Volume 1: Data Structures & Algorithms

Table of Contents

Competitive Programming for Infosys L3 Specialist Programmer

Authors: Technical Interview Preparation Team

Edition: 2025

Target: Infosys Specialist Programmer (L3/SP) Coding Round

Preface

This textbook is designed specifically for candidates preparing for the **Infosys L3 Specialist Programmer role**, with emphasis on the **3-hour coding round** featuring Easy, Medium (Greedy), and Hard (Dynamic Programming) problems.

The content focuses on **C++ as the primary language** with **Python as a secondary option**, providing production-grade implementations and interview-ready templates.

Key Features:

- Pattern-based learning (not just problem solving)
- Time/space complexity analysis for every solution
- Edge case engineering and common bug patterns
- Infosys-specific question patterns from 2023-2025

Table of Contents

11. Bit Manipulation
12. Mathematics & Number Theory

Chapter 1: Arrays & Strings

1.1 Fundamental Concepts

Arrays and strings constitute **40% of Infosys L3 coding questions**[1]. Mastery requires understanding not just implementation, but **pattern recognition under time pressure**.

Core Mental Models:

1. **Indexing as State Machine**: Each array problem represents state transitions across indices
2. **In-place vs Auxiliary Space**: L3 problems often require O(1) space optimization
3. **Prefix Computation**: Transform O(n) range queries to O(1) lookup

Critical Patterns:

- Prefix Sum (range queries)
- Two Pointers (sorted array optimization)
- Sliding Window (subarray/substring conditions)
- Kadane's Algorithm (maximum subarray)

1.2 Pattern 1: Prefix Sum

Concept: Precompute cumulative sums to answer range queries in O(1) time.

Template (C++):

```
#include &lt;vector&gt;
using namespace std;

class PrefixSum {
public:
    vector&lt;int&gt; prefix;

    PrefixSum(vector&lt;int&gt;&amp; arr) {
        int n = arr.size();
        prefix.resize(n + 1, 0);
        for (int i = 1; i &lt;= n; i++) {
            prefix[i] = prefix[i-1] + arr[i-1];
        }
    }

    // Range sum [L, R] in O(1)
    int rangeSum(int L, int R) {
        return prefix[R + 1] - prefix[L];
    }
};
```

Time Complexity: O(n) preprocessing, O(1) per query
Space Complexity: O(n)

Problem: Subarray Sum Equals K (Infosys Medium)[2]

Statement: Given array and integer k, count subarrays with sum = k.

Input: nums = [1, 1, 1], k = 2
Output: 2 (subarrays [1,1] at indices 0-1 and 1-2)

Approach: Prefix sum + Hash map

Why Hash Map?: We need to find pairs (i, j) where `prefix[j] - prefix[i] = k`, i.e., `prefix[i] = prefix[j] - k`. Hash map stores frequency of each prefix sum.

C++ Solution:

```
int subarraySum(vector&lt;int&gt;&amp; nums, int k) {
    unordered_map&lt;int, int&gt; prefixCount;
    prefixCount[0] = 1; // Empty prefix

    int sum = 0, count = 0;
    for (int num : nums) {
        sum += num;
        // Check if (sum - k) exists
        if (prefixCount.find(sum - k) != prefixCount.end()) {
            count += prefixCount[sum - k];
        }
        prefixCount[sum]++;
    }
    return count;
}
```

Python Solution:

```
from collections import defaultdict

def subarraySum(nums, k):
    prefix_count = defaultdict(int)
    prefix_count[0] = 1

    current_sum = 0
    count = 0

    for num in nums:
        current_sum += num
        count += prefix_count[current_sum - k]
        prefix_count[current_sum] += 1

    return count
```

Complexity:

- Time: O(n) — single pass through array
- Space: O(n) — hash map storing prefix sums

Edge Cases:

1. k = 0 → Initialize `prefixCount[0] = 1`
2. Negative numbers → Hash map handles all integer sums
3. Single element equals k → Caught by initialization

1.3 Pattern 2: Two Pointers

Concept: Use two pointers to avoid nested loops, reducing $O(n^2)$ to $O(n)$.

When to Use:

- Sorted array (opposite direction pointers)
- Cycle detection (fast-slow pointers)
- In-place array modification (same direction)

Template (Opposite Direction):

```
vector&lt;int&gt; twoSum(vector&lt;int&gt;&amp; arr, int target) {
    int left = 0, right = arr.size() - 1;

    while (left &lt; right) {
        int sum = arr[left] + arr[right];
        if (sum == target) {
            return {left, right};
        } else if (sum &lt; target) {
            left++;  // Need larger sum
        } else {
            right--; // Need smaller sum
        }
    }
    return {-1, -1};
}
```

Time: O(n), **Space**: O(1)

Problem: Container With Most Water (Infosys Medium)[3]

Statement: Given heights array, find two lines that form container with maximum water.

Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49 (lines at indices 1 and 8)

Greedy Insight: Moving the pointer with the **smaller height** is always optimal (moving taller height can only decrease area).

C++ Solution:

```
int maxArea(vector&lt;int&gt;&amp; height) {
    int left = 0, right = height.size() - 1;
    int maxWater = 0;

    while (left &lt; right) {
        int h = min(height[left], height[right]);
        int w = right - left;
        maxWater = max(maxWater, h * w);

        // Move pointer with smaller height
        if (height[left] &lt; height[right]) {
            left++;
        } else {
            right--;
        }
    }
    return maxWater;
}
```

Proof of Correctness: Moving the taller line can only decrease area (width decreases, height bounded by min). Therefore, moving shorter line is optimal.

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

1.4 Pattern 3: Sliding Window

Concept: Maintain a window of elements, expand/shrink based on conditions.

Types:

1. **Fixed Window**: Window size constant (e.g., "max sum of size k")
2. **Variable Window**: Window size changes (e.g., "longest substring without repeating chars")

Fixed Window Template:

```
int maxSumFixedWindow(vector&lt;int&gt;&amp; arr, int k) {
    int sum = 0;
    // Initialize first window
    for (int i = 0; i &lt; k; i++) sum += arr[i];
    int maxSum = sum;

    // Slide: add right, remove left
    for (int i = k; i &lt; arr.size(); i++) {
        sum += arr[i] - arr[i - k];
        maxSum = max(maxSum, sum);
    }
    return maxSum;
}
```

Variable Window Template:

```
int longestSubarrayWithCondition(vector&lt;int&gt;&amp; arr) {
    int left = 0, maxLen = 0;
    // State variables (sum, freq map, etc.)

    for (int right = 0; right &lt; arr.size(); right++) {
        // Expand: add arr[right]

        // Shrink while invalid
        while (condition_violated &amp;&amp; left &lt;= right) {
            // Remove arr[left]
            left++;
        }

        maxLen = max(maxLen, right - left + 1);
    }
    return maxLen;
}
```

Problem: Longest Substring Without Repeating Characters (Infosys Medium-Hard)[4]

Statement: Find length of longest substring with all unique characters.

Input: s = "abcabcbb"
Output: 3 ("abc")

C++ Solution:

```
int lengthOfLongestSubstring(string s) {
    unordered_map&lt;char, int&gt; charIndex;
    int maxLen = 0, left = 0;

    for (int right = 0; right &lt; s.length(); right++) {
        char c = s[right];

        // If char seen and in current window
        if (charIndex.find(c) != charIndex.end() &amp;&amp;
            charIndex[c] &gt;= left) {
            left = charIndex[c] + 1; // Shrink window
        }

        charIndex[c] = right;
        maxLen = max(maxLen, right - left + 1);
    }
    return maxLen;
}
```

Python Solution:

```
def lengthOfLongestSubstring(s):
    char_index = {}
    max_len = 0
    left = 0

    for right, char in enumerate(s):
        if char in char_index and char_index[char] &gt;= left:
            left = char_index[char] + 1

        char_index[char] = right
        max_len = max(max_len, right - left + 1)

    return max_len
```

Critical Bug to Avoid:

```
// WRONG: Not checking if duplicate is in current window
if (charIndex.find(c) != charIndex.end()) {
    left = charIndex[c] + 1; // May move left backward!
}

// Example: "abba"
// When processing second 'a', left would incorrectly jump to index 1
// but current window is "bb", not including first 'a'
```

Complexity: O(n) time, O(min(m,n)) space where m = charset size

1.5 Pattern 4: Kadane's Algorithm

Concept: Maximum subarray sum using dynamic programming.

Key Insight: At each position, decide whether to extend current subarray or start new one.

C++ Solution:

```
int maxSubArray(vector&lt;int&gt;&amp; nums) {
    int maxSoFar = nums[0];
    int maxEndingHere = nums[0];
```

```
    for (int i = 1; i &lt; nums.size(); i++) {
        // Extend or start new?
        maxEndingHere = max(nums[i], maxEndingHere + nums[i]);
        maxSoFar = max(maxSoFar, maxEndingHere);
    }
    return maxSoFar;
}
```

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

Extension: Return Subarray Indices:

```
pair&lt;int, int&gt; maxSubArrayIndices(vector&lt;int&gt;&amp; nums) {
    int maxSoFar = nums[0], maxEndingHere = nums[0];
    int start = 0, end = 0, tempStart = 0;

    for (int i = 1; i &lt; nums.size(); i++) {
        if (nums[i] &gt; maxEndingHere + nums[i]) {
            maxEndingHere = nums[i];
            tempStart = i; // New subarray
        } else {
            maxEndingHere += nums[i];
        }

        if (maxEndingHere &gt; maxSoFar) {
            maxSoFar = maxEndingHere;
            start = tempStart;
            end = i;
        }
    }
    return {start, end};
}
```

1.6 Edge Case Engineering

Comprehensive Checklist:

Input Validation:

- [ ] n = 0 (empty array/string)
- [ ] n = 1 (single element)
- [ ] All elements identical
- [ ] All elements distinct

Boundary Values:

- [ ] INT_MIN, INT_MAX (overflow risk)
- [ ] Negative numbers
- [ ] Zero in array (division/modulo issues)

String-Specific:

- [ ] Empty string ""
- [ ] Single character
- [ ] All same characters "aaaa"
- [ ] Special characters/Unicode

Algorithm-Specific:

- [ ] Sliding window: k > n
- [ ] Two pointers: left > right at start
- [ ] Prefix sum: off-by-one indexing

Performance:

- [ ] n = 10^5 → O(n^2) will TLE
- [ ] String concatenation in loop → use StringBuilder
- [ ] Repeated sorting → single sort + manipulation

1.7 Common Bug Patterns

Bug 1: Off-by-One in Sliding Window

```
// WRONG:
for (int i = k; i &lt; n; i++) {
    sum += arr[i] - arr[i - k - 1]; // Index error!
}

// CORRECT:
for (int i = k; i &lt; n; i++) {
    sum += arr[i] - arr[i - k];
}
```

Bug 2: Integer Overflow

```
// WRONG (if sum &gt; INT_MAX):
int sum = 0;
for (int x : arr) sum += x;

// CORRECT:
long long sum = 0;
for (int x : arr) sum += x;
```

Bug 3: Not Handling Empty Input

```
// WRONG:
int maxElement(vector&lt;int&gt;&amp; arr) {
    int maxVal = arr[0]; // Crashes if empty!
}

// CORRECT:
int maxElement(vector&lt;int&gt;&amp; arr) {
    if (arr.empty()) return -1;
    int maxVal = arr[0];
}
```

1.8 Practice Problems

Easy (15-20 minutes each):

1. Two Sum
2. Remove Duplicates from Sorted Array
3. Best Time to Buy and Sell Stock
4. Valid Palindrome
5. Merge Sorted Array

Medium (25-35 minutes each):
6. Container With Most Water
7. Longest Substring Without Repeating Characters
8. Subarray Sum Equals K
9. Product of Array Except Self
10. Find All Anagrams in String

Hard (40-60 minutes each):
11. Minimum Window Substring
12. Trapping Rain Water
13. Sliding Window Maximum
14. Longest Consecutive Sequence
15. Maximum Subarray Sum (2D extension)

Chapter 2: Hashing & Frequency Problems

2.1 Hash Table Fundamentals

Mental Model: Hash table as **memory** — O(1) lookup is "remembering" previous computations.

C++ Hash Structures:

Structure	Order	Time	Use Case
`unordered_map`	No	O(1) avg	Key-value, no order needed
`map`	Yes (sorted)	O(log n)	Key-value, ordered iteration
`unordered_set`	No	O(1) avg	Membership check
`set`	Yes (sorted)	O(log n)	Membership + ordered

Python Hash Structures:

- `dict` → unordered_map
- `set` → unordered_set
- `Counter` → frequency map
- `defaultdict` → dict with default values

2.2 Pattern: Frequency Counting

Problem: First Unique Character (Infosys Easy)

C++ Solution:

```
int firstUniqChar(string s) {
    unordered_map&lt;char, int&gt; freq;

    for (char c : s) freq[c]++;

    for (int i = 0; i &lt; s.length(); i++) {
        if (freq[s[i]] == 1) return i;
    }
    return -1;
}

// Optimization for lowercase letters:
int firstUniqCharOptimized(string s) {
    int freq[26] = {0};

    for (char c : s) freq[c - 'a']++;

    for (int i = 0; i &lt; s.length(); i++) {
        if (freq[s[i] - 'a'] == 1) return i;
    }
    return -1;
}
```

Performance Note: Array[26] is **faster than unordered_map** for small, fixed alphabets.

2.3 Pattern: Two Sum Family

Problem: Two Sum (Infosys Classic)[5]

C++ Solution:

```
vector&lt;int&gt; twoSum(vector&lt;int&gt;&amp; nums, int target) {
    unordered_map&lt;int, int&gt; numIndex;

    for (int i = 0; i &lt; nums.size(); i++) {
        int complement = target - nums[i];

        if (numIndex.find(complement) != numIndex.end()) {
            return {numIndex[complement], i};
        }
        numIndex[nums[i]] = i;
    }
    return {-1, -1};
}
```

Python Solution:

```
def twoSum(nums, target):
    num_index = {}

    for i, num in enumerate(nums):
        complement = target - num
        if complement in num_index:
            return [num_index[complement], i]
        num_index[num] = i
```

```
    return [-1, -1]
```

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

Variation: Count Pairs:

```
int countPairs(vector&lt;int&gt;&amp; nums, int k) {
    unordered_map&lt;int, int&gt; freq;
    int count = 0;

    for (int num : nums) {
        count += freq[k - num];
        freq[num]++;
    }
    return count;
}
```

Edge Case: k = 2*num (same element used twice). Above code handles correctly by adding before incrementing.

Chapter 3: Dynamic Programming

3.1 DP Philosophy

Core Principle: Break problem into subproblems, avoid recomputation.

Recognition Signals:

- "Count ways to..."
- "Maximize/minimize with constraints..."
- "Longest/shortest..."
- Overlapping subproblems + optimal substructure

FAST Method (from Dynamic Programming for Interviews[6]):

1. **First solution**: Write recursive brute force
2. **Analyze**: Identify overlapping subproblems
3. **Subproblems**: Define DP state
4. **Turn around**: Convert to bottom-up

3.2 Pattern: 1D DP

Problem: Climbing Stairs (Infosys Easy)

Recursive (Brute Force):

```
int climbStairs(int n) {
    if (n &lt;= 2) return n;
    return climbStairs(n-1) + climbStairs(n-2);
}
```

Time: O(2^n) — exponential!

DP (Bottom-Up):

```
int climbStairs(int n) {
    if (n &lt;= 2) return n;

    vector&lt;int&gt; dp(n + 1);
    dp[1] = 1;
    dp[2] = 2;

    for (int i = 3; i &lt;= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

Time: O(n), Space: O(n)

Space Optimized (only need last 2 values):

```
int climbStairs(int n) {
    if (n &lt;= 2) return n;

    int prev2 = 1, prev1 = 2;
    for (int i = 3; i &lt;= n; i++) {
        int curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

Time: O(n), Space: O(1)

3.3 Pattern: 2D DP (Knapsack)

Problem: 0/1 Knapsack (Infosys Hard)[7]

Statement: Given weights, values, and capacity W, maximize value without exceeding weight.

DP State: `dp[i][w]` = max value using first i items with weight limit w

Recurrence:

$$dp[i][w] = \begin{cases} dp[i-1][w] & amp; \text{if } wt[i-1] \quad gt; w \\ \max(dp[i-1][w], val[i-1] + dp[i-1][w - wt[i-1]]) & amp; \text{otherwise} \end{cases} \tag{1}$$

C++ Solution:

```
int knapsack(vector&lt;int&gt;&amp; wt, vector&lt;int&gt;&amp; val, int W) {
    int n = wt.size();
    vector&lt;vector&lt;int&gt;&gt; dp(n + 1, vector&lt;int&gt;(W + 1, 0));

    for (int i = 1; i &lt;= n; i++) {
        for (int w = 1; w &lt;= W; w++) {
            if (wt[i-1] &lt;= w) {
                dp[i][w] = max(dp[i-1][w],
                              val[i-1] + dp[i-1][w - wt[i-1]]);
            } else {
                dp[i][w] = dp[i-1][w];
            }
        }
    }
```

```
    return dp[n][W];
}
```

Space Optimization (2D → 1D):

```
int knapsackOptimized(vector&lt;int&gt;&amp; wt, vector&lt;int&gt;&amp; val, int W) {
    int n = wt.size();
    vector&lt;int&gt; dp(W + 1, 0);

    for (int i = 0; i &lt; n; i++) {
        for (int w = W; w &gt;= wt[i]; w--) { // Reverse!
            dp[w] = max(dp[w], val[i] + dp[w - wt[i]]);
        }
    }
    return dp[W];
}
```

Why reverse order? Prevents using same item twice (we only have previous row data when processing right to left).

3.4 Pattern: String DP

Problem: Longest Common Subsequence (Infosys Hard)[8]

DP State: `dp[i][j]` = LCS of s1[0..i-1] and s2[0..j-1]

C++ Solution:

```
int longestCommonSubsequence(string s1, string s2) {
    int m = s1.length(), n = s2.length();
    vector&lt;vector&lt;int&gt;&gt; dp(m + 1, vector&lt;int&gt;(n + 1, 0));

    for (int i = 1; i &lt;= m; i++) {
        for (int j = 1; j &lt;= n; j++) {
            if (s1[i-1] == s2[j-1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }
    return dp[m][n];
}
```

Backtracking to Print LCS:

```
string printLCS(string s1, string s2, vector&lt;vector&lt;int&gt;&gt;&amp; dp) {
    int i = s1.length(), j = s2.length();
    string lcs = "";

    while (i &gt; 0 &amp;&amp; j &gt; 0) {
        if (s1[i-1] == s2[j-1]) {
            lcs = s1[i-1] + lcs;
            i--; j--;
        } else if (dp[i-1][j] &gt; dp[i][j-1]) {
            i--;
        } else {
            j--;
        }
    }
```

```
    return lcs;
}
```

References

[1] PrepInsta. (2025). Infosys Specialist Programmer Coding Questions. https://prepinsta.com/infosys-sp-and-dse/specialist-programmer/coding-questions/

[2] GeeksforGeeks. (2024). Infosys Interview Experience for Specialist Programmer. https://www.geeksforgeeks.org/interview-experiences/infosys-interview-experience-specialist-programmer-full-time-on-campus-2025

[3] InterviewCake. (2025). Breadth-First Search (BFS) - Shortest Paths. https://www.interviewcake.com/concept/java/bfs

[4] Reddit. (2025). How to best prepare for L2, L3 specialist programmer role at infosys. https://www.reddit.com/r/developersIndia/comments/1owve00/

[5] CCBP. (2025). Top Infosys Coding Questions with Solutions. https://www.ccbp.in/blog/articles/infosys-coding-questions

[6] Byte by Byte. (2019). Dynamic Programming for Interviews. https://www.byte-by-byte.com/wp-content/uploads/2019/04/Dynamic-Programming-for-Interviews.pdf

[7] DesignGurus. (2024). How to prepare an algorithm interview. https://www.designgurus.io/answers/detail/how-to-prepare-an-algorithm-interview

[8] LinkedIn. (2021). Most asked interview questions on OS, DBMS, CN. https://www.linkedin.com/pulse/most-asked-interview-questions-os-dbms-cn-harsh-agarwal