve a problem. It is commonly used in problems that can be broken down into smaller subproblems of the same type.

**Example: Factorial Using Recursion**

Factorial of n (n!) is defined as:

n! =n×(n−1)!

Base Case: 0! = 1

|  |
| --- |
| public class RecursionExample {  public static int factorial(int n) {  if (n == 0) return 1; // Base case  return n \* factorial(n - 1); // Recursive case  }  public static void main(String[] args) {  System.out.println(factorial(5)); // Output: 120  }  } |

**2. How Does Recursion Work Internally in Memory?**

Recursion works using the **call stack**, a Last-In-First-Out (LIFO) structure where each function call is stored. Each recursive call **pushes a new frame onto the stack**, and when a base case is reached, function calls are **popped from the stack**.

**Example: Factorial Call Stack**

|  |
| --- |
| factorial(3)  ⮑ factorial(2)  ⮑ factorial(1)  ⮑ factorial(0) (Base case, return 1)  ⮑ return 1 \* 1 = 1  ⮑ return 2 \* 1 = 2  ⮑ return 3 \* 2 = 6 |

Memory Usage: Each call **stores** a return address and local variables, which increases stack size.

**3. What is a Base Case in Recursion? Why is it Important?**

A **base case** is the condition that **stops recursion**. Without it, the function will call itself infinitely, leading to a **stack overflow**.

|  |
| --- |
| public static void printNumbers(int n) {  if (n == 0) return; // Base Case: Stop when n reaches 0  System.out.println(n);  printNumbers(n - 1);  } |

🔹 **Importance:**  
✅ Prevents infinite recursion  
✅ Reduces memory consumption  
✅ Ensures termination

**4. What Happens if Recursion Has No Base Case?**

Without a base case, recursion **never stops**, causing a **stack overflow error** due to excessive memory use.

|  |
| --- |
| public static void infiniteRecursion(int n) {  System.out.println(n);  infiniteRecursion(n - 1); // No base case! Causes stack overflow.  } |

**Error:** Exception in thread "main" java.lang.StackOverflowError

**5. What is Direct and Indirect Recursion? Give Examples.**

**🔹 Direct Recursion: A function calls itself directly.**

Example:

|  |
| --- |
| void direct(int n) {  if (n == 0) return;  direct(n - 1);  } |

**🔹 Indirect Recursion: Two or more functions call each other in a cycle.**

Example:

|  |
| --- |
| void A(int n) {  if (n == 0) return;  B(n - 1);}  void B(int n) {  if (n == 0) return;  A(n - 1);} |

**6. What is Tail Recursion and Non-Tail Recursion?**

🔹 **Tail Recursion:** The recursive call is **the last statement**. It allows **compiler optimization (Tail Call Optimization - TCO).**  
Example:

|  |
| --- |
| void tailRecursion(int n) {  if (n == 0) return;  System.out.println(n);  tailRecursion(n - 1); // Last operation  } |

🔹 **Non-Tail Recursion:** The function performs additional operations **after** the recursive call.  
Example:

|  |
| --- |
| int factorial(int n) {  if (n == 0) return 1;  return n \* factorial(n - 1); // Multiplication happens after recursive call  } |

**7. How Does Recursion Differ from Iteration?**

| **Feature** | **Recursion** | **Iteration** |
| --- | --- | --- |
| Memory Usage | High (Stack frames) | Low (Loop variables) |
| Speed | Slower | Faster |
| Readability | More readable for complex problems | Less readable |
| Stack Overflow | Possible | Not possible |

Example of **Recursion vs. Iteration** for factorial:

**// Recursion**

|  |
| --- |
| int factorial(int n) {  if (n == 0) return 1;  return n \* factorial(n - 1);  } |

**// Iteration**

|  |
| --- |
| int factorialIterative(int n) {  int result = 1;  for (int i = 1; i <= n; i++) result \*= i;  return result;} |

**8. What Are the Advantages of Recursion?**

✅ **Simplifies code** for complex problems (Tree Traversal, DFS, etc.)  
✅ **Divides problem into smaller subproblems**  
✅ **Uses mathematical logic** (e.g., Fibonacci, Factorial)

**9. What Are the Disadvantages of Recursion?**

❌ **Consumes more memory** (due to stack frames)  
❌ **Slower execution** (due to function calls)  
❌ **Risk of stack overflow**

**10. What is Stack Overflow in Recursion? How Can We Prevent It?**

🔹 **Stack Overflow** occurs when recursion runs indefinitely, filling the stack memory.  
🔹 **Prevention:**  
✔ Always use **a base case**  
✔ Convert recursion into **iteration** if possible  
✔ Use **memoization** to reduce redundant calls

**11. Real-World Applications of Recursion**

✅ **Tree Traversals (Preorder, Inorder, Postorder)**  
✅ **Graph Traversals (DFS, BFS)**  
✅ **Backtracking (N-Queens, Sudoku Solver)**  
✅ **Sorting (Merge Sort, Quick Sort)**  
✅ **Dynamic Programming (Fibonacci, LCS)**

**12. What is Memoization in Recursion? How Does It Help?**

🔹 **Memoization** stores previously computed results to avoid redundant recursive calls.  
🔹 Used in **Dynamic Programming** problems.

Example:

|  |
| --- |
| int[] dp = new int[1000]; // Cache array  int fibonacci(int n) {  if (n <= 1) return n;  if (dp[n] != 0) return dp[n]; // Return stored result  return dp[n] = fibonacci(n - 1) + fibonacci(n - 2);  } |

✅ **Optimizes recursion**  
✅ **Reduces redundant calls**  
✅ **Improves performance**

**13. Time and Space Complexity of Recursion**

✅ **Time Complexity:** Depends on number of recursive calls  
✅ **Space Complexity:** Depends on call stack usage

Example: Fibonacci Recursion

* **Time Complexity:** O(2n)O(2^n)O(2n) (Exponential)
* **Space Complexity:** O(n)O(n)O(n) (Stack frames)

**14. What is Tree Recursion? Give an Example.**

🔹 When a function **calls itself multiple times**.  
Example: Fibonacci Sequence

|  |
| --- |
| int fibonacci(int n) {  if (n <= 1) return n;  return fibonacci(n - 1) + fibonacci(n - 2);  } |

🔹 **Time Complexity:** O(2n)O(2^n)O(2n) (Exponential)

**15. What is Recursion Depth in Programming?**

🔹 Recursion **depth** refers to the maximum number of recursive calls stored on the call stack.  
🔹 **Exceeding depth** leads to **StackOverflowError**.

**16. What is Functional Recursion?**

Functional recursion is a type of recursion where **the function only returns the result of a recursive call without performing any additional operations** before or after it.

**Example: Functional Recursion**

|  |
| --- |
| int sum(int n) {  if (n == 0) return 0; // Base case  return n + sum(n - 1); // Recursive call only  } |

🔹 **Characteristics:**  
✅ The function purely depends on the recursive call.  
✅ It does not store extra state or modify variables.  
✅ Common in functional programming paradigms.

**17. When Should Recursion Be Avoided?**

Recursion should be avoided when:

❌ **High memory usage is a concern** (e.g., deep recursion leading to stack overflow).  
❌ **Performance is critical** (e.g., recursive Fibonacci is slow without optimization).  
❌ **Tail call optimization (TCO) is not supported** by the compiler or language.  
❌ **An iterative solution is simpler and more efficient** (e.g., looping through an array).

**Example Where Iteration is Better**

🔴 **Inefficient Recursion (Can Cause Stack Overflow)**

|  |
| --- |
| void countDown(int n) {  if (n == 0) return;  System.out.println(n);  countDown(n - 1);  } |

✅ **Better Iterative Approach**

|  |
| --- |
| void countDownIterative(int n) {  while (n > 0) {  System.out.println(n);  n--;  }  } |

**18. Why is Recursion Slower Than Iteration?**

Recursion is often **slower** than iteration because:

1️. **Function Call Overhead:** Each recursive call **creates a new stack frame**, which takes time.  
2️.**Memory Consumption:** Recursion **uses extra memory** for function calls, whereas iteration reuses variables.  
3️.**Duplicate Calculations:** Some recursive functions (like Fibonacci) **repeat calculations**, slowing performance.

**Example: Fibonacci Without Optimization**

|  |
| --- |
| int fibonacci(int n) {  if (n <= 1) return n;  return fibonacci(n - 1) + fibonacci(n - 2); // Duplicate calls  } |

🔹 **Time Complexity:** O(2n)O(2^n)O(2n)  
🔹 **Space Complexity:** O(n)O(n)O(n) (due to call stack)

✅ **Optimized with Iteration**

|  |
| --- |
| int fibonacciIterative(int n) {  if (n <= 1) return n;  int a = 0, b = 1, sum;  for (int i = 2; i <= n; i++) {  sum = a + b;  a = b;  b = sum;}  return b;} |

🔹 **Time Complexity:** O(n)O(n)O(n)  
🔹 **Space Complexity:** O(1)O(1)O(1)

**19. What is Tree Recursion? Give an Example.**

**Tree recursion** occurs when a function **calls itself multiple times** within the same function execution.

**Example: Fibonacci Series (Tree Recursion)**

|  |
| --- |
| int fibonacci(int n) {  if (n <= 1) return n; // Base case  return fibonacci(n - 1) + fibonacci(n - 2); // Multiple recursive calls  } |

**Tree recursion characteristics:**  
✅ **Each call branches into multiple recursive calls**  
✅ **Grows exponentially in time complexity**  
✅ **Common in combinatorial problems (e.g., subset generation, backtracking)**

**Visual Representation of fibonacci(4)**

|  |
| --- |
| fib(4)  / \  fib(3) fib(2)  / \ / \  fib(2) fib(1) fib(1) fib(0)  / \  fib(1) fib(0) |

🔹 **Time Complexity:** O(2n)O(2^n)O(2n) (Exponential growth)  
🔹 **Space Complexity:** O(n)O(n)O(n) (Call stack depth)

**20. What is the Role of Recursion Depth in Programming?**

🔹 **Recursion depth** refers to **the maximum number of recursive calls that can be made before stack overflow occurs**.

**Role of Recursion Depth:**

✔ **Affects memory usage**: Higher depth consumes more memory.  
✔ **Limits recursive solutions**: Deep recursion may cause **StackOverflowError**.  
✔ **Used to optimize recursive functions**: Example - setting a **maximum recursion depth** in languages like Python.

**Example of Recursion Depth Causing an Error**

|  |
| --- |
| void infiniteRecursion(int n) {  System.out.println(n);  infiniteRecursion(n - 1); // No base case, causes StackOverflowError} |

🔹 **Fix:** Limit recursion depth

|  |
| --- |
| void limitedRecursion(int n) {  if (n == 0) return; // Base case prevents infinite recursion  System.out.println(n);  limitedRecursion(n - 1);  } |

🔹 **Alternative:** Convert deep recursion to **iteration** to prevent stack overflow.

