Interview Patterns

Master Essential Problem-Solving Techniques for Coding Interviews

Executive Summary

Coding interview success relies on recognizing and applying established problem-solving patterns rather than memorizing thousands of problems. By mastering key patterns like Two Pointers, Sliding Window, and DFS/BFS, you can solve 90% of interview questions efficiently. This summary introduces the most essential patterns used in technical interviews at FAANG companies and covers practical applications, recognition strategies, and example problems[1].

1. Introduction

Technical interviews test your ability to solve algorithmic problems efficiently under time constraints. Instead of approaching each problem as unique, experienced engineers recognize underlying patterns and apply tested solutions[2]. Pattern-based learning allows you to:

- **Solve problems faster** - Recognize the pattern, apply the solution approach
- **Handle novel variations** - Understand core concepts, adapt to new contexts
- **Explain reasoning clearly** - Demonstrate systematic problem-solving to interviewers
- **Build confidence** - Know you have a toolkit of reliable techniques

The most successful candidates master 10-15 core patterns that cover the majority of interview problems[3].

2. Pattern Recognition Framework

2.1 How to Identify Patterns

When facing an interview problem, ask these questions:

1. What is the data structure? (Array, String, Tree, Graph, LinkedList)
2. What operation is primary? (Search, Modify, Traverse, Optimize)
3. What constraints exist? (Sorted, Finite range, Memory limits)
4. What is being asked to find? (Substring, Subarray, Path, Value)
5. What are the performance requirements? (Time/Space complexity)

Your answers to these questions map to specific patterns[2].

2.2 Pattern Selection Guide

Problem Characteristics	Likely Patterns	Time Complexity
Sorted array, find element	Binary Search	$O(\log n)$
Contiguous subarray/substring	Sliding Window	$O(n)$
Array comparison or traversal	Two Pointers	$O(n)$
Tree/Graph exploration	DFS or BFS	$O(V + E)$
Optimization with choices	Dynamic Programming	Varies
Overlapping intervals	Merge Intervals	$O(n \log n)$
Find subset/combination	Backtracking	$O(2^n)$
Top K elements	Heap/Priority Queue	$O(n \log k)$

Table 1: Pattern Selection by Problem Type

3. Core Interview Patterns

3.1 Two Pointers Pattern

What It Is:
Use two pointers to traverse an array or linked list from different positions simultaneously[3].

When to Use:

- Array is sorted or needs comparison from both ends
- Find pairs or elements meeting a condition
- Reverse or rearrange elements
- Detect cycles in linked lists

Key Variants:

- **Opposite directions** - Start from opposite ends, move toward center (palindrome checking)
- **Same direction** - Both move forward at different speeds (cycle detection, slow/fast pointers)
- **Window-based** - Pointers define boundaries of a region to process

Time Complexity: $O(n)$ typically
Space Complexity: $O(1)$ (excluding output)

Example Problems:

- Two Sum (sorted array)
- Container With Most Water
- Remove Duplicates from Sorted Array
- Palindrome Validation

Sample Code Approach:
```
// Two pointers from opposite ends
int left = 0, right = arr.length - 1;
while (left < right) {
// Compare or process arr[left] and arr[right]
// Move pointers based on condition
if (condition) left++;
else right--;
}
```

3.2 Sliding Window Pattern

What It Is:
Maintain a "window" (subarray/substring) that slides through data to find optimal solutions[4].

When to Use:

- Find longest/shortest subarray or substring with property X
- Linear data structure (array or string)
- Avoid redundant calculations from previous iterations

Window Types:

- **Fixed-size window** - Window size constant (find max sum of K elements)
- **Variable-size window** - Window size changes based on condition (find longest substring)

Time Complexity: $O(n)$ (each element visited at most twice)
Space Complexity: $O(1)$ to $O(k)$ for tracking

Example Problems:

- Maximum Sum Subarray of Size K
- Longest Substring Without Repeating Characters
- Fruits Into Baskets
- Minimum Window Substring

Sample Code Approach:
```
// Variable-size sliding window
int left = 0;
for (int right = 0; right < arr.length; right++) {
// Expand window by adding arr[right]
// Shrink window from left while condition invalid
```

```
while (condition) {
left++;
}
// Process current window [left, right]
}
```

3.3 Binary Search Pattern

What It Is:
Efficiently find elements in sorted data by dividing search space in half repeatedly[5].

When to Use:

- Data is sorted (array, matrix, or conceptually)
- Need to find a value or boundary
- Optimization: minimize/maximize something in sorted space

Key Variants:

- **Standard binary search** - Find exact element
- **Left boundary** - Find first occurrence ($\geq$ target)
- **Right boundary** - Find last occurrence ($\leq$ target)
- **Search in rotated array** - Handle rotation edge case

Time Complexity: $O(\log n)$
Space Complexity: $O(1)$

Example Problems:

- Binary Search (standard)
- Search in Rotated Sorted Array
- Find Peak Element
- Square Root (using binary search on answer)

Sample Code Approach:

```
// Binary search template
int left = 0, right = arr.length - 1;
while (left <= right) {
int mid = left + (right - left) / 2;
if (arr[mid] == target) return mid;
else if (arr[mid] < target) left = mid + 1;
else right = mid - 1;
}
return -1; // Not found
```

3.4 DFS (Depth-First Search) Pattern

What It Is:
Traverse tree or graph by exploring as far as possible along each branch before backtracking[5].

When to Use:

- Tree or graph traversal needed

- Find all paths or solutions
- Check connectivity or cycles
- Backtracking problems

Implementation Styles:

- **Recursive** - Natural for tree structures, uses call stack
- **Iterative** - Uses explicit stack, avoids recursion depth limits

Time Complexity: $O(V + E)$ for graphs, $O(n)$ for trees
Space Complexity: $O(h)$ where h is height (recursion stack)

Example Problems:

- Inorder/Preorder/Postorder Tree Traversal
- Path Sum
- Number of Islands
- All Paths in Tree
- Detect Cycle in Graph

Sample Code Approach:
// DFS recursive approach
void dfs(Node node, List<Node> visited) {
if (node == null || visited.contains(node)) return;
visited.add(node);
process(node);
for (Node neighbor : node.neighbors) {
dfs(neighbor, visited);
}
}

3.5 BFS (Breadth-First Search) Pattern

What It Is:
Traverse tree or graph level-by-level, exploring all neighbors before moving deeper[3].

When to Use:

- Find shortest path in unweighted graph
- Level-order tree traversal needed
- Bipartite checking
- Connected components

Key Characteristics:

- Queue-based (FIFO)
- Guarantees shortest path in unweighted graphs
- Better for problems requiring level information

Time Complexity: $O(V + E)$
Space Complexity: $O(V)$ for queue

Example Problems:

- Binary Tree Level Order Traversal
- Shortest Path in Unweighted Graph
- Bipartite Graph Check
- Number of Islands (alternative to DFS)

Sample Code Approach:

```
// BFS approach with queue
Queue<Node> queue = new LinkedList<>();
Set<Node> visited = new HashSet<>();
queue.add(start);
visited.add(start);

while (!queue.isEmpty()) {
Node current = queue.poll();
process(current);
for (Node neighbor : current.neighbors) {
if (!visited.contains(neighbor)) {
visited.add(neighbor);
queue.add(neighbor);
}
}
}
```

3.6 Dynamic Programming Pattern

What It Is:
Solve optimization problems by breaking them into overlapping subproblems and storing results[6].

When to Use:

- Problem has optimal substructure (solution uses solutions to subproblems)
- Overlapping subproblems exist (same subproblem solved multiple times)
- Need to optimize resource usage (time or space)

Approaches:

- **Top-down (Memoization)** - Recursive with caching
- **Bottom-up (Tabulation)** - Iterative with table building

Time Complexity: Varies, typically polynomial for solvable problems
Space Complexity: $O(n)$ to $O(n^2)$ for DP table/cache

Example Problems:

- Fibonacci Sequence
- 0/1 Knapsack
- Longest Common Subsequence
- Edit Distance
- Coin Change

Sample Code Approach (Memoization):

```
// Top-down with memoization
```

Map<Integer, Integer> memo = new HashMap<>();

int solve(int n) {
if (n <= 1) return n;
if (memo.containsKey(n)) return memo.get(n);

```
int result = solve(n-1) + solve(n-2); // Recurrence relation
memo.put(n, result);
return result;
```

}

3.7 Backtracking Pattern

What It Is:
Explore all possible solutions by building candidates incrementally and abandoning ("backtracking") when conditions fail[2].

When to Use:

- Generate all permutations, combinations, or subsets
- Solve constraint satisfaction problems
- Exhaustive search with pruning

Key Concepts:

- Decision tree exploration
- Pruning to reduce search space
- Recursive candidate building

Time Complexity: Exponential ($O(2^n)$ or $O(n!)$)
Space Complexity: $O(n)$ for recursion depth

Example Problems:

- Permutations
- Combinations
- Subsets
- N-Queens Problem
- Sudoku Solver
- Word Search in Grid

Sample Code Approach:
// Backtracking template
void backtrack(int index, List<Integer> current, List<List<Integer>> result) {
// Base case: complete solution found
if (index == n) {
result.add(new ArrayList<>(current));
return;
}

```
// Try each choice
for (int choice : choices) {
    if (isValid(choice)) {
        current.add(choice);
        backtrack(index + 1, current, result);
        current.remove(current.size() - 1); // Backtrack
    }
}
```

}

3.8 Merge Intervals Pattern

What It Is:
Efficiently handle overlapping or adjacent intervals through sorting and merging[4].

When to Use:

- Merge overlapping intervals
- Insert new interval
- Find intersections
- Interval scheduling problems

Algorithm:

1. Sort intervals by start point
2. Iterate and merge overlapping intervals
3. Return merged list

Time Complexity: $O(n \log n)$ for sorting
Space Complexity: $O(n)$ for output

Example Problems:

- Merge Intervals
- Insert Interval
- Interval Intersection
- Interval Partitioning

3.9 Top K Elements Pattern

What It Is:
Efficiently find K largest or smallest elements using heaps (priority queues)[3].

When to Use:

- Find K largest/smallest elements
- Top K frequent elements
- Median in data stream
- K closest points

Approach:

- Use min-heap of size K for K largest
- Use max-heap of size K for K smallest
- Add/remove efficiently as needed

Time Complexity: $O(n \log k)$
Space Complexity: $O(k)$

Example Problems:

- K Largest Elements
- Top K Frequent Elements
- K Closest Points to Origin
- Median of Data Stream

4. Interview Strategy with Patterns

4.1 Problem-Solving Workflow

1. **Understand the problem** - Read carefully, ask clarifying questions, identify constraints
2. **Identify the pattern** - What data structure? What operation? What are constraints?
3. **Discuss approach** - Talk through your solution before coding
4. **Implement** - Write clean, readable code with explanations
5. **Test** - Use provided examples and edge cases
6. **Optimize** - Discuss time/space tradeoffs if time permits

4.2 Communication Tips

When discussing patterns in interviews:

- **Name the pattern** - "This looks like a Two Pointers problem because..."
- **Explain trade-offs** - "I could use approach X but it's O(n^2). Pattern Y is O(n)..."
- **Walk through examples** - Trace through your algorithm with test cases
- **Discuss complexity** - Be specific: "Time is O(n log n) due to sorting..."
- **Mention optimizations** - "We could optimize space by..." (only if relevant)

5. Practical Examples

Example 1: Container With Most Water

Problem: Given array of heights, find two lines that form container holding most water.

Pattern Recognition:

- Array problem, need to find optimal pair
- Two Pointers pattern applicable

Solution Approach:

- Start with pointers at both ends (maximum width)
- Calculate water area = width × min(height[left], height[right])

- Move pointer with smaller height inward
- Track maximum area found

Complexity: $O(n)$ time, $O(1)$ space

Example 2: Longest Substring Without Repeating Characters

Problem: Find length of longest substring without repeating characters.

Pattern Recognition:

- String problem, find longest substring
- Sliding Window pattern applicable

Solution Approach:

- Maintain window of unique characters
- Use HashMap to track character positions
- Expand window right, shrink left if duplicate found
- Track maximum window size

Complexity: $O(n)$ time, $O(\min(n, 26))$ space

Example 3: Number of Islands

Problem: Given grid of '1's and '0's, count distinct islands.

Pattern Recognition:

- Grid/Graph problem, find connected components
- DFS/BFS pattern applicable

Solution Approach:

- Iterate through grid
- When finding '1', use DFS to mark entire island as visited
- Count number of DFS calls (each is one island)

Complexity: $O(m \times n)$ time, $O(m \times n)$ space (for visited tracking)

6. Pattern Practice Roadmap

Week 1-2: Fundamentals

- Two Pointers (3-5 problems)
- Sliding Window (3-5 problems)
- Binary Search (2-3 problems)

Week 3-4: Traversal Patterns

- DFS (5-7 problems, increasing difficulty)
- BFS (3-5 problems)

Week 5-6: Complex Patterns

- Dynamic Programming (5-7 problems)
- Backtracking (3-5 problems)

Week 7-8: Advanced Patterns

- Merge Intervals (2-3 problems)
- Top K Elements (2-3 problems)
- Other patterns (monotonic stack, union-find)

Ongoing: Pattern Recognition

- Mix problems from different patterns
- Solve problems without hints
- Practice explaining your approach

7. Common Interview Mistakes to Avoid

- **Jumping to code immediately** - Think first, discuss approach, THEN code
- **Ignoring edge cases** - Empty input, single element, duplicates, etc.
- **Not optimizing from brute force** - Start simple, explain why optimization needed
- **Poor variable naming** - Use clear names, helps interviewer follow logic
- **Forgetting complexity analysis** - Always state time and space complexity
- **Getting stuck on pattern** - Ask if you're unsure; interviewers appreciate your thinking
- **Not testing with examples** - Trace through your code with test cases

8. Resources for Pattern Mastery

Practice Platforms:

- **LeetCode** - Organized by pattern/difficulty, 2000+ problems
- **AlgoMonster** - Pattern-focused learning
- **InterviewBit** - Organized question sets with solutions
- **HackerRank** - Good for structured learning

Study Approach:

- Focus on patterns, not individual problems
- Solve 3-5 problems per pattern minimum
- Review solutions of others for new insights
- Time yourself; simulate interview conditions

9. Key Takeaways

1. **Patterns > Memorization** - Learn patterns, apply to new problems
2. **Recognition is critical** - Quickly identify pattern from problem description
3. **Trade-offs matter** - Understand why one pattern better than another
4. **Communication is essential** - Explain your thinking throughout interview
5. **Practice with purpose** - Don't just solve; understand patterns deeply
6. **Edge cases matter** - Test your solution thoroughly

7. **Complexity analysis required** - Always state and justify time/space complexity
8. **Keep improving** - Learn new patterns as you encounter them

10. Conclusion

Pattern-based problem solving transforms technical interview preparation from memorizing thousands of problems to mastering 10-15 core patterns. By understanding when and how to apply each pattern, you dramatically increase your chances of solving interview problems efficiently[1].

Success in technical interviews comes from:

- **Deep understanding** of core patterns
- **Consistent practice** with varied problems
- **Clear communication** of your approach
- **Systematic problem-solving** that interviewers can follow
- **Continuous learning** from problems and feedback

Start with the fundamental patterns (Two Pointers, Sliding Window, Binary Search), master them thoroughly, then progressively learn advanced patterns. With dedicated practice and pattern-focused learning, you'll develop the confidence and skill to excel in any technical interview[2].

References

[1] Dev.to. (2025, July 12). Coding Interviews were HARD, until I learned these Patterns. https://dev.to/somadevtoo/coding-interviews-was-hard-until-i-learned-these-patterns-2ji7

[2] DesignGurus. (2024, September 4). Mastering the 20 Coding Patterns for Interviews. https://www.designgurus.io/blog/grokking-the-coding-interview-patterns

[3] AlgoMonster. (2020, December 31). Coding Interview Patterns: Your Personal Dijkstra's Algorithm to Success. https://algo.monster/problems/stats

[4] Interviewing.io. (2022, September 26). Sliding Window Interview Questions & Tips. https://interviewing.io/sliding-window-interview-questions

[5] iQuanta. (2025, May 7). Top 10 Algorithms for Coding Interview Questions in 2025. https://www.iquanta.in/blog/top-10-algorithms-for-coding-interview-questions-in-2025/

[6] Dev.to. (2021, March 5). The Ultimate Guide for Data Structures & Algorithm Interviews. https://dev.to/rahhularora/the-ultimate-guide-for-data-structures-algorithm-interviews-npo

[7] Interviewing.io. (2022, September 26). Two Pointers Interview Questions & Tips for Senior Engineers. https://interviewing.io/two-pointers-interview-questions

[8] Educative.io. (2025, July 28). 10+ top LeetCode patterns (2026) to ace FAANG coding interviews. https://www.educative.io/blog/coding-interview-leetcode-patterns

[9] Dev.to. (2025, April 15). Mastering Two Pointers & Sliding Window Techniques in Coding Interviews. https://dev.to/anilnayak_dev/mastering-two-pointers-sliding-window-techniques-in-coding-interviews-dao

[10] Dev.to. (2025, July 12). 6 Coding Patterns That Crack Most DSA Interview Problems. https://www.youtube.com/watch?v=o__d1w2tPFw