+91 96622 16697
jay@frameboxxers.com
F-201, Shilp Square-B, Nr. Shreeji Tower,
Drive-in Road, Vastrapur, Ahmedabad - 380015

📘 Lecture 11: Recursion

🧠 Introduction

Recursion is a powerful technique where a function calls itself to solve smaller instances of the same problem. It allows for elegant solutions to problems that are inherently repetitive or have a recursive structure.
In this lecture, we will dive deep into how recursion works, its components, and explore practical examples where recursion can be used effectively.

🔹 1. What is Recursion?

Recursion is a process where a function calls itself directly or indirectly to break down a complex problem into simpler sub-problems. Each recursive call solves a part of the problem, and the results of those sub-problems are combined to give the solution.
For recursion to work correctly, it must have two essential components:

1. **Base Case**: The condition that stops the recursion.

2. **Recursive Case**: The part where the function calls itself with a simpler or smaller version of the original problem.

🧪 Basic Example of Recursion:

Here's a simple example of recursion in JavaScript:
javascript
CopyEdit

```
function factorial(n) {
  if (n === 0) {  // Base case
    return 1;
  } else {
    return n * factorial(n - 1);  // Recursive case
  }
}

console.log(factorial(5)); // Output: 120
```

Explanation:

- **Base Case**: When n === 0, the function returns 1. This prevents infinite recursion.

- **Recursive Case**: The function calls itself with n - 1 until it reaches the base case.

In this example, the recursion works by breaking down the problem factorial(n) into smaller sub-problems like factorial(n - 1), and so on.

+91 96622 16697
jay@frameboxxers.com
F-201, Shilp Square-B, Nr. Shreeji Tower,
Drive-in Road, Vastrapur, Ahmedabad - 380015

🔹 2. How Recursion Works

Recursion works by breaking a problem into smaller instances of the same problem and using the results of those smaller instances to solve the larger problem.

Steps of Recursive Execution:

1. **Start**: The initial function call starts with the original problem.
2. **Recursive Calls**: Each call works on a smaller sub-problem.
3. **Base Case**: The base case is eventually reached, which stops further recursive calls.
4. **Unwinding**: The function returns from the recursive calls, combining the results to solve the original problem.

Example: Calculating Factorial

To calculate factorial(5), the recursion happens as follows:

matlab
CopyEdit

```
factorial(5)
= 5 * factorial(4)
= 5 * (4 * factorial(3))
= 5 * (4 * (3 * factorial(2)))
= 5 * (4 * (3 * (2 * factorial(1))))
= 5 * (4 * (3 * (2 * (1 * factorial(0)))))
= 5 * (4 * (3 * (2 * (1 * 1))))
= 120
```

🔹 3. Base Case and Recursive Case

Every recursive function needs both a **base case** and a **recursive case**.

- **Base Case**: This is the condition under which the function stops calling itself. Without a base case, recursion would continue infinitely, leading to a stack overflow error.
- **Recursive Case**: This part of the function reduces the problem into smaller pieces and calls the function on these smaller sub-problems.

Example: Sum of Natural Numbers

Consider the problem of calculating the sum of natural numbers up to n (i.e., 1 + 2 + 3 + ... + n).

javascript
CopyEdit

+91 96622 16697
jay@frameboxxers.com
F-201, Shilp Square-B, Nr. Shreeji Tower,
Drive-in Road, Vastrapur, Ahmedabad - 380015

```
function sum(n) {
  if (n === 0) {  // Base case
    return 0;
  } else {
    return n + sum(n - 1);  // Recursive case
  }
}

console.log(sum(5)); // Output: 15 (1 + 2 + 3 + 4 + 5)
```

- **Base Case**: When n === 0, the function returns 0 (the sum of numbers up to 0).
- **Recursive Case**: The function adds n to the sum of numbers up to n - 1.

🔹 **4. Advantages of Recursion**

- **Simplicity**: Recursive solutions often provide a more elegant and easier-to-understand implementation than their iterative counterparts.
- **Problem-Solving**: Recursion is naturally suited for problems that involve breaking down the problem into smaller sub-problems (e.g., tree traversal, sorting algorithms).
- **Readability**: Recursion often leads to more readable code for certain types of problems, like tree and graph traversal.

🔹 **5. Drawbacks of Recursion**

- **Memory Usage**: Each recursive call adds a new frame to the call stack, and excessive recursion can lead to a **stack overflow** if the recursion goes too deep.
- **Performance**: Recursive functions may be slower than their iterative counterparts due to the overhead of function calls and returning values.
- **Debugging Difficulty**: Recursion can sometimes be more challenging to debug, especially when there are logical errors in the base case or recursive case.

Example: Fibonacci Sequence (Recursive)
The Fibonacci sequence is defined as:

+91 96622 16697
jay@frameboxxers.com
F-201, Shilp Square-B, Nr. Shreeji Tower,
Drive-in Road, Vastrapur, Ahmedabad - 380015

r
CopyEdit
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2) (for n > 1)

Here's how you can implement this using recursion:
javascript
CopyEdit

```
function fibonacci(n) {
  if (n <= 1) {  // Base case
    return n;
  } else {
    return fibonacci(n - 1) + fibonacci(n - 2);  // Recursive case
  }
}

console.log(fibonacci(6)); // Output: 8 (0, 1, 1, 2, 3, 5, 8)
```

Explanation:
The function calculates the Fibonacci number for a given n by recursively calling itself with smaller values of n until it reaches the base case (n <= 1).

🔹 6. When to Use Recursion

Recursion is particularly useful in scenarios where a problem can be broken down into smaller sub-problems, such as:

- **Tree Traversal**: Recursion is commonly used to traverse hierarchical structures like trees (e.g., binary trees).
- **Graph Traversal**: Recursion is also used in graph traversal algorithms like Depth First Search (DFS).
- **Sorting Algorithms**: Algorithms like Merge Sort and Quick Sort use recursion to divide the problem into smaller sub-problems.
- **Backtracking Problems**: Problems like generating combinations or solving Sudoku puzzles often involve recursion.

🔹 7. Tail Recursion

+91 96622 16697
jay@frameboxxers.com
F-201, Shilp Square-B, Nr. Shreeji Tower,
Drive-in Road, Vastrapur, Ahmedabad - 380015

In a **tail recursion**, the recursive call is the last operation in the function, meaning there are no further computations after the function returns. This can optimize performance by allowing the compiler to optimize the call stack.

Example of Tail Recursion:

javascript
CopyEdit

```
function factorialTailRec(n, accumulator = 1) {
  if (n === 0) {
    return accumulator;
  } else {
    return factorialTailRec(n - 1, n * accumulator);
  }
}

console.log(factorialTailRec(5)); // Output: 120
```

In the tail-recursive version, instead of waiting for the recursion to return a result and then multiplying, the multiplication is done in the recursive call itself, making it more efficient.

🧩 Practice Tasks

1. Implement the **factorial function** using recursion and test it with different numbers.
2. Write a recursive function to calculate the **nth Fibonacci number**.
3. Solve the **Tower of Hanoi** problem recursively.
4. Implement a recursive function to **reverse a string**.
5. Write a function to **flatten a nested array** using recursion.
6. Create a recursive solution for **binary search**.

🧠 Summary

Concept	Explanation
Recursion	A technique where a function calls itself to solve a problem.
Base Case	The condition that stops the recursion.
Recursive Case	The part where the function calls itself on a smaller sub-problem.

+91 96622 16697
jay@frameboxxers.com
F-201, Shilp Square-B, Nr. Shreeji Tower,
Drive-in Road, Vastrapur, Ahmedabad - 380015

Advantages	Simplicity, readability, and elegance in solving problems like tree and graph traversal.
Drawbacks	High memory usage, potential stack overflow, and performance concerns.
Tail Recursion	A special form of recursion where the recursive call is the last operation, enabling optimization.

📌 **Key Takeaways**

- Recursion is a powerful problem-solving technique where functions call themselves.
- Always define a base case to prevent infinite recursion.
- Recursive solutions can be more elegant and easier to implement but can have performance issues due to deep recursion.
- Tail recursion is an optimized form of recursion that reduces the overhead.