Recursion and Backtracking

1. How Recursion Truly Works

At its heart, recursion is the process of a function calling itself to solve a smaller version of the same problem. To avoid an infinite loop, a recursive function must have two key parts:

1. Base Case: This is the simplest possible version of the problem, the "stopping condition." When the function receives this simple case, it returns a result directly without calling itself again.
2. Recursive Step: This is the "work" part. The function breaks the current problem into a smaller, simpler piece and calls itself on that smaller piece. It then uses the result from that recursive call to solve the current problem.

Part	Meaning		
✅ Base Case	The condition to stop recursion (the simplest input)		
🔁 Recursive Case	The part where the function calls itself with a smaller input		

Backtracking is a specific, advanced form of recursion. It's a technique for finding solutions to a problem by exploring all possible paths. If a path leads to a dead end or an invalid solution, it "backtracks" (returns from the recursive call) to the previous decision point and tries a different option.

2. Visualizing with the Call Stack

The magic behind recursion is the call stack. Think of it as a stack of plates.

- When a function is called, its state (local variables, parameters) is placed on top of the stack like a new plate.
- If that function calls itself, another new "plate" for the new call is placed on top of the first one.
- When a function hits its base case and returns, its plate is removed from the top of the stack, and execution returns to the function below it.

This Last-In, First-Out (LIFO) process ensures that the function "unwinds" in the correct order, with each call finishing its work after the smaller calls it depended on have finished.

Example: Factorial of N

```
int factorial(int n) {
    if (n == 0) return 1;              // ✅ Base Case     return n *
factorial(n - 1);       // 🔁 Recursive Case}
```

3. How Recursion Works Internally (Stack)

Every recursive call is pushed onto the **call stack**. When a base case is reached, functions start **returning** (unwinding the stack).

Think Like a Stack:

```
factorial(3)
→ 3 * factorial(2)
    → 2 * factorial(1)
         → 1 * factorial(0)
             → returns 1
        → returns 1
    → returns 2→
returns 6
```

💡 **LIFO** (Last In, First Out): Last function called is the first to complete.

4. Infinite Recursion & Stack Overflow

⚠️ If there's **no base case** or it's not reachable, the function keeps calling itself → causes a **StackOverflowError** in Java!

```
int recurse(int n) {
    return recurse(n - 1); // ❌ No base case}
```

5. Recursive Mindset

Ask yourself:

- Can I divide this problem into smaller parts?
- What's the simplest version of this problem (base case)?
- Is the recursive call moving toward the base case?

6. Try This Exercise (Dry Run on Paper)

Print numbers from N to 1 using recursion:

```
void printNto1(int n) {
    if (n == 0) return;    System.out.println(n);
    printNto1(n - 1);}
```

Try for `n = 3`. ✍️ Trace on paper:

```
printNto1(3)
→ print 3
→ print 2
→ print 1
→ return
```

Backtracking & Recursive Search

> **Goal:** Learn how to *try all possibilities*, *backtrack when needed*, and *build solutions step-by-step*.
> This is the core of problems like **N-Queens**, **Sudoku Solver**, and **Combinations**.

What is Backtracking?

> "Try → Check → Backtrack → Try Next"

You explore possible solutions using recursion, **undo** the step (backtrack) when the current path fails, and **try other paths**.

> Classic structure:

```
void backtrack(...) {
    if (solution found) return;

    for (choices) {
        make choice
        backtrack
        undo choice // 🔙 backtrack step
    }}
```

Core Problems to Master Backtracking

1. Subsets (Power Set)

📦 Input: `[1, 2, 3]`
🎯 Output: `[[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]`

```
void subsets(int[] nums, int index, List<Integer> current, List<List<Integer>>
result) {
    if (index == nums.length) {
        result.add(new ArrayList<>(current));
        return;
    }
    // include    current.add(nums[index]);
    subsets(nums, index + 1, current, result);

    // exclude (Backtrack part)    current.remove(current.size() - 1);
    subsets(nums, index + 1, current, result);
}
```

2. Subset Sum (Find all subset sums)

```
void subsetSums(int[] arr, int i, int sum, List<Integer> result) {
    if (i == arr.length) {
        result.add(sum);
        return;
    }
    subsetSums(arr, i + 1, sum + arr[i], result);  // include

    subsetSums(arr, i + 1, sum, result);  // exclude
}
```

3. Generate All Permutations

📦 Input: `[1,2,3]`
🎯 Output: `[1,2,3], [1,3,2], [2,1,3]...`

```
void permute(int[] nums, int index, List<List<Integer>> result) {

    if (index == nums.length) {
        List<Integer> perm = new ArrayList<>();
        for (int num : nums) perm.add(num);
        result.add(perm);
        return;
    }

    for (int i = index; i < nums.length; i++) {
```

```java
            swap(nums, index, i);
            permute(nums, index + 1, result);
            swap(nums, index, i); // backtrack
    }}
```

4. Palindrome Partitioning

📦 Input: `"aab"`
🎯 Output: `[["a", "a", "b"], ["aa", "b"]]`

```java
void partition(String s, int start, List<String> path, List<List<String>>
result) {
    if (start == s.length()) {
        result.add(new ArrayList<>(path));
        return;
    }

    for (int end = start; end < s.length(); end++) {
        String substr = s.substring(start, end + 1);
        if (isPalindrome(substr)) {
            path.add(substr);
            partition(s, end + 1, path, result);
            path.remove(path.size() - 1);
        }
    }}
```

Other Challenges

- **N-Queens Problem**
- **Sudoku Solver**
- **Word Search**
- **Combination Sum**
- **Rat in a Maze**

Other Patterns

Tree Recursion Patterns

🔹 Pattern 1: DFS Traversals (Preorder, Inorder, Postorder)

```
// Inorder: Left → Node → Right
void inorder(TreeNode root) {
    if (root == null) return;    inorder(root.left);
    System.out.print(root.val + " ");
    inorder(root.right);
}
```

- Preorder → `Node → Left → Right`
- Postorder → `Left → Right → Node`

🔹 Pattern 2: Find Height (Max Depth)

```
int height(TreeNode root) {
    if (root == null) return 0;
    return 1 + Math.max(height(root.left), height(root.right));
}
```

📌 Pattern: **Bottom-up recursion** 💡 Key: Return value bubbles up from the children.

🔹 Pattern 3: Count Paths / Path Sum / Root-to-Leaf

```
boolean hasPathSum(TreeNode root, int target) {
    if (root == null) return false;
    if (root.left == null && root.right == null) return root.val == target;
    return hasPathSum(root.left, target - root.val) ||
           hasPathSum(root.right, target - root.val);
}
```

📌 Pattern: **Top-down decision making**, no return value building (in simpler form).

🔹 Pattern 4: Lowest Common Ancestor (LCA)

```
TreeNode lca(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || root == p || root == q) return root;

    TreeNode left = lca(root.left, p, q);
    TreeNode right = lca(root.right, p, q);

    if (left != null && right != null) return root; // current is LCA
    return left != null ? left : right;}
```

📌 **Divide and conquer** with return values.

Linked List Recursion Patterns**

🔹 Pattern 1: Reverse a Linked List Recursively

```
ListNode reverse(ListNode head) {
    if (head == null || head.next == null) return head;
    ListNode newHead = reverse(head.next);
    head.next.next = head;
    head.next = null;
    return newHead;}
```

🔹 Pattern 2: Find Mid Node (Helper for Rec Merge Sort)

```
ListNode findMid(ListNode head) {
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            }
    return slow;
    }
```

📌 Used in **Divide & Conquer** patterns.

🔹 Pattern 3: Merge Two Sorted Linked Lists (Recursively)

```
ListNode merge(ListNode l1, ListNode l2) {
    if (l1 == null) return l2;

    if (l2 == null) return l1;

    if (l1.val < l2.val) {
        l1.next = merge(l1.next, l2);
        return l1;
    } else {
        l2.next = merge(l1, l2.next);
        return l2;
    }
}
```

💡 You can use this as part of **Merge Sort** too.

🔹 Pattern 4: Check Palindrome (Recursive Two Pointer)

```
ListNode frontPointer;

boolean isPalindrome(ListNode head) {
    frontPointer = head;
    return recursivelyCheck(head);
}

boolean recursivelyCheck(ListNode current) {

    if (current == null) return true;
    if (!recursivelyCheck(current.next)) return false;

    boolean isEqual = (current.val == frontPointer.val);
    frontPointer = frontPointer.next;

    return isEqual;}
```

🔥 Classic recursive technique using stack-like behavior.

Templates

Template 1: **Subsequence Pattern**

> Used when you need to include/exclude elements — classic recursion intro!

Core Structure:

```
void solve(int index, List<Integer> current, int[] arr) {
    if (index == arr.length) {
        System.out.println(current);
        return;
    }

    current.add(arr[index]);  // Pick the element

    solve(index + 1, current, arr);

    current.remove(current.size() - 1);  // Backtrack

    solve(index + 1, current, arr);  // Do NOT pick the element
}
```

Used In:

- All Subsequences
- Subset Sums
- Count number of subsequences with sum `k`
- Can be converted into DP with memoization later

Template 2: **Pick / Not Pick with Return Type (Functional Style)**

> Return-type version of subsequence for problems like:
> “Return all subsequences with sum X”, “Return subsets of a string”, etc.

Core Structure:

```
List<List<Integer>> solve(int index, int[] arr, List<Integer> current) {

    if (index == arr.length) {
        List<List<Integer>> base = new ArrayList<>();
        base.add(new ArrayList<>(current));
        return base;
    }

    current.add(arr[index]);   // Pick

    List<List<Integer>> left = solve(index + 1, arr, current);

    current.remove(current.size() - 1);

    List<List<Integer>> right = solve(index + 1, arr, current);   // Not Pick

    left.addAll(right); // Merge results

    return left;
}
```

Used In:

- Return all subsequences
- Power set

- Count/print/return variants
- Palindromic partitions (modified)

Template 3: Tree Recursion (Backtracking Style)

> The ultimate recursive tool: Used in permutations, N-Queens, Sudoku, etc.

Core Structure:

```
void solve(List<Integer> current, boolean[] used, int[] nums) {

    if (current.size() == nums.length) {
        System.out.println(current);
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        if (used[i]) continue;

        current.add(nums[i]);  // Choose
        used[i] = true;

        solve(current, used, nums);   // Recurse

        current.remove(current.size() - 1);  // Backtrack
        used[i] = false;
    }}
```

Used In:

- Permutations
- Combination Sum
- N-Queens
- Sudoku
- Path Finding in grid/maze

Key Pitfalls in Recursion and Backtracking

1. Missing or Incorrect Base Case

This is the most fundamental error in any recursive algorithm. The **base case** is the condition that stops the recursion. If it's missing, incorrect, or can never be reached, the function will call itself infinitely.

- **Why it's a problem**: This leads to a **Stack Overflow Error**, as the call stack grows indefinitely until the program runs out of memory and crashes.
- **Example**: A function to calculate a factorial that forgets to stop when `n` equals 0.

---

2. Forgetting to Backtrack (Corrupted State)

This is the signature mistake in backtracking problems. Backtracking involves making a choice, exploring its consequences recursively, and then **undoing that choice**.

- **Why it's a problem**: If you forget to undo your choice (e.g., remove a queen from a board, un-mark a cell as visited, remove an element from a subset), the state remains "dirty." All subsequent recursive calls at that level will operate on a corrupted state, leading to incorrect or missing solutions.
- **Example**: In a Sudoku solver, placing a `5` in a cell, finding it leads to a dead end, but forgetting to reset that cell to `empty`. The algorithm will then proceed as if the `5` was part of the original puzzle.

---

3. Creating Cycles or Redundant Computations

This is common in problems that traverse graphs, grids, or state spaces (like mazes).

- **Why it's a problem**: If you don't keep track of the nodes/states visited *in the current path*, the algorithm can enter an infinite loop by moving back and forth between two nodes. Even without a loop, it leads to massively redundant computations by exploring the same path multiple times.
- **Example**: In a maze pathfinding problem, moving from cell A to B, and then immediately allowing a recursive call to move from B back to A. This is prevented by a `visited` set or by marking the path on the grid itself.

---

4. Inefficient State Passing

This is a subtle but significant performance pitfall, especially in languages like C++.

- **Why it's a problem**: Passing large data structures (like a Sudoku board, a list of results, or the current subset) **by value** in each recursive call creates a full copy of that data structure every single time. This can be extremely slow and memory-intensive.
- **Example**: Passing a `vector<int>` for the current subset by value instead of **by reference**. The program will spend more time copying the vector than solving the actual problem.

---

5. Mishandling the Final Result

Many backtracking problems require you to find all valid solutions or the single best one. Managing how this result is stored can be tricky.

- **Why it's a problem**: If you use a single object to build a path (like a list of nodes) and add it to your final results list, you might be adding a *reference* to that object. When you later backtrack and modify that object, you inadvertently change the version you already saved in your results.
- **Example**: In the "All Paths" problem, adding the `currentPath` list to your results, but then continuing to modify `currentPath` through backtracking. The solution is to add a **deep copy** of `currentPath` to the results list.