Competitive Programming and LeetCode Mastery

Complete Problem-Solving Guide with Multiple Approaches and Interview Variations

Chapter 1: Competitive Programming Fundamentals

1.1 What is Competitive Programming?

Competitive Programming (CP) is a mind sport where participants solve algorithmic problems under time constraints. It tests:

- **Algorithm design** and optimization
- **Data structure** selection and implementation
- **Problem-solving** speed and accuracy
- **Code quality** under pressure

Major Platforms:

- **LeetCode:** Interview preparation, contests
- **Codeforces:** International competitive programming
- **CodeChef:** Long and short contests
- **AtCoder:** Japanese platform, strong math focus
- **HackerRank:** Skill-based assessments

1.2 Rating System and Progression

LeetCode Ratings:

- **1500-1700:** Entry level, consistent practice
- **1700-1900:** Intermediate, strong fundamentals
- **1900-2100:** Advanced, competitive
- **2100+:** Expert level, top percentile

Progression Path:

1. **Foundation (1-3 months):** Arrays, strings, basic algorithms
2. **Core Structures (3-6 months):** Trees, graphs, DP basics
3. **Advanced Topics (6-12 months):** Segment trees, flows, advanced DP
4. **Mastery (12+ months):** Competition-level optimization

Chapter 2: Problem-Solving Framework

2.1 The UMPIRE Method

Understand the problem

- Read carefully, identify inputs/outputs
- Ask clarifying questions
- Check constraints and edge cases

Match to known patterns

- Array manipulation, sliding window, two pointers
- Tree/graph traversal
- Dynamic programming

Plan the approach

- Pseudocode outline
- Time/space complexity analysis
- Consider multiple solutions

Implement the solution

- Write clean, readable code
- Handle edge cases
- Add comments for complex logic

Review and test

- Dry run with examples
- Test edge cases
- Check for off-by-one errors

Evaluate and optimize

- Analyze time/space complexity
- Look for optimization opportunities
- Consider alternative approaches

2.2 Complexity Analysis

Time Complexity Classes:

- $O(1)$: Constant - hash table lookup
- $O(\log n)$: Logarithmic - binary search
- $O(n)$: Linear - single pass through array

- $O(n \log n)$: Linearithmic - merge sort
- $O(n^2)$: Quadratic - nested loops
- $O(2^n)$: Exponential - recursive subsets
- $O(n!)$: Factorial - permutations

Space Complexity:

- In-place: $O(1)$ extra space
- Auxiliary: $O(n)$ extra arrays
- Recursive: $O(h)$ call stack depth

Chapter 3: Core Problem Patterns

Pattern 1: Two Pointers

Template:

```
def two_pointers_template(arr):
    left, right = 0, len(arr) - 1

    while left &lt; right:
        # Check condition
        if condition(arr[left], arr[right]):
            # Process and move pointers
            left += 1
        else:
            right -= 1

    return result
```

Problem: Two Sum II (Sorted Array)

```
def twoSum(numbers, target):
    """
    Find two numbers that add up to target in sorted array

    Approach 1: Two Pointers - O(n) time, O(1) space
    """
    left, right = 0, len(numbers) - 1

    while left &lt; right:
        current_sum = numbers[left] + numbers[right]

        if current_sum == target:
            return [left + 1, right + 1]  # 1-indexed
        elif current_sum &lt; target:
            left += 1  # Need larger sum
        else:
            right -= 1  # Need smaller sum
```

```
    return []

# Approach 2: Hash Table - O(n) time, O(n) space
def twoSumHash(numbers, target):
    seen = {}
    for i, num in enumerate(numbers):
        complement = target - num
        if complement in seen:
            return [seen[complement] + 1, i + 1]
        seen[num] = i
    return []

# Approach 3: Binary Search - O(n log n) time, O(1) space
def twoSumBinarySearch(numbers, target):
    for i in range(len(numbers)):
        left, right = i + 1, len(numbers) - 1
        complement = target - numbers[i]

        while left &lt;= right:
            mid = left + (right - left) // 2
            if numbers[mid] == complement:
                return [i + 1, mid + 1]
            elif numbers[mid] &lt; complement:
                left = mid + 1
            else:
                right = mid - 1
    return []
```

Interview Variations:

- 3Sum: Find triplets that sum to zero
- 4Sum: Find quadruplets that sum to target
- Container With Most Water
- Trapping Rain Water

Pattern 2: Sliding Window

Template:

```
def sliding_window_template(arr):
    window_start = 0
    max_value = float('-inf')
    window_sum = 0

    for window_end in range(len(arr)):
        # Expand window
        window_sum += arr[window_end]

        # Shrink window if needed
        while condition_violated(window_sum):
            window_sum -= arr[window_start]
```

```
            window_start += 1

        # Update result
        max_value = max(max_value, window_end - window_start + 1)

    return max_value
```

Problem: Longest Substring Without Repeating Characters

```
def lengthOfLongestSubstring(s):
    """
    Approach 1: Sliding Window with Hash Set - O(n) time, O(min(n,m)) space
    where m is character set size
    """
    char_set = set()
    left = 0
    max_length = 0

    for right in range(len(s)):
        # Shrink window until no duplicates
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1

        # Expand window
        char_set.add(s[right])
        max_length = max(max_length, right - left + 1)

    return max_length

# Approach 2: Optimized with Hash Map (skip duplicates faster)
def lengthOfLongestSubstringOptimized(s):
    char_index = {}  # char -&gt; last seen index
    left = 0
    max_length = 0

    for right in range(len(s)):
        if s[right] in char_index and char_index[s[right]] &gt;= left:
            # Jump left pointer to position after duplicate
            left = char_index[s[right]] + 1

        char_index[s[right]] = right
        max_length = max(max_length, right - left + 1)

    return max_length

# Approach 3: Brute Force - O(n^3) time (for comparison)
def lengthOfLongestSubstringBrute(s):
    max_length = 0
    for i in range(len(s)):
        for j in range(i, len(s)):
            if len(set(s[i:j+1])) == j - i + 1:
                max_length = max(max_length, j - i + 1)
            else:
```

```
                break
    return max_length
```

Interview Variations:

- Minimum Window Substring
- Longest Repeating Character Replacement
- Permutation in String
- Find All Anagrams in a String

Pattern 3: Fast & Slow Pointers (Floyd's Cycle Detection)

Template:

```
def has_cycle(head):
    slow = fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            return True

    return False
```

Problem: Linked List Cycle II (Find Cycle Start)

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

# Approach 1: Floyd's Algorithm - O(n) time, O(1) space
def detectCycle(head):
    """
    Mathematical proof:
    - When fast and slow meet, fast traveled 2x distance
    - If cycle starts at distance k from head, meeting point is k from cycle start
    - Reset one pointer to head, move both at same speed → meet at cycle start
    """
    slow = fast = head

    # Phase 1: Detect cycle
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            # Phase 2: Find cycle start
```

```
            slow = head
            while slow != fast:
                slow = slow.next
                fast = fast.next
            return slow

    return None

# Approach 2: Hash Set - O(n) time, O(n) space
def detectCycleHashSet(head):
    seen = set()
    current = head

    while current:
        if current in seen:
            return current
        seen.add(current)
        current = current.next

    return None

# Approach 3: Modify Node Values (destructive) - O(n) time, O(1) space
def detectCycleDestructive(head):
    sentinel = ListNode(-1)
    current = head

    while current:
        if current.next == sentinel:
            return current.next
        temp = current.next
        current.next = sentinel
        current = temp

    return None
```

Interview Variations:

- Happy Number
- Middle of Linked List
- Palindrome Linked List
- Reorder List

Pattern 4: Binary Search

Template:

```
def binary_search_template(arr, target):
    left, right = 0, len(arr) - 1

    while left &lt;= right:
        mid = left + (right - left) // 2  # Prevent overflow
```

```
        if arr[mid] == target:
            return mid
        elif arr[mid] &lt; target:
            left = mid + 1  # Search right half
        else:
            right = mid - 1  # Search left half

    return -1  # Not found
```

Problem: Search in Rotated Sorted Array

```
def search(nums, target):
    """
    Approach 1: Modified Binary Search - O(log n) time, O(1) space

    Key insight: At least one half is always sorted
    """
    left, right = 0, len(nums) - 1

    while left &lt;= right:
        mid = left + (right - left) // 2

        if nums[mid] == target:
            return mid

        # Determine which half is sorted
        if nums[left] &lt;= nums[mid]:  # Left half is sorted
            if nums[left] &lt;= target &lt; nums[mid]:
                right = mid - 1  # Target in left half
            else:
                left = mid + 1   # Target in right half
        else:  # Right half is sorted
            if nums[mid] &lt; target &lt;= nums[right]:
                left = mid + 1   # Target in right half
            else:
                right = mid - 1  # Target in left half

    return -1

# Approach 2: Find Pivot + Binary Search - O(log n) time, O(1) space
def searchWithPivot(nums, target):
    # Find pivot (smallest element index)
    left, right = 0, len(nums) - 1
    while left &lt; right:
        mid = left + (right - left) // 2
        if nums[mid] &gt; nums[right]:
            left = mid + 1
        else:
            right = mid
    pivot = left

    # Determine which half to search
    left, right = 0, len(nums) - 1
    if target &gt;= nums[pivot] and target &lt;= nums[right]:
        left = pivot
```

```
    else:
        right = pivot - 1

    # Standard binary search
    while left &lt;= right:
        mid = left + (right - left) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] &lt; target:
            left = mid + 1
        else:
            right = mid - 1

    return -1

# Approach 3: Linear Scan - O(n) time (baseline for comparison)
def searchLinear(nums, target):
    for i, num in enumerate(nums):
        if num == target:
            return i
    return -1
```

Interview Variations:

- Find Minimum in Rotated Sorted Array
- Search a 2D Matrix
- Koko Eating Bananas
- Capacity To Ship Packages Within D Days

Pattern 5: Dynamic Programming

Template (Top-Down Memoization):

```
def dp_memoization(n, memo=None):
    if memo is None:
        memo = {}

    # Base case
    if n &lt;= 1:
        return n

    # Check memo
    if n in memo:
        return memo[n]

    # Recursive case
    memo[n] = dp_memoization(n-1, memo) + dp_memoization(n-2, memo)

    return memo[n]
```

Problem: Coin Change

```
def coinChange(coins, amount):
    """
    Approach 1: Bottom-Up DP - O(amount * coins) time, O(amount) space

    dp[i] = minimum coins needed for amount i
    """
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0  # Base case: 0 coins for amount 0

    for i in range(1, amount + 1):
        for coin in coins:
            if coin &lt;= i:
                dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1

# Approach 2: Top-Down Memoization - O(amount * coins) time, O(amount) space
def coinChangeMemo(coins, amount):
    memo = {}

    def dp(remaining):
        if remaining == 0:
            return 0
        if remaining &lt; 0:
            return float('inf')
        if remaining in memo:
            return memo[remaining]

        min_coins = float('inf')
        for coin in coins:
            result = dp(remaining - coin)
            if result != float('inf'):
                min_coins = min(min_coins, result + 1)

        memo[remaining] = min_coins
        return min_coins

    result = dp(amount)
    return result if result != float('inf') else -1

# Approach 3: BFS (Shortest Path) - O(amount * coins) time, O(amount) space
from collections import deque

def coinChangeBFS(coins, amount):
    if amount == 0:
        return 0

    queue = deque([(amount, 0)])  # (remaining, num_coins)
    visited = {amount}

    while queue:
        remaining, num_coins = queue.popleft()

        for coin in coins:
            next_remaining = remaining - coin
```

```
            if next_remaining == 0:
                return num_coins + 1

            if next_remaining &gt; 0 and next_remaining not in visited:
                visited.add(next_remaining)
                queue.append((next_remaining, num_coins + 1))

    return -1
```

Interview Variations:

- Climbing Stairs
- House Robber (I, II, III)
- Longest Increasing Subsequence
- Edit Distance
- Word Break
- Partition Equal Subset Sum

Pattern 6: Graph Algorithms

DFS Template:

```
def dfs(graph, node, visited):
    visited.add(node)

    # Process node
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
```

BFS Template:

```
from collections import deque

def bfs(graph, start):
    queue = deque([start])
    visited = {start}

    while queue:
        node = queue.popleft()

        # Process node
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

Problem: Number of Islands

```
def numIslands(grid):
    """
    Approach 1: DFS - O(m*n) time, O(m*n) space (recursion stack)
    """
    if not grid:
        return 0

    rows, cols = len(grid), len(grid[0])
    count = 0

    def dfs(r, c):
        if (r &lt; 0 or r &gt;= rows or c &lt; 0 or c &gt;= cols or
            grid[r][c] == '0'):
            return

        grid[r][c] = '0'  # Mark as visited

        # Explore all 4 directions
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '1':
                dfs(r, c)
                count += 1

    return count

# Approach 2: BFS - O(m*n) time, O(min(m,n)) space (queue size)
def numIslandsBFS(grid):
    if not grid:
        return 0

    rows, cols = len(grid), len(grid[0])
    count = 0

    def bfs(r, c):
        queue = deque([(r, c)])
        grid[r][c] = '0'

        while queue:
            row, col = queue.popleft()

            for dr, dc in [(1,0), (-1,0), (0,1), (0,-1)]:
                nr, nc = row + dr, col + dc
                if (0 &lt;= nr &lt; rows and 0 &lt;= nc &lt; cols and
                    grid[nr][nc] == '1'):
                    grid[nr][nc] = '0'
                    queue.append((nr, nc))

    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '1':
```

```
                bfs(r, c)
                count += 1

    return count

# Approach 3: Union-Find - O(m*n * α(m*n)) time, O(m*n) space
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
        self.count = n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return

        if self.rank[px] &lt; self.rank[py]:
            self.parent[px] = py
        elif self.rank[px] &gt; self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[py] = px
            self.rank[px] += 1

        self.count -= 1

def numIslandsUF(grid):
    if not grid:
        return 0

    rows, cols = len(grid), len(grid[0])
    uf = UnionFind(rows * cols)
    water_count = 0

    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '0':
                water_count += 1
            else:
                for dr, dc in [(1,0), (0,1)]:
                    nr, nc = r + dr, c + dc
                    if (0 &lt;= nr &lt; rows and 0 &lt;= nc &lt; cols and
                        grid[nr][nc] == '1'):
                        uf.union(r * cols + c, nr * cols + nc)

    return uf.count - water_count
```

Interview Variations:

- Clone Graph

- Course Schedule (topological sort)
- Pacific Atlantic Water Flow
- Word Ladder
- Network Delay Time (Dijkstra)

Chapter 4: Advanced Data Structures

4.1 Trie (Prefix Tree)

Implementation:

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        """O(m) time where m is word length"""
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        """O(m) time"""
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

    def startsWith(self, prefix):
        """O(m) time"""
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True
```

4.2 Heap (Priority Queue)

Min Heap Operations:

```
import heapq

# Create min heap
heap = []

# Insert - O(log n)
heapq.heappush(heap, 5)

# Pop minimum - O(log n)
min_val = heapq.heappop(heap)

# Peek minimum - O(1)
min_val = heap[0]

# Heapify array - O(n)
arr = [3, 1, 4, 1, 5]
heapq.heapify(arr)

# Max heap (negate values)
max_heap = []
heapq.heappush(max_heap, -5)
max_val = -heapq.heappop(max_heap)
```

Problem: Top K Frequent Elements

```
import heapq
from collections import Counter

def topKFrequent(nums, k):
    """
    Approach 1: Min Heap - O(n log k) time, O(n) space
    """
    count = Counter(nums)

    # Min heap of size k
    heap = []
    for num, freq in count.items():
        heapq.heappush(heap, (freq, num))
        if len(heap) &gt; k:
            heapq.heappop(heap)

    return [num for freq, num in heap]

# Approach 2: Bucket Sort - O(n) time, O(n) space
def topKFrequentBucket(nums, k):
    count = Counter(nums)
    buckets = [[] for _ in range(len(nums) + 1)]

    for num, freq in count.items():
        buckets[freq].append(num)
```

```
    result = []
    for i in range(len(buckets) - 1, 0, -1):
        result.extend(buckets[i])
        if len(result) &gt;= k:
            return result[:k]

    return result
```

Chapter 5: Interview Preparation Strategy

5.1 Study Plan (3-Month Intensive)

Month 1: Foundations

- Week 1-2: Arrays, strings, hash tables (50 problems)
- Week 3-4: Linked lists, stacks, queues (40 problems)

Month 2: Core Algorithms

- Week 5-6: Binary search, sorting, two pointers (50 problems)
- Week 7-8: Trees, graphs, DFS/BFS (60 problems)

Month 3: Advanced Topics

- Week 9-10: Dynamic programming (50 problems)
- Week 11-12: Advanced DS, review weak areas (50 problems)

Total: ~300 problems

5.2 Problem Selection

Difficulty Distribution:

- 30% Easy: Build confidence, learn patterns
- 50% Medium: Interview standard
- 20% Hard: Push limits, competition prep

LeetCode Top Interview 150

- Curated list of most common interview problems
- Organized by pattern/topic
- Essential for preparation

Chapter 6: Contest Strategy

6.1 During Contest

Time Management:

- Read all problems first (5 min)
- Solve easiest problems first (build momentum)
- Budget: Easy (10 min), Medium (25 min), Hard (35 min)
- If stuck >15 min, move to next problem

Implementation Tips:

- Test with examples before submitting
- Check edge cases: empty input, single element, duplicates
- Use helper functions for clarity
- Add comments for complex logic

6.2 After Contest

Upsolving:

- Solve all unsolved problems within 24 hours
- Read editorial and discuss solutions
- Implement alternative approaches
- Add to spaced repetition system

Further Resources

Books:

- Competitive Programming 3 (Halim & Halim)
- Elements of Programming Interviews (Aziz, Lee, Prakash)
- Cracking the Coding Interview (McDowell)

Websites:

- LeetCode Patterns (seanprashad.com/leetcode-patterns)
- Neetcode.io (video explanations + roadmap)
- Visualgo (algorithm visualizations)

This comprehensive textbook provides complete problem-solving frameworks, multiple solution approaches for each pattern, and strategic guidance for achieving LeetCode 2000+ rating and acing technical interviews.