

# Arrays

DSA Notes in C++ | concepts, patterns, mnemonics and practice set

---

## What is inside

- 1. Array basics and what each operation really costs
- 2. The 8 core patterns (with a mnemonic to remember all of them)
- 3. Must-know tricks: rotation, majority element, product except self, next permutation
- 4. Edge-case checklist and common C++ traps
- 5. Clue-to-pattern lookup table
- 6. Practice questions, sorted by difficulty, with a tick box for each

**REMEMBER:** The 8 patterns in one sentence: **Two Students Kept Playing Hide-and-seek During Class Monday.**

= **Two** pointers, **Sliding** window, **Kadane**, **Prefix** sum, **Hashing**, **Dutch** flag, **Cyclic** sort, **Merge** intervals.

How to use these notes: read a pattern, cover the code, try to write it from the mnemonic alone, then check. Do not move to the next pattern until you can write the template without looking. Solving 3 to 4 questions per pattern from the practice set at the end matters more than reading this twice.

# 1. Array basics

An array is a block of **contiguous memory** holding elements of the same type. Because every element has the same size, the address of  $a[i]$  is just  $base + i * sizeof(element)$ . One multiplication and one addition, no searching. That is the entire reason indexing is  $O(1)$ , and it is also why inserting in the middle hurts: everything after the gap has to shuffle over.

```
int a[5] = {4, 8, 15, 16, 23};           // fixed size, lives on the stack
vector<int> v = {4, 8, 15, 16, 23};      // dynamic, use this in 95% of problems
vector<int> z(n, 0);                     // n zeros
vector<vector<int>> g(r, vector<int>(c, 0)); // r x c grid of zeros
```

## Cost of each operation

| Operation                    | Time             | Why                                                   |
|------------------------------|------------------|-------------------------------------------------------|
| Access $a[i]$                | $O(1)$           | direct address calculation                            |
| Search, unsorted             | $O(n)$           | may need to look at everything                        |
| Search, sorted               | $O(\log n)$      | binary search                                         |
| push_back (vector)           | $O(1)$ amortized | occasionally the vector doubles and copies everything |
| Insert / delete at index $i$ | $O(n)$           | elements after $i$ shift                              |
| Insert / delete at the end   | $O(1)$           | nothing shifts                                        |
| Insert at the front          | $O(n)$           | everything shifts; use deque if you need this a lot   |

**REMEMBER:** Array = **Fast to read, slow to rearrange**. If a question needs lots of middle inserts, the array is the wrong tool.

**WATCH OUT:** `a.size()` returns an **unsigned** type. On an empty vector, `a.size() - 1` becomes a gigantic number, not -1. Write `int n = a.size();` first, then use `n - 1`.

## Before you code anything, ask these four things

- Is the array **sorted**? (opens up two pointers and binary search)
- Can values be **negative**? (breaks the simple sliding window)
- Are the values inside a **small range like 1..n**? (cyclic sort or index-as-hash)
- Do I need to do it **in place** with  $O(1)$  extra space?

**REMEMBER: S N R P** = **S**orted? **N**egatives? **R**ange small? **P**lace (in place)? Say it as 'Sunday Night Rain Pours'.



## 2. The 8 core patterns

Almost every array question in interviews is one of these wearing a costume. The skill is spotting the clue, not inventing something new.

### 1. Two Pointers

**Clue words:** sorted array, pair or triplet, in-place, remove or move elements, palindrome-like check

Keep two indexes. **Opposite ends:** start at both ends and squeeze inward (works because the array is sorted, so you know which side to move). **Same direction:** a slow pointer marks the finished part and a fast pointer scans ahead.

**REMEMBER:** Opposite ends **squeeze**, same direction **chase**. Too small? move left up. Too big? move right down.

```
// opposite ends: pair with given sum in a SORTED array
vector<int> twoSumSorted(vector<int>& a, int target) {
    int l = 0, r = (int)a.size() - 1;
    while (l < r) {
        int s = a[l] + a[r];
        if (s == target) return {l, r};
        if (s < target) l++; // need a bigger sum
        else r--; // need a smaller sum
    }
    return {};
}

// same direction: remove duplicates from sorted array in place
int removeDuplicates(vector<int>& a) {
    if (a.empty()) return 0;
    int k = 1; // a[0..k-1] is the clean part
    for (int i = 1; i < (int)a.size(); i++)
        if (a[i] != a[k-1]) a[k++] = a[i];
    return k;
}
```

**Complexity:**  $O(n)$  time,  $O(1)$  space. 3Sum is this inside a loop: sort, fix one number, squeeze the other two, giving  $O(n^2)$ .

### 2. Sliding Window

**Clue words:** contiguous subarray or substring, 'longest', 'shortest', 'of size k', all values positive

A window  $[l, r]$  that moves right. **Fixed size:** add the new element, drop the old one. **Variable size:** expand  $r$  every step, and while the window is invalid, shrink from  $l$ .

**REMEMBER:** Expand right, shrink left while broken. Every element enters once and leaves once, so it is  $O(n)$  even with a while inside a for.

```
// fixed window: max sum of any k consecutive elements
long long maxSumK(vector<int>& a, int k) {
    long long win = 0, best = LLONG_MIN;
    for (int i = 0; i < (int)a.size(); i++) {
        win += a[i];
        if (i >= k) win -= a[i - k];          // element that just left
        if (i >= k - 1) best = max(best, win);
    }
    return best;
}

// variable window: smallest length with sum >= target (positives only)
int minLen(vector<int>& a, int target) {
    int l = 0, best = INT_MAX; long long sum = 0;
    for (int r = 0; r < (int)a.size(); r++) {
        sum += a[r];
        while (sum >= target) {                // window is valid, try to shrink
            best = min(best, r - l + 1);
            sum -= a[l++];
        }
    }
    return best == INT_MAX ? 0 : best;
}
```

**Complexity:**  $O(n)$  time,  $O(1)$  space.

**WATCH OUT:** Sliding window with a shrinking condition only works when adding an element moves the sum in one direction. With negative numbers the window logic breaks. Use prefix sum + hashmap instead (pattern 4).

### 3. Kadane's Algorithm

**Clue words:** maximum sum subarray, best contiguous stretch, profit-style questions

At each index you have exactly two choices: **extend** the running subarray or **start fresh** from here. Starting fresh is better whenever the running sum has gone negative.

**REMEMBER:** Carry the baggage only if it is positive. If your baggage is negative, drop it.

```
int kadane(vector<int>& a) {
    int cur = a[0], best = a[0];
    for (int i = 1; i < (int)a.size(); i++) {
        cur = max(a[i], cur + a[i]);          // restart or extend
        best = max(best, cur);
    }
    return best;
}
```

**Complexity:**  $O(n)$  time,  $O(1)$  space. Initialise with  $a[0]$ , not 0, or an all-negative array gives the wrong answer.

| i    | 0  | 1 | 2  | 3 | 4  | 5 | 6 | 7  | 8 |
|------|----|---|----|---|----|---|---|----|---|
| a[i] | -2 | 1 | -3 | 4 | -1 | 2 | 1 | -5 | 4 |
| cur  | -2 | 1 | -2 | 4 | 3  | 5 | 6 | 1  | 5 |
| best | -2 | 1 | 1  | 4 | 4  | 5 | 6 | 6  | 6 |

Dry run on the classic example. Answer is 6, from the subarray 4, -1, 2, 1.

## 4. Prefix Sum

**Clue words:** many range-sum queries, 'subarray sum equals k', negatives allowed, count of subarrays

Build  $P$  where  $P[i]$  is the sum of the first  $i$  elements, with  $P[0] = 0$ . Then the sum of  $a[l..r]$  is  $P[r+1] - P[l]$ . For 'count subarrays with sum  $k$ ', note that a subarray ending here sums to  $k$  exactly when an earlier prefix equals *current prefix* -  $k$ . Store prefixes in a hashmap.

**REMEMBER: Rich minus poor:**  $\text{sum}(l..r) = P[r+1] - P[l]$ . Big prefix minus small prefix leaves the middle.

```
// prefix array for range queries
vector<long long> P(n + 1, 0);
for (int i = 0; i < n; i++) P[i+1] = P[i] + a[i];
// sum of a[l..r] = P[r+1] - P[l]

// count subarrays whose sum is exactly k
int subarraySum(vector<int>& a, int k) {
    unordered_map<int,int> seen;
    seen[0] = 1; // the empty prefix, people forget this
    int pre = 0, count = 0;
    for (int x : a) {
        pre += x;
        if (seen.count(pre - k)) count += seen[pre - k];
        seen[pre]++; // add AFTER checking
    }
    return count;
}
```

**Complexity:**  $O(n)$  time,  $O(n)$  space. Same idea works for 2D grids (prefix sums in a matrix).

## 5. Hashing

**Clue words:** pair with target, duplicates, frequency, 'have I seen this before'

Trade space for time. Instead of a second loop to find the partner of  $a[i]$ , ask a hashmap whether the partner has already appeared.

**REMEMBER: Seen it? Ask the map** before writing another loop. Check first, insert after.

```
vector<int> twoSum(vector<int>& a, int target) {
    unordered_map<int,int> idx; // value -> index
    for (int i = 0; i < (int)a.size(); i++) {
        auto it = idx.find(target - a[i]);
        if (it != idx.end()) return {it->second, i};
        idx[a[i]] = i;
    }
    return {};
}
```

**Complexity:**  $O(n)$  average time,  $O(n)$  space. If the interviewer says  $O(1)$  space, they want sorting + two pointers instead.

## 6. Dutch National Flag (3-way partition)

**Clue words:** only 0/1/2 or three groups, partition around a pivot, sort in one pass

Three pointers split the array into four zones: **[0..low-1]** all zeros, **[low..mid-1]** all ones, **[mid..high]** unknown, **[high+1..end]** all twos. Shrink the unknown zone until it is empty.

**REMEMBER: Low swaps and both move. High swaps and only high moves.** The element swapped in from high is unchecked, so mid must stay put.

```
void sortColors(vector<int>& a) {
    int low = 0, mid = 0, high = (int)a.size() - 1;
    while (mid <= high) {
        if (a[mid] == 0) swap(a[low++], a[mid++]);
        else if (a[mid] == 1) mid++;
        else swap(a[mid], a[high--]); // do NOT move mid here
    }
}
```

**Complexity:**  $O(n)$  time, one pass,  $O(1)$  space.

## 7. Cyclic Sort

**Clue words:** array holds numbers in range  $1..n$  (or  $0..n$ ), find missing or duplicate,  $O(1)$  space demanded

Every value  $v$  has a home at index  $v - 1$ . Keep swapping  $a[i]$  to its home until the slot already holds the right value, then move on. Afterwards, any index  $i$  where  $a[i] \neq i + 1$  exposes a missing number ( $i + 1$ ) and a duplicate ( $a[i]$ ).

**REMEMBER:** Every number has a home: value  $v$  lives at index  $v - 1$ . Send each one home, the leftovers are your answer.

```
void cyclicSort(vector<int>& a) { // values in 1..n
    int i = 0, n = a.size();
    while (i < n) {
        int home = a[i] - 1;
        if (a[i] != a[home]) swap(a[i], a[home]); // send it home
        else i++; // already home, or a duplicate
    }
}
// now: for each i, if a[i] != i + 1 -> i + 1 is missing, a[i] is duplicated
```

**Complexity:**  $O(n)$  time even though there is a while loop, because each swap puts one value permanently in place.  $O(1)$  space.

## 8. Merge Intervals

**Clue words:** ranges, meetings, overlapping, schedules, insert interval

Sort by start time. Walk through; if the current interval starts after the last merged one ends, it is separate. Otherwise they overlap, so stretch the last end.

**REMEMBER:** Sort by start, compare with the last end. Overlap means  $\text{cur.start} \leq \text{last.end}$ .

```
vector<vector<int>> mergeIntervals(vector<vector<int>>& v) {
    sort(v.begin(), v.end());
    vector<vector<int>> res;
    for (auto& cur : v) {
        if (res.empty() || res.back()[1] < cur[0]) res.push_back(cur); // gap
        else res.back()[1] = max(res.back()[1], cur[1]); // overlap
    }
    return res;
}
```

**Complexity:**  $O(n \log n)$  for the sort,  $O(n)$  for the scan.

**WATCH OUT:** Use max when extending the end.  $[1,10]$  followed by  $[2,5]$  must stay  $[1,10]$ , not shrink to  $[1,5]$ .

### 3. Must-know tricks

#### Rotate an array by k (three reverses)

Rotating right by k means the last k elements jump to the front. Reversing the whole array puts them at the front but backwards; reversing each block fixes the order.

**REMEMBER: All, First k, Rest.** Reverse all, reverse the first k, reverse the rest. Always do  $k \% n$  first.

```
void rotateRight(vector<int>& a, int k) {
    int n = a.size(); k %= n;
    reverse(a.begin(), a.end());
    reverse(a.begin(), a.begin() + k);
    reverse(a.begin() + k, a.end());
}
```

#### Majority element (Boyer-Moore voting)

An element appearing more than  $n/2$  times survives a 'cancel one against one' contest, because everything else together cannot outvote it.

**REMEMBER: Count hits zero, pick a new champion.** Same as candidate = x, then +1 for friends and -1 for enemies.

```
int majority(vector<int>& a) {
    int cand = 0, cnt = 0;
    for (int x : a) {
        if (cnt == 0) cand = x;
        cnt += (x == cand) ? 1 : -1;
    }
    return cand;    // if a majority is NOT guaranteed, verify with a second pass
}
```

#### Product of array except self (no division)

Answer for index  $i$  = (product of everything left of  $i$ ) \* (product of everything right of  $i$ ). Fill the left products in the result array, then sweep from the right with a running variable.

**REMEMBER: Left pass, then right pass.** Division fails on zeros, so never reach for it here.

```
vector<int> productExceptSelf(vector<int>& a) {
    int n = a.size();
    vector<int> res(n, 1);
    for (int i = 1; i < n; i++) res[i] = res[i-1] * a[i-1];    // left products
    int right = 1;
    for (int i = n - 1; i >= 0; i--) { res[i] *= right; right *= a[i]; }
    return res;
}
```

## Next permutation

Find the next lexicographically larger arrangement, in place.

- Scan from the right, find the first index  $i$  with  $a[i] < a[i+1]$ . (If none exists, the array is the largest permutation, so just reverse it.)
- Scan from the right again, find the first  $j$  with  $a[j] > a[i]$ , and swap  $a[i]$  and  $a[j]$ .
- Reverse everything after position  $i$ .

**REMEMBER: Find the dip, swap with the just-bigger, flip the tail.**

```
void nextPermutation(vector<int>& a) {
    int n = a.size(), i = n - 2;
    while (i >= 0 && a[i] >= a[i+1]) i--;          // find the dip
    if (i >= 0) {
        int j = n - 1;
        while (a[j] <= a[i]) j--;                  // just-bigger on the right
        swap(a[i], a[j]);
    }
    reverse(a.begin() + i + 1, a.end());          // flip the tail
}
```

## 4. Edge cases and common mistakes

Most wrong answers on array problems are not wrong algorithms. They are one missed edge case. Run this checklist before submitting.

**REMEMBER: E N D S O = Empty, Negatives, Duplicates, Single element, Overflow.** 'Even New Developers Sometimes Overflow'.

| Trap                    | What goes wrong                                             | Fix                                              |
|-------------------------|-------------------------------------------------------------|--------------------------------------------------|
| Empty array             | $a[0]$ access crashes, $a.size()-1$ wraps around            | check $a.empty()$ first, cast size to int        |
| All negative numbers    | Kadane or max-product starts at 0 and returns 0             | start from $a[0]$ (or $INT\_MIN$ )               |
| Duplicates              | pair or triplet answers repeat                              | sort, then skip equal neighbours                 |
| Single element          | loops that start at index 1 or compare neighbours misbehave | handle $n == 1$ up front                         |
| Integer overflow        | sum of ints exceeds about 2.1 billion                       | use long long for sums and products              |
| Off by one              | window size $r - l + 1$ vs $r - l$                          | write a 3-element example and trace by hand      |
| Erasing while looping   | iterators and indexes skip elements                         | use two pointers or erase-remove                 |
| Passing vector by value | copies the whole array every call, TLE                      | pass $vector<int>&$ (const if you do not modify) |
| Wrong pointer move      | infinite loop in two-pointer code                           | make sure every branch changes $l$ or $r$        |

## Debugging habit that saves time

Take the smallest input that could break your code (empty, one element, two equal elements, all negatives), and trace it on paper line by line. Do this *before* you submit, not after the first wrong answer.

## 5. Clue to pattern lookup

Read the problem statement, find the clue in the left column, and you already know where to start.

| If the problem says...                                    | Think about...                                                |
|-----------------------------------------------------------|---------------------------------------------------------------|
| sorted array, find a pair or triplet                      | Two pointers                                                  |
| in place, remove or move elements, $O(1)$ extra space     | Two pointers (slow/fast), or cyclic sort if values are $1..n$ |
| subarray of size $k$                                      | Fixed sliding window                                          |
| longest or shortest subarray with a condition (positives) | Variable sliding window                                       |
| maximum sum of a contiguous subarray                      | Kadane                                                        |
| subarray sum equals $k$ , negatives allowed               | Prefix sum + hashmap                                          |
| many range-sum queries                                    | Prefix sum array                                              |
| two numbers that add up to target, unsorted               | Hashmap                                                       |
| contains duplicate, count frequency, first non-repeating  | Hashmap or set                                                |
| only 0, 1, 2 or three groups                              | Dutch national flag                                           |
| numbers 1 to $n$ , find missing or duplicate              | Cyclic sort, or index-as-hash                                 |
| ranges, meetings, overlapping                             | Sort + merge intervals                                        |
| element appearing more than $n/2$ times                   | Boyer-Moore voting                                            |
| left and right information both needed                    | Prefix and suffix arrays                                      |

## 6. Practice questions

LC = LeetCode problem number. Tick the box when you can solve it cold, without looking at a hint. Do roughly 3 to 4 per pattern, then revisit the ones you ticked after a week to check you still can.

### Easy: build the base

| Done                     | LC  | Problem                             | Pattern                      |
|--------------------------|-----|-------------------------------------|------------------------------|
| <input type="checkbox"/> | 1   | Two Sum                             | Hashing                      |
| <input type="checkbox"/> | 217 | Contains Duplicate                  | Hashing                      |
| <input type="checkbox"/> | 121 | Best Time to Buy and Sell Stock     | Running min / Kadane idea    |
| <input type="checkbox"/> | 26  | Remove Duplicates from Sorted Array | Two pointers                 |
| <input type="checkbox"/> | 283 | Move Zeroes                         | Two pointers                 |
| <input type="checkbox"/> | 88  | Merge Sorted Array                  | Two pointers (from the back) |
| <input type="checkbox"/> | 136 | Single Number                       | XOR trick                    |
| <input type="checkbox"/> | 268 | Missing Number                      | Sum formula / cyclic sort    |
| <input type="checkbox"/> | 169 | Majority Element                    | Boyer-Moore                  |
| <input type="checkbox"/> | 53  | Maximum Subarray                    | Kadane                       |

### Medium: where interviews live

| Done                     | LC  | Problem                         | Pattern                        |
|--------------------------|-----|---------------------------------|--------------------------------|
| <input type="checkbox"/> | 189 | Rotate Array                    | Three reverses                 |
| <input type="checkbox"/> | 75  | Sort Colors                     | Dutch flag                     |
| <input type="checkbox"/> | 238 | Product of Array Except Self    | Prefix / suffix                |
| <input type="checkbox"/> | 15  | 3Sum                            | Sort + two pointers            |
| <input type="checkbox"/> | 11  | Container With Most Water       | Two pointers                   |
| <input type="checkbox"/> | 209 | Minimum Size Subarray Sum       | Variable window                |
| <input type="checkbox"/> | 560 | Subarray Sum Equals K           | Prefix sum + hashmap           |
| <input type="checkbox"/> | 56  | Merge Intervals                 | Merge intervals                |
| <input type="checkbox"/> | 442 | Find All Duplicates in an Array | Cyclic sort / index marking    |
| <input type="checkbox"/> | 287 | Find the Duplicate Number       | Cyclic sort / Floyd cycle      |
| <input type="checkbox"/> | 31  | Next Permutation                | Dip, swap, flip                |
| <input type="checkbox"/> | 152 | Maximum Product Subarray        | Kadane variant (track min too) |
| <input type="checkbox"/> | 128 | Longest Consecutive Sequence    | Hash set                       |
| <input type="checkbox"/> | 18  | 4Sum                            | Sort + two pointers            |
| <input type="checkbox"/> | 73  | Set Matrix Zeroes               | In-place marking               |
| <input type="checkbox"/> | 930 | Binary Subarrays With Sum       | Prefix sum / window            |

## Hard: for when the rest feels easy

| Done | LC  | Problem                             | Pattern                         |
|------|-----|-------------------------------------|---------------------------------|
| [ ]  | 41  | First Missing Positive              | Cyclic sort                     |
| [ ]  | 42  | Trapping Rain Water                 | Two pointers / prefix max       |
| [ ]  | 239 | Sliding Window Maximum              | Window + monotonic deque        |
| [ ]  | 918 | Maximum Sum Circular Subarray       | Kadane twice                    |
| [ ]  | 992 | Subarrays with K Different Integers | Window: atMost(k) - atMost(k-1) |

**REMEMBER:** Suggested order: all Easy first, then Medium grouped by pattern (not by number), then Hard. If you get stuck for 25 minutes, read only the **pattern name** of the solution, close it, and try again.

## Final one-page recall (try this from memory)

1. Write the 8 patterns using the mnemonic. 2. For each, write its clue words. 3. Write the template for Kadane, sliding window and Dutch flag without looking. 4. List the five edge cases from END SO. If you can do all four, arrays are done.