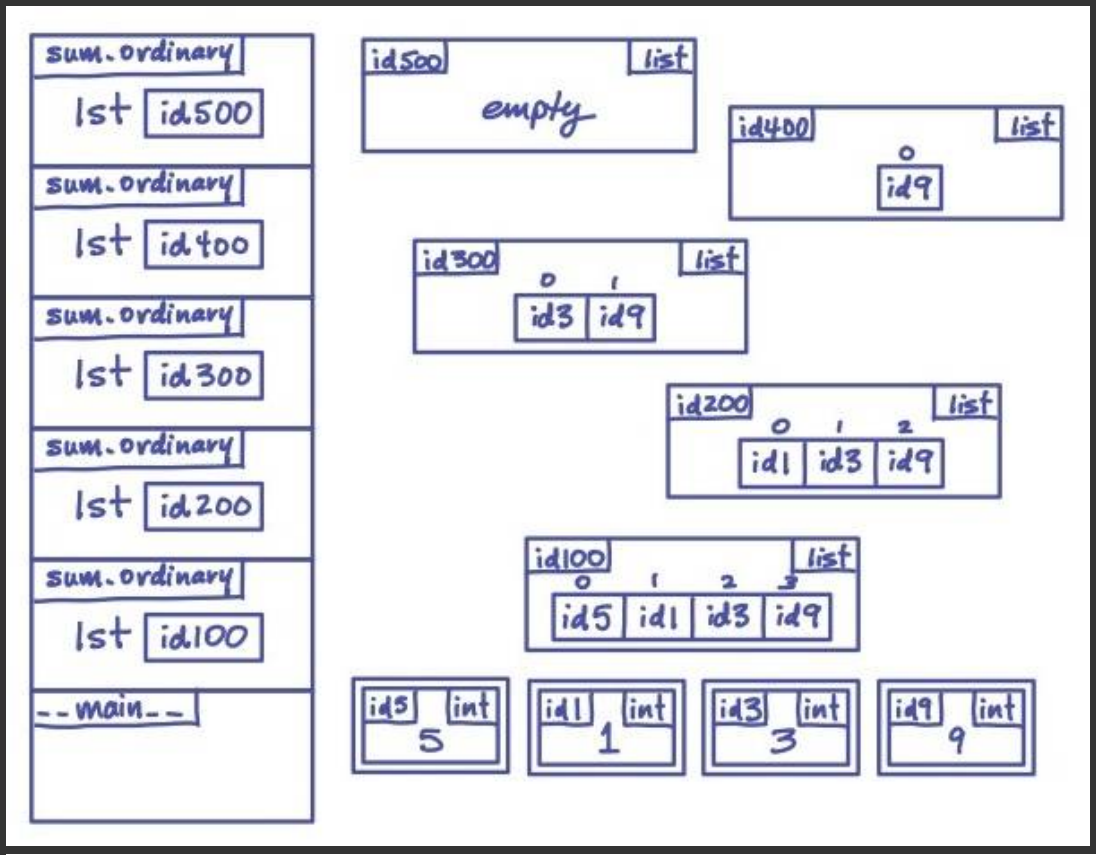
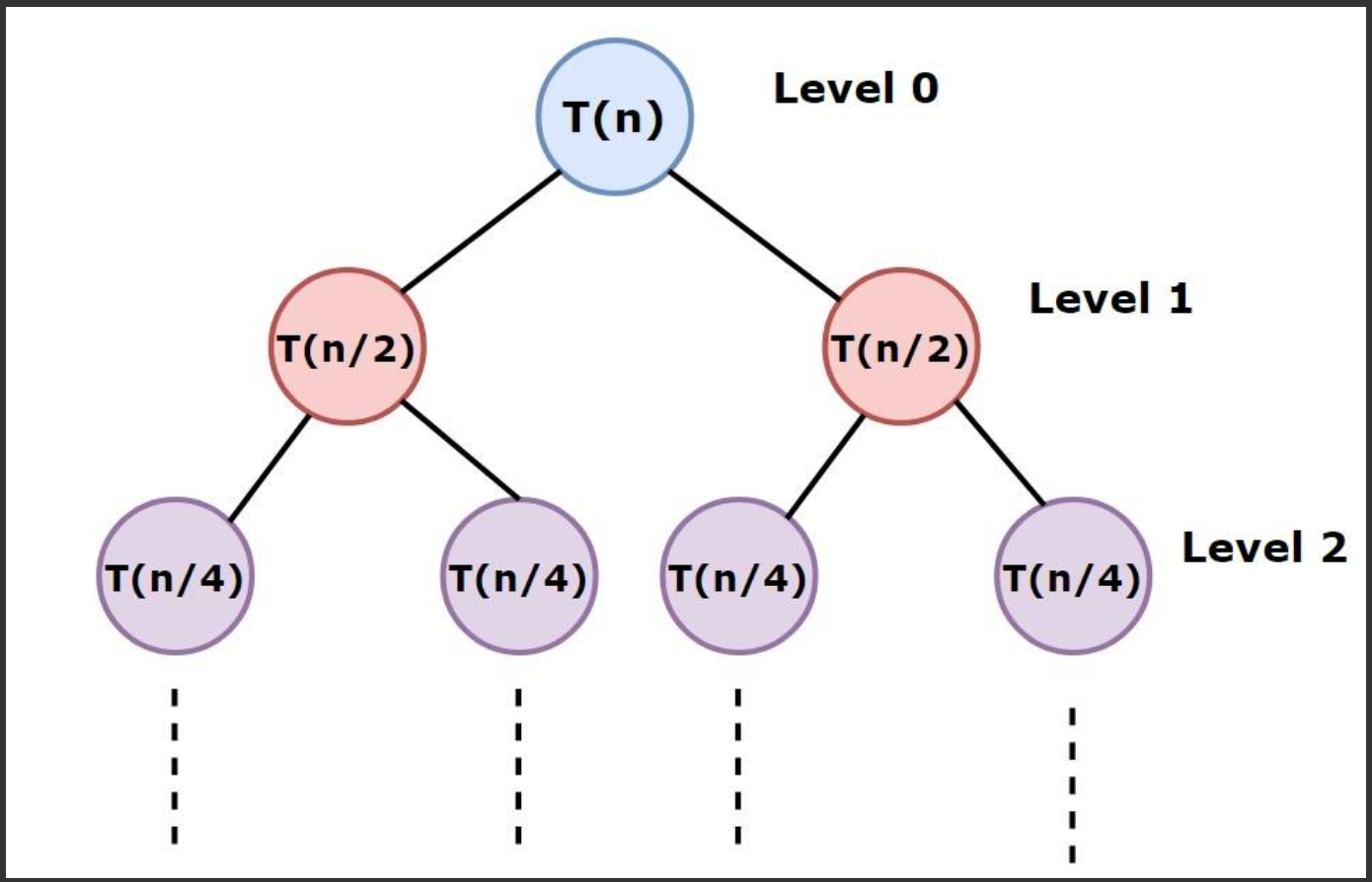
DSA - Recursion Python guide

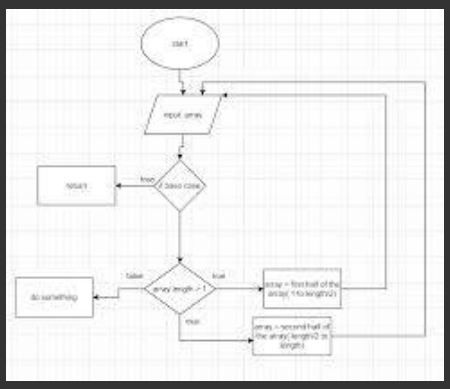
Recursion Python Full Tricks and When Where to use

Here is a **clean, practical, interview-oriented guide to Recursion in Python** — including **full tricks, patterns, when to use, when NOT to use**, and **DSA-type templates**.

🔥 RECURSION IN PYTHON — FULL TRICKS + WHEN & WHERE TO USE


sum.ordinary
lst id500
sum.ordinary
lst id400
sum.ordinary
lst id300
sum.ordinary
lst id200
sum.ordinary
lst id100
--main--
id500 list
empty
id400 list
0
id9
id300 list
0 1
id3 id9
id200 list
0 1 2
id1 id3 id9
id100 list
0 1 2 3
id5 id1 id3 id9
id5 int
5
id1 int
1
id3 int
3
id9 int
9


T(n)
Level 0
T(n/2)
T(n/2)
Level 1
T(n/4)
T(n/4)
T(n/4)
T(n/4)
Level 2



✅ 1. What is Recursion?

A function calling itself **until a stopping condition (base case)** is met.

✔️ Recursion requires:

1. **Base case** — stops recursion
2. **Recursive case** — reduces the problem

```
python

def f(n):
    if n == 0: return
    f(n-1)
```

✅ 2. When to Use Recursion

Use recursion when the problem is naturally defined in **smaller subproblems**:

Best Cases:

Scenario	Why recursion?
Trees (DFS, traversals)	Natural parent → children structure
Graphs (DFS)	Easy for exploring connected components
Backtracking (N-Queens, Subsets, Permutations)	Recursion maintains implicit state
Divide & Conquer	Reduce problem each step (e.g., merge sort)
Mathematical sequences	Factorial, Fibonacci, GCD

❌ 3. When NOT to Use Recursion

Avoid recursion when:

Bad Cases:

- You risk **RecursionError** (`max recursion depth` ≈ 1000)
- Iterative solution is simpler or faster
- Heavy memory usage (stack frames)
- Tail recursion (Python doesn't optimize tail calls)

Prefer loops for:

- DP tabulation
- Long linear problems
- Simple accumulations

🔥 4. Core Recursion TRICK: Think in Terms of "Faith"

Structure almost all recursion as:

Faith + Expectation + Combine

```
python

def solve(n):
    if n == 0: return something_base     # base
    small = solve(n-1)                   # faith
    return process(small, n)             # expectation
```

🔥 5. 10 MUST-KNOW RECURSION PATTERNS (WITH TEMPLATES)

PATTERN 1 — Simple Decreasing Recursion

```
python

def print_1_to_n(n):
    if n == 0: return
    print_1_to_n(n - 1)
    print(n)
```

PATTERN 2 — Backtracking

Useful for **subset**, **permutations**, **N-Queens**, **sudoku**.

```
python

def backtrack(i, path):
    if i == len(nums):
        ans.append(path[:])
        return

    path.append(nums[i])
    backtrack(i+1, path)
```

```
    path.pop()

    backtrack(i+1, path)
```

PATTERN 3 — Tree DFS Traversal

```
python

def dfs(root):
    if not root: return
    dfs(root.left)
    dfs(root.right)
```

PATTERN 4 — Graph DFS

```
python

def dfs(node, visited):
    visited.add(node)
    for nei in graph[node]:
        if nei not in visited:
            dfs(nei, visited)
```

PATTERN 5 — Divide & Conquer

E.g., **binary search**, **merge sort**, **quicksort**.

Binary Search:

```
python

def bs(arr, l, r, x):
    if l > r: return -1
    mid = (l+r)//2
    if arr[mid] == x: return mid
    if x < arr[mid]: return bs(arr, l, mid-1, x)
    return bs(arr, mid+1, r, x)
```

PATTERN 6 — Recursion with Memoization

Important for DP top-down.

```
python

from functools import lru_cache

@lru_cache(None)
def fib(n):
    if n < 2: return n
    return fib(n-1) + fib(n-2)
```

PATTERN 7 — Tail Recursion (Python DOES NOT optimize)

Not recommended in Python but for understanding:

```python
def tail(n, acc=1):
    if n == 0: return acc
    return tail(n-1, acc*n)
```

PATTERN 8 — Multi-branch Recursion

Used in **combinatorics**.

```python
def climb(n):
    if n <= 1: return 1
    return climb(n-1) + climb(n-2)
```

PATTERN 9 — Two-Pointer Recursion

Useful in string problems.

```python
def is_pal(s, l, r):
    if l >= r: return True
    return s[l] == s[r] and is_pal(s, l+1, r-1)
```

PATTERN 10 — Recursion on Arrays

```python
def sum_arr(arr, i=0):
    if i == len(arr): return 0
    return arr[i] + sum_arr(arr, i+1)
```

🔥 6. Recursion Debugging Tricks

✔️ Trick 1 — Add tracing prints

```python
def f(n):
    print("Enter:", n)
    if n == 0: return 0
    ans = n + f(n-1)
```

```
    print("Exit:", n)
    return ans
```

✔️ Trick 2 — Think from the top, not step-by-step

Recursive thinking → What does **f(n)** give assuming **f(n−1)** works.

✔️ Trick 3 — Use memoization for exponential recursions

🚀 Improves

- Fibonacci $O(2^n \rightarrow O(n))$
- Climbing stairs $O(2^n \rightarrow O(n))$

🔥 7. Recursion Cheat Sheet (Quick Rules)

Rule	Meaning
Always write **base case first**	Prevent infinite recursion
Shrink the input	Reduce towards base case
Trust the recursion	Don't simulate manually
Avoid heavy loops inside recursion	Leads to exponential time
Prefer **DP/tabulation** when depth is huge	Avoid stack overflow

---

🔥 8. Real Interview Questions Solved with Recursion

Let me know if you want full solutions for:

- Generate all subsets
- Generate permutations
- N-Queens
- Sudoku solver
- Balanced parentheses
- kth grammar symbol
- Reverse linked list
- Path sum in a tree

❤️ Want me to generate:

✔️ 1. PDF version of this?

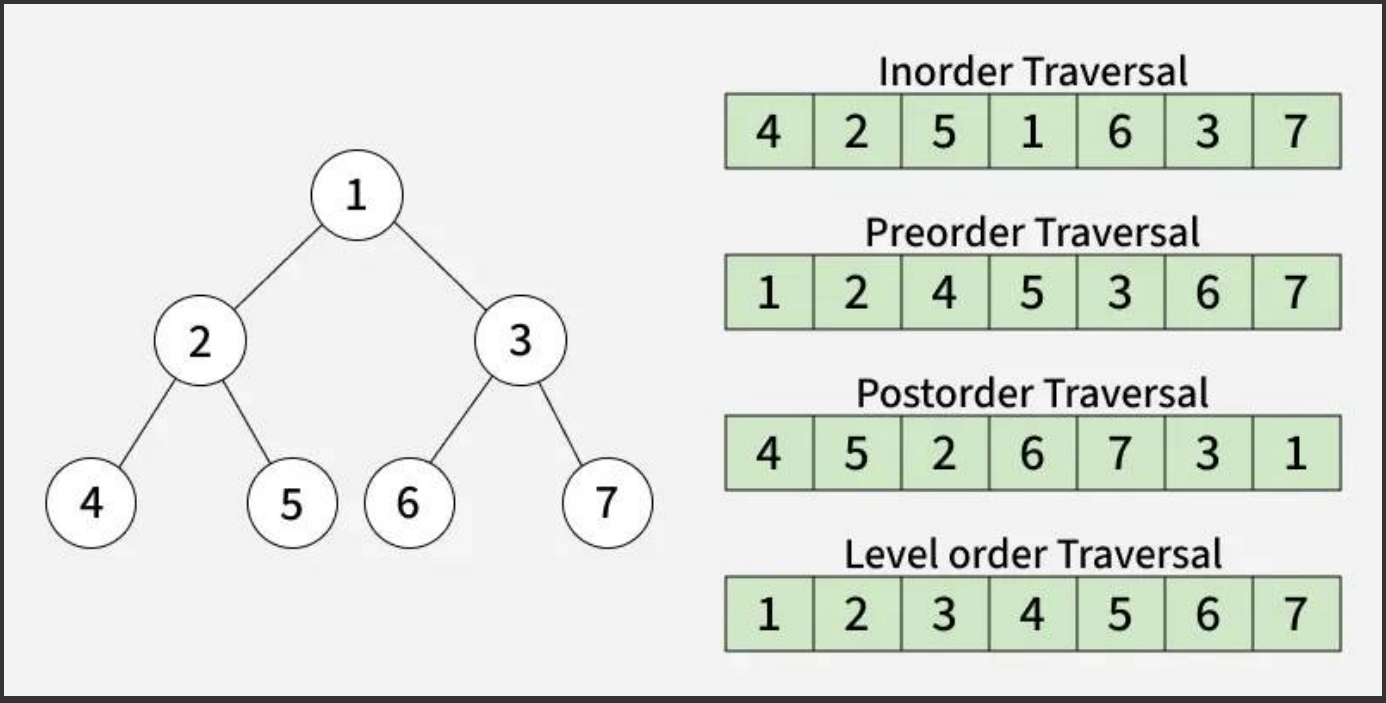
✔️ **2. Recursion 150 Tricks Complete List like your previous DSA request?**

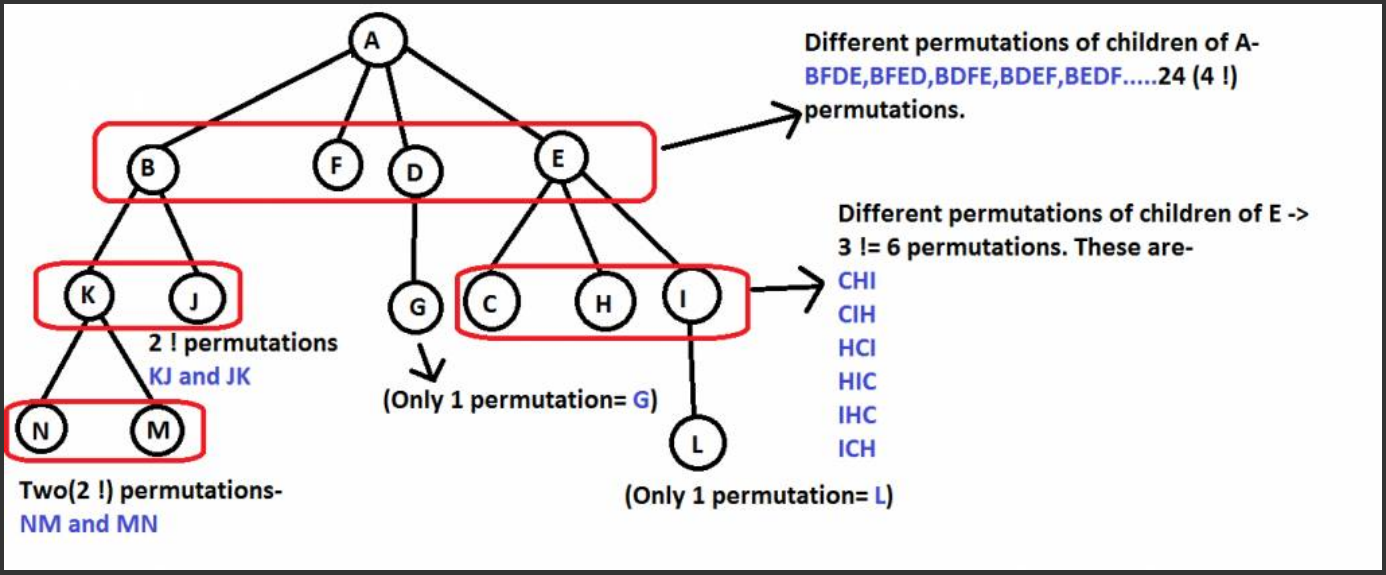
✔️ **3. Recursion practice problems from EASY → HARD with solutions?**

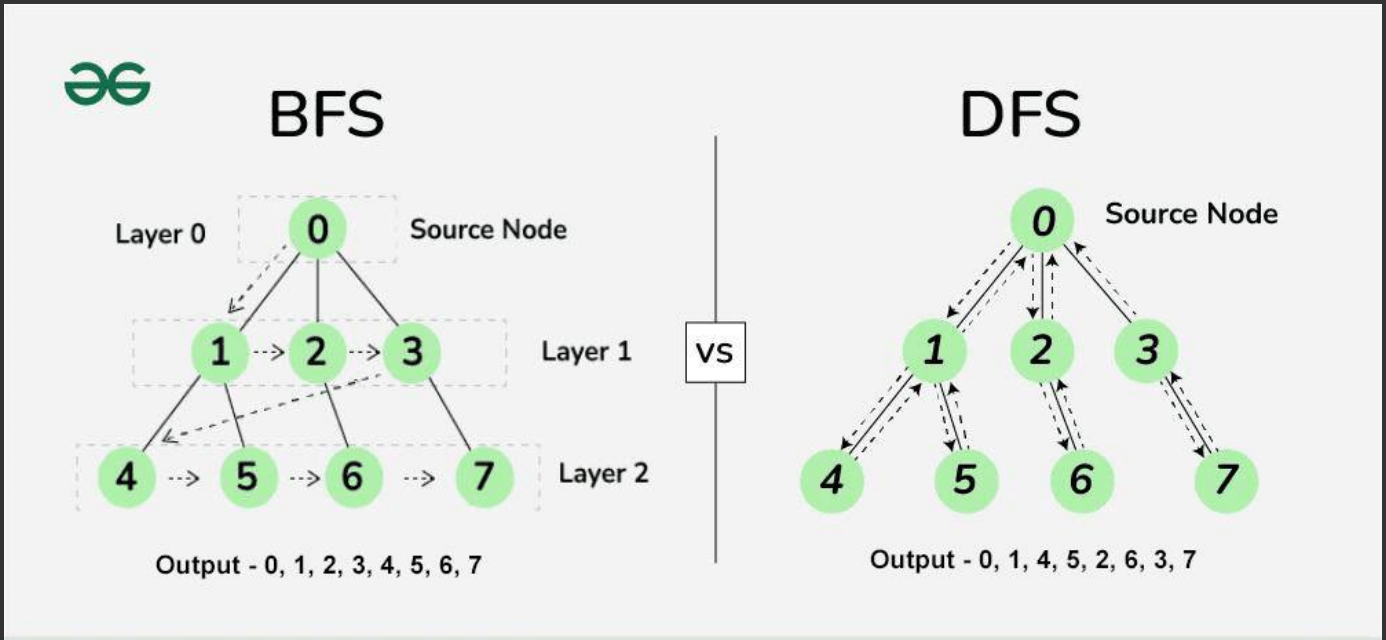
Just tell me **which one**!

Traversal Methods in Tree, Multi Child Trees, Graph

Here is a **complete, clean, interview-oriented guide** to **Traversal Methods in Trees, Multi-Child Trees, and Graphs** — with diagrams, templates, and when to use what.







🌳 1. TRAVERSAL METHODS IN BINARY TREES

Binary tree = each node has **0, 1 or 2 children**.

✅ 1.1 Depth-First Traversals (DFS)

✔️ Preorder (Root → Left → Right)

Used when:

- You want to **clone** a tree
- Convert to prefix expression
- Serialize a tree

```
python

def preorder(root):
    if not root: return
    print(root.val)
    preorder(root.left)
    preorder(root.right)
```

✔️ Inorder (Left → Root → Right)

Used when:

- You want **sorted order** from a Binary Search Tree (BST)

```python
def inorder(root):
    if not root: return
    inorder(root.left)
    print(root.val)
    inorder(root.right)
```

✔️ Postorder (Left → Right → Root)

Used for:

- **Deleting/freeing** a tree
- Evaluating postfix expressions

```python
def postorder(root):
    if not root: return
    postorder(root.left)
    postorder(root.right)
    print(root.val)
```

✅ 1.2 Level-Order Traversal (BFS)

Breadth-first traversal.

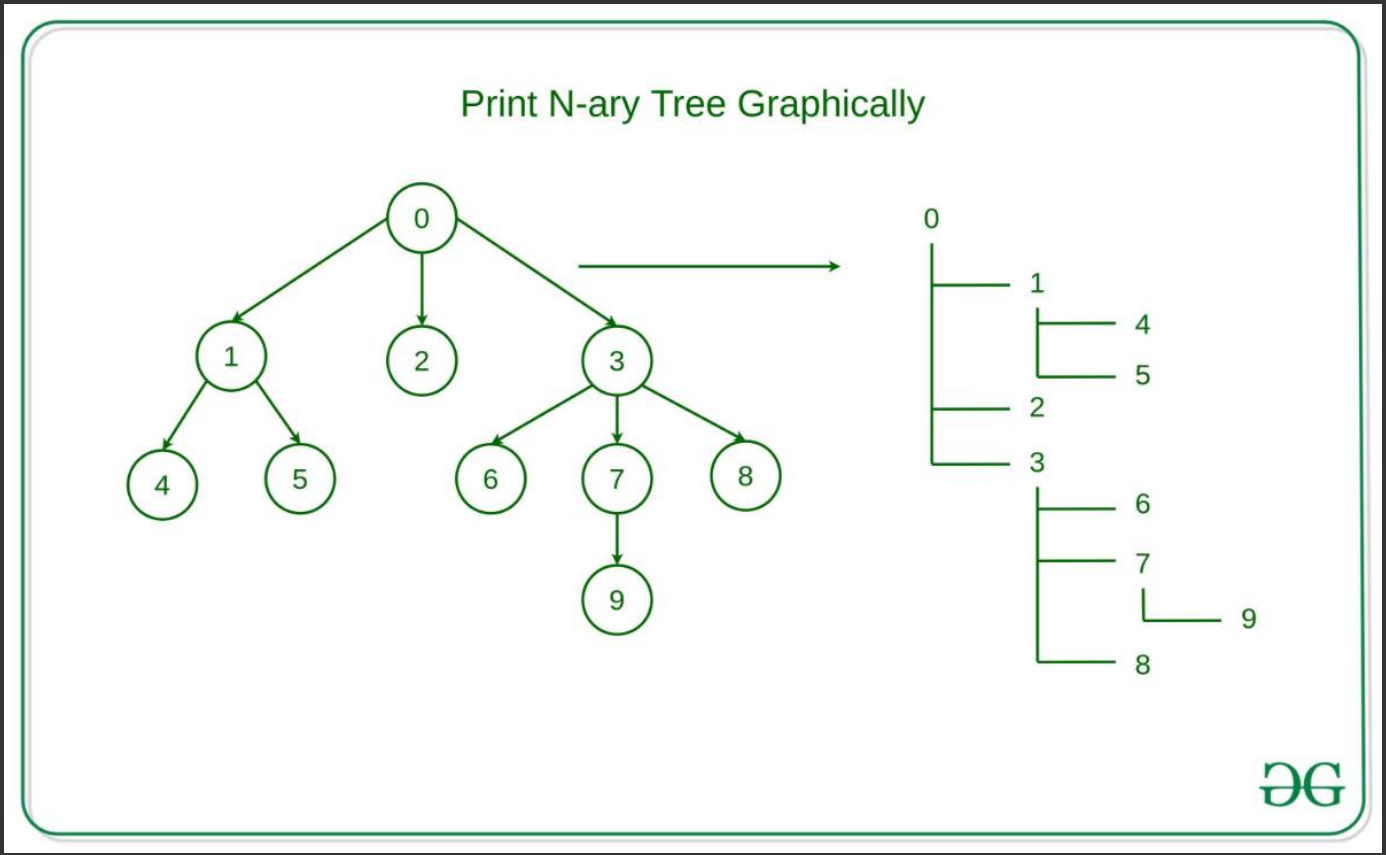
Used for:

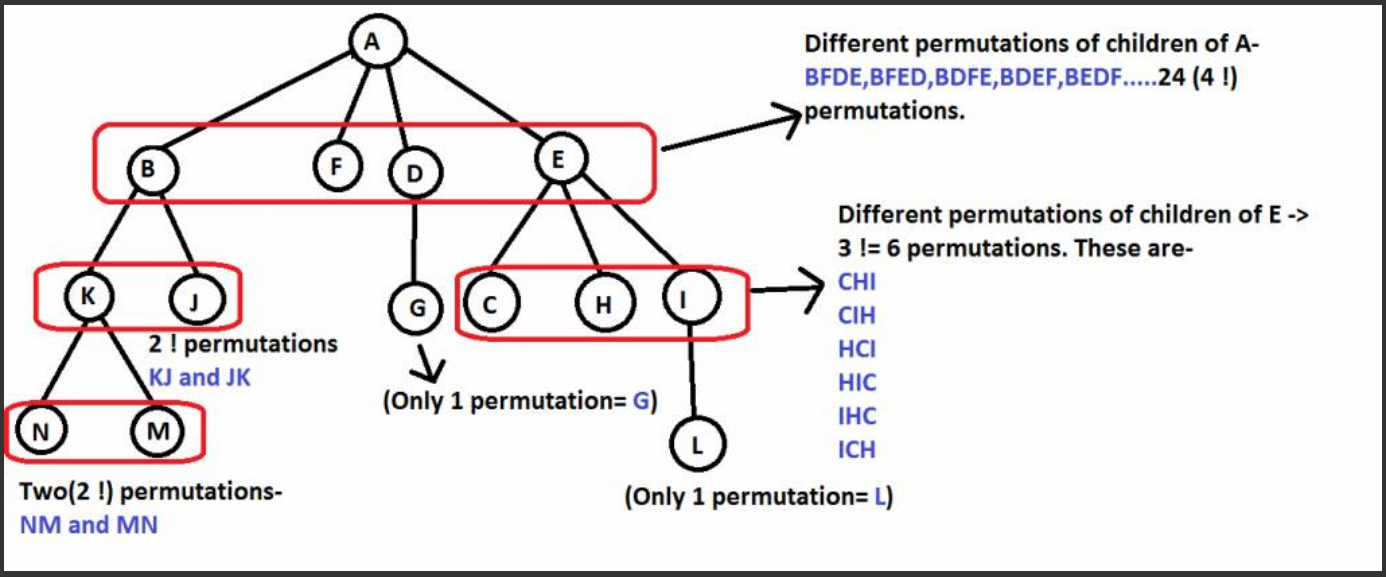
- Shortest path in a tree
- Zig-zag traversal
- Printing by levels

```python
from collections import deque

def level_order(root):
    if not root: return
    q = deque([root])
    while q:
        node = q.popleft()
        print(node.val)
        if node.left: q.append(node.left)
        if node.right: q.append(node.right)
```

🌲 2. TRAVERSAL METHODS IN MULTI-CHILD (N-ARY) TREES

N-ary tree → a node can have **more than two children**.





✅ 2.1 DFS Traversal in N-ary Trees

✔️ Preorder — Root → Children (left to right)

```
python

def preorder(root):
    if not root: return
    print(root.val)
    for child in root.children:
        preorder(child)
```

✔️ Postorder — Children → Root

```
python

def postorder(root):
    if not root: return
    for child in root.children:
        postorder(child)
    print(root.val)
```

✅ 2.2 Level-Order (BFS)

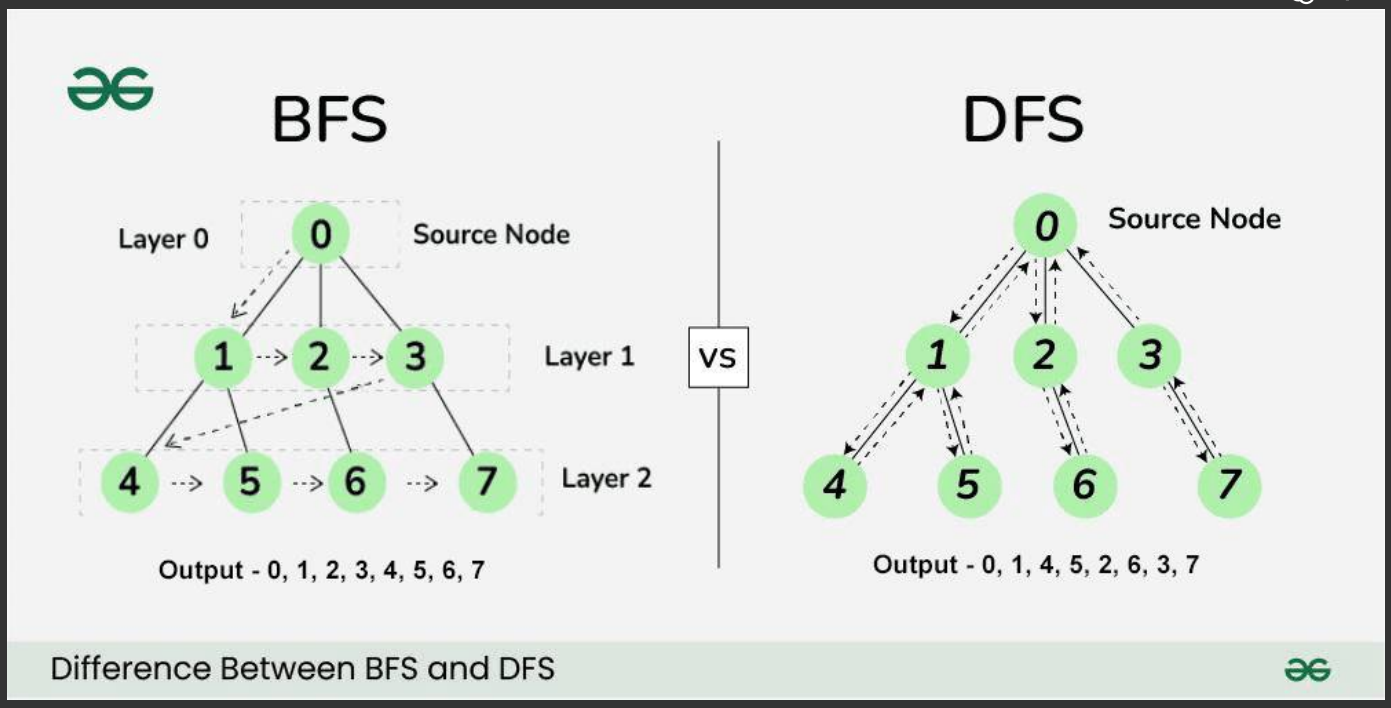
```
python

from collections import deque

def level_order(root):
    if not root: return
    q = deque([root])
    while q:
        node = q.popleft()
        print(node.val)
        for child in node.children:
            q.append(child)
```

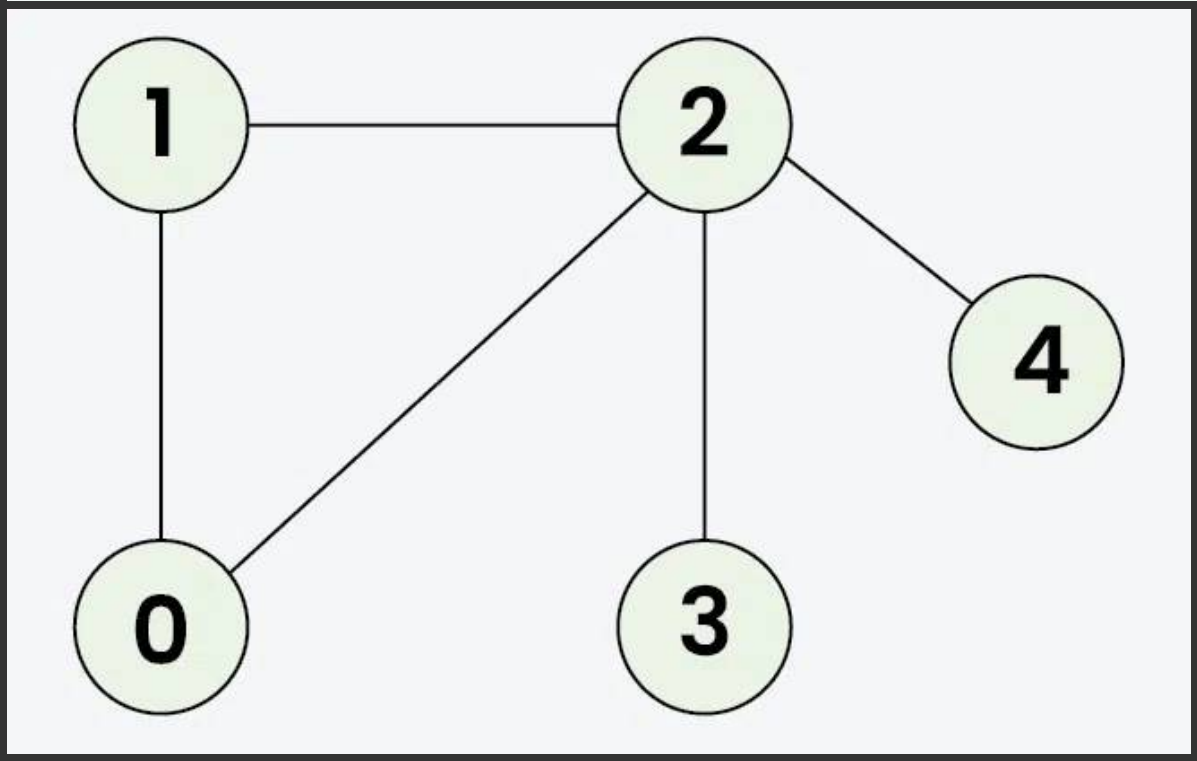
⭐ Additional Common N-ary Traversals:

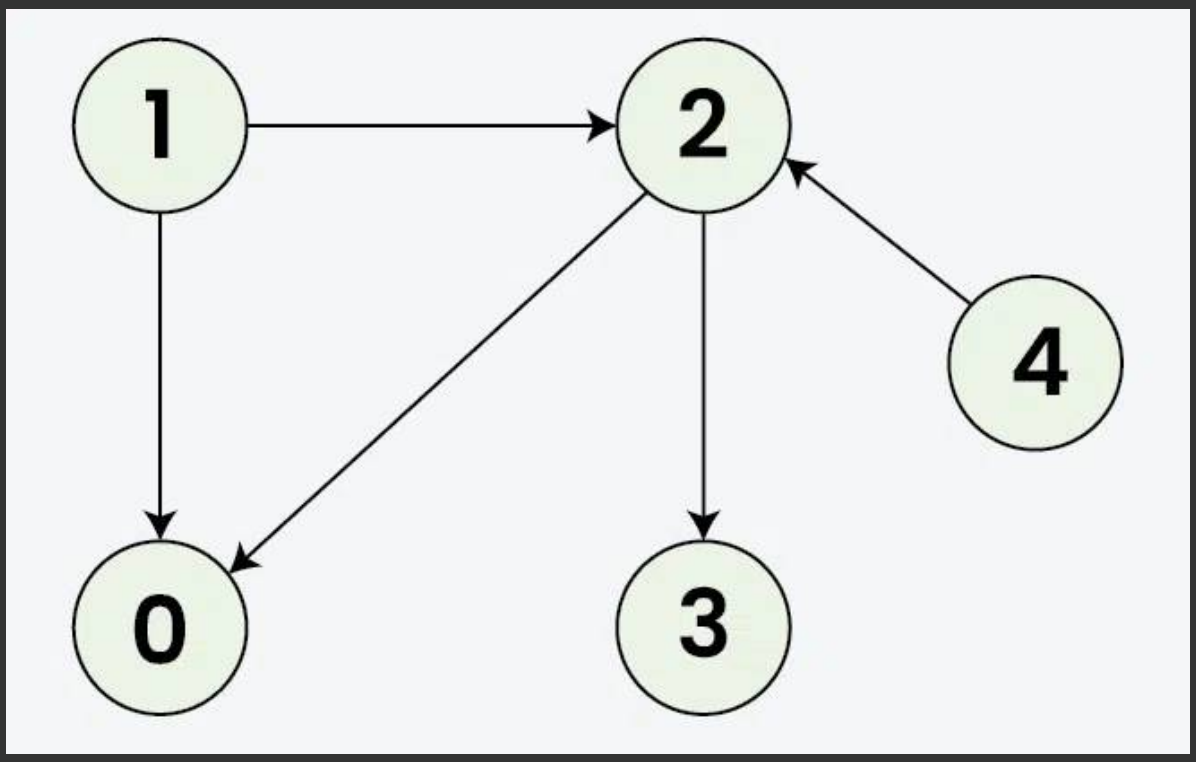
Traversal	Meaning
Zig-zag / Spiral	Alternate between left→right and right→left
Group by depth	Store children at each level in lists
DFS Path Printing	Print all paths root → leaf

🌐 3. GRAPH TRAVERSAL METHODS (UNDIRECTED / DIRECTED)

Graphs can have **cycles**, so traversal must track visited nodes.


BFS
DFS
Layer 0
0
Source Node
1
2
3
Layer 1
vs
4
5
6
7
Layer 2
Output - 0, 1, 2, 3, 4, 5, 6, 7
0
Source Node
1
2
3
4
5
6
7
Output - 0, 1, 4, 5, 2, 6, 3, 7
Difference Between BFS and DFS


1
2
4
0
3



📌 3.1 Depth-First Search (DFS)

Used for:

- Finding connected components
- Detect cycle
- Topological sort (for DAG)
- Path finding
- Tree/Articulation points

🔹 Recursive DFS:

```
python

def dfs(node, visited, graph):
    visited.add(node)
    print(node)
    for nei in graph[node]:
        if nei not in visited:
            dfs(nei, visited, graph)
```

🔹 Iterative DFS:

```
python

def dfs_iter(start, graph):
    stack = [start]
    visited = set()

    while stack:
        node = stack.pop()
```

```
        if node in visited: continue
        visited.add(node)
        print(node)
        for nei in graph[node]:
            stack.append(nei)
```

📌 3.2 Breadth-First Search (BFS)

Used for:

- **Shortest path in unweighted graphs**
- Finding distance levels
- Bipartite checking
- Flood fill

```
python

from collections import deque

def bfs(start, graph):
    q = deque([start])
    visited = set([start])

    while q:
        node = q.popleft()
        print(node)
        for nei in graph[node]:
            if nei not in visited:
                visited.add(nei)
                q.append(nei)
```

🎯 4. Differences Between Traversals

Structure	DFS	BFS
Binary Tree	preorder, inorder, postorder	level order
N-ary Tree	preorder, postorder	BFS
Graph	Stack/recursion	Queue

🎯 5. When to Use What?

✔️ Use DFS when:

- Explore full depth
- You need recursion or backtracking
- Detect cycles
- Solve tree problems

- Generate all possible states

✔️ Use BFS when:

- You need **shortest path**
- Problems with "minimum steps"
- Level-by-level traversal
- Flood-fill problems

🎁 Want More?

I can generate:

✅ A Combined PDF Cheat Sheet

✅ Tree + Graph Traversal 50 Practice Questions (Easy → Hard)

✅ Templates for all interview problems

✅ Animations for traversals (ASCII style)

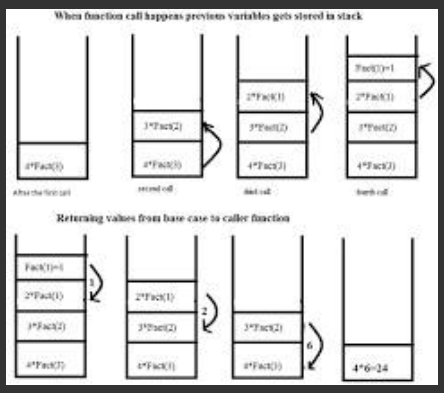
Tell me which one you want!

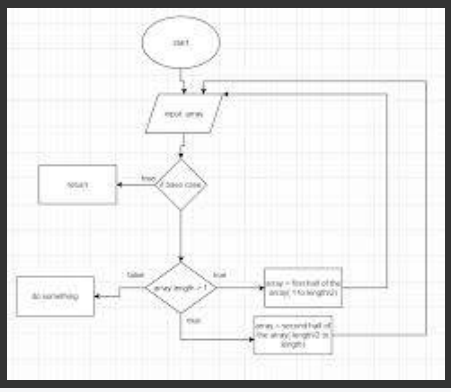
> Recursion Detailed Cheatsheet

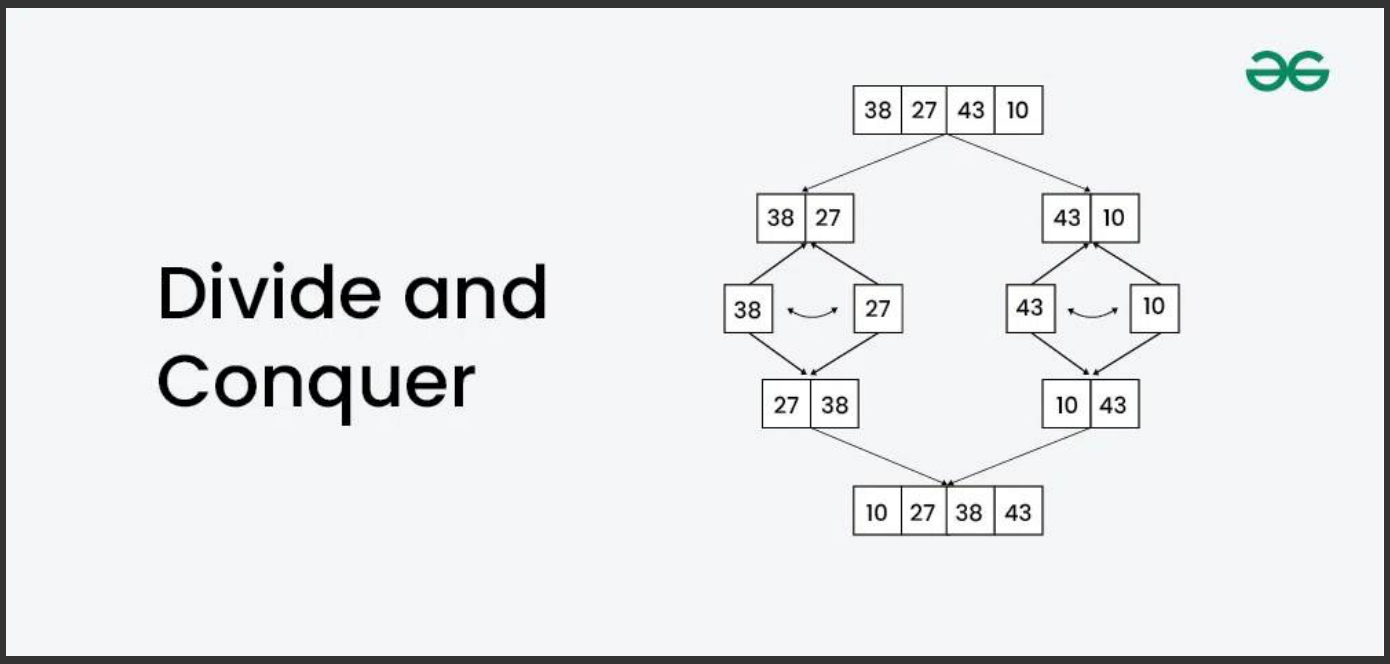
Here is the **Ultimate Recursion Detailed Cheat Sheet** — clean, compact, interview-oriented, but deeply detailed.
This is designed to be your **one-stop reference** for all recursion patterns in **DSA**, **trees**, **graphs**, **backtracking**, **DP**, and **mathematical recursion**.

If you want, I can also generate a **PDF version**.

🧠 RECURSION — DETAILED CHEAT SHEET (MASTER VERSION)







4

1️⃣ What is Recursion?

A function that calls itself until a **base case** is reached.

Every recursion must have:

✔️ Base Case

Stops infinite recursion.

✔️ Recursive Case

Reduces the problem and moves toward the base case.

2️⃣ How to Think Recursively (THE TRICK)

🔥 Master Trick: FAITH → EXPECTATION → COMBINE

1. **Faith:** Assume recursion will solve the smaller problem.
2. **Expectation:** Use that result to solve the current problem.
3. **Combine:** Return or print the final answer.

This mindset solves 90% of recursion problems.

3️⃣ Recursion Types

Type	Used In	Notes
Linear recursion	Factorial, sum of array	One recursive call
Tree recursion	Fibonacci, subsets	Two or more recursive calls
Backtracking	N-Queens, permutations	Explore → undo → explore
Divide & Conquer	Merge sort, binary search	Split → solve → merge
Indirect recursion	Mutually recursive functions	Rare in DSA
Tail recursion	Simple accumulations	Python does **not** optimize tail calls

---

4️⃣ Recursion Core Templates (Use These Again & Again)

⭐ Template 1: Simple decreasing recursion

```python
def f(n):
    if n == 0: return
    f(n-1)
```

⭐ Template 2: Recursion on Arrays

```python
def solve(arr, i=0):
    if i == len(arr): return
    solve(arr, i+1)
```

⭐ Template 3: Tree-like Multi-Branch Recursion

```python
def solve(n):
    if n <= 1: return 1
    return solve(n-1) + solve(n-2)
```

⭐ Template 4: Backtracking

(Try → Recurse → Undo)

```python
def backtrack(i, path):
    if i == len(nums):
        ans.append(path[:])
        return

    path.append(nums[i])      # choose
    backtrack(i+1, path)
    path.pop()                # undo

    backtrack(i+1, path)      # not choose
```

⭐ Template 5: DFS on Tree

```python
def dfs(node):
    if not node: return
    dfs(node.left)
    dfs(node.right)
```

⭐ Template 6: DFS on N-ary Tree

```python
def dfs(node):
    if not node: return
    for child in node.children:
        dfs(child)
```

⭐ Template 7: DFS on Graph

```python
def dfs(node):
    visited.add(node)
    for nei in graph[node]:
        if nei not in visited:
            dfs(nei)
```

⭐ Template 8: Divide & Conquer

```python
def solve(arr):
    if len(arr) == 1: return arr
    mid = len(arr)//2
    left = solve(arr[:mid])
    right = solve(arr[mid:])
    return merge(left, right)
```

5️⃣ Base Case Patterns (Recognize These!)

✔️ NUMBERS

```
if n == 0
if n == 1
```

✔️ ARRAYS / STRINGS

```
if i == len(arr)
if l >= r
```

✔️ TREES

```
if not root
```

✔️ GRAPH DFS

```
if node in visited
```

6️⃣ Common Recursion Problems & Their Patterns

⭐ Pattern A — Counting Ways / Fibonacci type

Tree recursion with 2 branches:

```python
def ways(n):
    if n <= 1: return 1
    return ways(n-1) + ways(n-2)
```

⭐ Pattern B — Subset / Subsequence Problems (2 choices)

```python
def subset(i, path):
    if i == len(arr):
        ans.append(path[:])
        return

    path.append(arr[i])
    subset(i+1, path)
    path.pop()

    subset(i+1, path)
```

⭐ Pattern C — Permutations (swap or used array)

```python
def perm(path):
    if len(path) == len(nums):
        ans.append(path[:])
        return

    for i in range(len(nums)):
        if not used[i]:
            used[i] = True
            path.append(nums[i])
            perm(path)
            path.pop()
            used[i] = False
```

⭐ Pattern D — Tree Traversals

Preorder

```python
print(root.val)
dfs(root.left)
dfs(root.right)
```

Inorder

```python
dfs(root.left)
print(root.val)
dfs(root.right)
```

Postorder

```python
dfs(root.left)
dfs(root.right)
print(root.val)
```

⭐ Pattern E — Graph DFS

```python
def dfs(node):
    if node in visited: return
    visited.add(node)
    for nei in graph[node]:
        dfs(nei)
```

⭐ Pattern F — Binary Search (Divide & Conquer)

```python
def bs(arr, l, r, target):
    if l > r: return -1
    mid = (l+r)//2
    if arr[mid] == target: return mid
    if target < arr[mid]:
        return bs(arr, l, mid-1, target)
    return bs(arr, mid+1, r, target)
```

7️⃣ Recursion Time Complexity Patterns

📌 1. Single branch

```
r

T(n) = T(n-1) + O(1) → O(n)
```

📌 2. Two equal branches

```
r

T(n) = 2T(n-1) → O(2^n)
```

📌 3. Divide & Conquer

```
r

T(n) = 2T(n/2) + O(n) → O(n log n)
```

📌 4. Backtracking worst-case

```scss
O(2^n), O(n!)
```

8️⃣ Recursion vs Iteration (When to Use What)

Use Recursion When	Use Iteration When
Natural tree structure	Simple loops
Backtracking / branching	Deep recursion > 1000 (Python limit)
Divide & Conquer	Dynamic programming bottom-up
Path exploration	Stack memory must be minimal

9️⃣ Common Recursion Mistakes

❌ Missing base case
❌ Wrong return value
❌ Changing list without undo (backtracking error)
❌ Infinite recursion (no shrink step)
❌ Not using memoization when needed

🔟 Memoization (TOP-DOWN DP)

Optimize heavy recursion:

```
python


from functools import lru_cache

@lru_cache(None)
def solve(n):
    if n <= 1: return n
    return solve(n-1) + solve(n-2)
```

1️⃣1️⃣ Backtracking — Full Pattern

Every backtracking problem must follow:

✔️ choose

✔️ explore (recursive)

✔️ un-choose (undo)

```
python


def backtrack(i):
    if i == end:
        process()
        return

    for choice in choices:
        make(choice)
        backtrack(i+1)
        undo(choice)
```

1️⃣2️⃣ Cheat Codes for Recursion Questions

✔️ If a problem asks for all combinations → subsets → permutations → paths, use backtracking.

✔️ If sorted output from BST → use inorder recursion.

✔️ **If shortest path → BFS not recursion.**

✔️ **If "minimum/maximum" in tree → simple DFS recursion.**

✔️ **If counting ways → Fibonacci style tree recursion.**

✔️ **If string palindromes → two pointer recursion.**

✔️ **If graph problem → DFS recursion with `visited`.**

🎁 WANT MORE?

I can generate:

🔥 **(A) 150 Recursion Tricks (like your "150 DSA Tricks" request)**

🔥 **(B) Full Recursion Practice Roadmap (Easy → Hard → Expert)**

🔥 **(C) One-page printable PDF Cheat Sheet**

🔥 **(D) Visual recursion trees for each pattern**

What would you like next?

A

Perfect — here it is.

🔥 150 RECURSION TRICKS (ULTIMATE INTERVIEW PLAYBOOK)

The most complete recursion cheat list you will ever need — structured, categorized, and optimized for DSA + interviews.

Use it as your **master reference** for LeetCode, interviews, and CP.

🧩 SECTION 1 — CORE RECURSION FUNDAMENTALS (1–20)

- **1. Always define base case first**
- **2. Always shrink input toward base case**
- **3. Recursion = solve a smaller subproblem + combine**
- **4. Use "FAITH → EXPECTATION → MERGE" thinking**

- 5. Recursion replaces a loop with stack frames
- 6. Print recursion entry/exit to debug
- 7. Prefer recursion for tree-like problems
- 8. Python max recursion ≈ 1000 → avoid deep linear recursion
- 9. Convert recursion to iteration using explicit stack
- 10. Time complexity grows with branching factor
- 11. Space complexity grows with recursion depth
- 12. Always check input validity before recursion
- 13. Global variables require careful backtracking
- 14. Use recursion for "generate all" problems
- 15. Use iteration for DP bottom-up
- 16. Memoize expensive recursive calls
- 17. Choose the smallest representation of state
- 18. Avoid passing large lists by value — use indexes
- 19. Recursion tree diagrams help understand complexity
- 20. Return results upward, don't print unless required

🌿 SECTION 2 — BASE CASE & INPUT SHRINKING PATTERNS (21–35)

- 21. `if n == 0:`
- 22. `if i == len(arr):`
- 23. `if l > r:`
- 24. `if not root:`
- 25. `if node in visited:`
- 26. `if sum < 0:` (optimization for subsets)
- 27. Shrink by index: `i+1`

- 28. Shrink by value: `n-1`
- 29. Shrink range: `(l+1, r-1)`
- 30. Move pointer in strings/arrays
- 31. Pick/remove–restore elements
- 32. Use sorted data to prune recursion
- 33. Combine results from children in trees
- 34. Return multiple values as tuple
- 35. Use optional parameters to control recursion flow

□ SECTION 3 — LINEAR RECURSION TRICKS (36–50)

- 36. Factorial pattern
- 37. Sum of array using `i+1`
- 38. Reverse array using `l, r` pointers
- 39. Recursively compute GCD
- 40. Recursively search an element
- 41. Recursively compute powers (fast exponentiation)
- 42. Tail recursion is NOT optimized in Python
- 43. Use helper function to carry state
- 44. Avoid unnecessary branching
- 45. Use accumulator variables
- 46. Return boolean expressions directly
- 47. Bubble-sort recursion possible but useless
- 48. Use recursion to simulate stacks
- 49. Freeze parameters to avoid mutation
- 50. Use default args carefully in recursion

🌲 SECTION 4 — TREE STRUCTURES RECURSION (51–80)

- 51. Preorder = Root → L → R
- 52. Inorder = L → Root → R
- 53. Postorder = L → R → Root
- 54. DFS natural with recursion
- 55. Compute tree height using recursion
- 56. Sum of nodes
- 57. Path-sum computation
- 58. Check balanced tree
- 59. Construct tree recursively
- 60. Traversals for BST extraction
- 61. LCA using recursion
- 62. Mirror tree recursively
- 63. Count leaves internal nodes
- 64. Serialize/Deserialize tree
- 65. Prune branches based on condition
- 66. Insert into BST
- 67. Delete from BST
- 68. Validate BST recursively
- 69. Recover BST using recursion
- 70. Evaluate expression trees
- 71. DFS on N-ary trees
- 72. Postorder on N-ary: process children first
- 73. Preorder on N-ary: process root first
- 74. Count children recursively

- 75. Maximum depth N-ary
- 76. Minimum depth binary tree
- 77. Find diameter of tree
- 78. Count good nodes
- 79. Find kth smallest in BST (inorder)
- 80. Convert sorted array to BST (divide & conquer)

SECTION 5 — GRAPH RECURSION TRICKS (81–105)

- 81. DFS implemented recursively
- 82. Mark visited immediately
- 83. Use adjacency list
- 84. Detect cycles with recursion stack
- 85. Topological sort via DFS
- 86. Find connected components
- 87. Graph coloring
- 88. Bipartite check
- 89. Recursive flood fill
- 90. Count islands in matrix
- 91. DFS in grid (4/8 directions)
- 92. Maze pathfinding
- 93. Detect bridges (Tarjan Algorithm)
- 94. Detect articulation points
- 95. Strongly connected components (Kosaraju uses DFS twice)
- 96. DFS for cycle detection in directed graph
- 97. Use recursion depth = number of nodes in worst case
- 98. Use tuple (r, c) as visited key in matrix

- 99. Avoid revisiting by marking on the grid
- 100. Use DFS to count clusters
- 101. Use recursion to generate adjacency lists
- 102. DFS for Euler path
- 103. DFS for Hamiltonian paths
- 104. Graph-based backtracking uses DFS tree
- 105. Always consider stack overflow in large graphs

♟️ SECTION 6 — BACKTRACKING MASTER TRICKS (106–135)

- 106. Choose → Explore → Undo
- 107. Maintain a `path` list
- 108. Maintain `used` array for permutations
- 109. Prune invalid states early
- 110. Sort input to remove duplicates
- 111. Backtracking always returns nothing
- 112. Undo step is crucial
- 113. Use helper function to isolate recursion
- 114. N-Queens uses column/diag sets
- 115. Generate subsets using include/exclude
- 116. Generate permutations via swapping
- 117. Combination Sum uses index to avoid duplicates
- 118. Partition problem uses recursion on remaining sum
- 119. Use recursion to generate binary strings
- 120. Restore IP addresses
- 121. Generate parentheses
- 122. Sudoku solver: recursive backtrack

- 123. Word search in grid
- 124. Palindrome partitioning
- 125. Letter combinations of phone
- 126. Combinations from 1...n choose k
- 127. Gray code generation
- 128. Build dirs from paths recursively
- 129. DFS-based backtracking for longest path
- 130. Use `yield` for recursive generators
- 131. Use memo + backtracking for pruning
- 132. Multi-branch recursion tree visualizes possibilities
- 133. Use lexicographical ordering via recursion
- 134. Always consider time complexity of full recursion tree
- 135. Use caching when recomputation is allowed

🧮 SECTION 7 — MATHEMATICAL RECURSION TRICKS (136–150)

- 136. Fibonacci naive = exponential
- 137. Fibonacci memo = linear
- 138. Fast power using divide & conquer
- 139. Tower of Hanoi recursion
- 140. Recursively check palindrome
- 141. Recursively compute binomial coefficient
- 142. Recursively generate Pascal triangle
- 143. Recursively compute digital root
- 144. Recursively flatten list
- 145. Recursively compute LCM via GCD
- 146. Recursively reverse number

- **147. Recursively generate k-th grammar**
- **148. Recursively compute integer partitions**
- **149. Solve Josephus problem using recursion**
- **150. Recursively derive recurrence relations**

🎉 DONE!

You now have the **Complete 150 Recursion Tricks Cheatsheet**.

💬 Want More?

I can create:

📌 A Printable PDF of this complete cheat sheet

📌 A Poster-style visual diagram of the 150 tricks

📌 A Recursion Practice Sheet (100 Problems: Easy → Hard)

📌 A Recursion Patterns Booklet (with code templates)

Tell me which one you want!

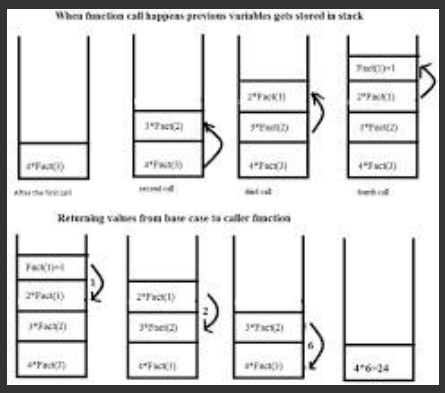
Detailed Roadmap to Master Recursion in LC

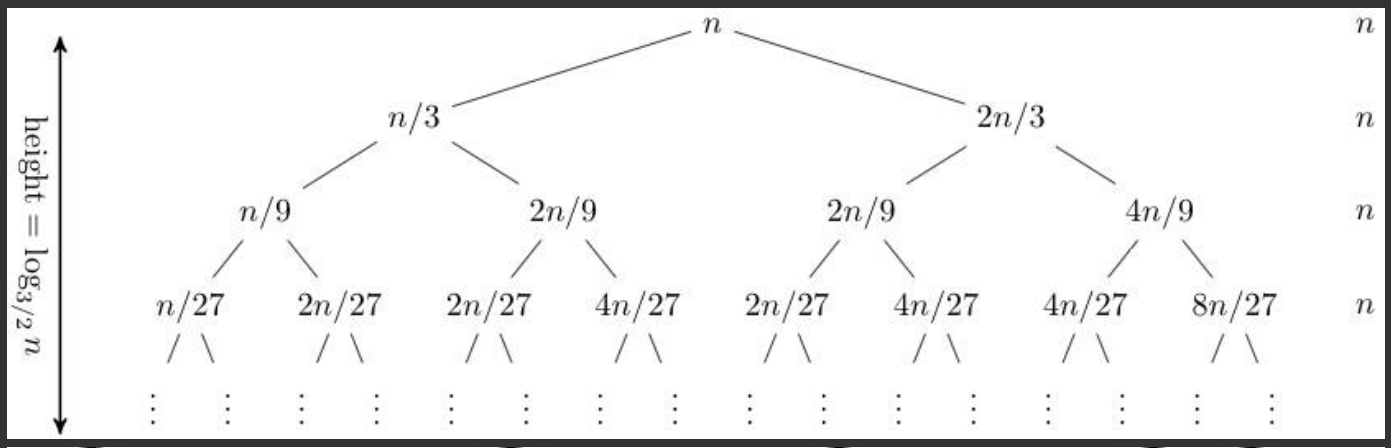
Below is the **Ultimate, Detailed, Structured Roadmap to Master Recursion for LeetCode (LC)** — designed to take you from *beginner → expert*, with curated problem sets, patterns, milestones, and progression paths.

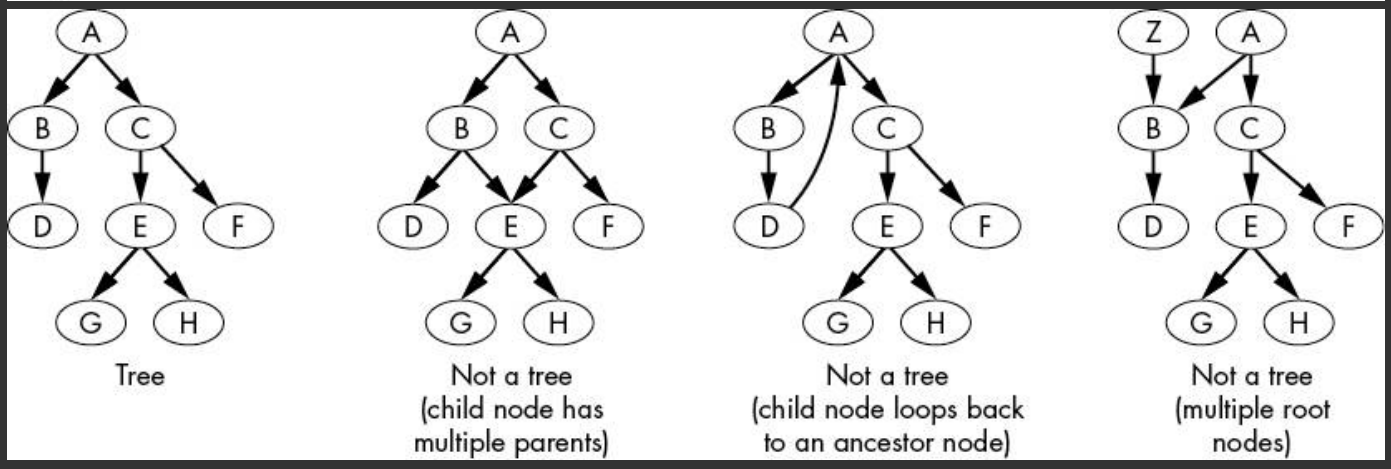
This roadmap is engineered for:

- DSA mastery
- FAANG interviews
- Competitive coding
- Understanding recursion deeply (trees, graphs, backtracking, DP)

🚀 RECURSION MASTERY ROADMAP FOR LEETCODE







🟩 STAGE 0 — Foundations of Recursion (Absolute Basics)

Goal: Learn how recursion works, what the stack does, and how to trace it.

🟦 Concepts to Learn:

- What is a base case?
- What is the recursive case?
- How the call stack expands & collapses
- How to trace a simple recursive function
- Limitations (Python recursion depth)

📝 Practice These Very Basic Problems:

1. Print 1 → N (simple recursion)
2. Factorial
3. Fibonacci (naive)
4. Reverse a string
5. Sum of array recursively

Time: 1 day

🟩 STAGE 1 — Linear Recursion Patterns

Goal: Master single-branch recursion using arrays/strings.

🟦 Learn These Patterns:

- Recursion with index (`i`)
- Recursion with two pointers (`l, r`)
- Recursion with accumulator variables
- Base-case-first technique

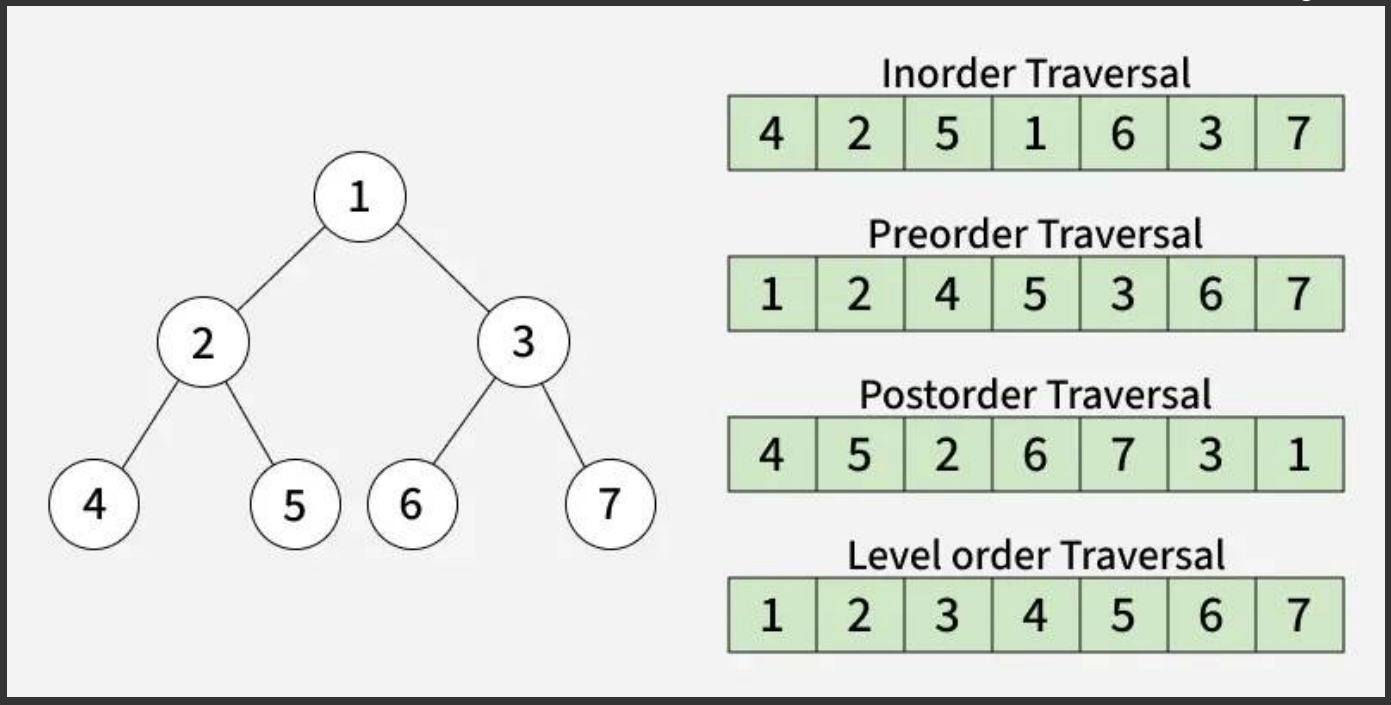
📝 LeetCode Problems:

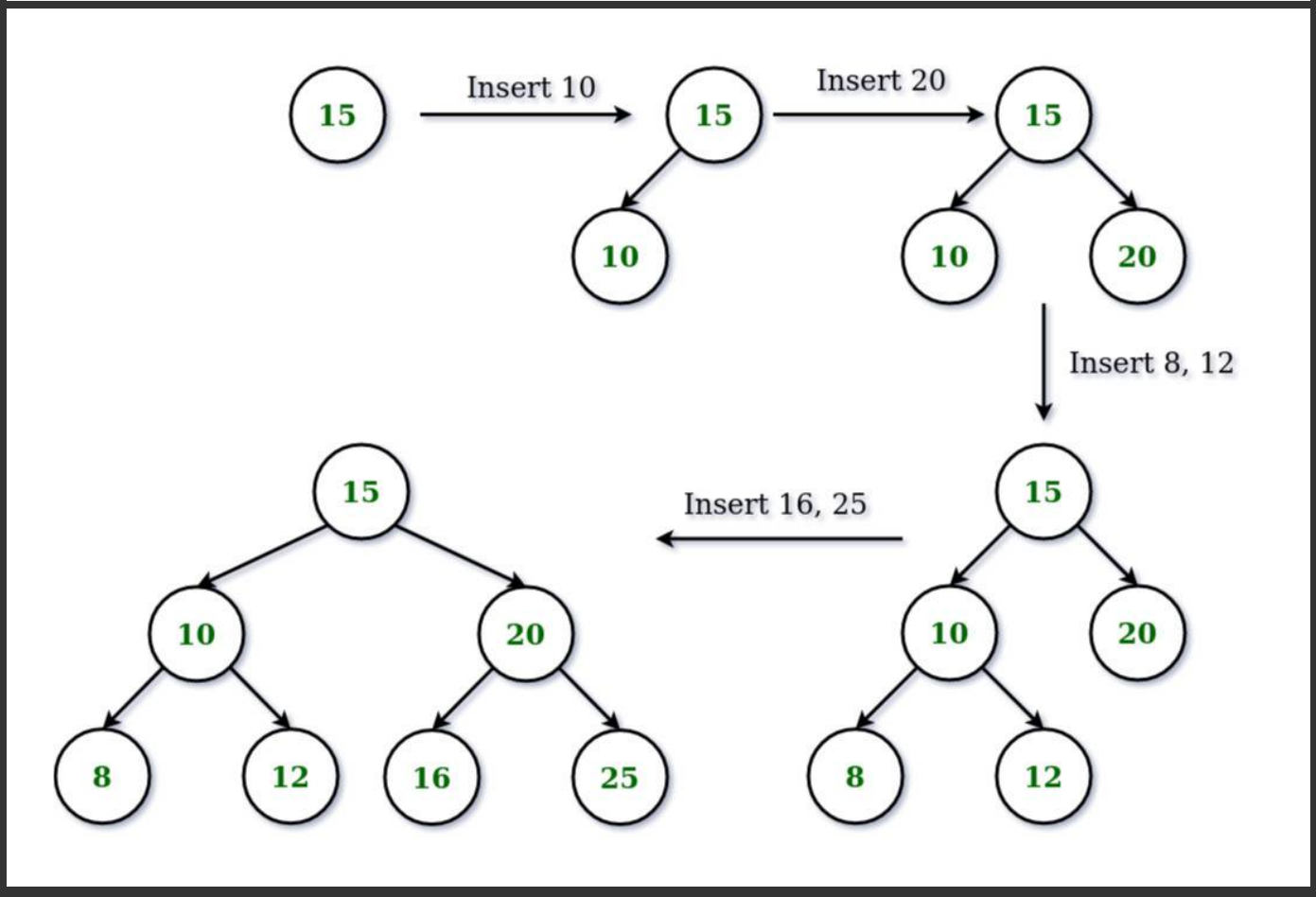
1. **344. Reverse String**
2. **226. Invert Binary Tree** (simple structural recursion)
3. **21. Merge Two Sorted Lists** (recursive version)
4. **24. Swap Nodes in Pairs**

Time: 1–2 days

🟩 STAGE 2 — Tree Recursion (DFS Mastery)

Goal: Build comfort with recursion on binary trees.





🟦 Learn These:

- Preorder / Inorder / Postorder
- DFS patterns (return value, void-style)
- Combining results from left + right child
- Recursive structural problems

📝 Core LC Problems:

1. **104. Maximum Depth of Binary Tree**
2. **110. Balanced Binary Tree**
3. **543. Diameter of Binary Tree**
4. **112. Path Sum**
5. **257. Binary Tree Paths**
6. **100. Same Tree**
7. **101. Symmetric Tree**

🧠 MASTER PATTERN:

```sql

left = dfs(root.left)
right = dfs(root.right)
return combine(left, right, root)
```

Time: 3–4 days

🟩 STAGE 3 — N-ary Tree Recursion

Goal: Strengthen multi-branch recursion intuition.

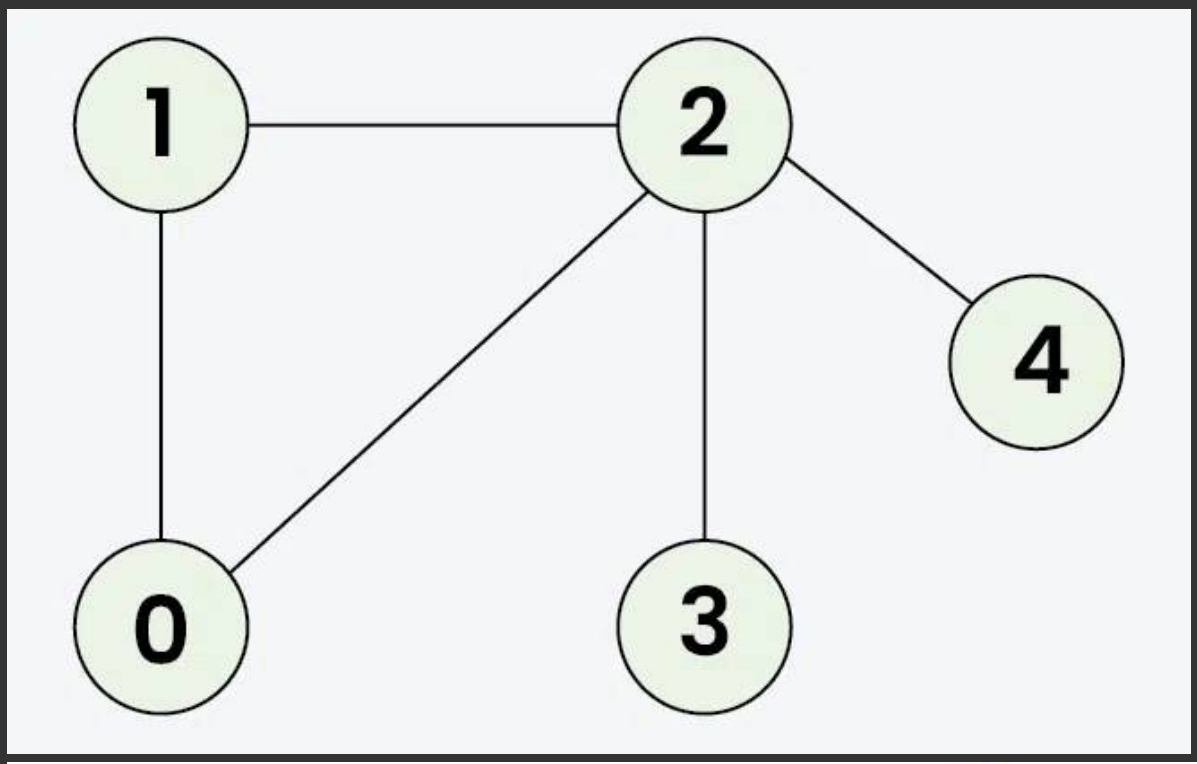
📝 LC Problems:

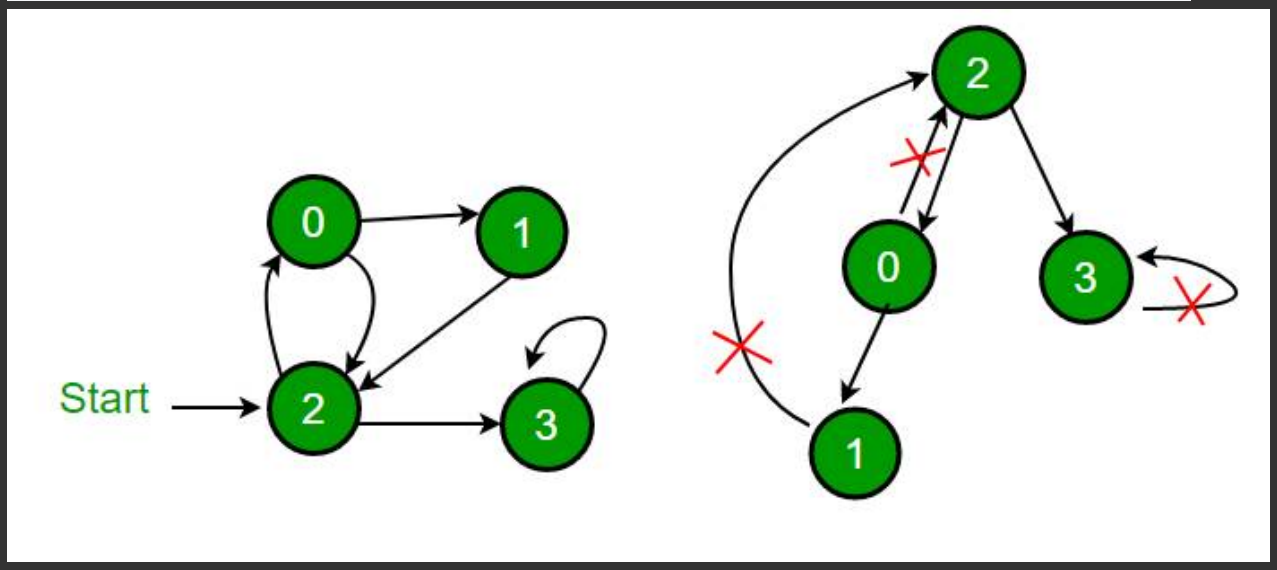
1. **589. N-ary Tree Preorder Traversal**
2. **590. N-ary Tree Postorder Traversal**
3. **429. N-ary Tree Level Order**

Time: 1 day

🟩 STAGE 4 — Graph DFS Recursion (Critical for Interviews)

Goal: Learn recursion with visited sets, cycles, adjacency lists.





🟦 Learn These:

- Visited sets
- Recursive DFS on matrix
- Handling cycles (directed vs undirected)
- Multi-directional recursion

📝 LC Problems (Increasing Difficulty):

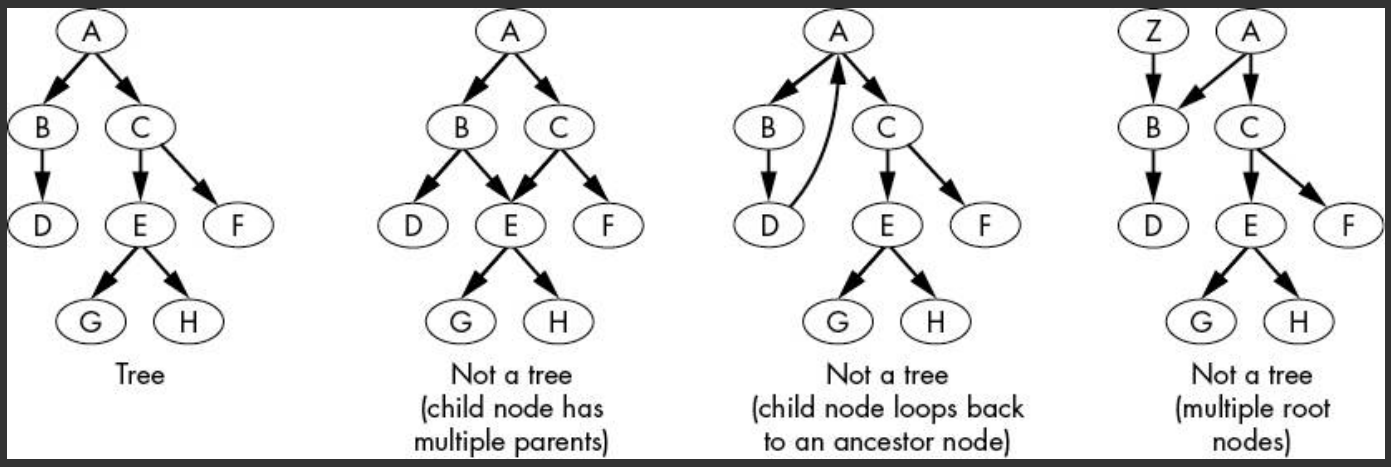
1. **200. Number of Islands**
2. **695. Max Area of Island**
3. **733. Flood Fill**
4. **207. Course Schedule (DFS + cycle detection)**

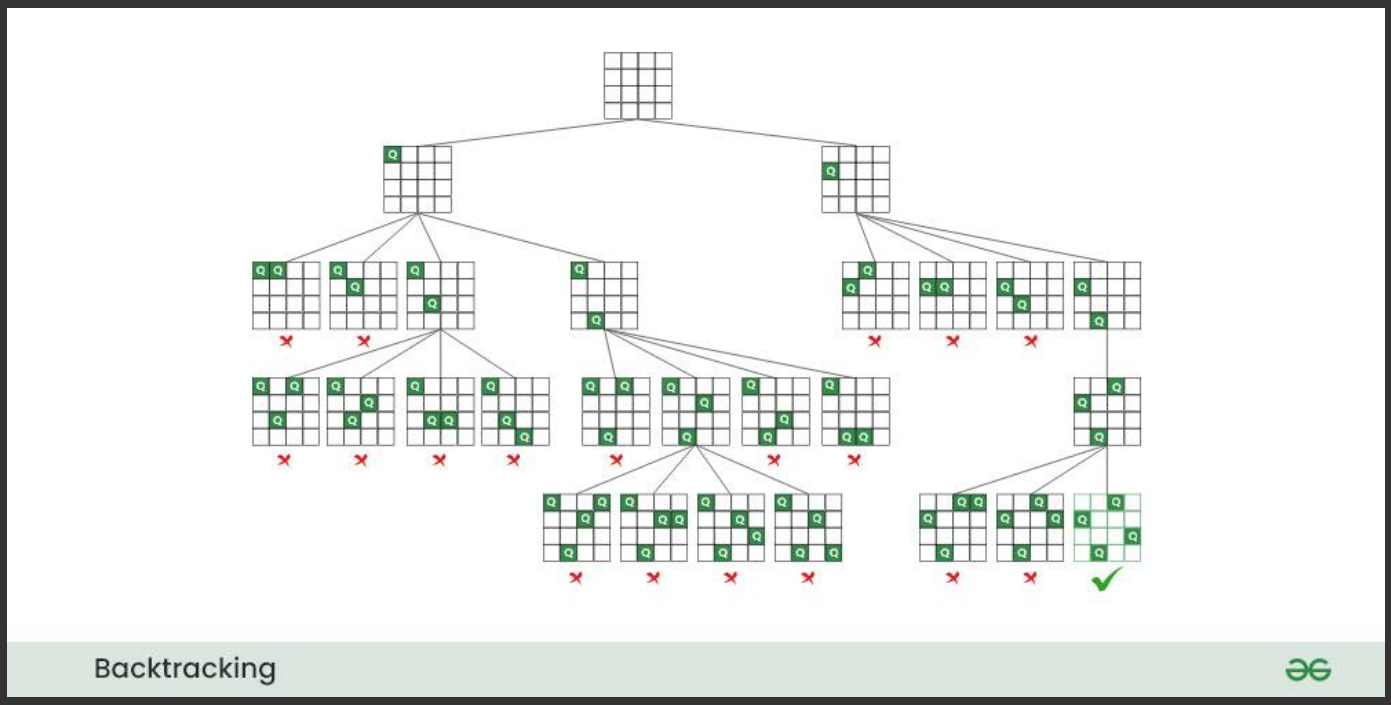
5. **130. Surrounded Regions**

Time: 3 days

🟧 STAGE 5 — Backtracking (Recursion Level UP)

Goal: Master CHOOSE → EXPLORE → UNDO.





📘 Learn Patterns:

- Subsets (include/exclude)
- Permutations (used array)
- Combinations (incremental index)
- N-Queens (state tracking)
- Palindromic recursion
- Multi-branch decision trees

🧠 BACKTRACKING TEMPLATE:

```python
def backtrack(start, path):
    if solution_reached:
        res.append(path[:])
        return

    for i in range(start, len(nums)):
        path.append(nums[i])
        backtrack(i+1, path)
        path.pop()
```

📝 Core LC Backtracking Problems:

Subsets → Combinations → Permutations

1. **78. Subsets**
2. **90. Subsets II**
3. **39. Combination Sum**
4. **40. Combination Sum II**
5. **46. Permutations**
6. **47. Permutations II**

String Backtracking

7. **17. Letter Combinations**
8. **22. Generate Parentheses**
9. **131. Palindrome Partitioning**

Grid Backtracking

10. **79. Word Search**

Hard Backtracking

11. **51. N-Queens**
12. **52. N-Queens II**

Time: 7–10 days

🟧 STAGE 6 — Recursion + Dynamic Programming (DP TOP-DOWN)

Goal: Convert exponential recursion → memoized recursion.

🟦 Learn:

- Recognizing overlapping subproblems

- Memoization with dictionary or `lru_cache`
- Recursion tree optimization

📝 LC Problems:

1. **70. Climbing Stairs**
2. **198. House Robber**
3. **139. Word Break**
4. **322. Coin Change**
5. **416. Partition Equal Subset Sum**
6. **494. Target Sum**

Time: 4–5 days

🔵 STAGE 7 — Advanced Recursion Patterns

📘 Topics:

- Divide & Conquer
- Recursion with multi-state returns
- Recursion building strings
- Recursion with pruning
- Recursion + sorting for deduplication

📝 LC Problems:

1. **241. Different Ways to Add Parentheses**
2. **932. Beautiful Array**
3. **124. Binary Tree Maximum Path Sum**
4. **236. Lowest Common Ancestor of Binary Tree**
5. **297. Serialize and Deserialize Binary Tree**

Time: 4–6 days

🔵 STAGE 8 — Master-Class Recursion + Graph Theory

📘 Advanced Topics:

- Tarjan's algorithm (bridges, articulation points)
- Strongly connected components
- Recursion depth optimization

- DFS-based DP on trees (tree DP)

📝 Problems:

1. **323. Number of Connected Components**
2. **207. Course Schedule (revisit)**
3. **210. Course Schedule II**
4. **1245. Tree Diameter** (tree DP)

Time: 5 days

🎯 FINAL STAGE — Recursion Grandmastery Challenge

Solve these **14 high-power recursion + backtracking problems**:

🔥 Absolute Must-Solves:

- **51. N-Queens**
- **212. Word Search II**
- **10. Regular Expression Matching**
- **44. Wildcard Matching**
- **282. Expression Add Operators**
- **698. Partition to K Equal Sum Subsets**
- **52. N-Queens II**
- **301. Remove Invalid Parentheses**
- **980. Unique Paths III**
- **37. Sudoku Solver**
- **802. Find Eventual Safe States**
- **140. Word Break II**
- **968. Binary Tree Cameras**
- **2246. Longest Path With Different Adjacent Characters**

If you master these → **your recursion level = Expert**.

🧭 TOTAL TIME TO MASTER RECURSION

Level	Time
Beginner	1 week
Intermediate	2–3 weeks
Strong	1 month

Level	Time
Expert	6–8 weeks

📌 BONUS: Daily Recursion Routine

20 minutes:

- 1 tree recursion
- 1 backtracking problem
- 1 DFS grid problem

Weekly:

- 1 advanced heavy recursion problem
- 1 memoization rewrite

💬 Want me to generate next:

✔️ A complete Recursion LC Study Plan (30 days)?

✔️ A PDF printable roadmap?

✔️ A curated Recursion LC playlist (with difficulty order)?

✔️ Code templates for every pattern in this roadmap?

Just tell me!