JOB-READY SERIES · C++ EDITION · SHORT

SHORT EDITION

DSA

Data Structures & Algorithms in C++

Start from C++ and STL basics, learn 20 patterns, and crack coding rounds. Every problem in simple steps: brute force, the key trick, a dry run, clean code and Big-O.

solution.cpp

```cpp
class Solution {
public:
    Offer crack(const Problem& p) {
        auto pt = spotPattern(p);   // 20 patterns
        auto t  = pt.keyTrick();    // the one idea
        return code(t);            // O(n), STL-powered
    }
};
```

Two Pointers · Sliding Window · Hashing · Binary Search · Backtracking · Linked Lists · Monotonic Stack · Trees · Heaps · BFS / DFS · Dijkstra · Union-Find · Dynamic Programming · Greedy · Tries · Bit Tricks

WRITTEN BY

Shreyash Tripathi

21 Short Chapters · 100 Must-Do Problems · Answers Included

Edition 2026

JOB-READY SERIES · C++ SHORT EDITION



Data Structures & Algorithms

Pattern-Based Interview Guide in C++, from basics to hard problems

1 Spot the pattern → 2 Find the trick → 3 Dry run → 4 Code → 5 Big-O

WRITTEN BY

Shreyash Tripathi

Short Edition, 2026

© Shreyash Tripathi. All rights reserved.

BEFORE YOU BEGIN

Preface

Coding rounds scare most people for one reason: every problem looks new. It is not. Almost every interview problem is one of about twenty **patterns** wearing a different costume. Once you can see the pattern, the "hard" problem turns into a template you have already practised. This book teaches you to see those patterns.

We start from the very basics of C++ in Chapter 0, so the language never gets in your way. Then each chapter takes one pattern, explains it with a simple analogy, gives you a reusable code template, and solves 4 to 5 classic interview problems the same way every time: the brute force first, then the **one key trick** that unlocks the problem, a step-by-step dry run, clean C++ code, and the time and space complexity. You will also learn what interviewers ask next, and the bugs that cost people offers.

Do not just read the solutions. Cover the code, try the problem for 20 minutes, then compare. That habit, repeated for 30 days, is the whole secret.

Shreyash Tripathi

How Every Problem Is Solved in This Book

1	2	3	4	5
Spot the pattern	**Brute force**	**Key trick**	**Dry run + code**	**Big-O + follow-ups**
Signal words in the problem tell you which chapter it belongs to.	Say the simple answer first and its Big-O.	The one idea that removes the wasted work.	Trace a tiny example, then write clean C++.	State complexity and handle what comes next.

Features of Every Chapter

💡 DO YOU KNOW?

Real facts and history.

😄 FUNNY SIDE

A joke that makes the idea stick.

🔑 KEY POINT

The line to remember.

🎯 INTERVIEW CORNER

Questions interviewers really ask, with model answers.

💼 **AT WORK**

How the concept shows up in a real job.

⚠️ **COMMON MISTAKE**

Traps that cost marks and cause bugs.

✅ **CHECK YOUR PROGRESS**

Short questions with answers after every topic, plus exercises and a full answer key at the end of each chapter.

Your 30-Day Plan

Days	Chapters	Goal
1 to 3	0 to 2	Language basics, problem-solving method, Big-O
4 to 12	3 to 8	Arrays, two pointers, sliding window, hashing, binary search, sorting
13 to 18	9 to 13	Backtracking, linked lists, stacks, trees, heaps
19 to 25	14 to 18	Graphs, dynamic programming, greedy
26 to 30	19 to 20	Tricks, then the 100 must-do problems against the clock

Contents

PART I: FOUNDATIONS

0 C++ and STL Basics for DSA (Start Here) 7
1 How to Approach Any DSA Problem 24
2 Time and Space Complexity 35

PART II: LINEAR PATTERNS

3 Arrays, Strings and Prefix Sums 46
4 Two Pointers 56
5 Sliding Window 66
6 Hashing 76
7 Binary Search 86
8 Sorting and Intervals 96

PART III: RECURSION AND LINEAR STRUCTURES

9 Recursion and Backtracking 108
10 Linked Lists 119
11 Stacks, Queues and Monotonic Stack 129

PART IV: TREES, HEAPS AND GRAPHS

12 Trees and Binary Search Trees 140
13 Heaps and Priority Queues 150
14 Graphs I: BFS, DFS and Topological Sort 160
15 Graphs II: Shortest Paths, Union-Find and MST 172

PART V: DYNAMIC PROGRAMMING AND GREEDY

16 Dynamic Programming I: Foundations 185
17 Dynamic Programming II: Grids, Strings and Subsequences 195
18 Greedy Algorithms 205

PART VI: TRICKS AND PRACTICE

19 Tries, Bit Manipulation and Math Tricks 216
20 Must-Do Problem Bank and Pattern Cheat Sheet 228

APPENDICES

A Pattern Recognition Cheat Sheet 242
B Big-O and C++ STL Cheat Sheet 245
C C++ Code Templates 248
D Glossary 254

PART I

Foundations

Language basics, how to approach any problem, and how to measure an algorithm.

CHAPTER 0

C++ and STL Basics for DSA (Start Here)

"You cannot bat well if you keep looking for the bat. Know your tools before the match starts."

After studying this chapter, you will be able to

- Write a full C++ program and a LeetCode-style `class Solution` without looking anything up.
- Choose between `int` and `long long`, and handle division and modulo with negative numbers.
- Pass vectors by reference and avoid silent, slow copies.
- Use vector, pair, string, unordered_map, map, set, stack, queue, deque and priority_queue fluently.
- Sort with lambda comparators and use lower_bound, upper_bound, accumulate and max_element.
- Read input fast and recognise the most common C++ bugs in coding rounds.

Many developers choose C++ for coding rounds because it is fast and the STL (Standard Template Library) gives ready-made data structures. The catch: unsigned sizes, silent overflow and accidental copies cost marks.

A dabbawala never thinks about how to hold the crate; his attention goes to delivering the right tiffin. This chapter makes C++ syntax your muscle memory, so in the interview your attention goes to the pattern, not to semicolons.

0.1 The Program Skeleton

On LeetCode-style platforms you fill in a method inside `class Solution`. On HackerRank or company online tests you often write a full program with `main`. Know both.

```
#include <bits/stdc++.h>      // pulls in the whole standard library (GCC)
using namespace std;            // lets you write vector instead of std::vector

class Solution {
public:                         // without this, the method is private
    int sumOf(const vector<int>& nums) {
        int total = 0;
        for (int x : nums) total += x;
        return total;
    }
};                              // note the semicolon after a class

int main() {
    Solution s;
    vector<int> v = {1, 2, 3};
    cout << s.sumOf(v) << '\n';   // 6
    return 0;
}
```

0.1.1 Why bits/stdc++.h and using namespace std are fine in contests

- `<bits/stdc++.h>` is a GCC header that includes every standard header. It saves you from forgetting `<algorithm>` or `<queue>`. It slows compilation slightly, which nobody notices in a test.
- `using namespace std;` removes the `std::` prefix everywhere. In a 50-line solution, name clashes are rare and typing speed matters.
- In production code both are bad habits: the header is not portable (MSVC lacks it) and the import can clash with your names. Saying this shows maturity.

> 🔑 **KEY POINT**
>
> Interview code: `#include <bits/stdc++.h>` + `using namespace std;` + `public:` inside the class + `;` after the closing brace of the class. Three of the most common compile errors are just these.

0.2 Types, Ranges and Overflow

Type	Size (typical)	Range	Use for
int	4 bytes	about ±2.1 × 10^9 (INT_MAX = 2,147,483,647)	indexes, counts, normal values
long long	8 bytes	about ±9.2 × 10^18 (LLONG_MAX)	sums, products, answers mod 10^9+7
char	1 byte	-128 to 127 (usually)	characters, small counts
bool	1 byte	true / false	flags, visited arrays
double	8 bytes	about 15 to 16 significant digits	averages, geometry (avoid for exact money)
size_t	8 bytes (64-bit)	0 to about 1.8 × 10^19, unsigned	what `.size()` returns

0.2.1 Overflow

Signed overflow in C++ is **undefined behaviour**: the program may give a wrong number, or the compiler may optimise in surprising ways. It never throws an error for you.

```
int a = 2000000000, b = 2000000000;
int bad = a + b;                  // overflow: undefined behaviour
long long alsoBad = a * b;        // multiplied as int FIRST, then widened
long long good = 1LL * a * b;     // 4e18, fits in long long
long long sum = (long long)a + b; // cast before the operation

const int MOD = 1e9 + 7;
long long x = 123456789;
x = (x * x) % MOD;                // x is long long, product fits
```

> ⚠️ **COMMON MISTAKE**
>
> Using `INT_MAX` as "infinity" and then adding to it, for example `dist[u] + w` in Dijkstra. INT_MAX + 5 overflows. Use `1e9` for int infinity or `LLONG_MAX / 4` for long long.

0.3 Integer Division and Modulo with Negatives

Dividing two integers gives an integer. C++ truncates towards zero, and `%` takes the sign of the left operand.

Expression	Result	Why
7 / 2	3	fraction dropped
-7 / 2	-3	towards zero, not -4
7 % 3	1	normal
-7 % 3	-1	sign of left side
((-7 % 3) + 3) % 3	2	safe positive modulo
7 / 2.0	3.5	one double makes it real division

```
int safeMod(long long x, int m) { return (int)(((x % m) + m) % m); }
bool isOdd(int x) { return x % 2 != 0; }   // x % 2 == 1 fails for -3
int ceilDiv(int a, int b) { return (a + b - 1) / b; }  // for positive a, b
```

CHECK YOUR PROGRESS 0.3

1. **What does `long long r = a * b;` give when a and b are ints near 10^5?**
   Answer: The product (about 10^10) is computed in int and overflows before it is widened. Write `1LL * a * b`.
2. **How do you move index i one step left in a circular array of size n without going negative?**
   Answer: `(i - 1 + n) % n`. Adding n first keeps the left operand non-negative.

0.4 Loops, Functions and References

```
for (int i = 0; i < n; i++) { }            // classic loop
for (int i = n - 1; i ⩾ 0; i--) { }        // backwards (int, not size_t!)
for (int x : v) { }                         // range-for, x is a COPY
for (int& x : v) x *= 2;                    // reference: changes v
for (const string& s : words) { }           // read-only, no copy
while (lo ⩽ hi) { }
```

0.4.1 Pass by value vs pass by reference

By default C++ **copies** every argument. Passing a vector of 10^5 elements by value copies 10^5 elements on every call. Inside a recursive DFS, that turns O(n) into $O(n^2)$ and can also blow the memory limit.

```
void byValue(vector<int> v)        { v[0] = 99; }  // caller's vector unchanged, O(n) copy
void byRef(vector<int>& v)         { v[0] = 99; }  // caller sees 99, no copy
int readOnly(const vector<int>& v) { return v[0]; } // no copy, cannot modify

void swapTwo(int& a, int& b) { int t = a; a = b; b = t; }  // works because of &
```

KEY POINT

Default for containers and strings: `const T&`. Use plain `T&` when the helper must change it. Plain ints and chars are cheap, so pass them by value.

To "return" two values, return a `pair<int,int>` and unpack it with structured bindings: `auto [lo, hi] = minMax(v);` (C++17).

0.5 Arrays vs vector

```
int arr[5] = {0};                       // fixed size, all 0; size cannot change
int big[100005];                         // large arrays: declare globally (stack is small)

vector<int> v(5, 0);                     // size 5, all 0
vector<int> w = {5, 2, 9};
w.push_back(7);                          // amortised O(1)
w.pop_back();                            // remove last
int n = w.size();                        // store in int to avoid unsigned surprises
vector<vector<int>> grid(rows, vector<int>(cols, 0));   // 2D grid
vector<bool> seen(n, false);
v.assign(n, -1);                         // reset all to -1
fill(v.begin(), v.end(), -1);            // same idea
```

Prefer `vector`: it knows its size and can grow. Raw arrays are fine for small fixed tables like `int cnt[26]`.

> ⚠ **COMMON MISTAKE**
>
> `v[i]` does not check bounds. Reading `v[n]` gives garbage or a crash, not a clear error. Use `v.at(i)` while debugging; it throws `out_of_range`. Also, a local `int a[1000000]` inside a function can overflow the stack; make it global or use a vector.

0.6 string Basics

C++ `string` is **mutable**, unlike Java and Python. You can change a character in place and `+=` a char in amortised O(1).

```
string s = "hello";
char c = s[1];                  // 'e'
s[0] = 'H';                     // "Hello", allowed
s += '!';                       // append a char, amortised O(1)
s += " world";                  // append a string
string sub = s.substr(1, 3);    // "ell": (start, LENGTH), not (start, end)
size_t pos = s.find("lo");      // index, or string::npos if absent
reverse(s.begin(), s.end());
sort(s.begin(), s.end());       // sorts characters

int d = '7' - '0';              // 7
int idx = 'c' - 'a';            // 2, index into int cnt[26]
char nx = 'a' + 3;              // 'd'
string t = to_string(42);       // "42"
int k = stoi("123");            // 123; stoll for long long
bool same = (s == t);           // == compares content in C++
```

⚠️ **COMMON MISTAKE**

`s = s + c;` inside a loop builds a brand new string each time, so it is $O(n^2)$ overall. Use `s += c;` or `s.push_back(c);`. Also remember: `substr` takes a length, so `s.substr(i, j - i)` gives indices i to j-1.

😄 **FUNNY SIDE**

A Java developer writes `s.substr(2, 5)` expecting characters 2 to 4 and gets five. A Python developer writes `s[-1]` and gets a crash. The compiler stays silent, like a strict principal who saves it all for the parent-teacher meeting.

0.7 STL Containers

Need	C++ container	Key operations	Time
Dynamic array	vector	push_back, pop_back, [i], size, back	O(1) (push amortised)
Two values together	pair	first, second, make_pair, {a, b}	O(1)
Key → value, fast	unordered_map	[k], count, find, erase	O(1) average, O(n) worst
Key → value, sorted	map	[k], count, lower_bound, begin, rbegin	O(log n)
Unique items, fast	unordered_set	insert, count, erase	O(1) average
Unique items, sorted	set (multiset allows duplicates)	insert, erase, lower_bound, begin	O(log n)
LIFO	stack	push, pop, top, empty	O(1)
FIFO	queue	push, pop, front, empty	O(1)
Both ends	deque	push_front/back, pop_front/back, front, back	O(1)
Max-heap	priority_queue<int>	push, pop, top	push/pop O(log n), top O(1)
Min-heap	priority_queue<int, vector<int>, greater<int>>	same	O(log n)

0.7.1 pair and unordered_map

```
pair<int, int> p = {3, 7};
cout << p.first << " " << p.second;       // 3 7
vector<pair<int,int>> edges = {{1, 2}, {2, 3}};

unordered_map<char, int> freq;
for (char c : string("banana")) freq[c]++;  // missing key starts at 0
if (freq.count('z')) { }                      // check without inserting
auto it = freq.find('a');
if (it ≠ freq.end()) cout << it→second;     // 3
for (auto& [ch, cnt] : freq) { }              // iterate (order not defined)
freq.erase('b');
```

⚠️ **COMMON MISTAKE**

`if (freq[x] > 0)` **inserts** x with value 0 when x is missing. In a loop this grows the map and can break logic that uses `freq.size()`, as in sliding window "distinct count" problems. Use `count` or `find` to check, and `erase` keys whose count drops to 0.

0.7.2 map, set and multiset

```
map<int, string> m;                      // keys kept sorted (red-black tree)
m[30] = "c"; m[10] = "a"; m[20] = "b";
for (auto& [k, val] : m) cout << k;       // 10 20 30
int smallest = m.begin()→first;           // 10
int largest  = m.rbegin()→first;          // 30

set<int> st = {5, 1, 9};
st.insert(5);                             // ignored, already present
auto it = st.lower_bound(4);              // first element ⩾ 4, here 5
if (it ≠ st.end()) cout << *it;
// largest element ⩽ x (floor):
auto f = st.upper_bound(6);               // first > 6, here 9
if (f ≠ st.begin()) cout << *prev(f);     // 5

multiset<int> ms = {2, 2, 3};
ms.erase(ms.find(2));                     // removes ONE 2
// ms.erase(2) would remove ALL 2s
```

🎯 **INTERVIEW CORNER**

Q: unordered_map vs map: when would you pick each?
A: unordered_map (hash table) gives O(1) average lookups when order does not matter. map (balanced tree) gives O(log n) but keeps keys sorted and supports lower_bound, smallest and largest key. Pick map only when you need order.

Q: Is unordered_map always O(1)?
A: On average. With many collisions it degrades to O(n) per operation. Contest hackers sometimes craft inputs against the default hash; a custom hash or a map avoids that.

0.7.3 stack, queue and deque

```
stack<int> stk;
stk.push(1); stk.push(2);
int top = stk.top(); stk.pop();          // pop() returns void! read top() first

queue<pair<int,int>> q;                  // BFS on grid cells
q.push({0, 0});
auto [r, c] = q.front(); q.pop();

deque<int> dq;
dq.push_back(5); dq.push_front(1);       // 1 5
dq.pop_back(); int f = dq.front();       // 1
```

0.7.4 priority_queue: max and min

```
priority_queue<int> maxHeap;                          // largest on top (default)
priority_queue<int, vector<int>, greater<int>> minHeap; // smallest on top
maxHeap.push(3); maxHeap.push(8); maxHeap.top();      // 8
minHeap.push(3); minHeap.push(8); minHeap.top();      // 3

// heap of {distance, node}, smallest distance first (Dijkstra)
priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;
pq.push({0, 1});
auto [d, u] = pq.top(); pq.pop();
```

> 🔑 **KEY POINT**
>
> C++ priority_queue is a **max-heap** by default, the opposite of Java and Python. For a min-heap add `vector<T>, greater<T>`. A quick trick for ints: push `-x` into a max-heap and negate when you pop.

0.7.5 lower_bound and upper_bound on sorted vectors

```
vector<int> a = {1, 3, 3, 5, 8};                    // must be sorted
int i = lower_bound(a.begin(), a.end(), 3) - a.begin();  // 1: first ≥ 3
int j = upper_bound(a.begin(), a.end(), 3) - a.begin();  // 3: first > 3
int countOf3 = j - i;                                    // 2
bool has = binary_search(a.begin(), a.end(), 5);         // true
int k = lower_bound(a.begin(), a.end(), 10) - a.begin(); // 5 == a.size(): not found
```

Subtracting `a.begin()` turns an iterator into an index. On a `set` or `map`, call the member `st.lower_bound(x)`; the free function `lower_bound(st.begin(), st.end(), x)` compiles but runs in O(n) because set iterators cannot jump.

CHECK YOUR PROGRESS 0.7

1. **How do you declare a min-heap of long long?**
   Answer: `priority_queue<long long, vector<long long>, greater<long long>> pq;`
2. **In a sorted vector, how do you count elements strictly less than x?**
   Answer: `lower_bound(a.begin(), a.end(), x) - a.begin()`, the index of the first element >= x, which equals the number of elements before it.

0.8 Algorithms and Iterators

0.8.1 Iterators in one minute

An iterator is a "pointer-like" position in a container. `v.begin()` points to the first element, `v.end()` points **one past** the last (never dereference it). `*it` reads the value, `it++` moves forward, and for maps `it->first` and `it->second` read key and value. Almost every STL algorithm takes a range `[begin, end)`.

0.8.2 The everyday algorithms

```
vector<int> v = {5, 2, 9, 1};
sort(v.begin(), v.end());                          // ascending, O(n log n)
sort(v.begin(), v.end(), greater<int>());          // descending
reverse(v.begin(), v.end());
int mx = *max_element(v.begin(), v.end());         // 9 (v must not be empty)
int mn = *min_element(v.begin(), v.end());         // 1
int pos = max_element(v.begin(), v.end()) - v.begin();  // index of max
long long total = accumulate(v.begin(), v.end(), 0LL);  // 17, 0LL = long long sum
int c = count(v.begin(), v.end(), 2);              // occurrences of 2
v.erase(unique(v.begin(), v.end()), v.end());      // dedupe AFTER sorting
swap(v[0], v[1]);
int big = max({3, 7, 5});                          // 7, max of a list
```

0.8.3 sort with a lambda comparator

```
vector<vector<int>> intervals = {{5, 6}, {1, 3}, {2, 4}};
sort(intervals.begin(), intervals.end(),
     [](const vector<int>& x, const vector<int>& y) { return x[0] < y[0]; });

vector<string> words = {"pear", "fig", "apple"};
sort(words.begin(), words.end(), [](const string& a, const string& b) {
    if (a.size() != b.size()) return a.size() < b.size();  // shorter first
    return a < b;                                          // then alphabetical
});
// pairs sort by first, then second, automatically
vector<pair<int,int>> ps = {{2, 1}, {1, 5}, {1, 2}};
sort(ps.begin(), ps.end());                        // {1,2} {1,5} {2,1}
```

⚠️ **COMMON MISTAKE**

A comparator means "a comes strictly before b", so it must return **false** for equal elements. `return a <= b;` breaks this rule and can crash `sort`. Take parameters by `const&` to avoid copies.

💡 **DO YOU KNOW?**

The STL was designed mainly by Alexander Stepanov, and it was accepted into the C++ standard in the mid-1990s. Since C++11, the standard requires `std::sort` to be O(n log n) even in the worst case; GCC's library meets this with introsort, a quicksort that switches to heapsort when recursion gets too deep.

0.9 Recursion

A recursive function solves a problem by calling itself on a smaller input. Every recursion needs a **base case** (smallest input, answered directly) and a **recursive step** that moves towards it.

```
long long fact(int n) {
    if (n <= 1) return 1;              // base case
    return n * fact(n - 1);            // smaller input
}

void dfs(int u, const vector<vector<int>>& adj, vector<bool>& seen) {
    seen[u] = true;                    // pass by reference: shared state
    for (int v : adj[u]) if (!seen[v]) dfs(v, adj, seen);
}

// lambda recursion inside class Solution (C++17 pattern)
function<int(int)> fib = [&](int n) { return n < 2 ? n : fib(n - 1) + fib(n - 2); };
```

Each call uses stack memory (often 1 to 8 MB in total), so recursion 10^5 or more levels deep, like DFS on a long chain, can crash with a segmentation fault. Use an explicit stack or BFS then.

🔑 **KEY POINT**

Before writing recursion, answer two questions aloud: "What does this function return for input n?" and "What is the smallest input?" If you can answer both, the code almost writes itself.

0.10 Fast I/O and Common Bugs

0.10.1 Fast input and output

```
int main() {
    ios::sync_with_stdio(false);   // stop syncing with C stdio: much faster cin/cout
    cin.tie(nullptr);              // do not flush cout before every cin
    int n; cin >> n;
    vector<long long> a(n);
    for (auto& x : a) cin >> x;
    string line; getline(cin >> ws, line);   // read a full line, skip leftover newline
    cout << accumulate(a.begin(), a.end(), 0LL) << '\n';   // '\n', not endl
}
```

After `sync_with_stdio(false)`, do not mix `scanf/printf` with `cin/cout`. Use `'\n'` instead of `endl`: `endl` flushes the buffer every time, which is slow for 10^5 lines.

0.10.2 The bugs that cost the most marks

Bug	Example	Fix
Unsigned size underflow	`for (int i = 0; i < v.size() - 1; i++)` with empty v loops about 1.8 × 10^19 times	`int n = v.size();` then use n
Overflow	`int sum` of 10^5 values up to 10^9	`long long`, `1LL *`
Accidental copy	`void dfs(vector<int> v)`	`const vector<int>&`
map[] inserts	`if (m[k])` on a missing key	`m.count(k)`
Empty container access	`st.top()`, `q.front()`, `*max_element` on empty	check `empty()` first
Iterator invalidation	erasing from a vector or map inside a range-for	`it = m.erase(it)` in an iterator loop
Uninitialised local	`int cnt;` then `cnt++`	always `int cnt = 0;`
Global not reset	global arrays reused across test cases	clear or `assign` per test

🎯 **INTERVIEW CORNER**

Q: Why is `v.size() - 1` dangerous?
A: `size()` returns an unsigned `size_t`. For an empty vector, 0 − 1 wraps to about 1.8 × 10^19 instead of -1, so loops run wild. Convert to int first.

Q: What does `ios::sync_with_stdio(false)` do?
A: It unties C++ streams from C stdio, letting cin and cout use their own buffers, which is much faster for large input. The cost: you must not mix printf/scanf with cout/cin afterwards.

0.11 Ten Warm-up Problems

Type each one without looking.

0.11.1 Reverse an Array In Place (Easy)

```
void reverseArr(vector<int>& a) {
    int l = 0, r = (int)a.size() - 1;
    while (l < r) swap(a[l++], a[r--]);   // two pointers meet in the middle
}
```

O(n) time, O(1) space. `reverse(a.begin(), a.end())` does the same.

0.11.2 Maximum in an Array (Easy)

```
int findMax(const vector<int>& a) {
    int best = a[0];                      // not 0: fails for all-negative input
    for (int x : a) best = max(best, x);
    return best;
}
```

0.11.3 Count Vowels (Easy)

```
int countVowels(const string& s) {
    string vowels = "aeiouAEIOU";
    int cnt = 0;
    for (char c : s) if (vowels.find(c) != string::npos) cnt++;
    return cnt;
}
```

0.11.4 Palindrome Check (Easy)

```
bool isPalindrome(const string& s) {
    int l = 0, r = (int)s.size() - 1;
    while (l < r) if (s[l++] != s[r--]) return false;
    return true;                          // "" and "a" are palindromes
}
```

0.11.5 First Unique Character (Easy)

```
int firstUniqChar(const string& s) {
    int cnt[26] = {0};
    for (char c : s) cnt[c - 'a']++;
    for (int i = 0; i < (int)s.size(); i++)
        if (cnt[s[i] - 'a'] == 1) return i;
    return -1;                            // "leetcode" → 0, "aabb" → -1
}
```

0.11.6 Contains Duplicate (Easy)

```
bool containsDuplicate(const vector<int>& nums) {
    unordered_set<int> seen;
    for (int x : nums) {
        if (seen.count(x)) return true;   // seen before
        seen.insert(x);
    }
    return false;
}
```

0.11.7 Two Sum (Easy)

```
vector<int> twoSum(const vector<int>& nums, int target) {
    unordered_map<int, int> indexOf;      // value → index
    for (int i = 0; i < (int)nums.size(); i++) {
        auto it = indexOf.find(target - nums[i]);
        if (it ≠ indexOf.end()) return {it→second, i};
        indexOf[nums[i]] = i;             // store AFTER checking
    }
    return {-1, -1};
}
```

[2,7,11,15], 9 → [0,1]. [3,3], 6 → [0,1]. O(n) time and space.

0.11.8 Valid Parentheses (Easy)

```
bool isValid(const string& s) {
    stack<char> st;
    for (char c : s) {
        if (c == '(' || c == '[' || c == '{') { st.push(c); continue; }
        if (st.empty()) return false;
        char o = st.top(); st.pop();
        if ((c == ')' && o ≠ '(') || (c == ']' && o ≠ '[') || (c == '}' && o ≠ '{'))
            return false;
    }
    return st.empty();                    // "((" must fail
}
```

0.11.9 Kth Largest Element (Medium)

```
int findKthLargest(const vector<int>& nums, int k) {
    priority_queue<int, vector<int>, greater<int>> pq;   // min-heap of size k
    for (int x : nums) {
        pq.push(x);
        if ((int)pq.size() > k) pq.pop();                  // drop the smallest
    }
    return pq.top();                      // [3,2,1,5,6,4], k=2 → 5
}
```

O(n log k) time, O(k) space.

0.11.10 Count of Prices at Most X (Easy)

Given sorted prices and many queries x, return how many prices are ≤ x.

```
int countAtMost(const vector<int>& sortedPrices, int x) {
    return upper_bound(sortedPrices.begin(), sortedPrices.end(), x) - sortedPrices.begin();
}   // {10,20,20,50}, x=20 → 3; x=5 → 0
```

O(log n) per query instead of O(n).

What Have We Learnt?

- Interview skeleton: `bits/stdc++.h`, `using namespace std`, `public:` and a semicolon after the class.
- int holds about ±2.1 × 10^9; use long long and `1LL *` for big sums and products.
- Division truncates towards zero and `%` keeps the left sign; use `((x % m) + m) % m`.
- C++ copies arguments; pass containers as `const T&` or `T&`.
- Prefer vector to raw arrays; store `size()` in an int before subtracting.
- string is mutable; use `+=`, and remember `substr(start, length)`.
- unordered_map/set for O(1) average; map/set for order and lower_bound; `m[k]` inserts.
- priority_queue is a max-heap by default; add `greater<T>` for a min-heap.
- sort with a strict lambda comparator; accumulate with `0LL`; lower/upper_bound on sorted ranges.
- Recursion needs a base case; very deep recursion can overflow the stack.
- Fast I/O: `ios::sync_with_stdio(false); cin.tie(nullptr);` and `'\n'`.

Key terms: bits/stdc++.h, namespace std, class Solution, long long, undefined behaviour, truncation, pass by reference, const reference, vector, size_t, string, pair, unordered_map, map, set, multiset, stack, queue, deque, priority_queue, greater, iterator, lower_bound, upper_bound, lambda comparator, accumulate, base case, fast I/O.

✅ CHECK YOUR PROGRESS 0

1. **Why is `void dfs(int u, vector<vector<int>> adj)` a performance bug?**
   Answer: The whole adjacency list is copied on every call, so a DFS over n nodes copies the graph n times. Use `const vector<vector<int>>& adj` .
2. **What is `-9 / 2` and `-9 % 2` in C++?**
   Answer: -4 and -1. Division truncates towards zero and the remainder takes the sign of the dividend.
3. **How do you get the largest key in a map?**
   Answer: `m.rbegin()→first` (check the map is not empty).
4. **Why must you read `st.top()` before calling `st.pop()` ?**
   Answer: `pop()` returns void in C++; it only removes the element.
5. **What is wrong with `accumulate(v.begin(), v.end(), 0)` for values up to 10^9?**
   Answer: The initial value 0 is an int, so the sum is computed in int and overflows. Use `0LL` .
6. **How do you remove just one copy of 2 from a multiset?**
   Answer: `ms.erase(ms.find(2));` (after checking find did not return end). `ms.erase(2)` removes all copies.

Quick Exercises

A. Multiple Choice Questions

1. What does `priority_queue<int> pq` return from `top()`?
   a. The smallest element
   b. The largest element
   c. The first inserted
   d. The last inserted

2. `string("abcdef").substr(1, 3)` is
   a. "bc"
   b. "bcd"
   c. "abc"
   d. "bcde"

3. Which check does NOT insert a missing key into an unordered_map m?
   a. m[k] == 0
   b. m[k]++
   c. m.count(k)
   d. if (m[k])

4. For an empty vector v, `v.size() - 1` is
   a. -1
   b. 0
   c. a huge positive number
   d. a compile error

5. Which comparator is valid for std::sort?
   a. return a <= b;
   b. return a < b;
   c. return a == b;
   d. return true;

B. Practice Problems

1. Valid Anagram: are two strings anagrams? Hint: one `int cnt[26]`, add for s, subtract for t.
2. Top K Frequent Elements. Hint: count with unordered_map, then keep a min-heap of {count, value} of size k.
3. Remove Duplicates and Print Sorted Unique Values. Hint: sort, then `erase(unique(...), end)`, or insert into a set.

Answer Key: Chapter 0

A. MCQ

1. (b) Default priority_queue is a max-heap.
2. (b) Start at 1, length 3: "bcd".
3. (c) count only looks; operator[] inserts a default value.
4. (c) size_t is unsigned, so 0 - 1 wraps to about 1.8 × 10^19.
5. (b) A comparator must be a strict "less than" and return false for equal items.

B. Practice Problems

1. If sizes differ, return false. Increment cnt for s, decrement for t, then check all 26 are zero. O(n) time, O(1) space.
2. Count frequencies in O(n). Push {count, value} into a min-heap with `greater<pair<int,int>>`; pop when size exceeds k. Remaining k values are the answer. O(n log k) time, O(n) space.
3. Sort O(n log n), then `v.erase(unique(v.begin(), v.end()), v.end())` keeps one copy of each value in order. A set gives the same result, also O(n log n), with O(n) extra space.

CHAPTER 1

How to Approach Any DSA Problem

"The interviewer is not checking if you have seen the problem. They are checking if you can think when you have not."

After studying this chapter, you will be able to

- Follow a seven-step framework for any coding interview problem.
- Read input constraints and guess the time complexity the interviewer expects.
- Map common problem signals to the right pattern and chapter of this book.
- Dry run code on paper using a step table, and talk clearly while you code.
- Pick the right C++ STL container quickly and avoid integer overflow.

Think of a chai stall owner at a busy railway station. Twenty people shout orders at once. A new helper panics and makes cups one by one in the order he heard them. The owner first listens, groups the orders (four cutting, two masala), checks how much milk he has, then makes everything in one go.

Most candidates who fail coding rounds do not fail because they do not know arrays or hash maps. They jump into code without a plan, get stuck, and go silent. This chapter gives you the owner's method: fixed steps, constraint reading, a pattern map and a C++ quick reference.

1.1 The Interview Problem-Solving Framework



Figure 1.1: The seven-step framework

Step	What you do
1. Understand	Restate the problem in your words. Ask short clarifying questions: empty input? negatives, zeros, duplicates? sorted? can I modify the input? what to return if no answer? how large are n and the values (int or long long)?
2. Examples	Solve the given example by hand, then one small example of your own and one edge case. If you cannot solve it by hand, you cannot code it.
3. Brute force	Say the simplest correct idea with its complexity: "Try every pair, $O(n^2)$ time, $O(1)$ space." Do not code it unless asked.

Step	What you do
4. Optimise	Ask what work is repeated. Use BUD: **B**ottlenecks, **U**nnecessary work, **D**uplicated work.
5. Code	Clean C++ with clear names (`left` , `right` , `seen`) and small helpers.
6. Test	Dry run your code (not your idea) on the example, then empty, one element, all equal, negatives, very large values.
7. Analyse	State time and space and the trade-off: "O(n) extra space brings time from $O(n^2)$ to O(n)."

Common tools that remove repeated work in step 4:

- **Hashing**: replace "search for x" (O(n)) with an unordered_map lookup (O(1) average).
- **Sorting**: once sorted, two pointers or binary search often apply.
- **Precomputation**: prefix sums, prefix max, frequency counts.
- **Sliding window**: reuse the previous window instead of recomputing.
- **Better structure**: heap for "top k", stack for "next greater", deque for "window max".
- **Caching recursion**: memoisation turns exponential recursion into DP.

1.1.1 The framework applied: Two Sum

Given `nums = [2, 7, 11, 15]` and `target = 9` , return indices of the two numbers that add up to target (LeetCode 1).

Step	What you say or do
Understand	"Exactly one answer, I cannot use the same element twice, return indices."
Examples	[2,7,11,15], 9 → [0,1]. Own case: [3,3], 6 → [0,1] (duplicates).
Brute force	Check every pair i < j. $O(n^2)$ time, O(1) space.
Optimise	For each x, I search for target - x. A map of value → index makes that O(1).
Test	[3,3]: i=0 stores 3→0; i=1 finds 3 in the map, returns [0,1].
Analyse	O(n) time, O(n) space.

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> indexOf;               // value → index
        for (int i = 0; i < (int)nums.size(); i++) {
            auto it = indexOf.find(target - nums[i]);
            if (it != indexOf.end()) return {it->second, i};  // partner seen earlier
            indexOf[nums[i]] = i;                      // store AFTER the check
        }
        return {-1, -1};                               // no pair found
    }
};
```

🔑 **KEY POINT**

Brute force first, out loud, with its complexity. It shows understanding, and the optimised solution almost always comes from staring at what the brute force repeats.

⚠️ **COMMON MISTAKE**

Coding before agreeing on the approach. If you code a wrong idea for 15 minutes, there is no time left to fix it. Ask "Shall I code this approach?" and wait for a nod.

✅ **CHECK YOUR PROGRESS 1.1**

1. **Why state the brute force even when you know a better solution?**

   Answer: It proves you understand the problem, gives a correct fallback and a baseline complexity to improve on. Interviewers often give partial credit for it.

2. **Name three clarifying questions that can change an array solution.**

   Answer: Is it sorted? Can it contain negatives or duplicates? Can I modify it? (Also: size of n, empty input behaviour.)

1.2 Reading Constraints to Guess the Target Complexity

Constraints like `1 ≤ n ≤ 10^5` are a hint. A typical judge runs roughly 10^8 simple operations per second (a rule of thumb). Pick a complexity that keeps the operation count near or below that.

Input size n	Target complexity	Typical technique
n ≤ 10	O(n!) or O(n · n!)	All permutations, backtracking
n ≤ 20	O(2^n) or O(n · 2^n)	Subsets, bitmask, backtracking
n ≤ 500	$O(n^3)$	Triple loops, interval DP, Floyd-Warshall
n ≤ 5,000	$O(n^2)$	Double loops, 2D DP (LCS, edit distance)
n ≤ 10^6	O(n log n) or O(n)	Sorting, heap, two pointers, hashing, sliding window

Input size n	Target complexity	Typical technique
n ≥ 10^9	O(log n) or O(1)	Binary search, math, fast exponentiation

1.2.1 Worked examples

1. **n ≤ 10^5**: n^2 = 10^10, far too slow. n log n ≈ 1.7 × 10^6, very fast. Aim for O(n log n) or O(n).
2. **n ≤ 20**: 2^20 ≈ 10^6. Trying every subset is fine.
3. **n ≤ 10^9** (e.g. integer square root): even O(n) is too slow. Binary search needs about 30 steps.
4. **m, n ≤ 1000**: O(m · n) = 10^6, fine. This hints at a 2D DP grid or grid BFS.

Value constraints matter too. If values reach 10^9 and you add two of them, 2 × 10^9 overflows an `int` (max about 2.147 × 10^9); use `long long`. If values are small (0 to 100), a counting array of size 101 can replace a hash map or sorting.

> 🔑 **KEY POINT**
>
> Small n is a hint, not a gift. n ≤ 20 means exponential search is intended. A huge n means logarithmic thinking. Read the constraints before you design.

> ✅ **CHECK YOUR PROGRESS 1.2**
>
> 1. **n ≤ 2 × 10^5. Is $O(n^2)$ acceptable?**
>
>   **Answer:** No. 4 × 10^10 operations is far above about 10^8 per second. Aim for O(n log n) or O(n).
>
> 2. **What does n ≤ 15 suggest?**
>
>   **Answer:** Exponential is allowed: 2^15 = 32,768. Think subsets, bitmask DP or backtracking.

1.3 Pattern Recognition: The Map of This Book

There are thousands of problems but only about 15 to 20 core patterns. Read the problem, notice the signal, pick the tool.

Problem signal	Likely pattern	Chapter
"Subarray sum", "range sum", many sum queries	Prefix sum (+ hash map)	3
Sorted array, "pair with sum", "in place", palindrome	Two pointers	4
"Longest/shortest substring or subarray with condition"	Sliding window	5
"Count frequency", "seen before", "group by"	Hashing	6
Sorted input, "minimum value that works", huge n	Binary search (on answer)	7
"All combinations / permutations / subsets"	Backtracking	9
"Reverse list", "cycle", "middle node"	Linked list, fast and slow pointers	10

Problem signal	Likely pattern	Chapter
"Next greater", "valid brackets", "previous smaller"	Stack / monotonic stack	11
Tree, "depth", "path", "level by level"	DFS / BFS on trees	12
"Top k", "k-th largest", "merge k sorted"	Heap (priority_queue)	13
Grid, "islands", "shortest path", dependencies	Graph BFS / DFS / topological sort	14, 15
"Connected groups", "redundant connection"	Union-Find	15
"Number of ways", "min cost", "max profit" with choices	Dynamic programming	16, 17
"Intervals", "meetings", "overlap"	Sort + greedy / merge intervals	18

Table 1.1: Signal to pattern map

1.3.1 How to use the map

1. Underline signal words: "contiguous", "sorted", "k", "minimum", "all".
2. Check the output type. "All solutions" almost always means backtracking.
3. Confirm with the constraint table that the pattern's complexity fits.
4. If two patterns fit, start with the simpler one.

😄 **FUNNY SIDE**

Every DSA learner goes through three stages. Stage 1: "Every problem is new and scary." Stage 2: "Every problem is a hash map." Stage 3: "Oh, this is just sliding window wearing a moustache." This book is the shortcut to Stage 3.

💡 **DO YOU KNOW?**

LeetCode has more than 3,000 problems, yet popular curated lists such as "Blind 75" and "NeetCode 150" cover the core interview patterns with about 75 to 150 problems. Breadth of patterns beats raw count.

✅ **CHECK YOUR PROGRESS 1.3**

1. **"Length of the longest substring without repeating characters": which pattern?**
   Answer: Sliding window with a set or a last-seen-index map. The signal is "longest substring with a condition".
2. **"All possible subsets": which pattern and complexity?**
   Answer: Backtracking. There are 2^n subsets, so O(n · 2^n) is unavoidable; n will usually be ≤ 20.

1.4 How to Dry Run

A dry run means executing your code by hand, line by line, tracking every variable in a table. Compilers catch syntax; dry runs catch logic.

1. Pick a small input (3 to 6 elements) that touches the interesting cases.
2. Draw a table: one column per variable, one row per iteration.
3. Follow the **code you wrote**, not the idea in your head.
4. Check the first and last iterations, then repeat with an edge case.

```
int findMax(const vector<int>& a) {
    int best = 0;                          // suspicious start value
    for (int x : a) if (x > best) best = x;
    return best;
}
```

x	x > best?	best
start	-	0
-5	no	0
-2	no	0
-9	no	0

For `a = [-5, -2, -9]` the answer should be -2, but the code returns 0. Fix: start with `best = a[0]` or `INT_MIN`. A test with only positives would never catch this, so choose test inputs on purpose.

> ⚠️ **COMMON MISTAKE**
>
> Dry running only the happy path. Most bugs live at the edges: index 0, index n-1, empty input, negatives, duplicates and integer limits. In C++ also check `v.size() - 1` on an empty vector (unsigned wrap-around).

> ✅ **CHECK YOUR PROGRESS 1.4**
>
> 1. **Why dry run the code you wrote, not your idea?**
>
>   **Answer:** Bugs live in the gap between idea and code: wrong boundary, wrong initial value, wrong statement order. Tracing the idea skips them.
>
> 2. **Give three edge cases for an array problem.**
>
>   **Answer:** Empty array, single element, all elements equal (also negatives and very large values).

1.5 How to Talk While Coding

The interviewer can only grade what they hear. Five minutes of silence looks the same as being stuck.

Stage	Example lines
Clarifying	"Can the array be empty? Should I return -1 then?"
Brute force	"Trying all pairs is $O(n^2)$. Let me see if we can do better."
Optimising	"The repeated work is searching for the complement. A hash map makes that O(1)."
Stuck	"I'm unsure how to handle duplicates. Let me try a small example."
Wrapping up	"Time O(n), space O(n). Sorting plus two pointers saves memory but loses the original indices."

A hint is not a failure. Say "thank you", restate it in your words and use it. Arguing with a hint is a strong negative signal.

> 🎯 **INTERVIEW CORNER**
>
> **Q:** What do you do if you are completely stuck?
> **A:** Say so calmly, solve a small example by hand, and name the patterns you are considering ("Could this be a sliding window? The condition isn't monotonic, so maybe not."). Thinking aloud often triggers a useful hint.
>
> **Q:** How long should I spend before coding?
> **A:** In a 45-minute round, roughly 5 to 10 minutes discussing, 15 to 20 minutes coding, and the rest testing and on follow-ups.

1.6 C++ STL Quick Reference

Chapter 0 covers the C++ basics and every container in detail, with examples and common bugs. Keep this table handy while solving.

Need	C++ STL	Key operations	Time
Dynamic array	vector<int>	push_back, pop_back, [i], size	O(1)
Key → value	unordered_map	[k], count, find, erase	O(1) average
Seen before?	unordered_set	insert, count, erase	O(1) average
Sorted keys, floor/ceiling	map / set	lower_bound, upper_bound, begin, rbegin	O(log n)
Stack / queue	stack / queue	push, pop, top / front	O(1)
Both ends, window max	deque	push/pop_front, push/pop_back	O(1)
Heap	priority_queue (max); add `vector<T>, greater<T>` for min	push, pop, top	O(log n)
Sort, search	sort with lambda, lower_bound	`sort(v.begin(), v.end(), cmp)`	O(n log n), O(log n)

🔑 **KEY POINT**

Four C++ habits prevent most wrong answers: use `long long` (and `1LL * a * b`) when values reach 10^9; pass containers as `const vector<int>&`; store `size()` in an `int` before subtracting; compute mid as `lo + (hi - lo) / 2`.

💼 **AT WORK**

Dry runs are what senior engineers do in code review. "What happens if the list is empty?" is a dry run in their head. A classic production bug: a dashboard showing "max latency 0 ms" because the max started at 0 and the day had no traffic. Another: totals in paise stored as 32-bit int going negative past about ₹2.1 crore.

What Have We Learnt?

- Use the seven steps: understand, examples, brute force, optimise, code, test, analyse.
- Clarifying questions about empty input, negatives, duplicates, sortedness and size change the solution.
- Optimise by removing bottlenecks, unnecessary work and duplicated work.
- A judge does roughly 10^8 operations per second; the constraint table predicts the target complexity.
- About 15 to 20 patterns cover most problems; signal words point to them.
- Dry run your actual code with a variable table, including edge cases.
- Talk continuously and use hints gracefully.
- Know the STL containers (Chapter 0) and use long long against overflow.

Key terms: problem-solving framework, clarifying question, brute force, BUD optimisation, constraint analysis, pattern recognition, dry run, edge case, unordered_map, priority_queue, integer overflow, long long.

✅ CHECK YOUR PROGRESS 1

1. **List the seven steps of the framework.**
   Answer: Understand, examples, brute force, optimise, code, test, analyse.
2. **Why should you dry run with negative numbers when finding a maximum?**
   Answer: A wrong initial value like 0 only fails when all values are negative; positive-only tests hide the bug.
3. **Which C++ container answers "have I seen this value?" in O(1) average?**
   Answer: `unordered_set` (use `count` or `insert(x).second`).
4. **What does `long long x = a * b;` give when a and b are ints near 10^5?**
   Answer: A wrong value: the product overflows in int before being widened. Write `1LL * a * b`.
5. **n ≤ 10^9 and you must find the integer square root. Which approach?**
   Answer: Binary search on the answer, about 30 iterations, O(log n).

Quick Exercises

A. Multiple Choice Questions

1. For n ≤ 10^5, which complexity is the safest target?
   a. $O(n^3)$ b. $O(n^2)$
   c. O(n log n) d. O(2^n)
2. Which STL container gives O(log n) "smallest key ≥ x" queries?
   a. unordered_map b. map
   c. stack d. vector (unsorted)
3. A problem asks for "all permutations" with n ≤ 8. The pattern is:
   a. Sliding window b. Backtracking
   c. Binary search d. Prefix sum
4. What does a default `priority_queue<int>` return from top()?
   a. Smallest b. Largest
   c. First inserted d. Random
5. What is the safe midpoint formula?
   a. (lo + hi) / 2 b. lo + (hi - lo) / 2
   c. hi - lo / 2 d. (lo * hi) / 2

B. Practice Problems

1. Contains Duplicate. Hint: what answers "have I seen this before?" in O(1)?
2. First Unique Character in a String. Hint: two passes with a count array of size 26.
3. Maximum in an Array with possibly all-negative values. Hint: think about the initial value.

Answer Key: Chapter 1

A. MCQ

1. (c) n log n ≈ 1.7 × 10^6 operations; n^2 = 10^10 is too slow.
2. (b) map.lower_bound(x) on a balanced tree.
3. (b) "All" plus tiny n means backtracking, O(n · n!).
4. (b) C++ priority_queue is a max-heap by default.
5. (b) Avoids overflow of lo + hi.

B. Practice Problems

1. Iterate with an unordered_set; if `insert(x).second` is false, a duplicate exists. O(n) time, O(n) space. Alternative: sort and compare neighbours, O(n log n), O(1) extra.
2. First pass fills `int cnt[26]`; second pass returns the first i with count 1, else -1. O(n) time, O(1) space.
3. Start with `best = a[0]` (or INT_MIN) and scan. Starting at 0 fails for all-negative arrays. O(n) time, O(1) space.