



Arrays & Hashing

A focused reference for interview preparation.

Arrays & Hashing

Recognition framework · verified code templates · complexity, preconditions and failure modes, all in one place

HASHMAP FUNDAMENTALS

FREQUENCY & GROUPING

PREFIX SUMS

ARRAY CONSTRUCTION

HOW TO RECOGNIZE THIS PATTERN

Ask these questions first — they confirm the pattern fits

- Do you need $O(1)$ average lookup for 'have I seen this before'?
- Does the problem ask about pairs, duplicates, or complements?
- Would grouping elements by a computed key (sorted chars, frequency) help?
- Is counting occurrences of each element central to the problem?
- Would a running prefix/suffix computation avoid an $O(n^2)$ approach?
- Is there a fixed small alphabet that would let you use $O(1)$ space instead?

Goal	Signal	Use	Complexity
Find a pair summing to a target	'Two Sum' style	Hashmap of value $\rightarrow$ index	$O(n)$
Detect any duplicate	'Contains duplicate'	Set size comparison	$O(n)$
Group items sharing a property	'Group anagrams'	Hashmap keyed by signature	$O(n \cdot k \log k)$
Find the k most/least frequent items	'Top K Frequent'	Counter + bucket sort	$O(n)$
Find the longest run of consecutive values	'Longest Consecutive'	Hash set + start-of-run check	$O(n)$
Count subarrays summing to k	'Subarray sum equals k'	Prefix sum + hashmap of counts	$O(n)$
Compute a value from all-other-elements	'Product except self'	Prefix and suffix running products	$O(n)$

THE CORE TRADE BEING MADE

Every technique here spends $O(n)$ extra space on a hashmap or set in exchange for turning an $O(n^2)$ nested-loop search into an $O(n)$ single pass. Whenever you catch yourself writing a nested loop to check *does some other element satisfy X*, ask whether a hashmap built in a single first pass would answer that in $O(1)$ instead.

HOW TO STATE SPACE COMPLEXITY

Separate **auxiliary** space from **output** space every time. Product of Array Except Self is $O(1)$ auxiliary but $O(n)$ output; Two Sum is $O(n)$ auxiliary and $O(1)$ output. Interviewers ask for this split deliberately, and volunteering it is cheap.

HASHMAP FUNDAMENTALS

Trade space for $O(1)$ average lookup

1 Two Sum

$O(n)$ time
 $O(n)$ auxiliary space

Store each value's index as you scan; before inserting, check whether its complement ($\text{target} - \text{value}$) has already been seen.

```
def two_sum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i
    return []
```

WHY IT WORKS

Checking *before* inserting is what stops an element from pairing with itself. Insert first and `[3]` with target 6 returns `[0, 0]`.

SAY THIS OUT LOUD

State the $O(n^2)$ double loop as the baseline, then say the hashmap converts 'does a complement exist' from a scan into a lookup. Naming the baseline first is worth real points.

INTERVIEWER FOLLOW-UP

"What if the array is sorted?" → two pointers, $O(1)$ space. "What if you need all pairs?" → see #10 and #11.

2 Contains Duplicate

$O(n)$ time
 $O(n)$ auxiliary space

A set only keeps unique values — if its size shrinks compared to the list, a duplicate exists.

```
def contains_duplicate(nums):
    return len(set(nums)) != len(nums)
```

INTERVIEWER FOLLOW-UP

"Now without extra space." If the array may be modified, sort and scan adjacent pairs — $O(n \log n)$ time, $O(1)$ space. Have that trade ready before it is asked.

3 Valid Anagram

$O(n)$ time
 $O(1)$ space for a fixed alphabet

Two strings are anagrams exactly when their character counts match.

```
from collections import Counter

def is_anagram(s, t):
    if len(s) != len(t):
        return False
    return Counter(s) == Counter(t)
```

BREAKS WHEN

The $O(1)$ space claim depends on a bounded alphabet. Counter is a dict sized by the number of *distinct* characters — $O(1)$ for 26 lowercase letters, $O(k)$ for arbitrary Unicode. State which you are assuming.

INTERVIEWER FOLLOW-UP

"What if the inputs are Unicode?" and "what if you get a stream of strings to compare against one target?" — the second turns into the Group Anagrams signature idea below.

FREQUENCY & GROUPING

4 Group Anagrams

$O(n \cdot k \log k)$ time
 $O(n \cdot k)$ space, $k = \text{max word length}$

Words that are anagrams share the same sorted-character signature; group them in a hashmap keyed by that signature.

```
from collections import defaultdict

def group_anagrams(strs):
    groups = defaultdict(list)
    for s in strs:
        key = ''.join(sorted(s))
        groups[key].append(s)
    return list(groups.values())
```

WHY IT WORKS

Any function that is constant across a group and distinct between groups works as a key. Sorting is simply the most obvious such function — that framing is what lets you invent the key for an unfamiliar grouping problem.

INTERVIEWER FOLLOW-UP

“Can you beat the $k \log k$?” Yes — key on a 26-length count tuple, `tuple(count)`, for $O(n \cdot k)$ total. Mention it unprompted; it proves you noticed the sort inside the loop.

5 Top K Frequent Elements — Bucket Sort

$O(n)$ time
 $O(n)$ space

Count frequencies, then bucket values by frequency (max frequency is at most n) — no sorting or heap required.

```
from collections import Counter

def top_k_frequent(nums, k):
    count = Counter(nums)
    buckets = [[] for _ in range(len(nums) + 1)]
    for num, freq in count.items():
        buckets[freq].append(num)

    result = []
    for freq in range(len(buckets) - 1, 0, -1):
        for num in buckets[freq]:
            result.append(num)
            if len(result) == k:
                return result
    return result
```

WHY IT WORKS

Frequency is bounded by n , which makes the frequency axis a legal array index. That bound is the entire reason the trick is allowed — say it out loud, because it is the step being tested.

SAY THIS OUT LOUD

Offer the heap solution first at $O(n \log k)$, then improve to $O(n)$. Arriving at the optimum by visibly improving a correct answer scores better than producing it with no derivation.

6 Longest Consecutive Sequence

$O(n)$ amortized time
 $O(n)$ space

Put all values in a set; only start counting a sequence from a value whose predecessor is absent, so each run is walked exactly once.

```
def longest_consecutive(nums):
    num_set = set(nums)
    best = 0
    for num in num_set:
        if num - 1 not in num_set:      # start of a sequence
            length = 1
            while num + length in num_set:
                length += 1
            best = max(best, length)
    return best
```

WHY IT WORKS

The inner while looks like it makes this $O(n^2)$, but the guard means each value is walked over by exactly one run. Total inner work across the whole outer loop is $O(n)$.

COMMON BUG

Iterate over `num_set`, not `nums`. With heavy duplicates, iterating the list re-walks the same runs and the amortization argument collapses.

INTERVIEWER FOLLOW-UP

The interviewer will challenge the $O(n)$ claim — that is the point of the problem. Have the amortization sentence ready verbatim.

PREFIX SUMS

Turn 'sum over a range' into two $O(1)$ lookups

7 Subarray Sum Equals K

$O(n)$ time
 $O(n)$ space

A subarray $(i, j]$ sums to k exactly when $\text{prefix}[j] - \text{prefix}[i] == k$. So while scanning, look up how many earlier prefixes equal $\text{running} - k$.

```
from collections import defaultdict

def subarray_sum(nums, k):
    counts = defaultdict(int)
    counts[0] = 1      # empty prefix: lets a prefix itself count as a match
    running = 0
    total = 0
    for num in nums:
        running += num
        total += counts[running - k]
        counts[running] += 1
    return total
```

COMMON BUG

Forgetting `counts[0] = 1`. Without it every subarray starting at index 0 is missed, and the answer is wrong only on some inputs — which is worse than being wrong on all of them.

BREAKS WHEN

Do not reach for a sliding window. Windows require non-negative values; this works with negatives precisely because it never shrinks a window. If the interviewer says 'all positive', the window becomes available and is $O(1)$ space.

SAME SKELETON

Same prefix-plus-hashmap skeleton: Contiguous Array (map 0→-1, find equal prefixes), Subarray Sums Divisible by K (key on running % k), Continuous Subarray Sum.

ARRAY CONSTRUCTION

Build the answer from running aggregates or an explicit framing

8 Product of Array Except Self

O(n) time
O(1) auxiliary, O(n) output

Compute running prefix products left-to-right, then multiply in running suffix products right-to-left — no division needed.

```
def product_except_self(nums):
    n = len(nums)
    result = [1] * n

    prefix = 1
    for i in range(n):
        result[i] = prefix
        prefix *= nums[i]

    suffix = 1
    for i in range(n - 1, -1, -1):
        result[i] *= suffix
        suffix *= nums[i]

    return result
```

WHY IT WORKS

Division is banned for a real reason, not an artificial one: a single zero breaks divide-the-total, and two zeros break the patched version too. Say that if the interviewer does not.

SAY THIS OUT LOUD

Claim O(1) auxiliary space explicitly and note that the output array is excluded by convention. Claiming O(1) without that caveat sounds like you have not thought about it.

9 Encode and Decode Strings

O(n) time
O(n) space

Prefix each string with its length and a delimiter so decoding never confuses a delimiter inside a payload with a real separator.

```
def encode(strs):
    return ''.join(f'{len(s)}#{s}' for s in strs)

def decode(s):
    result = []
    i = 0
    while i < len(s):
        j = i
        while s[j] != '#':
            j += 1
        length = int(s[i:j])
        result.append(s[j + 1: j + 1 + length])
        i = j + 1 + length
    return result
```

WHY IT WORKS

Length-prefixing beats escaping because the length tells the decoder exactly how far to read before it looks for the next delimiter. ["a#b"] encodes to 3#a#b and round-trips correctly.

SAY THIS OUT LOUD

Walk through the adversarial input yourself — a payload containing the delimiter. Producing your own counterexample to the naive `' , '.join` answer is the whole test here.

10 Two Sum — All Unique Value Pairs

$O(n)$ time
 $O(n)$ space

Same complement idea, but keep scanning and deduplicate by the normalized (low, high) key so each distinct value pair is reported once.

```
def two_sum_unique_pairs(nums, target):
    seen = set()
    found = set()
    result = []
    for num in nums:
        complement = target - num
        if complement in seen:
            key = (min(num, complement), max(num, complement))
            if key not in found:
                found.add(key)
                result.append(key)
        seen.add(num)
    return result
```

COMMON BUG

Deduplicate through a *set*, not by scanning the result list. `if key not in result` is $O(p)$ per check and quietly turns the whole function quadratic.

11 Two Sum — Count All Index Pairs

$O(n)$ time
 $O(n)$ space

If the question is *how many pairs of positions* sum to the target, count multiplicities instead: each new element pairs with every earlier occurrence of its complement.

```
from collections import Counter

def two_sum_count_pairs(nums, target):
    counts = Counter()
    total = 0
    for num in nums:
        total += counts[target - num]
        counts[num] += 1
    return total
```

BREAKS WHEN

“All pairs” is ambiguous and the two readings give different answers. For `[0, 0, 0, 0]` with target 0 there is **1** unique value pair but **6** index pairs. Pin the contract down before writing code.

COMMON BUG

A greedy match-and-consume loop (decrement the complement's count on a hit) answers *neither* version. It returns 1 pair for `[2, 2, 2]`, target 4 — the correct answers are 1 unique pair and 3 index pairs, and it only coincides with the first by accident.

QUICK-REFERENCE SUMMARY

One-line reminder for every technique on this sheet

Technique	What it's for
Two Sum	Hashmap of value $\rightarrow$ index; check the complement <i>before</i> inserting
Contains Duplicate	Compare <code>len(set(nums))</code> to <code>len(nums)</code>

Technique	What it's for
Valid Anagram	Compare character-count Counters; $O(1)$ space only for a bounded alphabet
Group Anagrams	Hashmap keyed by a signature that is constant within a group
Top K Frequent (Bucket Sort)	Frequency $\leq n$ makes frequency a legal array index
Longest Consecutive Sequence	Only start counting at a run's beginning — $O(n)$ amortized
Subarray Sum Equals K	Prefix sums in a hashmap; seed <code>counts[0] = 1</code>
Product of Array Except Self	Running prefix products, then running suffix products
Encode and Decode Strings	Length-prefixed strings make delimiter collisions harmless
Two Sum — Unique Pairs	Deduplicate by normalized (low, high) key through a set
Two Sum — Count Index Pairs	Counter of earlier values; add multiplicities

Verification & Practice

Arrays & Hashing

WHEN NOT TO USE THIS PATTERN

Avoid hashing when sorted order, range queries, or strict $O(1)$ auxiliary space is the real requirement.

EDGE-CASE CHECKLIST

- Empty and single-item inputs
- Duplicates, negative values, and hash collisions
- Mutable or non-hashable keys

HIGH-VALUE TEST CASES

- Two Sum: [3,3], target 6 -> [0,1]
- Longest consecutive: [100,4,200,1,3,2] -> 4
- Group anagrams: empty strings and repeated words

AUTHORITATIVE REFERENCES

<https://docs.python.org/3/library/stdtypes.html#mapping-types-dict>

<https://xlinux.nist.gov/dads/>

<https://www.w3.org/WAI/WCAG22/Understanding/contrast-minimum.html>