DSA Mastery: Complete Hashing Handbook

A Pattern-Driven Interview Guide

Table of Contents

1. Core Concepts & Theory

1.1 Hash Table Fundamentals

Time Complexity:

- Average Case: O(1) for insert, delete, search
- Worst Case: O(n) when all elements hash to same bucket
- Amortized: O(1) per operation including resizing

Space Complexity: O(n)

Load Factor (α = n/m):

- n = number of elements
- m = number of buckets
- Typical threshold: 0.75 (Java HashMap), 1.0 (C++ unordered_map)
- Rehashing occurs when threshold exceeded

Collision Resolution:

1. **Chaining**: Each bucket contains linked list
2. **Open Addressing**: Linear probing, quadratic probing, double hashing

1.2 When to Use Hashing

✅ **Use Hash Table When**:

- Need O(1) lookup, insert, delete
- Finding complements/pairs (Two Sum pattern)
- Counting frequencies
- Detecting duplicates
- Grouping/categorization
- Caching results (memoization)

❌ **Don't Use Hash Table When**:

- Need sorted order (use TreeMap/set instead)
- Need range queries (use TreeMap/set)
- Memory is severely constrained
- Working with small fixed dataset (array may be faster)

2. Data Structure Selection Guide

2.1 C++ Hash Containers

```
// UNORDERED_MAP - Key-Value pairs, O(1) average
unordered_map<int, int> map;
map[key] = value;
map.count(key);  // Returns 0 or 1
map.find(key) != map.end();  // Check existence

// UNORDERED_SET - Unique elements, O(1) average
unordered_set<int> set;
set.insert(element);
set.count(element);  // Returns 0 or 1
set.erase(element);

// MAP - Ordered, O(log n)
map<int, int> orderedMap;
orderedMap.lower_bound(key);  // First >= key
orderedMap.upper_bound(key);  // First > key

// SET - Ordered, O(log n)
set<int> orderedSet;
orderedSet.lower_bound(value);
```

2.2 Decision Matrix

Requirement	Data Structure	Time	When to Use
Fast lookup, no order	`unordered_map/set`	O(1) avg	Frequency counting, duplicates
Fast lookup + sorted	`map/set`	O(log n)	Range queries, closest elements
Insertion order matters	Custom or LinkedHashMap (Java)	O(1)	LRU cache, maintaining order
Count frequency	`unordered_map<T, int>`	O(1)	Character/element counting

Requirement	Data Structure	Time	When to Use
Group by key	`unordered_map<T, vector<T>>`	O(1)	Anagrams, categorization

3. Master Pattern Catalog

PATTERN 1: Frequency/Count Mapping

Pattern Recognition

- Keywords: "count", "frequency", "how many times", "occurrences"
- Goal: Track how often each element appears
- Time: O(n), Space: O(k) where k = unique elements

C++ Template

```
// Basic frequency counting
unordered_map<int, int> freq;
for (int num : arr) {
    freq[num]++;
}

// With character counting
unordered_map<char, int> charCount;
for (char c : str) {
    charCount[c]++;
}

// Find elements with specific frequency
for (auto& [element, count] : freq) {
    if (count == k) {
        // Process element
    }
}
```

Problem Set

1. **First Unique Character in String** - Find first non-repeating character
2. **Valid Anagram** - Check if two strings have same character frequencies
3. **Top K Frequent Elements** - Return k most frequent elements
4. **Sort Characters by Frequency** - Sort string by character frequency
5. **Find All Anagrams in String** - Find all anagram substrings
6. **Custom Sort String** - Sort based on custom character order
7. **Unique Number of Occurrences** - Check if all frequencies are unique
8. **Uncommon Words from Two Sentences** - Find words appearing once total
9. **K Most Frequent Words** - Return k most common words
10. **Subdomain Visit Count** - Aggregate domain visit counts

Interview Tips

- **Start with**: Creating frequency map, then query it
- **Optimization**: Use array instead of map for limited character set (26 lowercase letters)
- **Follow-up**: "Can you do it in one pass?" - Yes, build map while checking condition
- **Variant**: Asked to find elements appearing exactly/at least/at most k times

Remember for Interview

✅ Always check if you can use fixed-size array (lowercase letters → size 26)
✅ For string problems, ask about character set (ASCII, Unicode?)
✅ Frequency map is often the first step, not the final solution
✅ Consider using bucket sort for O(n) time when finding top k frequent

Edge Cases

- Empty input (return appropriate default)
- All elements same frequency
- Single element
- No element satisfies condition
- Tie in frequencies (ask how to handle)

PATTERN 2: Complement/Pair Finding

Pattern Recognition

- Keywords: "two elements", "pair", "sum equals target", "complement"
- Goal: Find pairs (a, b) where a + b = target or a = f(b)
- Time: O(n), Space: O(n)

C++ Template

```cpp
// Two Sum pattern
unordered_map<int, int> seen;
for (int i = 0; i < n; i++) {
    int complement = target - arr[i];
    if (seen.count(complement)) {
        return {seen[complement], i};
    }
    seen[arr[i]] = i;
}

// Store what you're looking for, not what you've seen
unordered_set<int> needed;
for (int num : arr) {
    if (needed.count(num)) {
        // Found pair
    }
    needed.insert(target - num);
}
```

Problem Set

1. **Two Sum** - Find two numbers that sum to target
2. **Two Sum II (Sorted)** - Two sum in sorted array
3. **3Sum** - Find all triplets summing to zero
4. **4Sum** - Find all quadruplets summing to target
5. **Two Sum IV (BST)** - Find pair in BST
6. **Subarray Sum Equals K** - Count subarrays with given sum
7. **Continuous Subarray Sum** - Subarray sum multiple of k
8. **Check If Array Pairs Divisible by K** - Pairs with sum divisible by k

9. **Max Number of K-Sum Pairs** - Maximum pairs with sum k
10. **Count Nice Pairs in Array** - Pairs where nums[i] + rev(nums[j]) = nums[j] + rev(nums[i])

Interview Tips

- **Key Insight**: Store complement in map, check if current element is someone's complement
- **Variation**: May need to return indices, count pairs, or find all unique pairs
- **3Sum/4Sum**: Reduce to 2Sum by fixing outer elements and using two pointers
- **Optimization**: For sorted input, consider two pointers instead of hash

Remember for Interview

✅ HashMap stores: key = number, value = index (for returning positions)
✅ For "all unique pairs", need to handle duplicates (sort + skip duplicates)
✅ Don't count same element twice (check index or use proper iteration)
✅ Ask: "Can same element be used twice?" "Need indices or just values?"

Edge Cases

- Array with duplicates
- Negative numbers in array
- Target is zero
- No valid pair exists
- Multiple valid pairs (return all vs. any one)
- Same element used twice (arr[i] + arr[i] = target)

PATTERN 3: Sliding Window with Hash

Pattern Recognition

- Keywords: "substring", "subarray", "window", "contiguous", "longest/shortest"
- Goal: Maintain window state efficiently while sliding
- Time: O(n), Space: O(k) where k = unique elements in window

C++ Template

```
// Longest substring with at most K distinct
unordered_map<char, int> window;
int left = 0, maxLen = 0;

for (int right = 0; right < s.length(); right++) {
    window[s[right]]++;

    // Shrink window if constraint violated
    while (window.size() > k) {
        window[s[left]]--;
        if (window[s[left]] == 0) {
            window.erase(s[left]);
        }
        left++;
    }

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

// Fixed size window
unordered_map<char, int> window;
for (int i = 0; i < s.length(); i++) {
    window[s[i]]++;

    if (i >= k) {  // Window size exceeded
        window[s[i - k]]--;
        if (window[s[i - k]] == 0) {
            window.erase(s[i - k]);
        }
    }

    if (i >= k - 1) {
        // Process window of size k
    }
}
```

Problem Set

1. **Longest Substring Without Repeating Characters** - Max length with unique chars
2. **Longest Substring with At Most K Distinct Characters** - Window with k distinct

3. **Minimum Window Substring** - Smallest window containing all characters
4. **Longest Repeating Character Replacement** - Max length after k replacements
5. **Permutation in String** - Check if permutation exists as substring
6. **Find All Anagrams in String** - All anagram starting positions
7. **Fruit Into Baskets** - Max fruits with 2 types (same as k=2 distinct)
8. **Subarrays with K Different Integers** - Count subarrays with exactly k distinct
9. **Minimum Window Subsequence** - Smallest window containing subsequence
10. **Longest Substring with At Most Two Distinct Characters** - K=2 variant

Interview Tips

- **Two pointers**: Left and right pointers define window boundaries
- **Expand**: Always expand right pointer in outer loop
- **Contract**: Use while loop to shrink left when condition violated
- **State**: Hash map tracks window state (frequency, presence)
- **Answer update**: Update result after ensuring valid window

Remember for Interview

✅ Window = [left, right] inclusive
✅ Always remove from left when shrinking: decrease count, erase if zero
✅ Check window validity BEFORE updating answer
✅ For "exactly k", use: atMost(k) - atMost(k-1)
✅ Map size = number of distinct elements in window

Edge Cases

- Empty string/array
- K = 0 or K > length
- All characters are same
- All characters are distinct
- Window size = 0 or 1
- Pattern longer than text

PATTERN 4: Prefix Sum + Hash

Pattern Recognition

- Keywords: "subarray sum", "range sum", "continuous subarray", "divisible by k"
- Goal: Find subarrays with specific sum properties
- Time: O(n), Space: O(n)

C++ Template

```cpp
// Subarray sum equals K
unordered_map<int, int> prefixSumCount;
prefixSumCount[0] = 1;  // Important: handle subarrays from index 0
int prefixSum = 0, count = 0;

for (int num : arr) {
    prefixSum += num;

    // If (prefixSum - K) exists, found subarray
    if (prefixSumCount.count(prefixSum - k)) {
        count += prefixSumCount[prefixSum - k];
    }

    prefixSumCount[prefixSum]++;
}

// For remainder/modulo problems
unordered_map<int, int> remainderIndex;
remainderIndex[0] = -1;  // Handle from beginning
int prefixSum = 0;

for (int i = 0; i < n; i++) {
    prefixSum += arr[i];
    int remainder = prefixSum % k;

    if (remainderIndex.count(remainder)) {
        int length = i - remainderIndex[remainder];
        // Check if valid
    } else {
        remainderIndex[remainder] = i;
    }
}
```

Problem Set

1. **Subarray Sum Equals K** - Count subarrays with sum k
2. **Continuous Subarray Sum** - Subarray sum multiple of k, length ≥ 2
3. **Contiguous Array** - Longest subarray with equal 0s and 1s
4. **Subarray Sums Divisible by K** - Count subarrays divisible by k

5. **Maximum Size Subarray Sum Equals K** - Longest subarray with sum k
6. **Binary Subarrays With Sum** - Count subarrays summing to goal
7. **Find Two Non-overlapping Sub-arrays Each With Target Sum** - Two disjoint subarrays
8. **Partition Array Into Two Arrays to Minimize Sum Difference** - Minimize partition difference
9. **Make Sum Divisible by P** - Smallest subarray to remove for divisibility
10. **Count Number of Nice Subarrays** - Exactly k odd numbers

Interview Tips

- **Core idea**: If prefixSum[j] - prefixSum[i] = k, then subarray (i, j] has sum k
- **Map stores**: prefixSum → count/index depending on problem
- **Initialize**: map[0] = 1 (or -1 for index) to handle subarrays starting at index 0
- **Modulo trick**: For divisibility, use remainder as key

Remember for Interview

✅ Map[0] = 1 is crucial - don't forget initialization
✅ For "at least k", store first occurrence; for "count", store frequency
✅ Negative numbers? Prefix sum approach still works
✅ For modulo: (a - b) % k = 0 means a % k = b % k
✅ Handle negative remainders: ((val % k) + k) % k

Edge Cases

- Target sum is 0
- All elements are 0
- Negative numbers present
- k = 0 (division by zero for modulo)
- Multiple subarrays with same sum
- Empty subarray counts or not

PATTERN 5: Grouping/Categorization

Pattern Recognition

- Keywords: "group by", "anagrams", "categorize", "common property"
- Goal: Group elements sharing common characteristic

- Time: O(n * k) where k = key computation time, Space: O(n)

C++ Template

```cpp
// Group anagrams
unordered_map<string, vector<string>> groups;
for (string& s : strs) {
    string key = s;
    sort(key.begin(), key.end());  // Sorted string as key
    groups[key].push_back(s);
}

// Alternative: character count as key
unordered_map<string, vector<string>> groups;
for (string& s : strs) {
    int count[26] = {0};
    for (char c : s) count[c - 'a']++;

    string key = "";
    for (int i = 0; i < 26; i++) {
        if (count[i] > 0) {
            key += string(count[i], 'a' + i);
        }
    }
    groups[key].push_back(s);
}

// Custom key function
unordered_map<KeyType, vector<ValueType>> groups;
for (auto& item : items) {
    KeyType key = computeKey(item);
    groups[key].push_back(item);
}
```

Problem Set

1. **Group Anagrams** - Group strings that are anagrams
2. **Group Shifted Strings** - Group strings with same shift pattern
3. **Isomorphic Strings** - Check if two strings are isomorphic
4. **Word Pattern** - Check if string follows pattern
5. **Valid Sudoku** - Check valid sudoku using groups

6. **Bulls and Cows** - Count bulls (correct position) and cows
7. **Encode and Decode Strings** - Serialize list of strings
8. **Design HashMap** - Implement HashMap with collision handling
9. **Design HashSet** - Implement HashSet
10. **Design Twitter** - Design simplified Twitter with feeds

Interview Tips

- **Key selection**: Critical part - must uniquely identify group
- **For anagrams**: Sorted string or character count array
- **For patterns**: Encode pattern (a→1, b→2 for "abb" → "122")
- **Hash collisions**: Real interviews rarely ask about this, but know chaining/open addressing

Remember for Interview

✅ Key must be immutable and hashable (string, tuple in Python, careful in C++)
✅ For C++, use string representation of arrays as key
✅ Sorting changes original, make copy if needed
✅ Character count array → convert to string for key
✅ Ask about constraints (all lowercase? special chars?)

Edge Cases

- Empty input list
- Single element groups
- All elements in one group
- All elements in separate groups
- Duplicate elements
- Empty strings in input

PATTERN 6: Index/Position Tracking

Pattern Recognition

- Keywords: "first occurrence", "last seen", "distance between", "duplicate within k"
- Goal: Track where elements appear to solve positional problems
- Time: O(n), Space: O(n)

C++ Template

```cpp
// Track last occurrence
unordered_map<int, int> lastIndex;
for (int i = 0; i < n; i++) {
    if (lastIndex.count(arr[i])) {
        int distance = i - lastIndex[arr[i]];
        // Check distance constraint
    }
    lastIndex[arr[i]] = i;
}

// Track all occurrences
unordered_map<int, vector<int>> allIndices;
for (int i = 0; i < n; i++) {
    allIndices[arr[i]].push_back(i);
}

// Track first occurrence (for prefix problems)
unordered_map<int, int> firstIndex;
for (int i = 0; i < n; i++) {
    if (!firstIndex.count(arr[i])) {
        firstIndex[arr[i]] = i;
    }
}
```

Problem Set

1. **Contains Duplicate II** - Duplicate within distance k
2. **Contains Duplicate III** - Duplicate with value difference within k
3. **First Unique Character in String** - First non-repeating character index
4. **Logger Rate Limiter** - Limit message frequency (time-based)
5. **Design Phone Directory** - Track available phone numbers
6. **Design Hit Counter** - Count hits in last 5 minutes
7. **Max Points on a Line** - Most points on same line
8. **Line Reflection** - Check if points symmetric across vertical line
9. **Number of Boomerangs** - Count boomerang triplets
10. **K-diff Pairs in Array** - Pairs with difference k

Interview Tips

- **Last seen**: Common for duplicate detection problems
- **All indices**: When need to process all occurrences
- **First occurrence**: Often combined with prefix sum pattern
- **Time-based**: Use timestamp as index, clean old entries

Remember for Interview

✅ Map stores: value → index (last/first occurrence)
✅ Map stores: value → list of indices (all occurrences)
✅ Distance check: currentIndex - lastIndex
✅ For duplicates within k: check if distance ≤ k
✅ Clean old entries in time-based problems to save space

Edge Cases

- No duplicates found
- All elements are duplicates
- Duplicate at boundaries (first/last position)
- k = 0 (no distance allowed)
- Single element
- Empty array

PATTERN 7: State Encoding

Pattern Recognition

- Keywords: "visited states", "cycle detection", "path tracking", "configuration"
- Goal: Encode complex states as hashable keys
- Time: O(states), Space: O(states)

C++ Template

```cpp
// Encode coordinates
unordered_set<string> visited;
string encodePos(int x, int y) {
    return to_string(x) + "," + to_string(y);
}
visited.insert(encodePos(x, y));

// Encode multiple values
string encodeState(int x, int y, int dir, int steps) {
    return to_string(x) + "," + to_string(y) + "," +
           to_string(dir) + "," + to_string(steps);
}

// Use pair/tuple for simpler cases (C++11+)
set<pair<int, int>> visited;
visited.insert({x, y});

// Custom hash for pair (unordered_set)
struct PairHash {
    size_t operator()(const pair<int,int>& p) const {
        return hash<int>()(p.first) ^ (hash<int>()(p.second) << 1);
    }
};
unordered_set<pair<int,int>, PairHash> visited;
```

Problem Set

1. **Robot Bounded in Circle** - Check if robot returns to origin
2. **Walking Robot Simulation** - Simulate robot with obstacles
3. **Unique Paths III** - Count paths visiting all cells
4. **Number of Distinct Islands** - Count unique island shapes
5. **Cracking the Safe** - Find shortest string with all combinations
6. **Sliding Puzzle** - Solve sliding puzzle game
7. **Shortest Path to Get All Keys** - Collect all keys in grid
8. **Minimum Knight Moves** - Minimum moves for knight
9. **Serialize and Deserialize Binary Tree** - Convert tree to/from string
10. **Design Tic-Tac-Toe** - Detect win in tic-tac-toe

Interview Tips

- **String encoding**: Simple but slower, works for any state
- **Pair/Tuple**: Faster, limited to 2-3 values
- **Custom hash**: Needed for unordered containers with pairs
- **Coordinate encoding**: "x,y" or "x_y" format common

Remember for Interview

✅ Delimiter choice matters: use comma or underscore (avoid ambiguity: "1,23" vs "12,3")
✅ For grid problems, encoding position is sufficient
✅ For games/robots, encode position + direction + other state
✅ Set for "visited", Map for "state → value" (distance, cost)
✅ Consider using bitmasking for boolean states instead

Edge Cases

- State already visited (cycle detection)
- Invalid state encoding (delimiter ambiguity)
- Hash collision (very rare with good hash function)
- State space too large (consider BFS limit)
- Initial state equals target state

PATTERN 8: Rolling Hash (Rabin-Karp)

Pattern Recognition

- Keywords: "substring matching", "repeated substring", "pattern search"
- Goal: Efficiently hash sliding windows of strings
- Time: O(n), Space: O(n) for storing hashes

C++ Template

```cpp
// Rabin-Karp for pattern matching
const int BASE = 31;
const int MOD = 1e9 + 7;

long long computeHash(string& s) {
    long long hash = 0;
    long long pow = 1;
    for (char c : s) {
        hash = (hash + (c - 'a' + 1) * pow) % MOD;
        pow = (pow * BASE) % MOD;
    }
    return hash;
}

// Rolling hash for window
long long hash = 0, pow = 1;
int m = pattern.length();

// Compute initial window hash
for (int i = 0; i < m; i++) {
    hash = (hash * BASE + text[i]) % MOD;
    if (i < m - 1) pow = (pow * BASE) % MOD;
}

// Slide window
for (int i = m; i < text.length(); i++) {
    // Remove leftmost character
    hash = (hash - text[i - m] * pow % MOD + MOD) % MOD;
    // Add new character
    hash = (hash * BASE + text[i]) % MOD;

    // Compare hash (still verify with actual comparison)
}
```

Problem Set

1. **Implement strStr()** - Find needle in haystack
2. **Repeated DNA Sequences** - Find 10-letter repeated sequences
3. **Longest Duplicate Substring** - Longest repeated substring
4. **Shortest Palindrome** - Shortest palindrome by adding prefix

5. **Longest Happy Prefix** - Longest prefix that is also suffix
6. **Find All Good Strings** - Count strings with specific properties
7. **Distinct Echo Substrings** - Count distinct echo substrings
8. **Maximize Palindrome Length From Subsequences** - Palindrome across two strings
9. **Sum of Scores of Built Strings** - Sum of Z-function values
10. **Number of Substrings With Fixed Ratio** - Count substrings with 0-1 ratio

Interview Tips

- **Always verify**: Hash match doesn't guarantee string match (collision possible)
- **Use prime base**: 31 or 37 works well for strings
- **Large modulo**: Use 10^9+7 or similar prime to reduce collisions
- **Power precomputation**: For multiple queries, precompute BASE^i
- **Polynomial rolling hash**: For comparing multiple patterns

Remember for Interview

✅ Rolling hash is O(1) window update vs O(k) for naive
✅ Hash collision possible - always verify actual strings
✅ Removing leftmost: hash = (hash - char * pow^(m-1)) * BASE
✅ Adding rightmost: hash = hash * BASE + char
✅ Handle negative modulo: (val % MOD + MOD) % MOD

Edge Cases

- Pattern longer than text
- Empty pattern or text
- Pattern equals entire text
- No match found
- Multiple matches at different positions
- Hash collision (must verify actual strings)

PATTERN 9: LRU Cache Design

Pattern Recognition

- Keywords: "least recently used", "cache", "O(1) get/put", "capacity limit"

- Goal: Implement cache with eviction policy
- Time: O(1) for get and put, Space: O(capacity)

C++ Template

```
class LRUCache {
    struct Node {
        int key, value;
        Node *prev, *next;
        Node(int k = 0, int v = 0) : key(k), value(v), prev(nullptr), next(nullptr) {}
    };

    int capacity;
    unordered_map<int, Node*> cache;
    Node *head, *tail;

    void removeNode(Node* node) {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }

    void addToFront(Node* node) {
        node->next = head->next;
        node->prev = head;
        head->next->prev = node;
        head->next = node;
    }

public:
    LRUCache(int cap) : capacity(cap) {
        head = new Node();
        tail = new Node();
        head->next = tail;
        tail->prev = head;
    }

    int get(int key) {
        if (!cache.count(key)) return -1;
        Node* node = cache[key];
        removeNode(node);
        addToFront(node);
        return node->value;
    }

    void put(int key, int value) {
        if (cache.count(key)) {
```

```
            Node* node = cache[key];
            node->value = value;
            removeNode(node);
            addToFront(node);
        } else {
            Node* node = new Node(key, value);
            cache[key] = node;
            addToFront(node);

            if (cache.size() > capacity) {
                Node* lru = tail->prev;
                removeNode(lru);
                cache.erase(lru->key);
                delete lru;
            }
        }
    }
};
```

Problem Set

1. **LRU Cache** - Implement LRU cache
2. **LFU Cache** - Implement LFU (Least Frequently Used) cache
3. **Design Browser History** - Browser back/forward functionality
4. **Design In-Memory File System** - File system with paths
5. **All O`one Data Structure** - Inc/dec with getMax/getMin in O(1)
6. **Design A Leaderboard** - Add/top/reset scores
7. **Time Based Key-Value Store** - Get value at timestamp
8. **Snapshot Array** - Array with snapshot functionality
9. **Design HashMap** - Implement HashMap from scratch
10. **Design HashSet** - Implement HashSet from scratch

Interview Tips

- **HashMap + Doubly Linked List**: Classic combination
- **HashMap**: O(1) access to nodes
- **DLL**: O(1) add/remove, maintains LRU order
- **Dummy nodes**: head/tail sentinels simplify edge cases
- **Update = Remove + AddToFront**: For both get and put

Remember for Interview

✅ HashMap stores key → node pointer
✅ DLL stores actual key-value pairs
✅ Most recent at head, least recent at tail
✅ On get: move to front (recently used)
✅ On put: add to front, evict tail if over capacity
✅ Always remove from current position before adding to front

Edge Cases

- Capacity = 1 (single element cache)
- Get non-existent key
- Put existing key (update value + move to front)
- Put when at capacity (evict LRU)
- Multiple operations on same key

4. Interview Strategy & Tips

4.1 Problem-Solving Framework

Step 1: Identify the Pattern (30 seconds)

```
Ask yourself:
- Do I need to count/track frequency? → Pattern 1
- Finding pairs with condition? → Pattern 2
- Substring/subarray with constraint? → Pattern 3
- Sum-based subarray problem? → Pattern 4
- Grouping by property? → Pattern 5
- Position/index matters? → Pattern 6
- Complex state tracking? → Pattern 7
- String matching/hashing? → Pattern 8
```

Step 2: Choose Data Structure (30 seconds)

```
- Need fast lookup? → unordered_map/set
- Need ordering? → map/set
- Tracking frequency? → unordered_map<T, int>
- Grouping? → unordered_map<Key, vector<Value>>
```

Step 3: Clarify Constraints (1 minute)

```
Always ask:
- Input size? (affects optimization decisions)
- Can elements be negative/zero?
- Duplicates allowed?
- Character set for strings? (26 lowercase → array, else map)
- Return any solution or all solutions?
- What if no solution exists?
```

Step 4: Code Template (3-5 minutes)

```
Start with pattern template, customize for problem
Always structure: initialization → iteration → return
```

Step 5: Test with Edge Cases (1-2 minutes)

```
Before running: mentally trace through edge cases
```

4.2 Time Optimization Tricks

Use Array Instead of Map When Possible

```
// ❌ Slower: O(1) but with hash overhead
unordered_map<char, int> freq;

// ✅ Faster: Direct O(1) array access
int freq[26] = {0};  // For lowercase letters
freq[c - 'a']++;
```

Reserve Capacity for Known Size

```
// ✅ Prevents rehashing
unordered_map<int, int> map;
map.reserve(expectedSize);
```

Use Count Instead of Find for Existence Check

```
// ❌ More typing
if (map.find(key) != map.end())

// ✅ Cleaner
if (map.count(key))
```

Erase Zero Counts to Save Space

```
// ✅ Keep map small
window[s[left]]--;
if (window[s[left]] == 0) {
    window.erase(s[left]);
}
```

4.3 Common Interview Mistakes to Avoid

❌ Forgetting to initialize map[0] for prefix sum problems

```
// Wrong: Missing base case
unordered_map<int, int> prefixCount;
// Correct: Handle subarrays from start
unordered_map<int, int> prefixCount;
prefixCount[0] = 1;
```

❌ Modifying map while iterating

```cpp
// Wrong: ConcurrentModificationException
for (auto& [key, val] : map) {
    if (condition) map.erase(key);
}

// Correct: Collect keys first
vector<int> toDelete;
for (auto& [key, val] : map) {
    if (condition) toDelete.push_back(key);
}
for (int key : toDelete) map.erase(key);
```

❌ Not checking existence before access

```cpp
// Wrong: May create unwanted entries
map[key]++;

// Correct: Check or use count
if (map.count(key)) map[key]++;
else map[key] = 1;
// Or simply: map[key]++ is fine if you want default 0
```

❌ Using map for ordered operations when unordered_map is sufficient

```cpp
// ❌ O(log n) when O(1) is possible
map<int, int> freq;

// ✅ Use unordered unless you need ordering
unordered_map<int, int> freq;
```

4.4 What Interviewers Look For

- ✅ **Pattern Recognition**: Quickly identify which pattern applies
- ✅ **Clean Code**: Use meaningful variable names (even if single letter is okay)
- ✅ **Edge Case Handling**: Proactively mention edge cases
- ✅ **Complexity Analysis**: Explain time/space complexity
- ✅ **Trade-offs**: Discuss alternative approaches
- ✅ **Optimization**: Suggest improvements (array vs map, one-pass vs two-pass)

4.5 How to Explain Your Approach

Template Answer Structure:

```
1. "This is a [PATTERN NAME] problem"
2. "I'll use [DATA STRUCTURE] to [WHY]"
3. "The algorithm is: [HIGH LEVEL STEPS]"
4. "Time complexity: O(n), Space: O(k) because [REASON]"
5. [Code while explaining]
6. "Edge cases to consider: [LIST]"
```

Example:

```
"This is a Frequency Counting problem. I'll use an unordered_map
to count character occurrences in O(1) per character.

The algorithm is:
1. Build frequency map while iterating through string
2. Find first character with frequency 1

Time: O(n) for single pass, Space: O(k) for k unique characters.
Edge cases: empty string, all characters same frequency, no unique character."
```

5. Edge Cases Checklist

5.1 Universal Edge Cases (Test Every Problem)

Input Validation

- ☐ Empty input (array, string, list)
- ☐ Null/nullptr input
- ☐ Single element input
- ☐ Two elements (minimum for pairs)

Boundary Values

- ☐ Minimum possible value (INT_MIN, 0, -∞)
- ☐ Maximum possible value (INT_MAX, ∞)

- [ ] Zero as element or target
- [ ] Negative numbers

Duplicates & Uniqueness

- [ ] All elements are same
- [ ] All elements are distinct
- [ ] Multiple duplicates of same value
- [ ] Using same element twice (i == j check)

Special Cases

- [ ] No solution exists
- [ ] Multiple valid solutions (return any or all?)
- [ ] Solution at boundaries (first/last position)

5.2 Pattern-Specific Edge Cases

Frequency/Counting:

- [ ] All elements have same frequency
- [ ] Only one element has target frequency
- [ ] Frequency is 0 or 1

Complement/Pairs:

- [ ] Pair uses same element twice: arr[i] + arr[i] = target
- [ ] Target sum is 0
- [ ] Multiple pairs with same sum
- [ ] No valid pair exists

Sliding Window:

- [ ] Window size = 0 or 1
- [ ] Window size > array length
- [ ] Pattern longer than text
- [ ] All characters in window are same
- [ ] No valid window exists

Prefix Sum:

- [ ] Sum is 0 at multiple positions

- [ ] Continuous subarray from index 0
- [ ] Negative prefix sums
- [ ] Division by zero (k = 0 for modulo)

Grouping:

- [ ] All items in one group
- [ ] Each item in separate group
- [ ] Empty groups after filtering

Index Tracking:

- [ ] Element at first position
- [ ] Element at last position
- [ ] Distance exactly k vs at most k
- [ ] Never seen before

State Encoding:

- [ ] Start state equals end state
- [ ] Cycle/loop in state transitions
- [ ] Unreachable states
- [ ] State space explosion

Rolling Hash:

- [ ] Hash collision (different strings, same hash)
- [ ] Pattern at start/end of text
- [ ] Pattern not found
- [ ] Entire text matches pattern

5.3 Quick Mental Test Checklist

Before submitting solution, test these in order:

```
1. Empty input → What should return?
2. Single element → Works?
3. All same elements → Handles correctly?
4. No solution case → Returns appropriate value?
5. Solution at boundaries → Found?
```

5.4 Common Edge Case Values to Test

```
// For integers
{} // empty
{0} // single zero
{-1} // single negative
{1, 1, 1} // all same
{INT_MIN, INT_MAX} // extremes

// For strings
"" // empty
"a" // single character
"aaa" // all same
"abc" // all distinct

// For targets/k
k = 0 // zero
k = 1 // minimum
k = n // equals size
k > n // exceeds size
```

6. Must Remember Points

6.1 Core Principles

🔥 Always Initialize map[0] for Prefix Sum

```
unordered_map<int, int> prefixCount;
prefixCount[0] = 1;  // DON'T FORGET THIS
```

🔥 Erase Zero-Count Entries

```
window[s[left]]--;
if (window[s[left]] == 0) {
    window.erase(s[left]);  // Keep map clean
}
```

🔥 Check Before Erasing in Sliding Window

```
while (window.size() > k) {
    // Shrink window
}
```

🔥 Use Array for Limited Character Set

```
// If "only lowercase letters"
int freq[26] = {0};  // NOT unordered_map
```

🔥 Handle Negative Modulo

```
int mod = ((val % k) + k) % k;  // Always positive
```

6.2 Complexity Quick Reference

Pattern	Time	Space	Key Factor
Frequency Count	O(n)	O(k)	k = unique elements
Two Sum	O(n)	O(n)	One pass
Sliding Window	O(n)	O(k)	k = window unique elements
Prefix Sum	O(n)	O(n)	Store all prefix sums
Grouping	O(n·m)	O(n)	m = key computation
Index Track	O(n)	O(n)	Worst: all unique
State Encode	O(states)	O(states)	Depends on state space
Rolling Hash	O(n+m)	O(1)	n = text, m = pattern

6.3 When to Use What

Use unordered_map when:

- Need O(1) average lookup
- Order doesn't matter
- Frequency counting
- Complement finding

Use map when:

- Need sorted order
- Range queries (lower_bound, upper_bound)
- Finding closest elements
- Sliding window maximum with ordering

Use array when:

- Fixed small character set (26 letters)
- Dense integer keys in small range
- Maximum performance needed

Use set instead of map when:

- Only care about existence, not associated value
- Duplicate detection
- Mathematical set operations

6.4 Interview Red Flags (What NOT to Do)

❌ Don't say "I'll sort to solve this" when hash can do O(n)
❌ Don't use nested loops when single pass with hash works
❌ Don't forget to ask about duplicates/edge cases
❌ Don't write code before explaining approach
❌ Don't ignore negative numbers in sum problems
❌ Don't assume small input (always analyze complexity)
❌ Don't use map when array is sufficient

6.5 Final Interview Checklist

Before Writing Code:

- [ ] Identified the pattern correctly
- [ ] Chose appropriate data structure
- [ ] Asked about edge cases
- [ ] Explained approach clearly
- [ ] Discussed time/space complexity

While Writing Code:

- [ ] Used meaningful variable names
- [ ] Added comments for tricky parts
- [ ] Handled edge cases in code
- [ ] Kept code clean and readable

After Writing Code:

- [ ] Traced through with example
- [ ] Tested edge cases mentally
- [ ] Stated final complexity
- [ ] Mentioned possible optimizations

7. Quick Reference Summary

Pattern Selection Flowchart

```
Need to count/frequency? → Pattern 1
Finding pairs with sum? → Pattern 2
Substring/subarray with constraint? → Pattern 3
Subarray sum equals K? → Pattern 4
Group by property? → Pattern 5
Track positions/indices? → Pattern 6
Complex state tracking? → Pattern 7
String matching? → Pattern 8
Design cache with eviction? → Pattern 9
```

Data Structure Decision Tree

```
Need ordering?
  YES → map/set (O(log n))
  NO → unordered_map/set (O(1) avg)

Limited character set?
  YES → int array[26] (fastest)
  NO → unordered_map

Need both key and value?
  YES → map/unordered_map
  NO → set/unordered_set
```

Complexity Cheat Sheet

```
Hash insert/search/delete: O(1) average, O(n) worst
Array access: O(1)
Sorting: O(n log n)
Two pointers: O(n)
```

Most Common Mistakes

1. Forgetting map[0] = 1 in prefix sum
2. Not erasing zero counts in sliding window
3. Modifying map while iterating
4. Using map when array is sufficient
5. Not handling negative modulo
6. Forgetting to check key existence

Remember: In interviews, explaining your thought process and pattern recognition is as important as writing correct code. Practice identifying patterns quickly!