Problem Solving Approaches

Premanand S

Assistant Professor
School of Electronics Engineering (SENSE)
Vellore Institute of Technology - Chennai Campus

premanand.s@vit.ac.in

August 14, 2025

What Are Problem Approaches?

- **Definition:** An overall **strategy** or **method** to solve a problem efficiently — a thinking pattern before writing the actual code.
- **Why not just write code directly?** The same problem can be solved in many ways; some are faster or use less memory.
- **Example (Technical):** Problem: Sort a list of numbers Approaches: Bubble Sort (simple but slow) vs Merge Sort (Divide & Conquer, much faster)
- **Example (Layman):** Destination: Another city Approaches: Walk, drive, or take a train — different speeds, costs, and efficiency.

Key Idea

A *problem approach* is the chosen route from the problem to its solution.

Why Do We Need Approaches?

- **Problems are often too big to solve at once** Approaches give a structured way to break them into smaller, manageable parts.
- **Right approach saves time and resources** Example: Searching in a sorted list $\rightarrow$ Binary search (Divide & Conquer) is faster than linear search.
- **Helps in thinking like a problem solver** Recognize patterns and apply them to new problems without starting from scratch.
- **Efficiency matters** Better approaches mean faster execution, less memory usage, and easier maintenance.
- **Bridges the gap between "What" and "How"** Problem = *what needs to be done*, Approach = *how to do it efficiently*.

Analogy

Choosing an approach is like picking the best route for a road trip — the right path saves time, fuel, and avoids traffic.

Types of Problem Approaches

- **Top-Down:** Start from the big problem, break into smaller ones (recursion).
- **Bottom-Up:** Build from the simplest cases to the full problem (iteration).
- **Divide & Conquer:** Split into independent sub-problems, solve, and combine.
- **Backtracking:** Try, explore, and undo when a path fails.

Note

Other approaches exist (e.g., Greedy, Brute Force), but we'll focus on these four today.

Top-Down Approach: Analogy

- **Analogy:** Planning a birthday party
- Think big picture first: Decorations, Food, Games
- Break each into smaller tasks until you can do them directly
- **Mindset:** Big $\rightarrow$ Small

Top-Down Approach: Technical View

Definition

Start with the overall problem and break it into smaller sub-problems until each can be solved directly.

- Focus on planning, then implementation
- Common in **design-first** thinking

Top-Down Approach: Example

```
def print_numbers(n):
    if n == 0:
        return
    print_numbers(n-1)  # big to small
    print(n)

print_numbers(10)
```

Bottom-Up Approach: Analogy

- **Analogy:** Building a house
- Start from foundation $\rightarrow$ walls $\rightarrow$ roof
- **Mindset:** Small $\rightarrow$ Big

Bottom-Up Approach: Technical View

Definition

Solve smaller sub-problems first, then combine them to solve the bigger problem.

- Common in **dynamic programming**
- Implementation builds solution step-by-step

Bottom-Up Approach: Example

```
def fib(n):
    dp = [0, 1]
    for i in range(2, n+1):
        dp.append(dp[i-1] + dp[i-2])  # build from bottom
    return dp[n]

print(fib(10))
```

Divide & Conquer: Analogy

- **Analogy:** Sorting a huge pile of books
- Split into two halves, sort each, merge back
- **Mindset:** Divide $\rightarrow$ Solve $\rightarrow$ Combine

Divide & Conquer: Technical View

Definition

Break the problem into smaller independent sub-problems, solve them recursively, then combine the results.

- Common in sorting/search algorithms
- Uses recursion heavily

Divide & Conquer: Example

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

Divide & Conquer: Example

```
def merge(left, right):
    result = []
    while left and right:
        if left[0] < right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    return result + left + right
```

Backtracking: Analogy

- **Analogy:** Solving a maze
- Try a path, if blocked, go back and try another
- **Mindset:** Try $\rightarrow$ Fail $\rightarrow$ Backtrack

Backtracking: Technical View

Definition

A problem-solving technique where you build solutions incrementally and abandon them if they lead to a dead end.

- Common in puzzles, path finding, N-Queens
- Uses recursion + state undoing

Backtracking: Example

```
def solve_maze(maze, pos, path):
    if pos == (end_x, end_y):
        return True
    for move in moves:
        if valid_move(move):
            path.append(move)
            if solve_maze(maze, move, path):
                return True
            path.pop()  # backtrack
    return False
```