Quick corrections (important!)

- HashMap/HashSet ops: average O(1) per operation (lookup/insert/delete). For n operations total ≈ O(n). (Worst-case O(n) exists in theory but is rare in interviews.)
- Heap / priority queue:
    - push / pop are O(log n) each.
    - peek (top) is O(1).
    - Building a heap from an array is O(n).
- "All pairs / all triples" brute force:
    - Pairs: about $n(n-1)/2$ checks ⇒ $O(n^2)$.
    - Triples: about $n^3/6$ ⇒ $O(n^3)$.
      Hashing can sometimes *avoid* checking every pair (e.g., Two Sum), but not always.

What to use *when*: kth, nearest, pairs, etc.

1) "Give me the k-th smallest/largest"

Pick based on the situation:

- Single shot (static array), only need the k-th
    - Quickselect (Hoare's selection): O(n) average, O(1) extra space.
      Use when you don't need the array fully sorted.
- k is small and you're scanning once

    - Min-heap of size k for k-th largest (or max-heap for k-th smallest): O(n log k) time, O(k) space.
    - Recipe (k-th largest): push first k numbers; for each next x, if x > heap[0], pop then push.
- You also need the array sorted afterward
    - Sort: O(n log n), then index [k-1]. Simple and often fine.
- Streaming (numbers arrive over time) or sliding window k-th/median
    - Two-heap approach (max-heap for lower half, min-heap for upper): insert/delete in O(log n), read median/top in O(1).
    - Balanced BST / ordered multiset with counts (or order-statistic tree): supports insert/delete/search in O(log n) and can get the i-th element in O(log n) if it maintains subtree sizes.

Mini recipes

```
# k-th largest via min-heap of size k
import heapq
def kth_largest(nums, k):
    h = []
    for x in nums:
        if len(h) < k: heapq.heappush(h, x)
        elif x > h[0]:
            heapq.heapreplace(h, x)  # pop+push in O(log k)
    return h[0]
```

```
# Quickselect (k-th smallest), average O(n)
import random
def quickselect(a, k):  # k: 0-indexed
    l, r = 0, len(a)-1
    while True:
        pivot = a[random.randint(l, r)]
        i, j, t = l, r, l
        # three-way partition: < pivot | == pivot | > pivot
        while t <= j:
            if a[t] < pivot:
                a[i], a[t] = a[t], a[i]; i += 1; t += 1
            elif a[t] > pivot:
                a[t], a[j] = a[j], a[t]; j -= 1
            else:
                t += 1
        if k < i: r = i-1
        elif k > j: l = j+1
        else: return a[k]
```

2) “Find the nearest value(s) to x”

- One-dimensional, many queries:

    - Sort once (O(n log n)), then binary search each query (O(log n)).
    - In languages with ordered sets/maps: lower_bound/upper_bound are O(log n) per query.
- Streaming / dynamic insertions & nearest-query:
    - Balanced BST / ordered set (e.g., TreeSet/TreeMap, C++ set, Python's bisect on a *sorted list* for search; for inserts/deletes you want a balanced tree or a library like sortedcontainers).
    - Each insert/delete/query O(log n).
- Nearest pair sum (closest to target):
    - Sort + two pointers ⇒ O(n Log n) preprocess, then O(n) scan.
- Nearest in 2D/ND:
    - k-d tree / spatial index for true nearest neighbor; heaps/maps aren't ideal here.

Mini recipe (1D nearest)

```
import bisect

def nearest(sorted_arr, x):

    i = bisect.bisect_left(sorted_arr, x)

    cand = []

    if i > 0: cand.append(sorted_arr[i-1])

    if i < len(sorted_arr): cand.append(sorted_arr[i])

    # pick closer of candidates
```

```
    return min(cand, key=lambda y: (abs(y-x), y))
```

3) “Pairs / Triples” style questions

- Two Sum (exact target): hash set/map in one pass ⇒ O(n) time, O(n) space.
- 3Sum / 3Sum closest: sort + two pointers ⇒ O(n²) (can’t generally do better).
- All pairs with difference ≤ d: sort + sliding window/two pointers ⇒ O(n log n) + O(n).
- All unique pairs (outputting them): even just printing Θ(m) pairs costs Ω(m) time; beware “output size” lower bounds.

Why hashing doesn’t always kill the O(n²): if the property depends on two free choices simultaneously (like choosing both i and j) without a reducible “complement”, you still end up exploring Θ(n²) combinations (3Sum is the classic).

---

4) When do balanced trees help?

They give you order-aware operations in O(log n):

- lower_bound(x) / upper_bound(x) for nearest on the left/right.
- Maintain a dynamic ordered set for streaming queries.
- With order statistics (tree augmented with subtree sizes): get the k-th element or rank of a key in O(log n).
- Useful for: sliding window median, maintaining top-k with deletes, “first greater than x”, etc.

(If your language lacks a built-in ordered multiset with order statistics, use two heaps or a library.)

A practical decision guide (rephrased)

- Only the k-th needed once (static) → Quickselect (avg O(n)); if k is tiny use min-heap of size k (O(n log k)); if you also need sorted order, sort.
- k-th / median with inserts/deletes (streaming or sliding window) → two heaps or balanced BST (O(log n) per update).
- Nearest value(s) in 1D → sort + binary search; for dynamic data → balanced BST.
- 2-sum → hash set/map (O(n)).
  3-sum / 3-sum closest → sort + two pointers ($O(n^2)$).
  Pairs with constraint after sorting → two pointers / sliding window.
- Need membership/frequency fast → hash set/map (O(1) avg).
  Need order/closest/rank → balanced BST or sort + binary search.
  Need top-k → heap (O(n log k)) or partial sort/nth_element (C++).

Tiny complexity "flash cards"

- Hash map/set op: O(1) avg
- Heap push/pop: O(log n) (peek: O(1))
- Sort: O(n log n)
- Two pointers scan: O(n) after sorting

- Quickselect: O(n) avg
- All pairs: $O(n^2)$; all triples: $O(n^3)$

1) Kill nested loops by changing the question

Ask: “Can I turn a two-index search into a one-index lookup?”

- Equality/complements → Hash set/map (avg O(1) per op).
    - e.g., Two Sum: $O(n^2)$ → O(n) with a set/map.
- Counting/frequency → Hash map.
    - e.g., anagram/group-by problems: O(n) time, $O(\sigma)$ space.

2) If the condition is about order/closest

You need ordering info.

- One-off queries → Sort + binary search/two pointers.
    - e.g., 3Sum / closest pair sum: $O(n^3)$ → $O(n^2)$ after sort.
- Many dynamic queries (insert/delete/nearest/k-th online) → Balanced tree or two-heap pattern.
    - Per update/query O(log n); peek/median O(1).

3) Convert “over many subarrays” to linear time

- Sliding window (grow/shrink with set/map of counts): O(n) amortized.
    - e.g., Longest substring w/o repeat; Longest subarray with sum ≤ K (monotone tricks).
- Prefix sums + hash for sum/target patterns: O(n).
    - e.g., subarray sum = k, subarray divisible by k.
- Window min/max → Monotonic deque: O(n) vs naive O(nk).

4) Selection & top-K (optimize beyond sorting)

- k-th smallest/largest (single shot) → Quickselect (avg O(n), O(1) extra).
- Top-K while scanning → Heap of size K: O(n log K), returns K largest in the heap.
- Top-K frequent → Count map +
    - Bucket sort (O(n) when buckets = n), or
    - Min-heap of size K (O(n log K)).

5) Use range/domain info to beat maps

- Keys in small integer range → Counting array: $O(n + K)$ time, $O(K)$ space (often faster constants than a hash map).

6) Know the "can't do better" lower bounds

- All pairs inherently $\Theta(n^2)$ checks if you must *consider/report many pairs*.
- 3Sum is $\Omega(n^2)$ for comparison models—so $O(n^2)$ is already optimal up to logs.
- If the output size is m, expect $\Omega(n + m)$ at least.

Quick upgrade recipes (copyable ideas)

- Two Sum: set/map → $O(n)$.
- 3Sum: sort + two pointers → $O(n^2)$.
- Subarray sum = k: prefix sum + map of counts → $O(n)$.
- Longest substring w/o repeat: sliding window + map → $O(n)$.
- Sliding window maximum: monotonic deque → $O(n)$.

- k-th largest: quickselect (O(n)) or min-heap size K (O(n log K)).
- Top-K frequent: count → buckets (O(n)) or min-heap K (O(n log K)).

Micro-optimizations that actually matter in interviews

- Initialize running min/max with first element (or ±∞), not `0`, to avoid logic bugs.
- Use the right container:
    - Presence only → set; counts → dict/Counter; order needed → sorted list + bisect (static) or tree/two-heap (dynamic).
- Avoid repeated work:
    - Don't recompute sums—prefix them.
    - Don't rescan windows—move pointers once.
- Exploit constraints:
    - If values are bounded, prefer arrays over maps.
    - If K is tiny, heap of size K beats sorting.

A mental flowchart (10-sec)

1. Membership/equality? → Hash (O(1) avg).

2. Order/nearest/k-th? → Sort + BS/two-ptr (static) or Tree/Heaps (dynamic).

3. Over subarrays/windows? → Sliding window / Prefix+Hash / Monotone deque.

4. Small integer domain? → Counting/buckets.

5. Is O(n²) unavoidable? If yes (e.g., 3Sum), jump straight to the best O(n²) pattern.

`list.pop()` (or `deque.pop()`) removes from the right end → LIFO (stack).

Short answer: `list.append` stores a *reference* to the same list object, not a frozen snapshot. When you later do `pop()`, you mutate that same object, so the element you already appended in `new_list` changes too.

Without copy:

```python
path = [1, 2]
new_list = []
new_list.append(path)   # store a REFERENCE to the same list object
path.pop()              # mutate that object
print(new_list)         # -> [[1]]
```

The element inside `new_list` changed because it's the *same object* as `path`.

With copy:

```python
path = [1, 2]
new_list = []
new_list.append(path.copy())  # snapshot
path.pop()                    # mutate path
print(new_list)               # -> [[1, 2]]  (unchanged snapshot)
```

Key differences (`sort` vs `sorted`):

Mutation: `list.sort()` modifies in place; `sorted()` creates a new list.

Return: `list.sort()` → `None`; `sorted()` → new sorted list.

Inputs: `list.sort()` works only on lists; `sorted()` works on any iterable.
Both support `key=` and `reverse=` and are stable, adaptive Timsort.

```python
a = [3, 1, 2]

b = a.sort()          # in-place
print(a)              # -> [1, 2, 3]
print(b)              # -> None   (common gotcha!)
```

```python
a = [3, 1, 2]

b = sorted(a)         # returns a NEW list
print(a)              # -> [3, 1, 2]  (unchanged)
print(b)              # -> [1, 2, 3]
```

- `list.sort()` **modifies** the list and returns **None**.
- `sorted(iterable)` **does not modify** the input and **returns** a new list.

2) What they work on

```python
s = "bca"
# s.sort()            # ❌ AttributeError: strings don't have .sort()

print(sorted(s))      # -> ['a', 'b', 'c']  # works on ANY iterable
```

3) Keeping the original vs saving memory

- Keep original: use `sorted(a)` (or `b = sorted(a, reverse=True)`).
- Save memory / huge list: mutate in place with `a.sort()`.

- Save memory / huge list: mutate in place with `a.sort()`.

4) Same features: `key=` and `reverse=`

Both support them and are **stable** (equal keys keep their original order).

```python
people = [
    {"name":"Ann",  "age":20},
    {"name":"Zoe",  "age":18},
    {"name":"Amy",  "age":20},  # same age as Ann
]

# In-place
people.sort(key=lambda p: p["age"])
print([p["name"] for p in people])  # -> ['Zoe', 'Ann', 'Amy']  (Ann before Amy preserved)

# New list
people2 = sorted(people, key=lambda p: p["name"], reverse=True)
print([p["name"] for p in people2]) # -> ['Zoe', 'Ann', 'Amy']
```

Quick rules

- Need the result **and** to keep the original: `sorted(iterable)`.
- Fine mutating the list (and want to avoid allocating a new list): `list.sort()`.
- Never do `b = a.sort()` — `b` will be `None`.

:

🧩 1️⃣ The error: `TypeError: 'method' object is not iterable`

You wrote:

`return list(group.values)`

💥 That's the problem.

Here, you forgot to call the `.values()` method.xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Without parentheses, `group.values` refers to the method object itself, not the actual list of values.

✅ Correct version:

```
return list(group.values())
```

Now `group.values()` returns something iterable (a view of all the values), and `list()` converts it to a list.

🧠 2️⃣ `.values`, `.keys`, `.items` — are methods

Every dictionary (or defaultdict) has these methods:

Method	What it returns	Example
`.keys()`	all keys	`{'a':1,'b':2}.keys()` → `dict_keys(['a','b'])`
`.values()`	all values	→ `dict_values([1,2])`
`.items()`	key-value pairs	→ `dict_items([('a',1), ('b',2)])`

If you forget the parentheses — you're just referring to the function itself, not calling it.
That's why Python says "not iterable" — a function is not a collection.

🧩 3️⃣ "It only auto-creates values when you access a missing key" — explained

Let's see the difference between `dict` and `defaultdict` clearly:

💡 Example 1 — Normal `dict`

```
d = {}
print(d['x'])
```

💥 **Error:**

```
KeyError: 'x'
```

Normal `dict` does not create missing keys.

💡 Example 2 — `defaultdict(list)`

```
from collections import defaultdict
d = defaultdict(list)
print(d['x'])
```

✅ **Output:**

```
[]
```

Here's what happened internally:

1. **`'x'` was missing.**
2. **Python said, "Oh, the default factory is `list`."**
3. **It created an empty list: `[]`**
4. **It stored `'x': []` in the dictionary.**
5. **Then it returned that list.**

So now `d` becomes:

```
{'x': []}
```

That's what we mean by auto-create.

🧩 4️⃣ What about `.get()` and `.setdefault()`?

These methods behave normally — they don't trigger auto-creation.

Example:

```
d = defaultdict(list)
print(d.get('y'))
```

✅ **Output:**

```
None
```

Even though it's a defaultdict, `.get()` will not create `'y'`.
Auto-creation only happens when you access directly (`d['y']`), not with `.get()`.

🧩 5️⃣ "It doesn't magically decide what to append"

This means:
Even though `defaultdict` automatically gives you an empty list for new keys, you still control what goes inside that list.

Example:

```
groups = defaultdict(list)
groups['a'].append('apple')  # you decide to append 'apple'
groups['a'].append('ant')    # you append 'ant'
```

`defaultdict` only made sure that `'a'` had an empty list `[]` when you started.
It doesn't "decide" to append anything automatically — that's your job.

🧩 6️⃣ Summary — `dict` vs `defaultdict`

Feature	`dict`	`defaultdict(list)`

Missing key access	❌ `KeyError`	✅ **Creates new key automatically**
Default value	**None**	**Return value of factory (e.g. `list()`)**
When auto-created	**Never**	**When you access key directly (`d[key]`)**
`.get()` behavior	**Normal**	**Normal (no auto-create)**
Example use case	**Storing known keys**	**Grouping or counting items**
Typical example	`d[key] = value`	`d[key].append(value)`

Array & Hashing

Hashing :

1.Hash Function
2. Collision - How to handle collision

Now, let's say our hash function isn't perfect:

- hash(14) -> returns index 4
- hash(24) -> also returns index 4 (This is a collision!)

The problem is that a simple array slot like main_array[4] can only hold **one direct value**. You can't do main_array[4] = 14 and then main_array[4] = 24. The second operation would overwrite the first.

Your Question: "Can't we use an array to store multiple values in the same bucket?"

Yes, we can! But it changes the performance. This is the concept of "separate chaining". The bucket itself is no longer a single value, but a **pointer to another data structure**.

Let's see what happens if we use a dynamic array (like a List or Vector) for our bucket:

1. We have our main_array. Each slot holds a *pointer* to another array (or is null).
2. To add(14):
    - hash(14) -> 4.
    - main_array[4] is empty, so we create a new list: list_at_4 = [].
    - Add 14 to it: list_at_4 is now [14].

    - Point main_array[4] to this list.
3. To add(24):
    - hash(24) -> 4.
    - Go to main_array[4]. We find our existing list.
    - Add 24 to it: list_at_4 is now [14, 24].

Here is the crucial difference:

Now, to check contains(24)?:

1. Calculate hash(24), which gives me 4.
2. Go directly to main_array[4]. I find the list [14, 24].
3. **I must now search through this second list.** Is the first element 24? No. Is the second element 24? Yes. Found it.

This search is **not O(1)**. It is O(N), where N is the number of items that have collided in that one bucket. We've lost our constant-time lookup and now have a linear search.

- **The "perfect world" O(1)** relies on the main array slot holding the value *directly*.
- When collisions happen, a single slot main_array[index] cannot hold multiple values directly.
- Instead, main_array[index] holds a reference to a **container** (like a Linked List or another Array) that stores all the colliding values.
- Once you're directed to that container, you have to **search within it**, which is what degrades the performance from O(1) to O(N) (or O(log N) if you use a tree).

The search **within that specific bucket's linked list** is O(k), where k is the number of items in that particular bucket.

So, if the claim is that a hash set is O(1) on average, how can this be true?

It comes down to the difference between **Average-Case** and **Worst-Case** performance, which is managed by two key ideas:

1. The Goal: A Good Hash Function & Distribution

A well-designed hash function distributes items evenly across all the buckets. The goal is to keep the number of items in any single bucket (k) as small as possible.

- If you have N total items and M buckets, a perfect hash function would put N/M items in each bucket.
- A good hash set implementation also performs **resizing**. When the number of items grows too large for the current number of buckets (i.e., the lists are getting long), it creates a new, larger main_array and re-distributes all the elements.

This ensures that, on average, the linked lists remain very short. Searching a list of length 1, 2, or 3 is practically a constant-time operation. Therefore, the **average time complexity is O(1)**.

2. The Worst-Case Scenario: O(N)

Now, let's consider the worst possible situation.

- Imagine you have a terrible hash function where every single item hashes to the *same index*. For example, hash(item) -> always returns 2.
- If you add N items to your hash set, they will all end up in the linked list at buckets[2].
- Your hash set has now degenerated into a single linked list.

In this worst-case scenario, to find an item, you would have to:

1. Calculate the hash to get index 2. (O(1))
2. Traverse the entire linked list of N items. (O(N))

So, the **worst-case time complexity is O(N)**.

Summary

Case	Time Complexity	Why?
Average	**O(1)**	A good hash function and resizing keep the linked lists (chains) at a small, constant length. The O(k) search is effectively O(1).
Worst	**O(N)**	A bad hash function puts all N elements into a single bucket, turning the hash set into a single linked list.

This is why, when discussing hash tables/sets, you will almost always hear their performance described as **"O(1) average time"**. The entire design is optimized to make the average case the normal case and the worst case extremely rare.

HASH SET : add, remove,contains
HASH MAP: put,get, containsKey, remove

Dict : Sorted Function only sort keys so if you want both keys and values use items() and you need to sort only values add lambda expression.

freq[num] = freq.get(num, 0) + 1

Top K Frequent Elements:

Python3 • Auto

```python
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        dict={}
        for num in nums:
            if num not in dict:
                dict[num]=1
            else:
                dict[num]=+1
        sorted_items=sorted(dict.items(), key=lambda x:x[1], reverse=True)
        return [item[0] for item in sorted_items[:k]]
```

Use `.items()` when you need both number and frequency.
Use `.values()` only if you care about frequencies alone (no mapping back to numbers).

Counter : most_common, elements, update

Substring

- A **substring** is a **contiguous** block of characters from the string (no gaps).
- Think: one clean slice of the original string.
- Every substring keeps original order *and* has no skips.

Subsequence

- A **subsequence** keeps the **order**, but you're allowed to **skip** characters.
- Think: highlight some characters left-to-right; you can skip any in between.
- Not required to be contiguous.

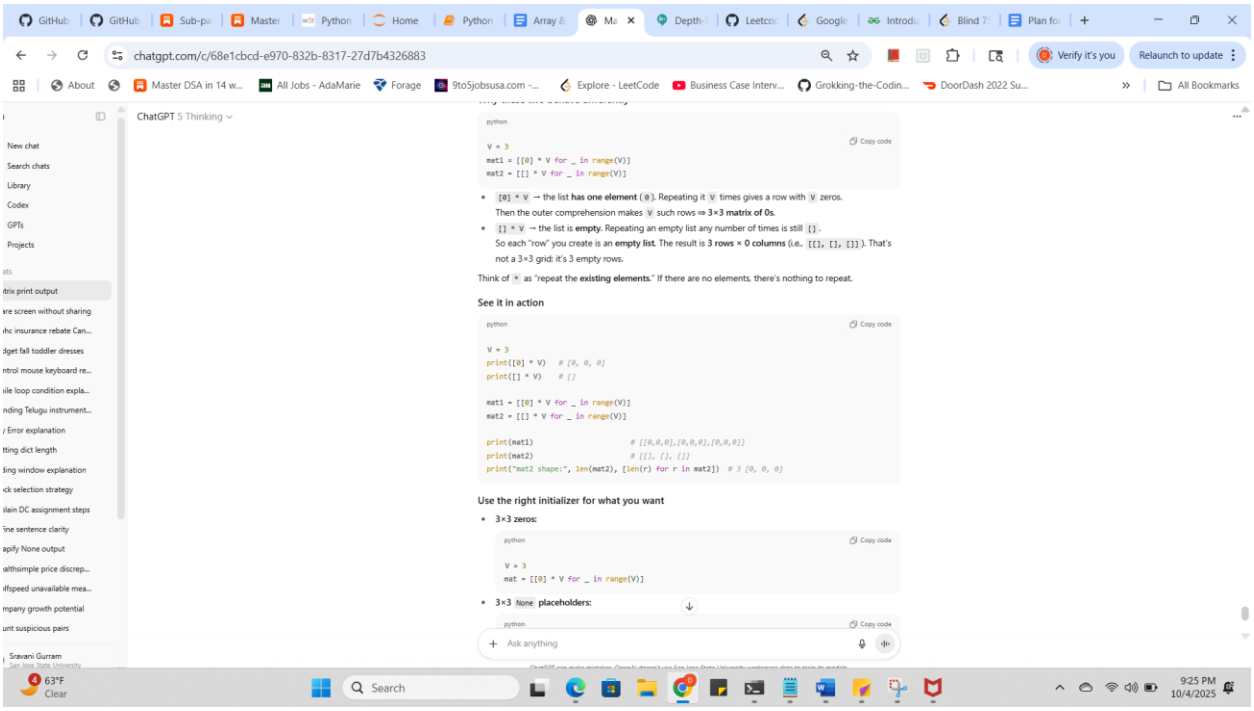
With `s = "pwwkew"`

Indices: `p(0) w(1) w(2) k(3) e(4) w(5)`

- **"wke"** uses indices `(2,3,4)` → contiguous → **substring** ✅ (and also a subsequence, since every substring is one).
- **"pwke"** uses indices `(0,2,3,4)` → there's a gap between `p` and the next `w` → **subsequence** ✅ but **not** a substring ❌.

GRAPHS:

- `[0] * V` → **the list has one element (`0`). Repeating it `V` times gives a row with `V` zeros.**
  Then the outer comprehension makes `V` such rows ⇒ 3×3 matrix of 0s.
- `[] * V` → **the list is empty. Repeating an empty list any number of times is still `[]`.**
  So each "row" you create is an empty list. The result is 3 rows × 0 columns (i.e., `[[], [], []]`). That's not a 3×3 grid; it's 3 empty rows.



CONTAINS DUPLICATE:

1) HashSet "length trick" (the quick one-liner)

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

How it works: building `set(nums)` removes duplicates. If any duplicates existed, the set will be smaller.

- Time: O(n) average (must process all n items).
- Space: O(n) in the worst case (all items unique).
- Pros: shortest code; very fast in CPython because the whole set build runs in C.
- Cons: no early exit—it always scans the entire list, even if the first two numbers already duplicate.

Example `[2, 14, 18, 22, 22]`
`set(nums) -> {2, 14, 18, 22}` → `len(nums)=5`, `len(set)=4` → duplicate ⇒ `True`.

2) HashSet “check with early exit”

```
def containsDuplicate(nums):
    seen = set()
    for x in nums:
        if x in seen:      # average O(1)
            return True    # early exit as soon as a dup appears
        seen.add(x)
    return False
```

How it works: walk the list once, tracking what you’ve seen. The first time you see an element twice, return immediately.

- Time: O(n) average, best-case O(1) if a duplicate appears very early; worst-case O(n) if no duplicates (or very late).
- Space: O(n) worst case.
- Pros: can short-circuit early—great when duplicates are common/early.
- Cons: Python loop overhead (bytecode) vs the C-level loop of `set(nums)`; if duplicates are rare or late, this can be a tad slower than the one-liner.

Example `[2, 14, 18, 22, 22]`
`seen = {2,14,18}` → see 22 → add → next 22 is already in `seen` → return True one element earlier than the length trick finishes.

Which is “quickest”?

It depends on the data:

- Duplicates likely/early? Use early-exit version → can stop fast.
- Duplicates rare/unknown and you just want raw speed + simplicity? The length trick is often fastest in practice because the heavy lifting runs in C and there’s no Python-level loop.
- Need the actual duplicate value / index? Use the early-exit version (you can return `x` or the index when found).

Both are O(n) time and O(n) space on average; the difference is constant factors and early-exit behavior.

3) Brute force with two loops (your “left/right” idea)

> “I am traversing from 0 to n elements in every loop, Time Complexity: O(n) * O(n)”

Close—formally:

- Outer loop runs ~`n-1` times.
- Inner loop runs decreasing counts: `n-1, n-2, …, 1`.
- Total comparisons = `n(n-1)/2` = O(n²).

That's why it's slower than the hash-set and sort approaches.

4) Sorting method (for completeness)

```
def containsDuplicate(nums):
    nums.sort()
    for i in range(len(nums) - 1):
        if nums[i] == nums[i+1]:
            return True
    return False
```

- Time: sort is O(n log n); scan is O(n) → overall O(n log n).
- Space: O(1) extra if you sort in place.
- When to use: if you can mutate the array and want low extra memory; otherwise prefer a set.

Guaranteed O(nlog⁡n) and truly in place: Heapsort.

Usually fastest in practice, uses left/right pointers: Quicksort (use median-of-three or random pivot to avoid O(n2)O(n^2)O(n2)).

FYI: Python's `list.sort()` (Timsort) is O(nlog⁡n)O(n\log n)O(nlogn) and very fast, but it's not strict O(1)O(1)O(1) extra space (so not "in-place" in the academic sense).