

# FAANG Coding Interview Patterns & Templates

Python Reference Guide

Master These Patterns for Interview Success

**GOAL:** Practice these templates until they become muscle memory

**COVERAGE:** 12 Essential Patterns + Recognition Guide

**TIMELINE:** 4-Week Intensive Study Plan

**TARGET:** FAANG/MAANG Interview Readiness

Based on 2024 FAANG Interview Analysis

September 29, 2025

## Contents

<b>1</b>	<b>Quick Pattern Recognition Guide</b>	<b>3</b>
<b>2</b>	<b>Core Algorithm Patterns</b>	<b>3</b>
2.1	1. Depth-First Search (DFS)	3
2.1.1	Recursive DFS Template	3
2.1.2	Iterative DFS Template	4
2.2	2. Breadth-First Search (BFS)	4
2.2.1	Standard BFS Template	4
2.2.2	BFS for Shortest Path	5
2.3	3. Two Pointers	5
2.3.1	Opposite Ends Template	5
2.3.2	Fast and Slow Pointers	6
2.4	4. Sliding Window	7
2.4.1	Fixed Size Window	7
2.4.2	Variable Size Window	7
2.5	5. Binary Search	8
2.5.1	Standard Binary Search	8
2.5.2	Find First and Last Occurrence	9
2.5.3	Binary Search on Answer	9
2.6	6. Dynamic Programming	10
2.6.1	1D DP Template	10
2.6.2	2D DP Template	11
2.7	7. Backtracking	11
2.7.1	General Backtracking Template	12
2.7.2	Combinations and Subsets	12
2.8	8. Heap Operations	13
2.8.1	Min/Max Heap Templates	13
2.8.2	Advanced Heap Applications	14
2.9	9. Union-Find (Disjoint Set)	15
2.10	10. Trie (Prefix Tree)	16
2.11	11. Topological Sort	17
2.12	12. Interval Problems	18
2.13	13. Matrix Traversal	19
2.14	14. Tree Construction	20
2.15	15. Special Algorithms	21
2.15.1	Kadane's Algorithm - Maximum Subarray	21
2.15.2	Prefix Sum	22
2.15.3	Monotonic Stack	22
2.15.4	Cyclic Sort	23
<b>3</b>	<b>Pattern Recognition Guide</b>	<b>23</b>
3.1	Decision Tree for Pattern Selection	24
<b>4</b>	<b>4-Week Study Schedule</b>	<b>24</b>
4.1	Week 1: Foundation Patterns	24
4.2	Week 2: Dynamic Programming & Backtracking	25
4.3	Week 3: Advanced Data Structures	25
4.4	Week 4: Special Algorithms & Integration	25
<b>5</b>	<b>Daily Practice Routine</b>	<b>26</b>
5.1	Morning Routine (30 minutes)	26
5.2	Evening Session (60-90 minutes)	26
5.3	Weekly Goals	26

<b>6</b>	<b>Progress Tracking</b>	<b>26</b>
6.1	Template Mastery Checklist . . . . .	26
6.2	Problem Categories to Master . . . . .	27
<b>7</b>	<b>Final Tips for Success</b>	<b>27</b>
7.1	Red Flags to Avoid . . . . .	28
7.2	Last-Minute Review (Day Before Interview) . . . . .	28
<b>A</b>	<b>Daily Progress Tracking Worksheets</b>	<b>29</b>
A.1	Template Mastery Progress Tracker . . . . .	29
A.1.1	Daily Template Practice (Write from Memory) . . . . .	29
A.1.2	Problem Solving Progress . . . . .	29
A.2	4-Week Study Schedule Tracker . . . . .	30
A.2.1	Week 1: Foundation Patterns . . . . .	30
A.2.2	Week 2: Dynamic Programming & Backtracking . . . . .	30
A.2.3	Week 3: Advanced Data Structures . . . . .	30
A.2.4	Week 4: Special Algorithms & Integration . . . . .	30
A.3	Scientific Spaced Repetition Schedule . . . . .	32
A.3.1	Week 1: Foundation Building . . . . .	32
A.3.2	Week 2: Pattern Reinforcement . . . . .	32
A.3.3	Week 3: Advanced Structures . . . . .	32
A.3.4	Week 4: Mastery and Integration . . . . .	32
A.3.5	Daily Review Strategy . . . . .	32
A.4	Daily Routine Checklist . . . . .	33
A.4.1	Morning Routine (30 minutes) . . . . .	33
A.4.2	Evening Session (60-90 minutes) . . . . .	33
A.4.3	Weekly Reflection . . . . .	34
A.5	Mock Interview Tracker . . . . .	35
A.5.1	Interview Readiness Checklist . . . . .	35

## 1 Quick Pattern Recognition Guide

Pattern	Key Indicators	When to Use
Two Pointers	Sorted array, pairs, palindromes	Finding pairs with target sum, removing duplicates
Sliding Window	Subarray/substring, contiguous	Max/min subarray, longest substring problems
DFS	Tree/graph traversal, explore all paths	Path finding, connectivity, topological sort
BFS	Shortest path, level-order	Minimum steps, level traversal
Binary Search	Sorted data, search space	Finding element, first/last occurrence
Dynamic Programming	Optimal substructure, overlapping subproblems	Optimization, counting, decision problems
Backtracking	Generate all combinations/permutations	N-Queens, Sudoku, subset generation
Heap	Top K, merge K, priority	Finding extremes, scheduling
Union-Find	Connected components, cycles	Graph connectivity, MST
Trie	Prefix matching, word games	Autocomplete, word search
Intervals	Overlapping ranges, scheduling	Meeting rooms, merge intervals
Matrix Traversal	2D grid problems, connected components	Islands, flood fill
Tree Construction	Build tree from traversals	Preorder/inorder, serialize

## 2 Core Algorithm Patterns

### 2.1 1. Depth-First Search (DFS)

#### DFS - Tree Traversal

**Use Case:** Tree/graph traversal, path finding, cycle detection

**Time:**  $O(V + E)$  for graphs,  $O(n)$  for trees

**Space:**  $O(h)$  where  $h$  is height/depth

#### 2.1.1 Recursive DFS Template

```

1 def dfs_recursive(root):
2     if not root:
3         return # Base case
4
5     # Process current node
6     print(root.val)
7
8     # Recurse on children
9     dfs_recursive(root.left)
10    dfs_recursive(root.right)
11
12 # With result collection
13 def dfs_collect_paths(root, path=[], all_paths=[]):
14     if not root:
15         return
16
17     path.append(root.val)
18
19     # Leaf node - save path
20     if not root.left and not root.right:
21         all_paths.append(path[:]) # Copy path

```

```
22     dfs_collect_paths(root.left, path, all_paths)
23     dfs_collect_paths(root.right, path, all_paths)
24
25     path.pop() # Backtrack
26     return all_paths
27
```

### 2.1.2 Iterative DFS Template

```
1 def dfs_iterative(root):
2     if not root:
3         return
4
5     stack = [root]
6     while stack:
7         node = stack.pop()
8         print(node.val) # Process node
9
10        # Add children (right first for left-to-right order)
11        if node.right:
12            stack.append(node.right)
13        if node.left:
14            stack.append(node.left)
15
16    # Graph DFS with visited set
17    def dfs_graph(graph, start):
18        visited = set()
19        stack = [start]
20
21        while stack:
22            node = stack.pop()
23            if node not in visited:
24                visited.add(node)
25                print(node) # Process node
26
27                # Add neighbors
28                for neighbor in graph[node]:
29                    if neighbor not in visited:
30                        stack.append(neighbor)
```

## 2.2 Breadth-First Search (BFS)

### BFS - Level-by-Level Traversal

**Use Case:** Shortest path, level-order traversal, minimum steps

**Time:**  $O(V + E)$  for graphs,  $O(n)$  for trees

**Space:**  $O(w)$  where  $w$  is maximum width

### 2.2.1 Standard BFS Template

```
1 from collections import deque
2
3 def bfs_level_order(root):
4     if not root:
5         return []
6
7     result = []
8     queue = deque([root])
9
10    while queue:
11        level_size = len(queue)
12        level = []
13
14        for _ in range(level_size):
15            node = queue.popleft()
16            level.append(node.val)
17
```

```
18         if node.left:
19             queue.append(node.left)
20         if node.right:
21             queue.append(node.right)
22
23     result.append(level)
24     return result
25
26 # Simple BFS without levels
27 def bfs_simple(root):
28     if not root:
29         return
30
31     queue = deque([root])
32     while queue:
33         node = queue.popleft()
34         print(node.val) # Process node
35
36         if node.left:
37             queue.append(node.left)
38         if node.right:
39             queue.append(node.right)
```

### 2.2.2 BFS for Shortest Path

```
1 def bfs_shortest_path(graph, start, target):
2     queue = deque([(start, 0)]) # (node, distance)
3     visited = set([start])
4
5     while queue:
6         node, dist = queue.popleft()
7
8         if node == target:
9             return dist
10
11        for neighbor in graph[node]:
12            if neighbor not in visited:
13                visited.add(neighbor)
14                queue.append((neighbor, dist + 1))
15
16    return -1 # Target not reachable
```

## 2.3 3. Two Pointers

### Two Pointers - Efficient Array/String Processing

**Use Case:** Sorted arrays, palindromes, pairs with target sum

**Time:**  $O(n)$  typically

**Space:**  $O(1)$  space optimization

### 2.3.1 Opposite Ends Template

```
1 # Problem: Two Sum II - Input Array Is Sorted
2 # Given a sorted array, find two numbers that add up to target
3 # Return indices of the two numbers (assume exactly one solution exists)
4 def two_sum_sorted(nums, target):
5     left, right = 0, len(nums) - 1
6
7     while left < right:
8         current_sum = nums[left] + nums[right]
9
10        if current_sum == target:
11            return [left, right]
12        elif current_sum < target:
13            left += 1
14        else:
15            right -= 1
```

```
16     return []
17
18
19 # Problem: Valid Palindrome
20 # Given a string, determine if it's a palindrome (reads same forwards/backwards)
21 # Consider only alphanumeric characters and ignore case
22 def is_palindrome(s):
23     left, right = 0, len(s) - 1
24
25     while left < right:
26         if s[left] != s[right]:
27             return False
28         left += 1
29         right -= 1
30
31     return True
32
33 # Problem: Reverse Array
34 # Reverse an array in-place using two pointers
35 def reverse_array(arr):
36     left, right = 0, len(arr) - 1
37
38     while left < right:
39         arr[left], arr[right] = arr[right], arr[left]
40         left += 1
41         right -= 1
```

### 2.3.2 Fast and Slow Pointers

```
1 # Problem: Linked List Cycle (Floyd's Cycle Detection)
2 # Detect if a linked list has a cycle using fast/slow pointers
3 def has_cycle(head):
4     if not head or not head.next:
5         return False
6
7     slow = fast = head
8
9     while fast and fast.next:
10         slow = slow.next
11         fast = fast.next.next
12
13         if slow == fast:
14             return True
15
16     return False
17
18 # Problem: Middle of the Linked List
19 # Find the middle node of a linked list
20 def find_middle(head):
21     slow = fast = head
22
23     while fast and fast.next:
24         slow = slow.next
25         fast = fast.next.next
26
27     return slow
28
29 # Problem: Remove Nth Node From End of List
30 # Remove the nth node from the end of a linked list
31 def remove_nth_from_end(head, n):
32     dummy = ListNode(0)
33     dummy.next = head
34     slow = fast = dummy
35
36     # Move fast n+1 steps ahead
37     for _ in range(n + 1):
38         fast = fast.next
39
40     # Move both until fast reaches end
41     while fast:
42         slow = slow.next
43         fast = fast.next
```

```
44
45 # Remove nth node
46 slow.next = slow.next.next
47 return dummy.next
```

## 2.4 4. Sliding Window

### Sliding Window - Contiguous Subarray/Substring

**Use Case:** Maximum/minimum subarray, longest substring

**Time:**  $O(n)$  single pass

**Space:**  $O(k)$  for tracking window contents

#### 2.4.1 Fixed Size Window

```
1 # Problem: Maximum Sum Subarray of Size K
2 # Find the maximum sum of any contiguous subarray of size k
3 def max_sum_subarray(nums, k):
4     if len(nums) < k:
5         return 0
6
7     # Calculate first window
8     window_sum = sum(nums[:k])
9     max_sum = window_sum
10
11    # Slide window
12    for i in range(k, len(nums)):
13        window_sum = window_sum - nums[i - k] + nums[i]
14        max_sum = max(max_sum, window_sum)
15
16    return max_sum
17
18 def find_averages(nums, k):
19     result = []
20     window_sum = 0
21
22     for i in range(len(nums)):
23         window_sum += nums[i]
24
25         if i >= k - 1:
26             result.append(window_sum / k)
27             window_sum -= nums[i - k + 1]
28
29     return result
```

#### 2.4.2 Variable Size Window

```
1 # Problem: Longest Substring with At Most K Distinct Characters
2 # Find the length of the longest substring with at most k distinct characters
3 def longest_substring_k_distinct(s, k):
4     if k == 0:
5         return 0
6
7     char_count = {}
8     left = 0
9     max_length = 0
10
11    for right in range(len(s)):
12        # Expand window
13        char_count[s[right]] = char_count.get(s[right], 0) + 1
14
15        # Contract window if needed
16        while len(char_count) > k:
17            char_count[s[left]] -= 1
18            if char_count[s[left]] == 0:
19                del char_count[s[left]]
20            left += 1
```



```

21         max_length = max(max_length, right - left + 1)
22
23     return max_length
24
25
26 def min_window_substring(s, t):
27     if not s or not t:
28         return ""
29
30     t_count = {}
31     for char in t:
32         t_count[char] = t_count.get(char, 0) + 1
33
34     left = 0
35     min_len = float('inf')
36     min_start = 0
37     matched = 0
38     window_count = {}
39
40     for right in range(len(s)):
41         char = s[right]
42         window_count[char] = window_count.get(char, 0) + 1
43
44         if char in t_count and window_count[char] == t_count[char]:
45             matched += 1
46
47         while matched == len(t_count):
48             if right - left + 1 < min_len:
49                 min_len = right - left + 1
50                 min_start = left
51
52             left_char = s[left]
53             window_count[left_char] -= 1
54             if left_char in t_count and window_count[left_char] < t_count[left_char]:
55                 matched -= 1
56             left += 1
57
58     return s[min_start:min_start + min_len] if min_len != float('inf') else ""

```

## 2.5 5. Binary Search

### Binary Search - Efficient Search in Sorted Space

**Use Case:** Finding element, first/last occurrence, search space

**Time:**  $O(\log n)$

**Space:**  $O(1)$  iterative,  $O(\log n)$  recursive

#### 2.5.1 Standard Binary Search

```

1 # Problem: Binary Search
2 # Find target value in a sorted array, return index or -1 if not found
3 def binary_search(nums, target):
4     left, right = 0, len(nums) - 1
5
6     while left <= right:
7         mid = left + (right - left) // 2
8
9         if nums[mid] == target:
10             return mid
11         elif nums[mid] < target:
12             left = mid + 1
13         else:
14             right = mid - 1
15
16     return -1
17
18 def binary_search_recursive(nums, target, left=0, right=None):
19     if right is None:
20         right = len(nums) - 1

```

```
21
22     if left > right:
23         return -1
24
25     mid = left + (right - left) // 2
26
27     if nums[mid] == target:
28         return mid
29     elif nums[mid] < target:
30         return binary_search_recursive(nums, target, mid + 1, right)
31     else:
32         return binary_search_recursive(nums, target, left, mid - 1)
```

### 2.5.2 Find First and Last Occurrence

```
1 def find_first_occurrence(nums, target):
2     left, right = 0, len(nums) - 1
3     result = -1
4
5     while left <= right:
6         mid = left + (right - left) // 2
7
8         if nums[mid] == target:
9             result = mid
10            right = mid - 1 # Continue searching left
11        elif nums[mid] < target:
12            left = mid + 1
13        else:
14            right = mid - 1
15
16    return result
17
18 def find_last_occurrence(nums, target):
19     left, right = 0, len(nums) - 1
20     result = -1
21
22    while left <= right:
23        mid = left + (right - left) // 2
24
25        if nums[mid] == target:
26            result = mid
27            left = mid + 1 # Continue searching right
28        elif nums[mid] < target:
29            left = mid + 1
30        else:
31            right = mid - 1
32
33    return result
34
35 def search_range(nums, target):
36     return [find_first_occurrence(nums, target),
37            find_last_occurrence(nums, target)]
```

### 2.5.3 Binary Search on Answer

```
1 def find_peak_element(nums):
2     left, right = 0, len(nums) - 1
3
4     while left < right:
5         mid = left + (right - left) // 2
6
7         if nums[mid] > nums[mid + 1]:
8             right = mid
9         else:
10            left = mid + 1
11
12    return left
13
14 def search_rotated_sorted_array(nums, target):
15     left, right = 0, len(nums) - 1
```

```
16
17 while left <= right:
18     mid = left + (right - left) // 2
19
20     if nums[mid] == target:
21         return mid
22
23     # Left half is sorted
24     if nums[left] <= nums[mid]:
25         if nums[left] <= target < nums[mid]:
26             right = mid - 1
27         else:
28             left = mid + 1
29     # Right half is sorted
30     else:
31         if nums[mid] < target <= nums[right]:
32             left = mid + 1
33         else:
34             right = mid - 1
35
36 return -1
```

## 2.6 6. Dynamic Programming

### Dynamic Programming - Optimal Substructure

**Use Case:** Optimization, counting, decision problems

**Time:** Varies (often  $O(n^2)$  or  $O(n*m)$ )

**Space:**  $O(n)$  to  $O(n*m)$  depending on dimensions

#### 2.6.1 1D DP Template

```
1 # Problem: Fibonacci Number
2 # Calculate the nth Fibonacci number using dynamic programming
3 def fibonacci(n):
4     if n <= 1:
5         return n
6
7     # Bottom-up approach
8     dp = [0] * (n + 1)
9     dp[1] = 1
10
11     for i in range(2, n + 1):
12         dp[i] = dp[i-1] + dp[i-2]
13
14     return dp[n]
15
16 # Space optimized
17 def fibonacci_optimized(n):
18     if n <= 1:
19         return n
20
21     prev2, prev1 = 0, 1
22
23     for i in range(2, n + 1):
24         current = prev1 + prev2
25         prev2, prev1 = prev1, current
26
27     return prev1
28
29 def climb_stairs(n):
30     if n <= 2:
31         return n
32
33     dp = [0] * (n + 1)
34     dp[1], dp[2] = 1, 2
35
36     for i in range(3, n + 1):
37         dp[i] = dp[i-1] + dp[i-2]
```

```

38     return dp[n]
39
40
41 def house_robber(nums):
42     if not nums:
43         return 0
44     if len(nums) == 1:
45         return nums[0]
46
47     dp = [0] * len(nums)
48     dp[0] = nums[0]
49     dp[1] = max(nums[0], nums[1])
50
51     for i in range(2, len(nums)):
52         dp[i] = max(dp[i-1], dp[i-2] + nums[i])
53
54     return dp[-1]

```

### 2.6.2 2D DP Template

```

1 def unique_paths(m, n):
2     # Create DP table
3     dp = [[1 for _ in range(n)] for _ in range(m)]
4
5     for i in range(1, m):
6         for j in range(1, n):
7             dp[i][j] = dp[i-1][j] + dp[i][j-1]
8
9     return dp[m-1][n-1]
10
11 def longest_common_subsequence(text1, text2):
12     m, n = len(text1), len(text2)
13     dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]
14
15     for i in range(1, m + 1):
16         for j in range(1, n + 1):
17             if text1[i-1] == text2[j-1]:
18                 dp[i][j] = dp[i-1][j-1] + 1
19             else:
20                 dp[i][j] = max(dp[i-1][j], dp[i][j-1])
21
22     return dp[m][n]
23
24 def min_path_sum(grid):
25     m, n = len(grid), len(grid[0])
26
27     # Initialize first row and column
28     for i in range(1, m):
29         grid[i][0] += grid[i-1][0]
30
31     for j in range(1, n):
32         grid[0][j] += grid[0][j-1]
33
34     # Fill the DP table
35     for i in range(1, m):
36         for j in range(1, n):
37             grid[i][j] += min(grid[i-1][j], grid[i][j-1])
38
39     return grid[m-1][n-1]

```

## 2.7 7. Backtracking

### Backtracking - Explore All Possibilities

**Use Case:** Permutations, combinations, N-Queens, Sudoku

**Time:** Exponential  $O(2^n)$  or  $O(n!)$

**Space:**  $O(\text{depth})$  for recursion stack

### 2.7.1 General Backtracking Template

```
1 def backtrack_template(candidates, target, path=[], result=[]):
2     # Base case - solution found
3     if is_valid_solution(path, target):
4         result.append(path[:]) # Make a copy
5         return
6
7     # Explore all possibilities
8     for i, candidate in enumerate(candidates):
9         # Skip invalid candidates
10        if not is_valid_candidate(candidate, path):
11            continue
12
13        # Make choice
14        path.append(candidate)
15
16        # Recurse with remaining candidates
17        backtrack_template(candidates[i+1:], target, path, result)
18
19        # Undo choice (backtrack)
20        path.pop()
21
22    return result
23
24 def generate_permutations(nums):
25     def backtrack(path, remaining):
26         if not remaining:
27             result.append(path[:])
28             return
29
30         for i in range(len(remaining)):
31             # Choose
32             path.append(remaining[i])
33             # Explore
34             backtrack(path, remaining[:i] + remaining[i+1:])
35             # Unchoose
36             path.pop()
37
38     result = []
39     backtrack([], nums)
40     return result
```

### 2.7.2 Combinations and Subsets

```
1 def generate_subsets(nums):
2     def backtrack(start, path):
3         result.append(path[:]) # Add current subset
4
5         for i in range(start, len(nums)):
6             path.append(nums[i])
7             backtrack(i + 1, path)
8             path.pop()
9
10    result = []
11    backtrack(0, [])
12    return result
13
14 def combination_sum(candidates, target):
15     def backtrack(start, path, remaining):
16         if remaining == 0:
17             result.append(path[:])
18             return
19
20         for i in range(start, len(candidates)):
21             if candidates[i] > remaining:
22                 break
23
24             path.append(candidates[i])
25             # Can reuse same element
26             backtrack(i, path, remaining - candidates[i])
```

```

27         path.pop()
28
29     result = []
30     candidates.sort()
31     backtrack(0, [], target)
32     return result
33
34 def letter_combinations(digits):
35     if not digits:
36         return []
37
38     phone = {
39         '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
40         '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
41     }
42
43     def backtrack(index, path):
44         if index == len(digits):
45             result.append("".join(path))
46             return
47
48         for letter in phone[digits[index]]:
49             path.append(letter)
50             backtrack(index + 1, path)
51             path.pop()
52
53     result = []
54     backtrack(0, [])
55     return result

```

## 2.8 8. Heap Operations

### Heap - Priority Queue for Top K Problems

**Use Case:** Top K elements, merge K sorted, scheduling

**Time:**  $O(\log n)$  insert/delete,  $O(1)$  peek

**Space:**  $O(n)$  for heap storage

### 2.8.1 Min/Max Heap Templates

```

1 import heapq
2
3 def find_kth_largest(nums, k):
4     # Use min heap of size k
5     min_heap = []
6
7     for num in nums:
8         heapq.heappush(min_heap, num)
9         if len(min_heap) > k:
10             heapq.heappop(min_heap)
11
12     return min_heap[0]
13
14 def find_k_largest_elements(nums, k):
15     # Min heap approach
16     min_heap = []
17
18     for num in nums:
19         if len(min_heap) < k:
20             heapq.heappush(min_heap, num)
21         elif num > min_heap[0]:
22             heapq.heapreplace(min_heap, num)
23
24     return sorted(min_heap, reverse=True)
25
26 # For max heap, negate values
27 def max_heap_operations():
28     max_heap = []
29

```

```
30 # Insert (negate for max heap)
31 heapq.heappush(max_heap, -value)
32
33 # Extract max (negate result)
34 max_val = -heapq.heappop(max_heap)
35
36 # Peek max
37 max_val = -max_heap[0]
```

## 2.8.2 Advanced Heap Applications

```
1 def merge_k_sorted_lists(lists):
2     import heapq
3
4     min_heap = []
5     # Add first element from each list
6     for i, lst in enumerate(lists):
7         if lst:
8             heapq.heappush(min_heap, (lst.val, i, lst))
9
10    dummy = ListNode(0)
11    current = dummy
12
13    while min_heap:
14        val, list_idx, node = heapq.heappop(min_heap)
15        current.next = node
16        current = current.next
17
18        # Add next element from same list
19        if node.next:
20            heapq.heappush(min_heap, (node.next.val, list_idx, node.next))
21
22    return dummy.next
23
24 def top_k_frequent_elements(nums, k):
25     from collections import Counter
26     import heapq
27
28     count = Counter(nums)
29
30     # Min heap of size k
31     min_heap = []
32     for num, freq in count.items():
33         heapq.heappush(min_heap, (freq, num))
34         if len(min_heap) > k:
35             heapq.heappop(min_heap)
36
37     return [num for freq, num in min_heap]
38
39 class MedianFinder:
40     def __init__(self):
41         self.small = [] # max heap (negated)
42         self.large = [] # min heap
43
44     def addNum(self, num):
45         # Add to max heap first
46         heapq.heappush(self.small, -num)
47
48         # Balance: move largest from small to large
49         heapq.heappush(self.large, -heapq.heappop(self.small))
50
51         # Ensure small has more or equal elements
52         if len(self.large) > len(self.small):
53             heapq.heappush(self.small, -heapq.heappop(self.large))
54
55     def findMedian(self):
56         if len(self.small) > len(self.large):
57             return -self.small[0]
58         return (-self.small[0] + self.large[0]) / 2
```

## 2.9 9. Union-Find (Disjoint Set)

### Union-Find - Connected Components

**Use Case:** Graph connectivity, cycle detection, MST

**Time:**  $O(\alpha(n))$  amortized per operation

**Space:**  $O(n)$  for parent and rank arrays

```

1 class UnionFind:
2     def __init__(self, n):
3         self.parent = list(range(n))
4         self.rank = [0] * n
5         self.components = n
6
7     def find(self, x):
8         # Path compression
9         if self.parent[x] != x:
10             self.parent[x] = self.find(self.parent[x])
11         return self.parent[x]
12
13     def union(self, x, y):
14         root_x = self.find(x)
15         root_y = self.find(y)
16
17         if root_x == root_y:
18             return False # Already connected
19
20         # Union by rank
21         if self.rank[root_x] < self.rank[root_y]:
22             self.parent[root_x] = root_y
23         elif self.rank[root_x] > self.rank[root_y]:
24             self.parent[root_y] = root_x
25         else:
26             self.parent[root_y] = root_x
27             self.rank[root_x] += 1
28
29         self.components -= 1
30         return True
31
32     def connected(self, x, y):
33         return self.find(x) == self.find(y)
34
35     def count_components(self):
36         return self.components
37
38 # Applications
39 def number_of_islands(grid):
40     if not grid:
41         return 0
42
43     m, n = len(grid), len(grid[0])
44     uf = UnionFind(m * n)
45     islands = 0
46
47     for i in range(m):
48         for j in range(n):
49             if grid[i][j] == '1':
50                 islands += 1
51                 # Check 4 directions
52                 for di, dj in [(0,1), (1,0), (0,-1), (-1,0)]:
53                     ni, nj = i + di, j + dj
54                     if (0 <= ni < m and 0 <= nj < n and
55                         grid[ni][nj] == '1'):
56                         if uf.union(i*n + j, ni*n + nj):
57                             islands -= 1
58
59     return islands
60
61 def redundant_connection(edges):
62     uf = UnionFind(len(edges) + 1)
63 
```



```

64     for u, v in edges:
65         if not uf.union(u, v):
66             return [u, v] # This edge creates a cycle
67
68     return []

```

## 2.10 10. Trie (Prefix Tree)

### Trie - Efficient String Storage and Search

**Use Case:** Autocomplete, word search, prefix matching

**Time:**  $O(m)$  where  $m$  is string length

**Space:**  $O(\text{ALPHABET\_SIZE} * N * M)$  worst case

```

1 class TrieNode:
2     def __init__(self):
3         self.children = {}
4         self.is_end_of_word = False
5
6 class Trie:
7     def __init__(self):
8         self.root = TrieNode()
9
10    def insert(self, word):
11        node = self.root
12        for char in word:
13            if char not in node.children:
14                node.children[char] = TrieNode()
15            node = node.children[char]
16        node.is_end_of_word = True
17
18    def search(self, word):
19        node = self.root
20        for char in word:
21            if char not in node.children:
22                return False
23            node = node.children[char]
24        return node.is_end_of_word
25
26    def starts_with(self, prefix):
27        node = self.root
28        for char in prefix:
29            if char not in node.children:
30                return False
31            node = node.children[char]
32        return True
33
34    def find_words_with_prefix(self, prefix):
35        # Find the prefix node
36        node = self.root
37        for char in prefix:
38            if char not in node.children:
39                return []
40            node = node.children[char]
41
42        # DFS to find all words
43        words = []
44        self._dfs(node, prefix, words)
45        return words
46
47    def _dfs(self, node, path, words):
48        if node.is_end_of_word:
49            words.append(path)
50
51        for char, child_node in node.children.items():
52            self._dfs(child_node, path + char, words)
53
54    # Word Search II using Trie
55    def find_words_in_board(self, board, words):
56        # Build trie

```

```

57     trie = Trie()
58     for word in words:
59         trie.insert(word)
60
61     result = set()
62     m, n = len(board), len(board[0])
63
64     def dfs(i, j, node, path):
65         if node.is_end_of_word:
66             result.add(path)
67
68         if (i < 0 or i >= m or j < 0 or j >= n or
69             board[i][j] not in node.children):
70             return
71
72         char = board[i][j]
73         board[i][j] = '#' # Mark visited
74
75         # Explore 4 directions
76         for di, dj in [(0,1), (1,0), (0,-1), (-1,0)]:
77             dfs(i + di, j + dj, node.children[char], path + char)
78
79         board[i][j] = char # Restore
80
81     # Try starting from each cell
82     for i in range(m):
83         for j in range(n):
84             if board[i][j] in trie.root.children:
85                 dfs(i, j, trie.root, "")
86
87     return list(result)

```

## 2.11 11. Topological Sort

### Topological Sort - Ordering with Dependencies

**Use Case:** Course scheduling, dependency resolution

**Time:**  $O(V + E)$

**Space:**  $O(V)$  for in-degree array and queue

```

1  from collections import defaultdict, deque
2
3  def topological_sort_kahn(num_courses, prerequisites):
4      # Build graph and in-degree array
5      graph = defaultdict(list)
6      in_degree = [0] * num_courses
7
8      for course, prereq in prerequisites:
9          graph[prereq].append(course)
10         in_degree[course] += 1
11
12     # Initialize queue with nodes having in-degree 0
13     queue = deque([i for i in range(num_courses) if in_degree[i] == 0])
14     result = []
15
16     while queue:
17         node = queue.popleft()
18         result.append(node)
19
20         # Reduce in-degree of neighbors
21         for neighbor in graph[node]:
22             in_degree[neighbor] -= 1
23             if in_degree[neighbor] == 0:
24                 queue.append(neighbor)
25
26     # Check if topological sort is possible
27     return result if len(result) == num_courses else []
28
29 def can_finish_courses(num_courses, prerequisites):
30     topo_order = topological_sort_kahn(num_courses, prerequisites)

```

```

31     return len(topo_order) == num_courses
32
33 # DFS-based topological sort
34 def topological_sort_dfs(graph):
35     visited = set()
36     rec_stack = set()
37     result = []
38
39     def dfs(node):
40         if node in rec_stack:
41             return False # Cycle detected
42         if node in visited:
43             return True
44
45         visited.add(node)
46         rec_stack.add(node)
47
48         for neighbor in graph.get(node, []):
49             if not dfs(neighbor):
50                 return False
51
52         rec_stack.remove(node)
53         result.append(node) # Add to result in post-order
54     return True
55
56 for node in graph:
57     if node not in visited:
58         if not dfs(node):
59             return [] # Cycle detected
60
61 return result[::-1] # Reverse for correct order

```

## 2.12 12. Interval Problems

### Interval Problems - Merge and Insert Intervals

**Use Case:** Meeting rooms, overlapping intervals, scheduling

**Time:**  $O(n \log n)$  for sorting,  $O(n)$  for merging

**Space:**  $O(n)$  for result storage

```

1 # Problem: Merge Intervals
2 # Given collection of intervals, merge overlapping intervals
3 def merge_intervals(intervals):
4     if not intervals:
5         return []
6
7     # Sort by start time
8     intervals.sort(key=lambda x: x[0])
9     merged = [intervals[0]]
10
11     for current in intervals[1:]:
12         last = merged[-1]
13
14         # Overlapping intervals
15         if current[0] <= last[1]:
16             # Merge by updating end time
17             last[1] = max(last[1], current[1])
18         else:
19             # Non-overlapping, add to result
20             merged.append(current)
21
22     return merged
23
24 # Problem: Insert Interval
25 # Insert new interval and merge if necessary
26 def insert_interval(intervals, new_interval):
27     result = []
28     i = 0
29     n = len(intervals)
30

```

```

31 # Add all intervals before new_interval
32 while i < n and intervals[i][1] < new_interval[0]:
33     result.append(intervals[i])
34     i += 1
35
36 # Merge overlapping intervals
37 while i < n and intervals[i][0] <= new_interval[1]:
38     new_interval[0] = min(new_interval[0], intervals[i][0])
39     new_interval[1] = max(new_interval[1], intervals[i][1])
40     i += 1
41
42 result.append(new_interval)
43
44 # Add remaining intervals
45 while i < n:
46     result.append(intervals[i])
47     i += 1
48
49 return result
50
51 # Problem: Meeting Rooms II
52 # Find minimum number of meeting rooms needed
53 def min_meeting_rooms(intervals):
54     if not intervals:
55         return 0
56
57     # Separate start and end times
58     starts = sorted([interval[0] for interval in intervals])
59     ends = sorted([interval[1] for interval in intervals])
60
61     rooms = 0
62     end_ptr = 0
63
64     for start in starts:
65         # If meeting starts after another ends, reuse room
66         if start >= ends[end_ptr]:
67             end_ptr += 1
68         else:
69             # Need new room
70             rooms += 1
71
72     return rooms

```

## 2.13 13. Matrix Traversal

### Matrix Problems - 2D Grid Traversal

**Use Case:** Island problems, path finding, flood fill

**Time:**  $O(m*n)$  where m,n are matrix dimensions

**Space:**  $O(m*n)$  for visited array or recursion stack

```

1 # Problem: Number of Islands
2 # Count connected components of '1's in 2D grid
3 def num_islands(grid):
4     if not grid or not grid[0]:
5         return 0
6
7     m, n = len(grid), len(grid[0])
8     count = 0
9
10    def dfs(i, j):
11        # Boundary check and water check
12        if (i < 0 or i >= m or j < 0 or j >= n or
13            grid[i][j] != '1'):
14            return
15
16        # Mark as visited
17        grid[i][j] = '0'
18
19        # Explore 4 directions

```

```

20         for di, dj in [(0,1), (1,0), (0,-1), (-1,0)]:
21             dfs(i + di, j + dj)
22
23     for i in range(m):
24         for j in range(n):
25             if grid[i][j] == '1':
26                 dfs(i, j)
27                 count += 1
28
29     return count
30
31 # Problem: Rotting Oranges (Multi-source BFS)
32 # Find time for all oranges to rot
33 def oranges_rotting(grid):
34     from collections import deque
35
36     m, n = len(grid), len(grid[0])
37     queue = deque()
38     fresh = 0
39
40     # Find all rotten oranges and count fresh ones
41     for i in range(m):
42         for j in range(n):
43             if grid[i][j] == 2:
44                 queue.append((i, j))
45             elif grid[i][j] == 1:
46                 fresh += 1
47
48     if fresh == 0:
49         return 0
50
51     time = 0
52     directions = [(0,1), (1,0), (0,-1), (-1,0)]
53
54     while queue:
55         time += 1
56         size = len(queue)
57
58         for _ in range(size):
59             x, y = queue.popleft()
60
61             for dx, dy in directions:
62                 nx, ny = x + dx, y + dy
63
64                 if (0 <= nx < m and 0 <= ny < n and
65                     grid[nx][ny] == 1):
66                     grid[nx][ny] = 2
67                     fresh -= 1
68                     queue.append((nx, ny))
69
70     return time - 1 if fresh == 0 else -1

```

## 2.14 14. Tree Construction

### Tree Construction - Build Trees from Traversals

**Use Case:** Construct tree from preorder/inorder, serialize/deserialize

**Time:**  $O(n)$  with hashmap optimization

**Space:**  $O(n)$  for hashmap and recursion stack

```

1 # Problem: Construct Binary Tree from Preorder and Inorder
2 # Build tree given preorder and inorder traversal arrays
3 def build_tree_preorder_inorder(preorder, inorder):
4     if not preorder or not inorder:
5         return None
6
7     # Build hashmap for O(1) inorder lookups
8     inorder_map = {val: i for i, val in enumerate(inorder)}
9     self.preorder_idx = 0
10

```

```

11     def build(left, right):
12         if left > right:
13             return None
14
15         # Root is current preorder element
16         root_val = preorder[self.preorder_idx]
17         self.preorder_idx += 1
18         root = TreeNode(root_val)
19
20         # Find root position in inorder
21         root_idx = inorder_map[root_val]
22
23         # Build left subtree first (preorder property)
24         root.left = build(left, root_idx - 1)
25         root.right = build(root_idx + 1, right)
26
27         return root
28
29     return build(0, len(inorder) - 1)
30
31 # Problem: Serialize and Deserialize Binary Tree
32 # Convert tree to string and back
33 def serialize(root):
34     def preorder(node):
35         if not node:
36             result.append("null")
37             return
38
39         result.append(str(node.val))
40         preorder(node.left)
41         preorder(node.right)
42
43     result = []
44     preorder(root)
45     return ",".join(result)
46
47 def deserialize(data):
48     def build():
49         val = next(values)
50         if val == "null":
51             return None
52
53         node = TreeNode(int(val))
54         node.left = build()
55         node.right = build()
56         return node
57
58     values = iter(data.split(","))
59     return build()

```

## 2.15 15. Special Algorithms

### 2.15.1 Kadane's Algorithm - Maximum Subarray

```

1 def max_subarray_sum(nums):
2     if not nums:
3         return 0
4
5     max_sum = current_sum = nums[0]
6
7     for i in range(1, len(nums)):
8         # Either extend existing subarray or start new one
9         current_sum = max(nums[i], current_sum + nums[i])
10        max_sum = max(max_sum, current_sum)
11
12    return max_sum
13
14 def max_subarray_with_indices(nums):
15     max_sum = current_sum = nums[0]
16     start = end = temp_start = 0
17
18     for i in range(1, len(nums)):

```

```
19         if current_sum < 0:
20             current_sum = nums[i]
21             temp_start = i
22         else:
23             current_sum += nums[i]
24
25         if current_sum > max_sum:
26             max_sum = current_sum
27             start = temp_start
28             end = i
29
30     return max_sum, start, end
```

### 2.15.2 Prefix Sum

```
1 class PrefixSum:
2     def __init__(self, nums):
3         self.prefix = [0]
4         for num in nums:
5             self.prefix.append(self.prefix[-1] + num)
6
7     def range_sum(self, i, j):
8         # Sum from index i to j (inclusive)
9         return self.prefix[j + 1] - self.prefix[i]
10
11 def subarray_sum_equals_k(nums, k):
12     count = 0
13     prefix_sum = 0
14     sum_count = {0: 1} # prefix_sum -> frequency
15
16     for num in nums:
17         prefix_sum += num
18         if prefix_sum - k in sum_count:
19             count += sum_count[prefix_sum - k]
20         sum_count[prefix_sum] = sum_count.get(prefix_sum, 0) + 1
21
22     return count
```

### 2.15.3 Monotonic Stack

```
1 def next_greater_element(nums):
2     result = [-1] * len(nums)
3     stack = [] # Store indices
4
5     for i in range(len(nums)):
6         while stack and nums[i] > nums[stack[-1]]:
7             index = stack.pop()
8             result[index] = nums[i]
9         stack.append(i)
10
11     return result
12
13 def daily_temperatures(temperatures):
14     result = [0] * len(temperatures)
15     stack = []
16
17     for i, temp in enumerate(temperatures):
18         while stack and temp > temperatures[stack[-1]]:
19             prev_index = stack.pop()
20             result[prev_index] = i - prev_index
21         stack.append(i)
22
23     return result
24
25 def largest_rectangle_in_histogram(heights):
26     stack = []
27     max_area = 0
28
29     for i, h in enumerate(heights + [0]): # Add sentinel
30         while stack and h < heights[stack[-1]]:
```

```
31         height = heights[stack.pop()]
32         width = i if not stack else i - stack[-1] - 1
33         max_area = max(max_area, height * width)
34         stack.append(i)
35
36     return max_area
```

### 2.15.4 Cyclic Sort

```
1 def cyclic_sort(nums):
2     i = 0
3     while i < len(nums):
4         correct_index = nums[i] - 1
5         if nums[i] != nums[correct_index]:
6             nums[i], nums[correct_index] = nums[correct_index], nums[i]
7         else:
8             i += 1
9     return nums
10
11 def find_missing_number(nums):
12     # Array contains n numbers in range [0, n]
13     i = 0
14     n = len(nums)
15
16     while i < n:
17         if nums[i] < n and nums[i] != nums[nums[i]]:
18             nums[nums[i]], nums[i] = nums[i], nums[nums[i]]
19         else:
20             i += 1
21
22     # Find the missing number
23     for i in range(n):
24         if nums[i] != i:
25             return i
26
27     return n
28
29 def find_all_duplicates(nums):
30     duplicates = []
31
32     for i in range(len(nums)):
33         # Use array indices to mark presence
34         num = abs(nums[i])
35         if nums[num - 1] < 0:
36             duplicates.append(num)
37         else:
38             nums[num - 1] *= -1
39
40     return duplicates
```

## 3 Pattern Recognition Guide

### Complexity Analysis

#### Time Complexity Quick Reference:

- $O(1)$  - Hash table access, array index
- $O(\log n)$  - Binary search, heap operations
- $O(n)$  - Single pass through array, BFS/DFS
- $O(n \log n)$  - Merge sort, heap sort
- $O(n^2)$  - Nested loops, bubble sort
- $O(2^n)$  - Recursive algorithms without memoization



### 3.1 Decision Tree for Pattern Selection

#### 1. Array/String Problems:

- Sorted array → Binary Search
- Two elements with condition → Two Pointers
- Contiguous subarray/substring → Sliding Window
- All subarrays → Prefix Sum or DP

#### 2. Tree/Graph Problems:

- Path finding → DFS
- Shortest path → BFS
- Level-order traversal → BFS
- Connected components → Union-Find or DFS

#### 3. Optimization Problems:

- Optimal substructure → Dynamic Programming
- Multiple choices at each step → Backtracking
- Greedy choice property → Greedy Algorithm

#### 4. Priority/Ordering Problems:

- Top K elements → Heap
- Streaming data → Heap
- Dependencies → Topological Sort

#### 5. String Matching:

- Prefix operations → Trie
- Pattern matching → KMP or Rolling Hash
- Anagrams → Hash Map

#### Pro Tip

##### Red Flags for Common Patterns:

- "Maximum/Minimum subarray" → Sliding Window or Kadane's
- "All permutations/combinations" → Backtracking
- "Shortest path in unweighted graph" → BFS
- "Detect cycle" → DFS or Union-Find
- "Top K" or "K-th largest" → Heap

## 4 4-Week Study Schedule

### 4.1 Week 1: Foundation Patterns

#### Day 1-2: Two Pointers & Sliding Window

- Practice writing templates from memory
- Solve: Two Sum II, Container With Most Water, Longest Substring Without Repeating Characters

#### Day 3-4: DFS & BFS

- Master both recursive and iterative approaches

- Solve: Binary Tree Inorder Traversal, Binary Tree Level Order Traversal, Number of Islands

**Day 5-7: Binary Search**

- Practice standard and modified binary search
- Solve: Search in Rotated Sorted Array, Find First and Last Position, Search Insert Position

## 4.2 Week 2: Dynamic Programming & Backtracking

**Day 8-10: 1D & 2D Dynamic Programming**

- Focus on identifying optimal substructure
- Solve: Climbing Stairs, House Robber, Unique Paths, Longest Common Subsequence

**Day 11-14: Backtracking**

- Master the template and understand when to backtrack
- Solve: Permutations, Combinations, Subsets, N-Queens

## 4.3 Week 3: Advanced Data Structures

**Day 15-17: Heap Operations**

- Learn min/max heap operations and applications
- Solve: Kth Largest Element, Top K Frequent Elements, Merge K Sorted Lists

**Day 18-19: Union-Find**

- Understand path compression and union by rank
- Solve: Number of Islands II, Redundant Connection, Friend Circles

**Day 20-21: Trie**

- Build trie from scratch and understand applications
- Solve: Implement Trie, Word Search II, Add and Search Word

## 4.4 Week 4: Special Algorithms & Integration

**Day 22-24: Special Algorithms**

- Kadane's Algorithm, Prefix Sum, Monotonic Stack
- Solve: Maximum Subarray, Subarray Sum Equals K, Next Greater Element

**Day 25-26: Topological Sort**

- Both Kahn's algorithm and DFS approach
- Solve: Course Schedule, Course Schedule II, Alien Dictionary

**Day 27-28: Integration & Mock Interviews**

- Combine multiple patterns in complex problems
- Practice writing templates under time pressure
- Mock interview sessions

## 5 Daily Practice Routine

### 5.1 Morning Routine (30 minutes)

1. **Template Writing (10 min):** Write 3 random templates from memory
2. **Pattern Recognition (10 min):** Look at problem titles and identify patterns
3. **Complexity Analysis (10 min):** Review time/space complexity for each pattern

### 5.2 Evening Session (60-90 minutes)

1. **Problem Solving (45-60 min):**
  - Choose 2-3 problems focusing on current week's patterns
  - Spend 5 minutes identifying the pattern before coding
  - Write solution from scratch using templates
2. **Template Review (15-30 min):**
  - Review any templates you struggled with
  - Write them again from memory
  - Note common mistakes or variations

### 5.3 Weekly Goals

- **Week 1:** Master 4 foundational patterns
- **Week 2:** Add DP and backtracking to arsenal
- **Week 3:** Comfortable with advanced data structures
- **Week 4:** Integrate patterns and achieve fluency

#### Pro Tip

##### Success Metrics:

- Can write any template from memory in under 2 minutes
- Identify correct pattern within 30 seconds of reading problem
- Solve medium problems in 15-20 minutes
- Explain time/space complexity confidently

## 6 Progress Tracking

### 6.1 Template Mastery Checklist

Check off when you can write from memory in under 2 minutes:

- ☐ DFS (recursive)
- ☐ DFS (iterative)
- ☐ BFS (standard)
- ☐ BFS (level-order)
- ☐ Two Pointers (opposite ends)
- ☐ Two Pointers (fast/slow)

- ☐ Sliding Window (fixed)
- ☐ Sliding Window (variable)
- ☐ Binary Search (standard)
- ☐ Binary Search (first/last occurrence)
- ☐ 1D DP template
- ☐ 2D DP template
- ☐ Backtracking template
- ☐ Heap operations
- ☐ Union-Find (with optimizations)
- ☐ Trie implementation
- ☐ Topological Sort (Kahn's)
- ☐ Kadane's Algorithm
- ☐ Prefix Sum
- ☐ Monotonic Stack

## 6.2 Problem Categories to Master

Category	Easy	Medium	Hard
Arrays & Strings	10	15	5
Trees & Graphs	8	12	5
Dynamic Programming	5	10	5
Backtracking	3	8	3
Heaps & Priority Queues	5	8	3
Advanced Data Structures	5	10	5
<b>Total</b>	<b>36</b>	<b>63</b>	<b>26</b>

## 7 Final Tips for Success

### Pro Tip

#### Interview Day Strategy:

1. Spend 2-3 minutes understanding the problem completely
2. Identify the pattern (this should be automatic by now)
3. Explain your approach before coding
4. Write the template structure first, then fill in details
5. Test with simple examples
6. Analyze time and space complexity

### Complexity Analysis

#### Common Optimizations:

- Two-pass → One-pass with hash map
- Nested loops → Two pointers or hash map
- Recursion → DP with memoization
- Multiple data structures → Single optimized structure
- Extra space → In-place modifications

## 7.1 Red Flags to Avoid

- Jumping into coding without understanding the problem
- Not identifying the pattern before starting
- Overcomplicating simple problems
- Not testing with edge cases
- Forgetting to analyze complexity

## 7.2 Last-Minute Review (Day Before Interview)

1. Write all 12 main templates from memory
2. Review the pattern recognition guide
3. Practice one problem from each category
4. Get good sleep and stay confident

**Remember: Consistency beats intensity.  
Practice these templates daily until they become automatic!**

## A Daily Progress Tracking Worksheets

### A.1 Template Mastery Progress Tracker

#### Week \_\_\_\_ Progress Tracker

Date Range: \_\_\_\_\_ to \_\_\_\_\_

#### A.1.1 Daily Template Practice (Write from Memory)

Template	Mon	Tue	Wed	Thu	Fri	Sat	Sun
DFS Recursive	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DFS Iterative	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
BFS Standard	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Two Pointers (Opposite)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Two Pointers (Fast/Slow)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Sliding Window (Fixed)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Sliding Window (Variable)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Binary Search	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Binary Search (First/Last)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
1D DP Template	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2D DP Template	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Backtracking Template	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

**Goal:** Check ☐ if you can write the template from memory in under 2 minutes

#### A.1.2 Problem Solving Progress

Date	Problem Name	Pattern	Difficulty	Time (min)
_____	_____	_____	Easy/Med/Hard_____	
_____	_____	_____	Easy/Med/Hard_____	
_____	_____	_____	Easy/Med/Hard_____	
_____	_____	_____	Easy/Med/Hard_____	
_____	_____	_____	Easy/Med/Hard_____	
_____	_____	_____	Easy/Med/Hard_____	
_____	_____	_____	Easy/Med/Hard_____	
_____	_____	_____	Easy/Med/Hard_____	
_____	_____	_____	Easy/Med/Hard_____	
_____	_____	_____	Easy/Med/Hard_____	

## A.2 4-Week Study Schedule Tracker

### A.2.1 Week 1: Foundation Patterns

**Focus:** Two Pointers, Sliding Window, DFS, BFS, Binary Search

Day	Primary Focus	Goals	Done
Day 1-2	Two Pointers & Sliding Window	Master templates, solve: Two Sum II, Container With Most Water	<input type="checkbox"/>
Day 3-4	DFS & BFS	Recursive/iterative approaches, solve: Binary Tree Traversals	<input type="checkbox"/>
Day 5-7	Binary Search	Standard and modified, solve: Search in Rotated Array	<input type="checkbox"/>

**Week 1 Notes:**

### A.2.2 Week 2: Dynamic Programming & Backtracking

**Focus:** 1D & 2D DP, Backtracking patterns

Day	Primary Focus	Goals	Done
Day 8-10	Dynamic Programming	1D/2D templates, solve: Climbing Stairs, Unique Paths	<input type="checkbox"/>
Day 11-14	Backtracking	Template mastery, solve: Permutations, N-Queens	<input type="checkbox"/>

**Week 2 Notes:**

### A.2.3 Week 3: Advanced Data Structures

**Focus:** Heaps, Union-Find, Trie, Intervals, Matrix

Day	Primary Focus	Goals	Done
Day 15-17	Heap Operations	Min/max heap applications, solve: Top K Elements	<input type="checkbox"/>
Day 18-19	Union-Find	Path compression, solve: Number of Islands	<input type="checkbox"/>
Day 20-21	Trie & Intervals	Build from scratch, solve: Merge Intervals	<input type="checkbox"/>

**Week 3 Notes:**

### A.2.4 Week 4: Special Algorithms & Integration

**Focus:** Special algorithms, pattern combinations, mock interviews

Day	Primary Focus	Goals	Done
Day 22-24	Special Algorithms	Kadane's, Prefix Sum, Monotonic Stack	<input type="checkbox"/>
Day 25-26	Topological Sort	Kahn's algorithm, solve: Course Schedule	<input type="checkbox"/>
Day 27-28	Integration & Mocks	Combine patterns, timed practice sessions	<input type="checkbox"/>

Week 4 Notes:



### A.3 Scientific Spaced Repetition Schedule

#### Optimal Learning and Review Cadence for Maximum Retention

This schedule follows cognitive science principles with review intervals at 1 day, 3 days, 7 days, and 14 days to maximize long-term retention.

##### A.3.1 Week 1: Foundation Building

Day	New Learning	Learning Activity	Review (Spaced Intervals)	Done
Day 1	Two Pointers	Learn template, solve: Two Sum II	–	<input type="checkbox"/>
Day 2	Sliding Window	Learn template, solve: Longest Substring	Two Pointers (1-day)	<input type="checkbox"/>
Day 3	Binary Search	Learn template, solve: Search in Rotated Array	Sliding Window (1-day)	<input type="checkbox"/>
Day 4	DFS Fundamentals	Learn tree/graph traversal	Two Pointers (3-day), Binary Search (1-day)	<input type="checkbox"/>
Day 5	BFS Fundamentals	Learn level-order traversal	Sliding Window (3-day), DFS (1-day)	<input type="checkbox"/>
Day 6	Dynamic Programming	Learn basic DP patterns	Binary Search (3-day), BFS (1-day)	<input type="checkbox"/>
Day 7	Week 1 Integration	Combine learned patterns	DFS (3-day), DP (1-day)	<input type="checkbox"/>

##### A.3.2 Week 2: Pattern Reinforcement

Day	New Learning	Learning Activity	Review (Spaced Intervals)	Done
Day 8	Advanced DFS	Backtracking, path finding	Two Pointers (7-day), BFS (3-day), DP (1-day)	<input type="checkbox"/>
Day 9	Advanced DP	2D DP, optimization	Sliding Window (7-day), Advanced DFS (1-day)	<input type="checkbox"/>
Day 10	Greedy Algorithms	Local optimal choices	Binary Search (7-day), Advanced DP (1-day)	<input type="checkbox"/>
Day 11	Backtracking	Permutations, combinations	DFS (7-day), Greedy (1-day)	<input type="checkbox"/>
Day 12	Advanced Backtracking	N-Queens, Sudoku patterns	BFS (7-day), Backtracking (1-day)	<input type="checkbox"/>
Day 13	Interval Problems	Merge, overlap detection	DP (7-day), Advanced Backtracking (1-day)	<input type="checkbox"/>
Day 14	Week 2 Integration	Complex problem combinations	Week 1 Integration (7-day), Intervals (1-day)	<input type="checkbox"/>

##### A.3.3 Week 3: Advanced Structures

##### A.3.4 Week 4: Mastery and Integration

##### A.3.5 Daily Review Strategy

- **1-Day Review:** Quick template rewriting (5-10 minutes)
- **3-Day Review:** Solve 1 easy problem using the pattern (15-20 minutes)
- **7-Day Review:** Solve 1 medium problem, focus on edge cases (25-30 minutes)
- **14-Day Review:** Solve 1 challenging problem or combo pattern (35-40 minutes)

**Total Daily Time Investment:** 90-120 minutes (30-45 min new learning + 60-75 min reviews)

Day	New Learning	Learning Activity	Review (Spaced Intervals)	Done
Day 15	Heap Operations	Min/max heaps, K problems	Advanced DFS (7-day), Week 2 Integration (1-day)	<input type="checkbox"/>
Day 16	Priority Queues	Advanced heap applications	Advanced DP (7-day), Heaps (1-day)	<input type="checkbox"/>
Day 17	Union-Find	Path compression, optimization	Greedy (7-day), Priority Queues (1-day)	<input type="checkbox"/>
Day 18	Trie Operations	Prefix trees, word problems	Backtracking (7-day), Union-Find (1-day)	<input type="checkbox"/>
Day 19	Advanced Trie	Auto-complete, word search	Advanced Backtracking (7-day), Trie (1-day)	<input type="checkbox"/>
Day 20	Matrix Traversal	2D patterns, connectivity	Intervals (7-day), Advanced Trie (1-day)	<input type="checkbox"/>
Day 21	Week 3 Integration	Advanced data structure combos	Two Pointers (14-day), Matrix (1-day)	<input type="checkbox"/>

Day	New Learning	Learning Activity	Review (Spaced Intervals)	Done
Day 22	Special Algorithms	Kadane's, KMP, Manacher's	Sliding Window (14-day), Heaps (7-day), Week 3 Integration (1-day)	<input type="checkbox"/>
Day 23	Monotonic Stack	Next greater, histogram problems	Binary Search (14-day), Priority Queues (7-day), Special Algorithms (1-day)	<input type="checkbox"/>
Day 24	Topological Sort	Kahn's algorithm, dependencies	DFS (14-day), Union-Find (7-day), Monotonic Stack (1-day)	<input type="checkbox"/>
Day 25	Graph Algorithms	Shortest path, MST	BFS (14-day), Trie (7-day), Topological Sort (1-day)	<input type="checkbox"/>
Day 26	Advanced Graph	Strongly connected components	DP (14-day), Advanced Trie (7-day), Graph Algorithms (1-day)	<input type="checkbox"/>
Day 27	Mock Interview 1	Timed practice, pattern recognition	Week 1 Integration (14-day), Matrix (7-day), Advanced Graph (1-day)	<input type="checkbox"/>
Day 28	Mock Interview 2	Final assessment, weak areas	All patterns that showed weakness in Mock 1	<input type="checkbox"/>

## A.4 Daily Routine Checklist

### Daily Practice Checklist

Date: \_\_\_\_\_

#### A.4.1 Morning Routine (30 minutes)

- ☐ **Template Writing (10 min):** Write 3 random templates from memory
  - Template 1: \_\_\_\_\_ Time: \_\_\_\_ min
  - Template 2: \_\_\_\_\_ Time: \_\_\_\_ min
  - Template 3: \_\_\_\_\_ Time: \_\_\_\_ min
- ☐ **Pattern Recognition (10 min):** Look at problem titles and identify patterns
- ☐ **Complexity Analysis (10 min):** Review time/space complexity for each pattern

#### A.4.2 Evening Session (60-90 minutes)

- ☐ **Problem Solving (45-60 min):** 2-3 problems focusing on current week's patterns
  - Problem 1: \_\_\_\_\_
  - Problem 2: \_\_\_\_\_

– Problem 3: .....

- ☐ **Template Review (15-30 min):** Review and rewrite struggled templates

#### A.4.3 Weekly Reflection

**What went well this week?**

**What needs improvement?**

**Next week's focus:**

## A.5 Mock Interview Tracker

Date	Problem	Pattern Used	Time	Notes/Improvements
-----	-----	-----	----	-----
-----	-----	-----	----	-----
-----	-----	-----	----	-----
-----	-----	-----	----	-----
-----	-----	-----	----	-----

### A.5.1 Interview Readiness Checklist

#### Before Your Real Interview:

- ☐ Can write all 12 main templates from memory in under 2 minutes each
- ☐ Completed at least 100 problems across all difficulty levels
- ☐ Can identify correct pattern within 30 seconds of reading problem
- ☐ Solve medium problems consistently in 15-20 minutes
- ☐ Explain time/space complexity confidently for all solutions
- ☐ Completed at least 5 timed mock interview sessions
- ☐ Comfortable with edge cases and boundary conditions
- ☐ Can clearly communicate approach before coding

#### Final Preparation Notes: