

# Recursion – Recursion Tree

## 1. Introduction

A **Recursion Tree** is a **visual representation** of how recursive function calls are executed.

It shows how a problem is broken down into **smaller sub-problems** and how recursive calls branch and expand.

Recursion trees are extremely useful for:

- Understanding recursion flow
  - Analyzing time complexity
  - Explaining recursive algorithms clearly
- 

## 2. What is a Recursion Tree?

A recursion tree:

- Represents each recursive call as a **node**
- Shows recursive calls as **branches**
- Displays how the function expands until it reaches base cases

Each level of the tree represents **one level of recursion depth**.

---

## 3. Why Do We Use a Recursion Tree?

Recursion trees help to:

- Visualize recursive calls step by step
  - Understand overlapping sub-problems
  - Analyze time complexity
  - Explain recursion in exams and viva
  - Debug recursive logic easily
-

## 4. Structure of a Recursion Tree

A recursion tree has:

- **Root node:** Original problem
  - **Internal nodes:** Recursive calls
  - **Leaf nodes:** Base cases
  - **Levels:** Depth of recursion
- 

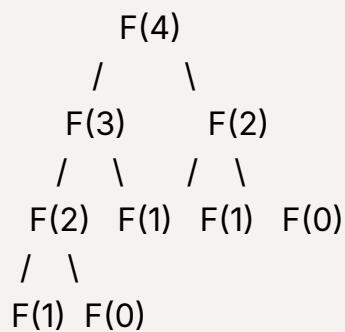
## 5. Basic Idea Behind Recursion Tree

The idea is:

- Each recursive call creates new sub-calls
  - These sub-calls form branches
  - Branching continues until base cases are reached
  - Results return back while the tree collapses
- 

## 6. Example: Recursion Tree for Fibonacci (Conceptual)

For  $F(4)$ :



- Root  $\rightarrow F(4)$
  - Leaves  $\rightarrow F(1)$  and  $F(0)$  (base cases)
- 

## 7. Understanding Overlapping Sub-Problems

From the tree:

- $F(2)$  is computed multiple times
- $F(1)$  is repeated several times

This explains why **recursive Fibonacci is inefficient** and motivates **Dynamic Programming**.

---

## 8. Recursion Tree for Factorial (Conceptual)

Factorial recursion tree is linear:

```
factorial(4)
  |
factorial(3)
  |
factorial(2)
  |
factorial(1)
```

- No branching
  - Single recursive call per level
  - Much more efficient than Fibonacci recursion
- 

## 9. Using Recursion Tree to Find Time Complexity

Steps:

1. Count number of nodes at each level
2. Count total nodes
3. Determine work done at each node
4. Combine results

Example:

- Fibonacci → Exponential growth →  $O(2^n)$

- Factorial → Linear growth →  $O(n)$
- 

## 10. Advantages of Recursion Tree

- Clear visualization of recursion
  - Helps analyze performance
  - Explains recursion depth
  - Useful for teaching and learning
- 

## 11. Limitations of Recursion Tree

- Trees can grow very large
  - Hard to draw for deep recursion
  - Not memory-efficient representation
  - Used mainly for analysis, not execution
- 

## 12. Real-World Importance

- Algorithm analysis
  - Understanding divide-and-conquer algorithms
  - Interview explanations
  - Academic learning and exams
- 

## 13. Common Mistakes While Using Recursion Trees

- Ignoring base cases
  - Forgetting repeated sub-problems
  - Miscounting nodes
  - Assuming all recursion trees are linear
-

## 14. Summary

- Recursion tree visualizes recursive calls
  - Shows branching and depth
  - Helps analyze time complexity
  - Highlights inefficiencies
  - Essential for recursion understanding
-