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

**LEETCODE PROBLEMS: 2095, 167, 238, 230,98,79,695**

**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.**

**Validate Binary search tree – Do it once every month,  
Rotate array – Need to do next week**

* 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)
* **reverse** – returns a LIST
* **reversed** – returns an iteration
* **remove** – method removes a value, not an index – So it always starts from first
* **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**

A screenshot of a chat

AI-generated content may be incorrect.

* If an object is **immutable**, reassignment just redirects the arrow — original unaffected.
* If an object is **mutable**, in-place changes affect everyone pointing to it.

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].  
  
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

**SORTINGS**

**BUBBLE\_SORT**: Adding the swapped variable to bubble sort improves its best-case time complexity from O(n2) 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  
  
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 |
| 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 about understanding Python class method calls, variable scoping, and proper recursion/backtracking logic.

Great questions—these are the exact spots where merge sort feels slippery. Let’s straighten them out.

**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

**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.

A screenshot of a computer

AI-generated content may be incorrect.

.

**🔹 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.”

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

* 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.

A screenshot of a math problem

AI-generated content may be incorrect. A white background with black text

AI-generated content may be incorrect.

A screenshot of a math problem

AI-generated content may be incorrect.

A screenshot of a math test

AI-generated content may be incorrect.

**BUCKET SORT:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

A screenshot of a computer

AI-generated content may be incorrect.  
  
A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a cell phone

AI-generated content may be incorrect.