Expertise in Python and Python-based ML frameworks (e.g.,LangGraph, Pydantic AI, PyTorch)

So Based on previous mistakes, I made mistakes when defining classes and functions inside class, Local variables errors in functions, when to use self , when to access using self.

- After i pass, the last i elements are already sorted and don't need to be checked again.
- BUT → **slicing** in Python always creates a new list, it does not just give you a view. (Allocates a new list of the same size as l. Iterates through all n elements of l, one by one, in reverse order. Copies each element into the new list)

Slicing (`lst[start:stop]`) — creates a new list (shallow copy)

Key fact: `lst[a:b]` returns a *new list object* containing references to the same items that were in the slice (shallow copy). So reassigning the slice result does not affect the original list — but mutating *inner* (nested) objects may show up in both because only the outer list was copied.

Example — simple elements:

```python
a = [1,2,3,4,5]
b = a[1:4]
print(b)        # [2,3,4]
print(a is b)   # False -> b is a new list
```

Slicing copies the elements (shallowly), so altering `b` won't change `a`:

```python
b.append(99)
print(a)  # [1,2,3,4,5] unchanged
print(b)  # [2,3,4,99]
```

But **shallow copy caveat** (nested/mutable items):

```python
a = [[1], [2]]
b = a[:]        # or a[0:2] — also a shallow copy
print(a is b)   # False (different outer lists)
b[0].append(99)
print(a)        # [[1, 99], [2]]  <-- inner list changed in both
print(b)        # [[1, 99], [2]]
```

Why? `a[:]` made a new outer list but copied references to the *same* inner lists. Mutating an inner list affects both.

If you need a full independent copy (no shared nested objects), use `copy.deepcopy()`:

Slice assignment vs slicing to read

Be careful: `lst = lst[1:3]` *rebinds* the variable to a new list. But `lst[1:3] = [100, 200]` *mutates the original list in-place* (replaces that slice).

Example:

```python
a = [1,2,3,4]
a[1:3] = ['a','b']   # changes the original list
print(a)  # [1, 'a', 'b', 4]
```

`del a[1:3]` removes that slice in-place as well.

Operation	Code	Effect
Slice read	`b = a[1:4]`	Creates new list (shallow copy). Original unchanged.
Slice assignment	`a[1:4] = [...]`	Modifies the original list. May change length.

- **reverse** – returns a LIST

🧠 3 Comparison Table

Aspect	`list.reverse()`	`reversed(list)`
Type	List method	Built-in function
Works on	Lists only	Any reversible sequence
Returns	`None`	Iterator
In-place?	✅ Yes	❌ No
Original list changes?	✅ Yes	❌ No
Space complexity	O(1)	O(1) (iterator only)
Time complexity	O(n)	O(n) (when iterated)
Use case	Modify existing list	Get reversed view or make new reversed list

When we say a method like

```python
a.reverse()
```

"**returns None**", it means:

> the method does *not* produce or return any new value that you can store in a variable — it just performs an action (modifies the list in place).

Let's see what that actually looks like 👇

Example

```python
a = [1, 2, 3, 4]
result = a.reverse()
print(result)
```

Output:

```
None
```

🧠 2 Common list methods that return `None`

Method	What it does	Modifies original list?	Returns
`.append(x)`	Adds one element at end	✅ Yes	`None`
`.extend(iterable)`	Adds multiple elements	✅ Yes	`None`
`.insert(i, x)`	Inserts element at index	✅ Yes	`None`
`.remove(x)`	Removes first occurrence of x	✅ Yes	`None`
`.pop(i)`	Removes and **returns** element	✅ Yes	✅ element (not None)
`.sort()`	Sorts list in place	✅ Yes	`None`
`.reverse()`	Reverses list in place	✅ Yes	`None`
`.clear()`	Removes all elements	✅ Yes	`None`

🧩 4️⃣ Contrast: Functions that return new objects

Function	Original changes?	Returns new object?
`sorted(list)`	❌ No	✅ Yes
`reversed(list)`	❌ No	✅ Yes (iterator)
`copy.copy(list)`	❌ No	✅ Yes
`list()`	❌ No	✅ Yes

- **reversed** – returns an iteration
- **remove** – method removes a value, not an index – So it always starts from first ->O(n)
- **POP** – Removes an item at that index -> **O(n)** if not last element, **O(1)** if last element

Python lists are **dynamic arrays** under the hood (not linked lists).So when you remove something **from the middle or front**,
all elements **after that index** must be **shifted left** to fill the gap

✅ Example 3: Multiple same values

```python
letters = ['a', 'b', 'a', 'c', 'a']
letters.remove('a')
print(letters)
```

✅ Output:

```css
['b', 'a', 'c', 'a']
```

👉 Only the **first** '**a**' is removed — the others stay.

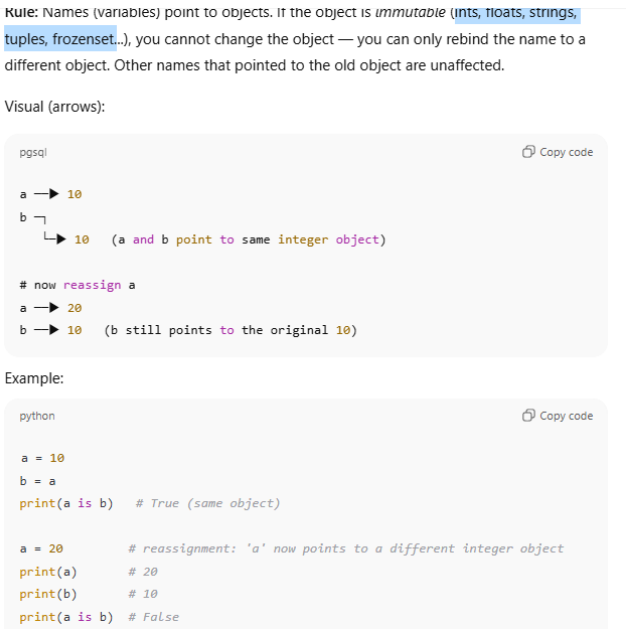
- **APPEND & EXTEND METHOD**: **append(l)** puts the entire box into your new box as a single item (just 1 slot).
  extend(l) takes every card out of the box and puts each card into your new box (needs many slots — one per card). // reverse.extend([l[element]]) (EXPECTS AN ITERABLE-> reverse.extend(l[element]). That's why append(l) is O(1) extra space (one slot), while extend(l) is O(n) extra space (one slot per element).

append() → nests the list → constant space → reference stored. // [10, 20, 30, [40, 50, 60, 70]]
extend() → flattens the list → linear space → all elements copied. // [10, 20, 30, 40, 50, 60, 70]

- append is natural when adding **one element**.
- extend is natural when adding **multiple elements at once**. Takes that **1-element list**, then unpacks it and adds that **same one integer**
- Use `append(x)` when you're adding **one thing**.
- Use `extend(iterable)` when you're adding **many things at once**

- If an object is **immutable**, reassignment just redirects the arrow — original unaffected.

immutable: ints, floats, strings, tuples, frozenset.



- If an object is **mutable**, in-place changes affect everyone pointing to it.

Rule: If the object is *mutable* (list, dict, set, bytearray, most custom objects), operations that modify the object in-place change that single object — and all names that reference that object see the change.

Visual:

```pgsql
lst1 ──▶ [1,2,3]
lst2 ──┘
# both point to same list object

lst1.append(4)
# the single list object now is [1,2,3,4]
# both lst1 and lst2 see the change
```

```
lst1 = [1,2,3]
lst2 = lst1
lst1.append(4)
print(lst2)   # [1, 2, 3, 4] — changed because lst1 and lst2 refer to same list
```

Contrast with rebinding:

```python
lst1 = [1,2,3]
lst2 = lst1
lst1 = [9,9]   # lst1 now points to a new list object
print(lst2)    # [1,2,3]  — still the original list
```

So **mutating** (e.g. append, pop, sort, extend, clear, item assignment lst[0]=x) modifies the object. **Reassigning** the name binds it to a new object and does not change the original.

Check NESTED LIST & number difference:

If a list has only 1 number, then the product of all other numbers = product of nothing. Product of nothing =1 (rule) thats why the code gives [1]. So , "**product of nothing = 1**" just means -when there are no numbers to multiply, the result is **1** by definition — because 1 is the neutral value for multiplication.

A variable must be assigned every time the function is run, no matter how the if-else conditions go.

If a variable is assigned only in some conditions but used after those conditions, Python will raise an error when that variable is used without assignment

```python
def f(x):
    if x > 0:
        y = 10
    print(y)
```

Let's break it down:

When you call `f(1)`:

- The `if` condition is true, so `y = 10` happens.
- Then `print(y)` prints `10`.

  ✅ Works fine.

When you call `f(-1)`:

- The `if` condition is false, so the line `y = 10` is skipped.
- Python reaches `print(y)` — but `y` has never been assigned in this run!

  💥 Boom →

```pgsql
UnboundLocalError: local variable 'y' referenced before assignment
```

✅ **Fix:**

Initialize `y` with a default value **before** the `if`:

```python
def f(x):
    y = None     # ensures y exists no matter what
    if x > 0:
        y = 10
    print(y)
```

Now:

- If `x>0`, you'll print `10`.
- If `x<=0`, you'll print `None`.

  No error, because `y` is always assigned.

🧩 Example 2: The global / local confusion

```python
x = 5

def g():
    print(x)
    x = 3

g()
```

At first glance, you might think:

> "There's a global `x=5`, so `print(x)` should print 5."

But Python says:

```pgsql
UnboundLocalError: local variable 'x' referenced before assignment
```

Why?

Here's what's happening internally:

When Python compiles the function `g`, it sees **an assignment to** `x` inside it:

```python
x = 3
```

That's enough for Python to say:

> "Okay, `x` is a local variable for this function."

So when it hits `print(x)`, Python **does not look at the global x**.
It looks for a *local x* — but that local x hasn't been assigned yet → error.

✅ **Fix 1: If you meant to use the global variable**

Tell Python explicitly using the `global` keyword:

```python
x = 5
def g():
    global x
    print(x)   # uses global x
    x = 3      # updates global x

g()
print(x)   # now global x has changed to 3
```

✅ **Fix 2: If you wanted a local variable**

Just move `print(x)` after the assignment:

```python
def g():
    x = 3
    print(x)
```

Example 3: Variable defined only inside a loop

```python
for i in []:
    last = i
print(last)
```

Here:

- The loop body never runs because the list is empty.
- So `last = i` never executes.
- When you try `print(last)`, there is no `last` variable →

```pgsql
NameError: name 'last' is not defined
```

If the list wasn't empty:

```python
for i in [1,2,3]:
    last = i
print(last)
```

✅ Now `last` = 3 (the last value assigned).

Summary — "Rules of variable life"

Situation	What happens	Fix
Variable assigned in only some branches	May cause `UnboundLocalError` when branch skipped	Always assign a default
Variable assigned later in function	Python still treats it as local from start	Move reads after assignment
Variable name exists globally but reassigned in function	Python ignores global and treats local version	Use `global` if needed
Variable only created in loop that may not run	Will cause `NameError` if loop skipped	Initialize before loop

Here are the mistakes with respect to usage of classes and functions in your Word Search solution code:

1. Incorrect Passing of self

- You were calling self.helper(self, board, row, col, w) instead of just self.helper(board, row, col, w).
- In Python class instance methods, self is implicitly passed and should NOT be passed explicitly when calling instance methods.
- Mistake here caused self to be misinterpreted as the parameter board, breaking length checks.

2. Variable Scope for word

- Your recursive helper function used the variable word but it was not passed as a parameter or accessed via self.word.

- This caused NameError because word was undefined in the local scope of helper.
- You must either:
    - Pass word to helper explicitly every time.
    - Or store the word in self.word in exist and access via self.word in helper.

3. Positional Arguments After Keyword Arguments

- In self.helper, you had keyword arguments like row=0, col=0, w=0 followed by a positional argument word.
- **Python syntax requires all positional arguments to come before keyword arguments.**
- You must reorder to pass arguments entirely positionally, or entirely as keywords.

4. Using len() on Integer Variables

- row and col are integers representing indices, but you used len(row) and len(col) in conditions.
- Only strings or collections have length; mistake caused TypeError.
- You should compare row or col directly to the len() of the grid dimensions.

5. Recursion & Backtracking Logic in helper

- You had complex row/col resetting logic (row = 0, col = 0) inside helper, which caused infinite recursion.
- Backtracking in DFS-based grid search works by simply returning from invalid moves and recursion stack naturally restores previous state.
- You neither marked visited cells nor had clean base cases, which caused infinite recursion and revisit cycles.

Summary Table of Mistakes

Mistake	**Recommended Fix**
Extra self parameter in calls	Call methods like self.helper(...)
undefined word in helper	Pass word argument or use self.word
Positional after keyword args	Pass positional args first, then keyword or all positional

Mistake	Recommended Fix
len() used on int indices	Use row or col directly without len()
Incorrect recursion base cases	Use clear base case, remove wrap logic, mark visited cells

---

The core issues are understanding Python class method calls, variable scoping, and proper recursion/backtracking logic.

SORTINGS

BUBBLE_SORT: Adding the swapped variable to bubble sort improves its best-case time complexity from O(n^2) to O(n) by allowing the algorithm to detect if the array is already sorted and exit early, rather than always making a full set of comparisons

0) "In-place" vs space complexity (why it's still O(n))

- In most tutorials, **"in-place merge sort" just means "mutates the input list"**, i.e., it writes the merged values back into arr.
- **It still needs O(n) extra memory** to perform the *merge* step (you either slice to make left/right copies, or use a single auxiliary buffer).
- Plus **O(log n)** call-stack space from recursion.
- A *true* O(1)-extra merge sort exists, but it's complicated/slow and rarely used.

So even if you "don't use any data structure" explicitly, Python **slicing creates new lists** behind the scenes, and many in-place versions still allocate temporary arrays. That's the O(n).

1) What does this do?

```
left  = merge_sort(arr[:mid])
right = merge_sort(arr[mid:])
```

- arr[:mid] and arr[mid:] **create new lists** (copies of the halves).
- merge_sort(...) returns **new sorted lists**.
- left and right point to those new lists.
  Original arr is unchanged.

2) What about calling without keeping the result?

```
merge_sort(arr[:mid])
merge_sort(arr[mid:])
```

- Same slices are created and sorted, **but you throw the results away**.
- If your merge_sort is the "returns-new-list" version (pure/functional), this has **no effect** on arr.
- Only an **in-place** implementation (one that mutates the list it receives) would have an effect when you ignore the return value.

3) Why does this print a sorted list even though arr stays the same?

print(merge_sort(arr))

- The function returns a new sorted list, and you print it.
- arr itself is **still unsorted**.
  If you later do print(arr), you'll see the original.

4) Why does this not sort arr?

merge_sort(arr)

print(arr)

- With the "returns-new-list" version, you didn't capture the result:
- arr2 = merge_sort(arr)
- print(arr2) # sorted
- print(arr) # original, unchanged
- If you want arr itself to become sorted, you need an **in-place** version:
- merge_sort_in_place(arr)
- print(arr) # now sorted

⚖️ Final Comparison

Type	Extra Space	Recursive Stack	Total Space	Time Complexity	Practical?
Normal Merge Sort	O(n)	O(log n)	**O(n)**	O(n log n)	✅ Fast and common
In-place Merge Sort	O(1)	O(log n)	**O(log n)**	O(n log n) but slow constants	⚠️ Rarely used
Bubble Sort	O(1)	O(1)	**O(1)**	$O(n^2)$	❌ Slow for large n

Mental model cheat-sheet

- **Slicing:** arr[a:b] → **new list** (copy). Costs memory/time.
- **Pure/functional version:** returns a sorted **new list**.
    - Use: arr2 = merge_sort(arr).
    - Printing merge_sort(arr) won't change arr.
- **Mutating ("inline") version:** takes indices, writes back into arr.
    - Use: merge_sort_in_place(arr) then print(arr).
    - Still **O(n) extra space** (aux buffer or temporary halves) + **O(log n)** stack.
- **True O(1) extra memory** merge sort exists but is complex and not typical.

If you want, I can instrument the in-place version to print exactly how lo/mid/hi and the array change at each merge on your sample input so you can watch it happen step by step.

⚖️ Direct Comparison

Expression	What it does	Recursion Behavior
`arr[:mid]`	Always takes first half of the **current sub-array**	Shrinks each call → ✅ terminates
`arr[low:mid]` with `low=0`	Always starts from index 0 again (not relative)	Doesn't shrink → ❌ infinite recursion

🔹 **Step 1: What slicing (arr[:mid], arr[mid:]) does in Python**

When you do:

left = arr[:mid]

right = arr[mid:]

Python creates **brand-new lists**.

- These are **copies** of the portion of the array, not "views" into the same memory.
- Internally, Python allocates a new list object and then copies elements from the original one into it.

So for example:

arr = [5, 2, 8, 1]

left = arr[:2] # [5, 2] → new array

right = arr[2:] # [8, 1] → new array

Now left and right are **completely independent lists**. Changing arr later won't affect them.

🔹 **Step 2: Why Program 1 = many arrays alive**

In Program 1:

left = l[low:mid] # creates new copy

right = l[mid:] # creates new copy

left = merge_sorted(left) # returns *another* new array

right = merge_sorted(right) # returns *another* new array

return merge_sorted_list(left, right) # creates *another* new merged array

👉 So for each recursion:

1. **Slicing** → creates new arrays (left, right)
2. **Recursion return** → returns *another* new array
3. **Merging** → builds *yet another* array

That's why **so many arrays pile up** → total copying over the recursion = O(n log n).

🔹 **Step 3: Why Program 2 is better**

In Program 2:

left = arr[:mid] # slicing still makes a new array

right = arr[mid:] # slicing still makes a new array

merge_sorted(left)

merge_sorted(right)

return merge_sorted_list(left, right, arr)

⚠️ Notice the difference:

- Yes, slicing **still makes new arrays** (left and right).
- BUT after recursion, you don't return a new array — instead you merge **directly into the original arr**.

So:

- You don't create an extra merged array at every level.
- The **original array gets reused** as the output buffer.

👉 That's why total memory is bounded by O(n) at any point, instead of piling up to O(n log n).

🔹 **Internal Mechanics of Slicing**

Suppose:

arr = [5, 2, 8, 1]

left = arr[:2]

Python internally does something like:

1. Allocate a new empty list [] of length 2.
2. Copy arr[0] → left[0], arr[1] → left[1].
3. Return that new list.

So left is a **separate array** in memory, not a pointer to the original slice.

🔹 **Summary**

- **Slicing always makes new arrays** in Python (not like NumPy views).
- **Program 1** → keeps building and returning new arrays → total allocations O(n log n).
- **Program 2** → also slices, but merges results **into the same original array**, so allocations don't pile up → O(n) peak space.

👉 So the optimization in Program 2 isn't "slicing is avoided" (you still slice).
It's "we don't create a new merged array at every level — we reuse the original arr to hold results."

✅ Quick check:

If I had written:

```python
buckets = [[]] * k
```

This would create **k references to the same list** (bad).

- Modifying one bucket would modify them all.
- Example:

```python
buckets = [[]] * 3
buckets[0].append(5)
print(buckets)  # [[5], [5], [5]]
```

But with **list comprehension**, each `[]` is a fresh, separate list:

```python
buckets = [[] for _ in range(3)]
buckets[0].append(5)
print(buckets)  # [[5], [], []]
```

[2]:
```
bucket=[]
for i in range(10):
    bucket.append([])
print(bucket)
```

```
[[], [], [], [], [], [], [], [], [], []]
```

[4]:
```
bucket=[]
for i in range(10):
    bucket.extend([])
print(bucket)
```

```
[]
```

- Here, the iterable you passed to extend is [] (an **empty list**).
- An empty list has **no elements**, so extend does nothing.
- That's why your bucket stays empty.

SORTING

Bubble Sort: We will compare element next to that element, where it takes O(n^2) Time complexity but if we check that the array is sorted or not at first using Swapped =False. So here after completing one inner loop if there is no swap happens means that the array is already sorted which takes O(n) time.

Merge Sort:

Quick Sort: Takes first element as pivot and then next we will check highest and lowest element compared with pivot

Selection sort: Select minimums.
Step 1: We will take first minimum element in the array and then we will swap it with first element, and first element goes to the minimum element (first element is in Correct Position)

Step 2: I know that the extremely smallest element is in first place, and then we need to find second minimum element from next position (first 2 elements are sorted in the array)

Step 3: Need to find 3rd minimum element from position 3 because first 2 elements are sorted.

If we have n elements, we will take n-2 steps to sort the elements in ascending order.

Swap at node 0 & min node -> [0,n-1]

Swap at node 1 & min node -> [1,n-1]

Swap at node 2 & min node -> [2,n-1]

Swap at node n-2 & min node ->

Rule of Thumb

- **If inner loop runs a *fixed full n times* for every outer iteration**, we multiply:

O(n)×O(n)=O(n^2)

If inner loop shrinks depending on outer index, we add the actual counts instead of multiplying.

Here:

- When `i=0`, inner loop runs **n-1 times**
- When `i=1`, inner loop runs **n-2 times**
- ...
- When `i=n-2`, inner loop runs **1 time**
- When `i=n-1`, inner loop runs **0 times**

So total work =

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$$

That simplifies to **$O(n^2)$**.

```python
for i in range(n):          # O(n)
    for j in range(i+1, n): # O(n-i-1) → shrinks
        print(i, j)
```

🔹 First: `O(n) + O(n²)` → What happens?

When we have **different terms** in Big-O, we keep only the **dominant (largest) one** as `n → ∞`.

$$O(n) + O(n^2) = O(n^2)$$

✅ Because as `n` grows, the quadratic term dwarfs the linear term.

Example: if `n=1000`,

- `O(n)` ≈ 1000
- `O(n²)` ≈ 1,000,000
  So we just keep **O(n²)**.

Example: `n = 6`

Outer loop → `i = 0 … 4`

`i` value	inner loop = `range(i+1, n)`	length of inner loop
0	range(1, 6) → [1,2,3,4,5]	5 = n-1
1	range(2, 6) → [2,3,4,5]	4 = n-2
2	range(3, 6) → [3,4,5]	3 = n-3
3	range(4, 6) → [4,5]	2 = n-4
4	range(5, 6) → [5]	1 = n-5

So the inner loop length is always:

$$n - (i + 1) = n - i - 1$$

That's why we say **inner loop complexity = O(n - i - 1)**, and as `i` increases, the number shrinks.

BUCKET SORT:

Intuition (short)

Think of bucket sort like sorting mail by ZIP-code prefixes before putting letters in order.

1. You first **scatter** items into bins (buckets) according to their value range.
2. Each bucket now contains only items close in value, so sorting each bucket is cheap.
3. Finally **collect** buckets in order → fully sorted list.

Bucket sort is great when values are fairly **uniformly distributed** across the range.

Algorithm (step-by-step)

1. Pick number of buckets `k` (common choices: `k = n`, or `k` proportional to `n`).
2. Find `min_v` and `max_v` of the array.
3. For each element `x`, compute a bucket index `idx` that maps `x` into a bucket representing a subrange of `[min_v, max_v]`.
   - A common mapping (real numbers):

     csharp Copy code

     ```
     idx = int((x - min_v) / (max_v - min_v) * k)
     if idx == k: idx = k-1
     ```

   - For values already in [0,1), you can use `idx = int(x * k)`.
4. Append `x` to `buckets[idx]`. (Insertion keeps original order so stability is possible.)
5. Sort each bucket individually (insertion sort is common because buckets are small).
6. Concatenate buckets from index `0` to `k-1` → result.

Worked example 1 — classic floats (CLRS)

Input:

```ini
A = [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]
k = 10   # buckets for ranges [0,0.1), [0.1,0.2), ..., [0.9,1.0)
```

Mapping since values in [0,1): `idx = int(x * k)`.

Distribute:

- 0.78 → idx = 7 → bucket[7] = [0.78]
- 0.17 → idx = 1 → bucket[1] = [0.17]
- 0.39 → idx = 3 → bucket[3] = [0.39]
- 0.26 → idx = 2 → bucket[2] = [0.26]
- 0.72 → idx = 7 → bucket[7] = [0.78, 0.72]
- 0.94 → idx = 9 → bucket[9] = [0.94]
- 0.21 → idx = 2 → bucket[2] = [0.26, 0.21]
- 0.12 → idx = 1 → bucket[1] = [0.17, 0.12]
- 0.23 → idx = 2 → bucket[2] = [0.26, 0.21, 0.23]
- 0.68 → idx = 6 → bucket[6] = [0.68]

Now sort each bucket (usually trivial small sorts):

- bucket[1] [0.17, 0.12] → [0.12, 0.17]
- bucket[2] [0.26, 0.21, 0.23] → [0.21, 0.23, 0.26]
- bucket[7] [0.78, 0.72] → [0.72, 0.78]
  (others are either singletons or empty)

Concatenate buckets in order:

```csharp
[0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]
```

🔹 1. "Pick `k` proportional to the range"

When data are **integers** and you know the **min and max**, you can decide how many buckets to use based on that **range**.

👉 Example:

```ini
Array = [29, 25, 3, 49, 9, 37, 21, 43]
min = 3, max = 49
Range = 49 - 3 = 46
```

- If I say **k proportional to the range**, it means:
  - If range = 46, I might pick k = 5, k = 10, or even k = 46.
  - For example, one bucket per 10 values → k = range / 10 = ~5 buckets.
  - Or one bucket per integer → k = range = 46 buckets (in fact, this basically becomes **counting sort**).

👉 The idea: you don't have to fix k = n; instead, you can pick k that "matches" how spread out the numbers are.

🔹 2. "Uniform floats"

This refers to **floating-point numbers (decimals)** that are **spread evenly across a range**, usually **[0, 1)**.

- **Uniform** = each sub-range is equally likely to have elements.
- Example: `[0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]`
  - These values look random between 0 and 1.
  - They don't "cluster" heavily in one corner (like all being near 0.9).
- Why important?
  - If data is **uniform**, buckets get **balanced loads**.
  - If data is **skewed** (say 90% of numbers are between 0.9 and 1.0), then most elements fall into one bucket → bad performance.

🔹 1. What if `min = max` in the array?

Example:

```python
array = [7, 7, 7, 7]
min = 7
max = 7
```

👉 Problem:

- The **range = max - min = 0**.
- When placing elements into buckets, we usually do something like:

  ```python
  index = (value - min) * k // (max - min + 1)
  ```

  But if `max = min`, the denominator becomes `0`. ❌ Division by zero.

👉 Solution:

- If all values are the same, the array is already **sorted**.
- So you can just return it directly without doing bucket sort.
- In practice, we add a condition:

  ```python
  if min == max:
      return array
  ```