Interview & Competitive Programming Prep

Targeted Practice to Master Patterns and Speed

Minseok Jeon
DGIST

November 2, 2025

Outline

Introduction

Course Overview

Goal

Master data structures and algorithms for technical interviews and competitive programming

Key Topics:

- Common patterns for arrays, strings, stacks, queues
- Daily problem-solving routines
- Time complexity estimation techniques
- Pattern recognition and templates
- Mock interviews and reviews
- Progress tracking and metrics

Why This Preparation Matters

Technical Interviews:

- FAANG companies emphasize DSA
- 45-60 minute coding interviews
- Communication is crucial
- Optimization expected
- Pattern recognition wins

Competitive Programming:

- Speed + accuracy under time pressure
- Multiple problems in limited time
- Edge case handling
- Template mastery
- Algorithm optimization

Key Insight

Success requires deliberate practice with patterns, not just solving random problems

Common Patterns

Pattern Category Overview

Data Structure	Common Patterns	Time Complexity
Arrays/Strings	Two pointers, Sliding window	$O(n)$
Stacks	Monotonic stack, Parentheses	$O(n)$
Queues	BFS, Sliding window max	$O(n)$
Hash Tables	Frequency count, Anagrams	$O(n)$

Learning Strategy:

1. Recognize the pattern from problem description
2. Apply the standard template
3. Modify template for specific constraints
4. Verify with test cases

Two Pointers: Concept

Pattern Description:

- Use two indices moving through data
- Avoid nested loops ($O(n^2) \to O(n)$)
- Works on sorted or unsorted arrays

Variants:

- Opposite direction (start/end)
- Same direction (fast/slow)
- Multiple arrays

Signal Words:

- "Pairs summing to..."
- "Remove duplicates"
- "Reverse in-place"
- "Container/water problem"
- "Palindrome check"

Complexity:

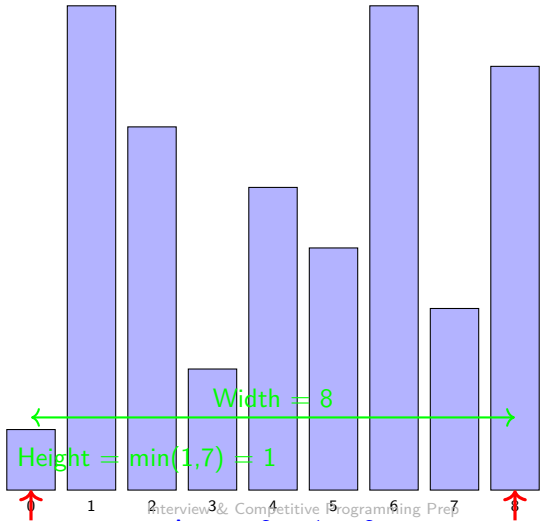
- Time: $O(n)$ single pass
- Space: $O(1)$ in-place

Two Pointers: Template

```python
def two_pointers_template(arr):
    """Opposite direction pointers."""
    left, right = 0, len(arr) - 1

    while left < right:
        # Check condition
        if condition_met(arr[left], arr[right]):
            # Process and move both
            left += 1
            right -= 1
        elif need_larger_value:
            left += 1
        else:
            right -= 1

    return result
```

Two Pointers: Visual Example

Problem: Container with Most Water



Sliding Window: Concept

Pattern Description:

- Maintain a window over data
- Expand/contract window dynamically
- Track window properties
- Avoid recomputing from scratch

Types:

- **Fixed size**: Window size constant
- **Variable size**: Window grows/shrinks

Signal Words:

- "Contiguous subarray"
- "Longest substring"
- "Maximum/minimum in window"
- "K consecutive elements"

Complexity:

- Time: $O(n)$ (each element visited ≤ 2 times)
- Space: $O(k)$ for window state

Sliding Window: Template

```python
def sliding_window_variable(s):
    """Variable size sliding window."""
    left = 0
    result = 0
    window = {}  # Track window state

    for right in range(len(s)):
        # Expand window: add s[right]
        window[s[right]] = window.get(s[right], 0) + 1

        # Contract window if condition violated
        while window_invalid(window):
            window[s[left]] -= 1
            if window[s[left]] == 0:
                del window[s[left]]
            left += 1

        # Update result with current window
        result = max(result, right - left + 1)

    return result

def sliding_window_fixed(arr, k):
    """Fixed size sliding window."""
    window_sum = sum(arr[:k])  # Initial window
    max_sum = window_sum

    for i in range(k, len(arr)):
        window_sum = window_sum - arr[i-k] + arr[i]  # Slide
        max_sum = max(max_sum, window_sum)
```

Sliding Window: Example

Problem: Longest Substring Without Repeating Characters

```
s = "abcabcbb"
```

Step	Window	Left	Right	Max Length
1	[a]	0	0	1
2	[a,b]	0	1	2
3	[a,b,c]	0	2	3
4	[a,b,c,a] → [b,c,a]	1	3	3
5	[b,c,a,b] → [c,a,b]	2	4	3
6	[c,a,b,c] → [a,b,c]	3	5	3

Answer: 3 (substring "abc")

Key Idea:

- Use hash map to track character positions
- When duplicate found, contract window from left
- Track maximum window size seen

Stack Patterns: Overview

Common Stack Patterns:

- **Monotonic stack**: Next greater/smaller element
- **Parentheses matching**: Valid expressions
- **Expression evaluation**: Calculator, RPN
- **Backtracking**: DFS traversal

Signal Words:

- "Next greater element"
- "Valid parentheses"
- "Evaluate expression"
- "Largest rectangle"
- "Trapping rain water"

Why Stacks?

- LIFO matches nested structures
- Efficient backtracking
- $O(1)$ push/pop operations

Monotonic Stack Pattern

```python
def next_greater_element(nums):
    """Find next greater element for each element."""
    n = len(nums)
    result = [-1] * n
    stack = []  # Store indices

    for i in range(n):
        # While current element is greater than stack top
        while stack and nums[i] > nums[stack[-1]]:
            idx = stack.pop()
            result[idx] = nums[i]
        stack.append(i)

    return result

# Example: nums = [2, 1, 2, 4, 3]
# Result: [4, 2, 4, -1, -1]
```

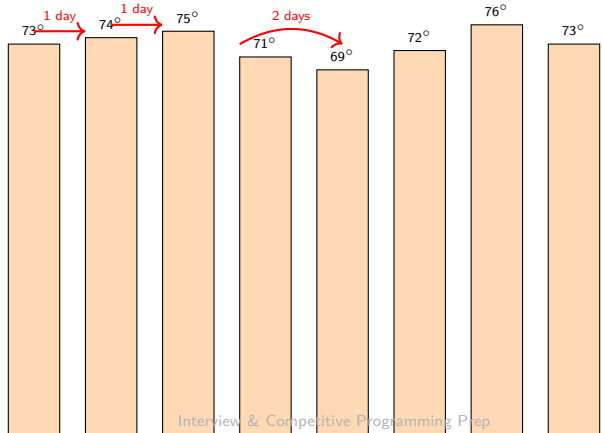
Key Idea:

- Maintain stack in increasing/decreasing order
- Pop elements that violate monotonic property
- Each element pushed and popped once $\rightarrow O(n)$

Monotonic Stack: Visual Example

Problem: Daily Temperatures

Given temperatures = [73, 74, 75, 71, 69, 72, 76, 73], find how many days until warmer temperature.



Valid Parentheses Pattern

```python
def is_valid_parentheses(s):
    """Check if parentheses are balanced."""
    stack = []
    pairs = {'(': ')', '[': ']', '{': '}'}

    for char in s:
        if char in pairs:  # Opening bracket
            stack.append(char)
        else:  # Closing bracket
            if not stack or pairs[stack[-1]] != char:
                return False
            stack.pop()

    return len(stack) == 0

# Examples:
# "()" -> True
```

Queue Patterns: Overview

Common Queue Patterns:

- **BFS traversal**: Level-order, shortest path
- **Sliding window maximum**: Deque optimization
- **Level processing**: Binary tree levels
- **Multi-source BFS**: Multiple starting points

Signal Words:

- "Level order traversal"
- "Shortest path"
- "Minimum steps"
- "Layer by layer"
- "Sliding window maximum"

Why Queues?

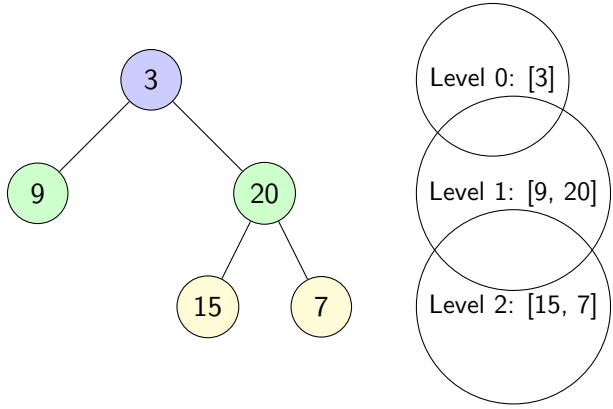
- FIFO ensures level-order processing
- Guarantees shortest path in unweighted graphs
- Natural for breadth-first exploration

BFS Template

```python
from collections import deque

def bfs_template(start):
    """Standard BFS traversal."""
    queue = deque([start])
    visited = {start}

    while queue:
        node = queue.popleft()

        # Process current node
        process(node)

        # Add neighbors
        for neighbor in get_neighbors(node):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    return result

def bfs_level_order(root):
    """BFS with level tracking."""
    if not root:
        return []

    queue = deque([root])
    result = []

    while queue:
```

BFS: Visual Example

Binary Tree Level Order Traversal



Result: [[3], [9, 20], [15, 7]]

Key Points:

Daily Problem-Solving Routine

Building Consistent Practice Habits

Core Principle

Consistency beats intensity. Daily practice builds pattern recognition and speed.

Recommended Schedule:

- **Morning (30 min)**: 1 medium problem, fresh mind
- **Evening (20 min)**: Review solution, optimize, add to notes
- **Weekly (2 hours)**: Mock interview session
- **Monthly**: Review all problems, identify weak areas

Difficulty Progression:

- Week 1-2: Easy problems (build confidence)
- Week 3-4: Mix 70% easy, 30% medium
- Week 5+: Mix 30% easy, 60% medium, 10% hard

Topic Rotation Strategy

Weekly Rotation:

- **Monday**: Arrays/Strings
- **Tuesday**: Stacks/Queues
- **Wednesday**: Trees/Graphs
- **Thursday**: Dynamic Programming
- **Friday**: Hash Tables/Heaps
- **Saturday**: Mixed review
- **Sunday**: Mock interview

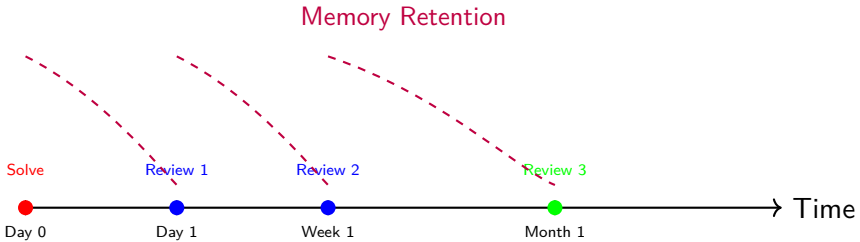
Why Rotation?

- Prevents burnout on single topic
- Builds breadth of knowledge
- Reinforces pattern recognition
- Mimics interview randomness

Time Boxing:

- Easy: 15-20 min
- Medium: 25-35 min
- Hard: 40-50 min
- Stop and review solution if stuck

Review Cycle: Spaced Repetition



Review Schedule:

- **Day 1**: Re-solve without looking at notes (should be fresh)
- **Week 1**: Solve again, focus on optimization
- **Month 1**: Final review, add to template library if pattern

Goal: Move problems from short-term to long-term memory

Problem-Solving Workflow

1. **Read & Understand** (2-3 min)
   - Read problem carefully, underline key constraints
   - Ask clarifying questions (inputs, outputs, edge cases)
   - Identify problem type/pattern
2. **Plan Approach** (3-5 min)
   - Discuss brute force solution first
   - Identify optimizations (data structure, algorithm)
   - Estimate time/space complexity
3. **Code Solution** (15-20 min)
   - Write clean, readable code
   - Use meaningful variable names
   - Add comments for complex logic
4. **Test & Debug** (5-7 min)
   - Test with example cases
   - Check edge cases (empty, single element, large input)
   - Fix bugs systematically

Time Complexity Estimation

Big-O Notation: Quick Reference

Notation	Name	Example
$O(1)$	Constant	Hash table lookup
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Single array pass
$O(n \log n)$	Linearithmic	Merge sort, heap operations
$O(n^2)$	Quadratic	Nested loops
$O(n^3)$	Cubic	Triple nested loops
$O(2^n)$	Exponential	Recursive backtracking
$O(n!)$	Factorial	All permutations

Golden Rule

If unsure, count nested loops and recursion depth. Each nesting level typically multiplies complexity.

Input Size Guidelines

Input Size n	Maximum Acceptable Complexity
$n \leq 10$	$O(n!)$, $O(2^n)$
$n \leq 20$	$O(2^n)$
$n \leq 100$	$O(n^3)$
$n \leq 1,000$	$O(n^2)$
$n \leq 10,000$	$O(n^2)$ with small constant
$n \leq 100,000$	$O(n \log n)$
$n \leq 1,000,000$	$O(n)$ or $O(n \log n)$
$n \leq 10,000,000$	$O(n)$

Rule of Thumb:

- Modern computers: $\sim 10^8$ to 10^9 operations per second
- Time limit typically 1-2 seconds
- Estimate: $n \times$ complexity factor $< 10^8$

Complexity Analysis Examples

Example 1: Nested Loops

for i in range(n) :
  for j in range(n) :
    // $O(1)$ work

Analysis: $O(n \times n) = O(n^2)$

Example 2: Divide and Conquer

$$T(n) = 2T(n/2) + O(n)$$

Analysis: $O(n \log n)$ (Merge Sort)

Example 3: Binary Search Tree

while node $\neq$ null :
  node $=$ node.left or right

Analysis: $O(\log n)$ balanced, $O(n)$ worst

Example 4: Sliding Window

for right in range(n) :
  while invalid :
    left$+ = 1$

Analysis: $O(n)$ (each element visited ≤ 2 times)

Space-Time Tradeoffs

When to Use Extra Space:

- Hash maps for $O(1)$ lookup
- Memoization in DP
- Caching intermediate results
- Preprocessing for multiple queries

When to Save Space:

- Memory-constrained systems
- Large datasets
- In-place modifications allowed
- Space complexity matters

Example: Two Sum

- Brute force: $O(n^2)$ time, $O(1)$ space
- Hash map: $O(n)$ time, $O(n)$ space

Example: Reverse Array

- Extra array: $O(n)$ space
- Two pointers in-place: $O(1)$ space

Interview Tip

Always discuss both time and space complexity. Mention trade-offs explicitly.

Pattern Recognition

Common Problem Categories

Category	Signal Words	Approach
Sliding Window	Contiguous subarray, substring	Two pointers, hash map
Two Pointers	Sorted array, pairs, palindrome	Opposite or same direction
Backtracking	All combinations, permutations	Recursive DFS with pruning
Dynamic Programming	Optimal, maximum/minimum	Memoization or tabulation
Graph Traversal	Connected components, paths	DFS for paths, BFS for shortest
Binary Search	Sorted, find target/range	Divide search space
Greedy	Locally optimal choices	Sort + greedy selection
Divide & Conquer	Break into subproblems	Merge sort, quick sort pattern

Pattern Recognition Strategy:

1. Read problem, identify keywords
2. Check constraints (sorted? tree? graph?)
3. Recall similar problems solved before
4. Apply matching template

Template Library: Binary Search

```python
def binary_search_exact(arr, target):
    """Find exact target."""
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

def binary_search_left_bound(arr, target):
    """Find leftmost occurrence."""
    left, right = 0, len(arr)
    while left < right:
        mid = left + (right - left) // 2
        if arr[mid] < target:
            left = mid + 1
        else:
            right = mid
    return left

def binary_search_on_answer(check, low, high):
    """Binary search on answer space."""
    while low < high:
        mid = low + (high - low) // 2
        if check(mid):
            high = mid
```

Template Library: Backtracking

```python
def backtrack_template(nums):
    """Generate all combinations/permutations."""
    result = []

    def backtrack(path, start):
        # Base case: path complete
        if is_complete(path):
            result.append(path[:])  # Copy current path
            return

        # Recursive case: try all choices
        for i in range(start, len(nums)):
            # Choose
            path.append(nums[i])

            # Explore
            backtrack(path, i + 1)  # i+1 for combinations, 0 for permutations

            # Unchoose (backtrack)
            path.pop()

    backtrack([], 0)
    return result

# Pruning optimization:
def backtrack_with_pruning(path, start):
    if not is_valid(path):  # Prune invalid paths early
        return
    # ... rest of backtracking logic
```

Similar Problem Mapping

Recognizing Problem Variations:

New Problem	Maps To
Coin Change	Unbounded Knapsack DP
House Robber	Linear DP with non-adjacent constraint
Longest Increasing Subsequence	DP or Binary Search
Edit Distance	2D DP (Levenshtein distance)
Word Break	DP with string matching
Maximum Subarray	Kadane's algorithm
Trapping Rain Water	Two pointers or monotonic stack
Merge Intervals	Sorting + greedy
Number of Islands	DFS/BFS on grid
Clone Graph	DFS/BFS with hash map

Strategy: Maintain a mental library of classic problems. New problems are often variations.

Mock Interviews

Mock Interview Structure

Simulating Real Interview Conditions (45-60 min):

1. **Introduction** (2-3 min)
   - Brief self-introduction
   - Interviewer explains format
2. **Problem Presentation** (2-3 min)
   - Read problem carefully
   - Ask clarifying questions
   - Confirm understanding
3. **Discussion** (5-10 min)
   - Explain brute force approach
   - Discuss optimizations
   - Agree on approach with interviewer
4. **Coding** (20-25 min)
   - Write clean, commented code
   - Think aloud while coding

Communication Best Practices

Do's:

- Think aloud consistently
- Ask clarifying questions
- Explain your reasoning
- Discuss trade-offs
- Admit when stuck, ask for hints
- Be open to feedback
- Stay calm and positive

Example Phrases:

- "I'm thinking of using a hash map because..."
- "The time complexity would be $O(n)$ since..."

Don'ts:

- Code in silence
- Jump to coding without discussion
- Ignore interviewer's hints
- Get defensive about mistakes
- Give up when stuck
- Skip testing
- Ignore edge cases

Red Flags:

- Not asking questions
- Poor variable naming
- Skipping complexity analysis
- Not testing solution

Mock Interview Platforms

Free Platforms:

- **LeetCode Mock Interviews**: Timed problems with evaluation
- **Pramp**: Peer-to-peer mock interviews
- **CodeSignal**: Assessment practice
- **HackerRank**: Interview prep kits

Paid Platforms:

- **interviewing.io**: Anonymous interviews with engineers
- **Exponent**: Mock interviews with feedback
- **Brilliant.org**: Interactive problem solving

Practice Partners:

- Study groups
- Friends preparing for interviews
- Online communities (Reddit, Discord)
- University career services

Recording & Review:

- Record your sessions (with permission)
- Review recordings later
- Identify communication gaps
- Track improvement over time

Post-Mortem Analysis

After Each Mock Interview:

1. **What Went Well:**
   - Clear communication
   - Correct solution found
   - Good time management
   - Handled edge cases
2. **What to Improve:**
   - Missed initial edge case (empty array)
   - Took too long to code (30 min instead of 20 min)
   - Didn't discuss space complexity
   - Nervous, spoke too fast
3. **Alternative Approaches:**
   - Could have used DP instead of recursion
   - Hash map would have been simpler
   - Greedy approach also works here

Progress Tracking

Problem Log Template

Spreadsheet/Notion Tracking:

Date	Problem	Difficulty	Time	Status	Notes
11/01	Two Sum	Easy	15 min	✓	Hash map approach
11/01	Valid Parentheses	Easy	20 min	✓	Stack, clean solution
11/02	Longest Substring	Medium	45 min	✓	Struggled with sliding window
11/02	Merge Intervals	Medium	35 min	✓	Sorting + greedy
11/03	Binary Tree Level Order	Medium	30 min	✓	BFS with queue
11/03	Coin Change	Medium	60 min	×	Need to review DP

Key Metrics to Track:

- Solve time per difficulty
- Success rate per topic
- Common mistakes
- Review schedule adherence

Topic Coverage Checklist

Data Structures:

- ✓ Arrays (85% proficiency)
- ✓ Strings (80% proficiency)
- ✓ Stacks (90% proficiency)
- ✓ Queues (85% proficiency)
- ✓ Hash Tables (80% proficiency)
- △ Trees (65% proficiency)
- △ Graphs (60% proficiency)
- × Heaps (45% proficiency)
- × Tries (40% proficiency)

Algorithms:

- ✓ Two Pointers (85%)
- ✓ Sliding Window (80%)
- ✓ Binary Search (85%)
- △ Backtracking (60%)
- △ Dynamic Programming (55%)
- × Graph Algorithms (50%)
- ✓ Sorting (90%)
- △ Greedy (65%)

Legend:

- ✓ $\geq 80\%$: Strong
- △ 50-79%: Needs practice
- × $< 50\%$: Priority focus

Speed Metrics and Goals

Difficulty	Target Time	Current Avg	Status
Easy	15-20 min	18 min	✓On track
Medium	25-35 min	40 min	△ Need improvement
Hard	40-50 min	65 min	× Priority

Speed Improvement Strategies:

- Focus on pattern recognition (reduce planning time)
- Master templates (reduce coding time)
- Practice typing speed and shortcuts
- Time-box practice sessions
- Review fast solutions from top submissions

Goal

Reduce medium problem time to 30 min within 2 months through daily practice

Identifying Weak Areas

Analysis Methods:

1. **Problem Log Analysis**
   - Filter by unsolved or struggled problems
   - Group by topic/pattern
   - Identify recurring difficulties
2. **Time Analysis**
   - Which topics take longest?
   - Where do you get stuck? (planning, coding, debugging)
   - Compare to target times
3. **Success Rate by Topic**
   - % solved on first try
   - % requiring hint/solution lookup
   - % passing all test cases
4. **Mock Interview Feedback**
   - Interviewer notes

Contest Participation

Weekly Contests:

- **LeetCode Weekly/Biweekly**: Saturday/Sunday
- **Codeforces**: Multiple times per week
- **AtCoder Beginner Contest**: Weekly
- **CodeChef**: Long/short contests

Benefits:

- Time pressure practice
- Ranking/percentile tracking
- Exposure to new problems
- Community solutions

Contest Strategy:

- Read all problems first (5 min)
- Solve easiest problems first
- Submit early and often
- Don't get stuck on one problem
- Review solutions after contest

Rating Goals:

- LeetCode: Target 1800+ (top 10%)
- Codeforces: Target Expert (1600+)
- Track rating trends monthly

Summary

Key Takeaways

1. **Master Common Patterns**
   - Two pointers, sliding window, monotonic stack, BFS
   - Templates save time and reduce errors
   - Pattern recognition is the key to speed
2. **Consistent Daily Practice**
   - 1-3 problems daily with topic rotation
   - Spaced repetition for long-term retention
   - Time-boxed sessions to build speed
3. **Complexity Analysis**
   - Estimate before coding (saves time)
   - Know input size $\rightarrow$ complexity mapping
   - Discuss time/space trade-offs
4. **Communication Matters**
   - Think aloud in interviews
   - Ask clarifying questions
   - Explain approach before coding

12-Week Study Plan

Structured Preparation Timeline:

- **Weeks 1-2: Foundations**
  - Easy problems only (arrays, strings, stacks, queues)
  - Build confidence and basic pattern recognition
  - Target: 40-50 easy problems
- **Weeks 3-4: Expand Patterns**
  - Mix 60% easy, 40% medium
  - Trees, graphs, hash tables
  - Target: 30 easy + 20 medium
- **Weeks 5-8: Core Medium Problems**
  - Mix 30% easy, 60% medium, 10% hard
  - Focus on DP, backtracking, advanced patterns
  - Weekly mock interviews
  - Target: 15 easy + 40 medium + 5 hard
- **Weeks 9-12: Interview Ready**

Resources and References

Problem Platforms:

- LeetCode (most popular)
- HackerRank
- Codeforces
- AtCoder
- CodeChef

Books:

- "Cracking the Coding Interview" (Gayle Laakmann McDowell)
- "Elements of Programming Interviews"
- "Competitive Programming" (Halim & Halim)

Online Resources:

- NeetCode (pattern roadmap)
- AlgoExpert (curated problems)
- Blind 75 (essential problems)
- Grind 75 (study plan)

Communities:

- r/leetcode
- LeetCode Discuss
- Codeforces Blogs
- Discord servers

Final Advice

Remember

Interview preparation is a marathon, not a sprint. Consistent practice beats cramming.

Mental Preparation:

- Don't compare yourself to others (focus on your growth)
- It's okay to struggle with problems
- Learn from mistakes (they're valuable)
- Take breaks to avoid burnout
- Celebrate small wins (solved a hard problem!)

During the Interview:

- Stay calm and think clearly
- Communication is 50% of success
- It's okay to ask for hints
- Show your problem-solving process
- Be enthusiastic and positive

Thank You!

Questions?

"The only way to learn to program is by writing programs." – Dennis Ritchie

Apply this wisdom to interview prep:
The only way to get better is by solving problems daily!

Good luck with your interviews!