Static array and dynamic array : Static arrays have a **fixed size** determined at compile-time and are typically stored on the **stack**, whereas dynamic arrays have a **flexible size** that can grow or shrink at runtime and are stored on the **heap.**.

Characteristics of Static Arrays

- **Immutability:** Once defined, the capacity is permanent. If you need more space than originally allocated, you must create a new, larger array and manually copy data over

Characteristics of Dynamic Arrays

- **Automatic Resizing:** Most modern implementations (like Python's `list` or C++'s `std::vector`) double in capacity when they become full.

✅ What you already know (half true)

When numbers are positive, `%` *looks like* "remainder":

- `10 % 3 = 1` (because 10 = 3*3 + 1)

So it feels like "remainder".

🔥 The real definition in Python

Python defines `%` using this identity:

$$a = (a//b) \times b + (a\%b)$$

This must be true **always**, even for negative numbers.

So `%` is not a "free" remainder — it is the value that makes this equation true **after Python has already decided** `a // b` **using floor**.

Step-by-step example: `-7 % 3`

Step 1: Compute `-7 // 3`

First do true division:

- `-7 / 3 = -2.333...`

Now apply **floor**:

- floor means "go DOWN on number line"
- Down from -2.333 goes to **-3**

So:

- `-7 // 3 = -3` ✅ (your answer #1 is correct)

Step 2: Use the formula to find `%`

Step 2: Use the formula to find `%`

Now plug into:

$$a = (a//b) \times b + (a\%b)$$

$$-7 = (-3) \times 3 + r$$

$$-7 = -9 + r$$

$$r = 2$$

✅ So:

```python
-7 % 3 = 2
```

Why is it not `-1`?

Because Python chooses the remainder so that:

- it matches the "floor division result"
- and (for positive divisor) the remainder is **0 to b-1**

Since divisor is `3` (positive), remainder must be one of:
`0, 1, 2`

So `-1` is not allowed in Python's modulo rule when divisor is positive.

⭐ The easiest way to remember this

When the divisor is positive (`% 3`, `% 5`, `% 10`...):

✅ The result of `a % b` is always in:

$$0 \text{ to } b-1$$

So:

- `-7 % 3` must be `0,1,2` → it is `2`

String Comparison in PYTHON:

2) Strings compare character by character (NOT numbers)

When you do:

```python
"10" > "2"
```

Python is **not** comparing 10 and 2.

It's comparing **text** like dictionary order.

How Python compares strings

It compares from left to right:

Compare first characters:

- `"10"` first char is `'1'`
- `"2"` first char is `'2'`

Now compare `'1'` vs `'2'`:

- `'1'` comes before `'2'`

So:

```python
"10" > "2"   # False
```

✅ Correct answer for #3 is **False**, not True.

ERRORS:

✅ PART 2 — What does this error mean?

```csharp
ValueError: invalid literal for int() with base 10: ''
```

This means:

👉 You tried to convert an **empty string** into an integer:

```python
int("")
```

This ALWAYS causes:

```csharp
ValueError: invalid literal for int()
```

DICT METHODS & LAMBDA EXPRESSIONS, LIST COMPREHENSION:

1) `sorted` vs `.sort`

- `sorted(iterable, key=..., reverse=...)` → **returns a new list**. Works on **any iterable** (list, set, dict, generator, …). Does **not** modify the original.
- `list.sort(key=..., reverse=...)` → **in-place** sort of an existing **list**. Returns `None`.

Same algorithm (Timsort), same time **O(m log m)** for m items.

2) What does `key=...` do?

Think **"decorate → sort → undecorate"**:

- For each element `x` in the iterable, Python computes `key(x)` once.

- It sorts the elements using those key values.
- It returns the elements themselves in that order.

Examples you'll use all the time:

- `sorted(words, key=len)` → by length
- `sorted(pairs, key=lambda kv: kv[1])` → by tuple's second item
- `sorted(names, key=str.casefold)` → case-insensitive
- **Top-K frequent** trick: `sorted(freq, key=freq.get, reverse=True)`
  Dict iteration yields **keys**; `freq.get(k)` gives that key's **count**. So this sorts **keys by their counts**.

Concrete mini-trace:

```
freq = {1:3, 2:5, 7:2}
sorted(freq, key=freq.get, reverse=True)
# keys are 1,2,7; key values are 3,5,2 → order by (5,3,2) → [2,1,7]
The key idea is: sorted iterates over its first argument. When that
first argument is a dict, iterating it yields its keys by default

sorted(freq, key=freq.get)                  # iterate dict → keys
sorted(freq.keys(), key=freq.get)             # explicit keys
sorted(freq.items(), key=lambda kv: kv[1]) # (key, value) pairs, sort
by value
```

3) Lambda: tiny, one-line functions

- Syntax: `lambda x: <expression using x>`
- Use in `key=...`, `map`, `filter`, or anywhere a short function is handy.

If lambdas feel fuzzy, define a normal function and pass it:

```
 def by_count(k): return freq[k]
sorted(freq, key=by_count, reverse=True)
```

-

4) Comprehensions: build a new collection from an iterable

- **List:** `[expr for x in it if cond]`
- **Dict:** `{k(x): v(x) for x in it if cond}`
- **Set:** `{expr for x in it if cond}`

Same result as a loop + append/assign; just shorter and often clearer.

Examples:

```
# extract the first k keys after sorting pairs by count
pairs = [(2,5), (1,3), (7,2)]
topk = [k for (k, _) in pairs[:2]]          # → [2, 1]

# dict comp: squares of even numbers
squares = {x: x*x for x in range(6) if x % 2 == 0}  # {0:0,2:4,4:16}
```

5) Dict essentials you'll use constantly

- `freq.get(k, 0)` → fetch with default (great for counting).
- `freq.items()` → `(key, value)` pairs.
- `freq.keys()` / `freq.values()` → views of keys/values.

- `setdefault(k, default)` or `defaultdict(list)` → painless grouping (e.g., anagrams).
    - In anagrams: key is something immutable like `tuple(sorted(word))`.

6) Same thing, three ways (top-k frequent)

Assume `freq = {1:3, 2:5, 7:2}`, `k=2`.

A. Sort keys by counts (no lambda):

```
def by_count(x): return freq[x]
keys = list(freq)
keys.sort(key=by_count, reverse=True)  # [2,1,7]
topk = keys[:k]                        # [2,1]
```

B. Sort items by value (with `itemgetter`):

```
from operator import itemgetter
pairs = list(freq.items())             # [(1,3),(2,5),(7,2)]
pairs.sort(key=itemgetter(1), reverse=True)  # by count
topk = [k for k, _ in pairs[:k]]       # [2,1]
```

C. One-liner you've seen:

```
topk = sorted(freq, key=freq.get, reverse=True)[:k]  # [2,1]
```

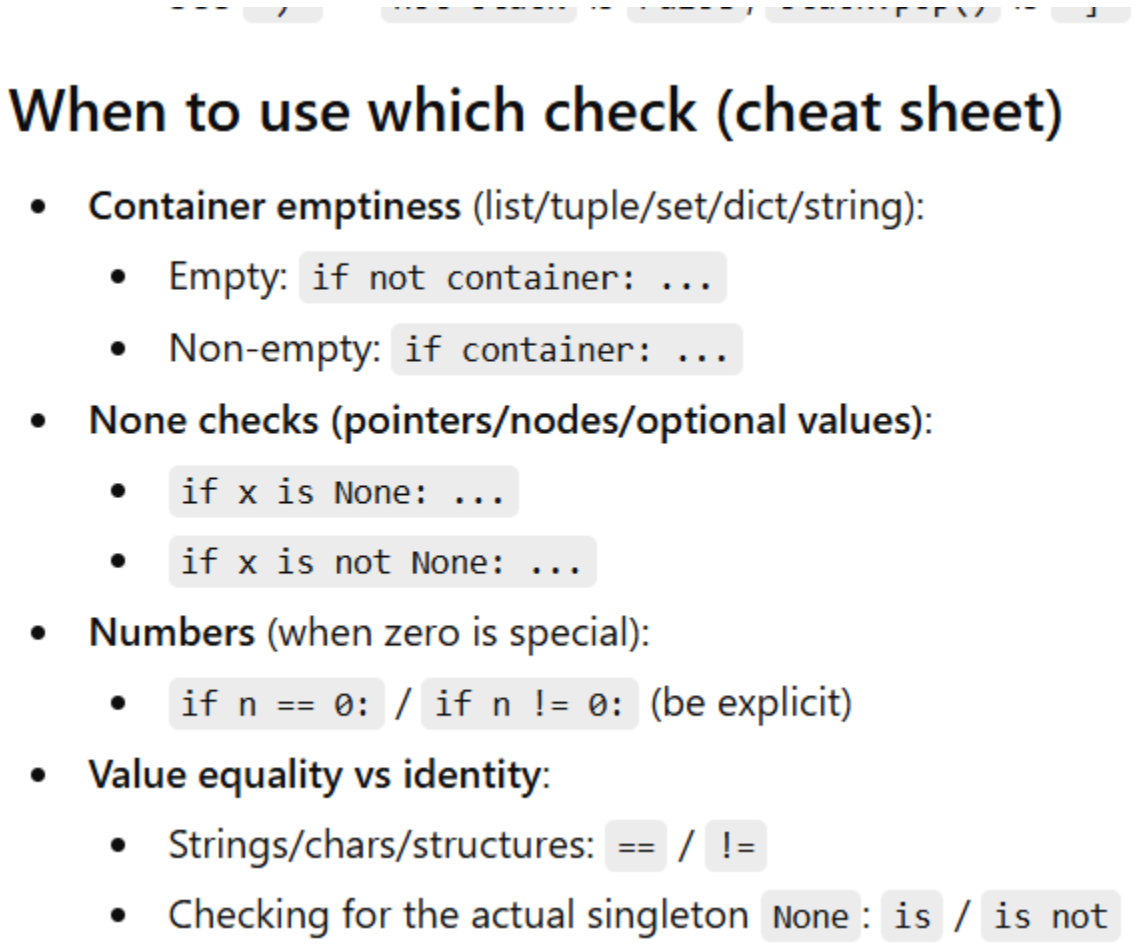
Time for all three sorting approaches: **O(u log u)** where `u = len(freq)`.
If **k ≪ u**, the **min-heap** alternative is **O(u log k)**:

```
import heapq
topk = heapq.nlargest(k, freq.keys(), key=freq.get)
```

7) How to get confident fast (tiny drills)

Do these without looking, then check:

1. Sort names by length (desc).
2. Sort list of `(name, score)` by score (desc).
3. From a dict `{char:count}`, produce the 3 highest-count chars.
4. Group words by their sorted-letters key (anagrams).
5. Build `{word: len(word)}` for words with length ≥ 4 via dict comp.



When to use which check (cheat sheet)

- **Container emptiness** (list/tuple/set/dict/string):
  - Empty: `if not container: ...`
  - Non-empty: `if container: ...`
- **None checks (pointers/nodes/optional values)**:
  - `if x is None: ...`
  - `if x is not None: ...`
- **Numbers** (when zero is special):
  - `if n == 0:` / `if n != 0:` (be explicit)
- **Value equality vs identity**:
  - Strings/chars/structures: `==` / `!=`
  - Checking for the actual singleton `None`: `is` / `is not`

Cheat sheet

Object you generate	How many exist	Time to generate all (typical)	Space (extra)
All subsequences of length n	2^n	$\Theta(2^n)$ to visit; $\Theta(n \cdot 2^n)$ if you copy/output elements	recursion $O(n)$ (plus output)
All subsets (power set) of n	2^n	$\Theta(n \cdot 2^n)$ (each subset avg size ~$n/2$)	$O(n)$
k-combinations from n	$\binom{n}{k}$	$\Theta(\binom{n}{k} \cdot k)$	$O(k)$
All combinations (all k)	2^n total	$\Theta(n \cdot 2^n)$	$O(n)$
k-permutations from n	$P(n,k) = \frac{n!}{(n-k)!}$	$\Theta(P(n,k) \cdot k)$	$O(k)$
All permutations of n	$n!$	$\Theta(n! \cdot n)$	$O(n)$
All substrings / subarrays of length n	$\frac{n(n+1)}{2} = \Theta(n^2)$	$\Theta(n^2)$ to enumerate index ranges; $\Theta(n^3)$ if you *materialize every substring* (sum of lengths is $\Theta(n^3)$)	$O(1)$ if you yield indices
Distinct substrings	$\leq \Theta(n^2)$	Naive set insert: $\tilde{O}(n^3)$ if slicing copies; suffix structures can reduce to $\tilde{O}(n)$–$\tilde{O}(n^2)$	varies
Valid parentheses strings of n pairs	Catalan $C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$	$\Theta(C_n \cdot n)$ (each output length $2n$)	$O(n)$

> **Common mix-up:**
> - **"Subsequence count is 2^n (not contiguous)."**
> - **"Substring count is $\Theta(n^2)$ (contiguous).**
>   **If you actually build all substring strings, total characters across outputs is $\Theta(n^3)$, hence $\Theta(n^3)$ time."**

Notes & rules of thumb

- **Output-sensitive**: you can't beat the count of outputs. Big-O often equals "#outputs × cost per output".
- **Copy vs. yield**: If you *copy* each result (e.g., build a list/string), multiply by its size. If you *yield indices* or an iterator, you avoid that factor.
- **With duplicates**:
  - *Unique subsets/subsequences*: still worst-case 2^n; add a sort $O(n \log n)$ and skip-dupe logic.
  - *Unique permutations*: $\leq n!$; with counts $n! / \prod f_i!$.
- **Feasibility heuristics (interview sanity checks)**:
  - $n!$: usually only feasible up to ~10–11.
  - 2^n: feasible up to ~20–25 (depends on constant factors).
  - n^2: fine up to ~10^5 with care; n^3: usually small n (≤ 2–3k) only.

1. **How many strings exist** of length $2n$ over the alphabet $\{(,)\}$.
2. **How many of those are valid parenthesis strings**.

Generate Parentheses (LeetCode 22)

- Each answer has **length** $2n$ (n opens + n closes).
- If you ignore validity, there are 2^{2n} different strings (two choices at each of $2n$ positions).
- But **most are invalid**. The number of **valid** strings is the **Catalan number**:

$$C_n = \frac{1}{n+1}\binom{2n}{n} \quad (1, 1, 2, 5, 14, 42, \dots)$$

  - For **n = 2**, $C_2 = 2$ → the only valid strings are: `"(())"` and `"()()"`.
    (Not 4.)

Great q. The short answer:

Because there are $2n$ positions and each position has 2 choices, the count is 2^{2n}, not 2^n.

Why 2^{2n}?

General rule: **#strings = (alphabet size)^length**.

Here the alphabet is `{ '(', ')' }` → size = 2.

Each valid/invalid string we might generate has length $2n$.

So the number of *all possible* strings (ignoring validity) is:

$$2^{2n} = \underbrace{2 \times 2 \times \cdots \times 2}_{2n \text{ times}}$$

Concrete examples

- $n = 1 \Rightarrow$ length $2 \Rightarrow 2^2 = 4$ strings: `()`, `)(`, `((`, `))`.
- $n = 2 \Rightarrow$ length $4 \Rightarrow 2^4 = 16$ strings (most are invalid).
- $n = 3 \Rightarrow$ length $6 \Rightarrow 2^6 = 64$ strings.

(Also note $2^{2n} = (2^2)^n = 4^n$.)

Don't mix up $2n$, 2^n, and 2^{2n}

- $2n$ is linear in n (just the length).
- 2^n is exponential (choose/skip per **n** positions).
- 2^{2n} is exponential with $2n$ positions (bigger than 2^n for $n \geq 1$).

0) Golden rules

- **Contiguous windows** (substrings/subarrays) → about n^2 of them.
- **Pick/skip each element** (subsequences/subsets) → 2^n of them.
- **Order matters** (permutations) → factorials.
- **Choose exactly** k (combinations/permutations of size k) → $\binom{n}{k}$ or $P(n,k)$.

When you **output/copy** every object, multiply by its **size** (often $O(k)$ or $O(n)$).
When you just **enumerate indices** (or yield), you can avoid that extra factor.

- **Both substrings and subarrays must consist of elements that are consecutive in the original data structure**
- Subsequence: A subsequence is a sequence that can be derived from another sequence by removing zero or more elements, without changing the order of the remaining elements.
- For the same example, there are 15 sub-sequences. They are (1), (2), (3), (4), (1,2), (1,3),(1,4), (2,3), (2,4), (3,4), (1,2,3), (1,2,4), (1,3,4), (2,3,4), (1,2,3,4). More generally, we can say that for a sequence of size n, we can have (2^n-1) non-empty sub-sequences in total.
  A string example to differentiate: Consider strings "geeksforgeeks" and "gks". "gks" is a subsequence of "geeksforgeeks" but not a substring. "geeks" is both a subsequence and subarray. Every subarray is a subsequence. More specifically, Subsequence is a generalization of substring.
- A subarray or substring will always be contiguous, but a subsequence need not be contiguous. That is, subsequences are not required to occupy consecutive positions within the original sequences. But we can say that both contiguous subsequence and subarray are the same.
- A subarray is a contiguous part of the array. An array that is inside another array.

2) Subsequence (not necessarily contiguous, order kept)

For `"abc"`:

All subsequences (keep each char or skip it):
`"" , a, b, c, ab, ac, bc, abc` → **8** = 2^3.

- **Count:** 2^n.
- **Visit only (no copying):** $\Theta(2^n)$.
- **If you copy each subsequence:** total length across outputs
  $\sum_{k=0}^{n} k\binom{n}{k} = n2^{n-1} = \Theta(n2^n)$.

> Big takeaway: **subsequence ≠ substring**.
> - "**Substring = contiguous, $\Theta(n^2)$.**"
> - "**Subsequence = pick/skip, 2^n.**"

---

3) Subset (power set) of a set (order ignored)

For $\{1, 2, 3\}$:
`∅, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}` → **8** = 2^3.

- **Count:** 2^n.
- **Time (copying each subset):** $\Theta(n2^n)$ (avg subset size ≈ $n/2$).

> Subset vs subsequence: counts match 2^n, but subsets ignore order; subsequences preserve order of the original sequence.

Tiny examples side-by-side (n=3; items a, b, c)

- **Substrings (contiguous on "abc"):**

  `a, b, c, ab, bc, abc` → 6 = $3 \cdot 4/2$.
- **Subsequences ("abc", keep/skip):**

  `∅, a, b, c, ab, ac, bc, abc` → 8 = 2^3.
- **Subsets of {a,b,c} (order ignored):**

  same 8 as above but sets: `∅, {a}, {b}, {c}, {a,b}, {a,c}, {b,c}, {a,b,c}`.
- **k-combinations (k=2):** `{a,b}, {a,c}, {b,c}` → 3 = $\binom{3}{2}$.
- **All permutations:**

  `abc, acb, bac, bca, cab, cba` → 6 = $3!$.
- **k-permutations (k=2):**

  `ab, ac, ba, bc, ca, cb` → 6 = (P(3,2)=3·2`.

D) Quick memory hooks

- **Contiguous** windows → **Substrings/Subarrays** → ~ **n^2**.
- **Pick/Skip** with order preserved → **Subsequences** → **2^n**.
- **Choose k, order ignored** → **Combinations** → **C(n,k)**.
- **Choose k, order matters** → **Permutations** → **P(n,k)**.
- **All permutations** → **n!**.

E) One-line clarifications to your points

- "Order ignored means I can write `BA` instead of `AB`."
  - For **combinations/subsets**, `{A,B} == {B,A}` (same object).
  - For **permutations**, `AB` and `BA` are **different** outputs.
  - For **subsequences**, whether `BA` exists depends on the original sequence's order.
- "Substrings and subarrays are n(n+1)/2; why $O(n^3)$?"
  - Count is $\Theta(n^2)$. If you **return strings**, total copying is $\Theta(n^3)$. If you just output index ranges, $\Theta(n^2)$.

Great question! Here's a quick cheat-sheet for **Python dict** lookups when the key might be missing.

What happens if the key is missing?

Operation	If key exists	If key is missing	Raises error?	Mutates dict?	Notes
`d[k]`	returns the value	**KeyError**	✅	❌	Direct indexing ("must exist").
`d.get(k)`	value	`None`	❌	❌	Read-only, safe default `None`.
`d.get(k, default)`	value	`default`	❌	❌	Read-only with custom default.
`k in d`	`True` / `False`	`False`	❌	❌	Membership test.
`d.setdefault(k)`	value	**inserts** `k: None`, returns `None`	❌	✅	Be careful: missing key is added with `None`.
`d.setdefault(k, default)`	value	**inserts** `k: default`, returns `default`	❌	✅	"Create-or-return existing" in one call.
`d.pop(k)`	removes & returns value	**KeyError**	✅	✅	Use when you must remove and it must exist.
`d.pop(k, default)`	removes & returns value	returns `default`	❌	✅	Safe remove with fallback.
`del d[k]`	deletes entry	**KeyError**	✅	✅	Like `pop(k)` but returns nothing.

STRINGS:

- Loop + string concatenation → bad (O(n²)).
- Loop + list.append → good (O(n)).

- **You don't need to "avoid a loop" entirely—you just need to avoid concatenating strings in the loop. Gather pieces, then join once.**

Rule of thumb:
If you're building a long string from many pieces, use a list (or builder) + one final join. That gives you overall O(n) time.

The `appendleft()` method is used with a `collections.deque` object in Python to add an element to the left end (front) of the deque.

`split(" ")` splits only on a single space and keeps empty strings when there are multiple spaces.
Example: `"a good example".split(" ")` → `['a', 'good', '', '', 'example']`

LinkedList: Not contiguous memory , you can increase/decrease the size of the array. Used in Stack/ Queue - No Fixed Size in /stack and Queue

In real life, you always use it in a browser.

Depending on the system, memory space used is different