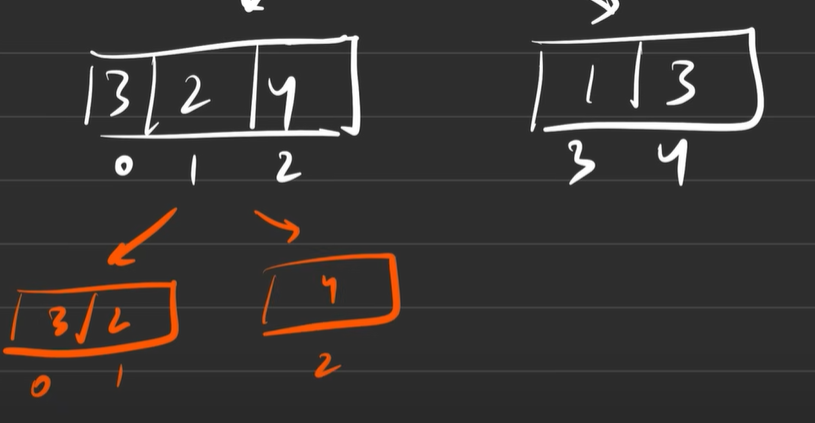
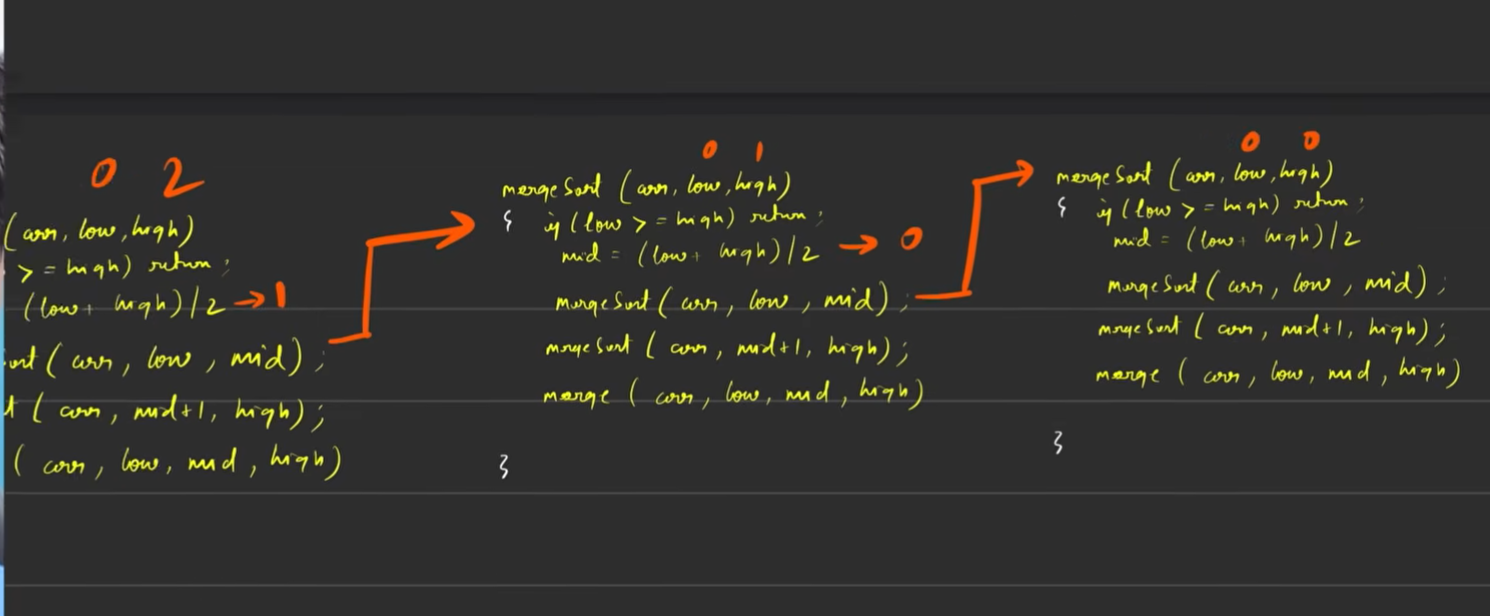
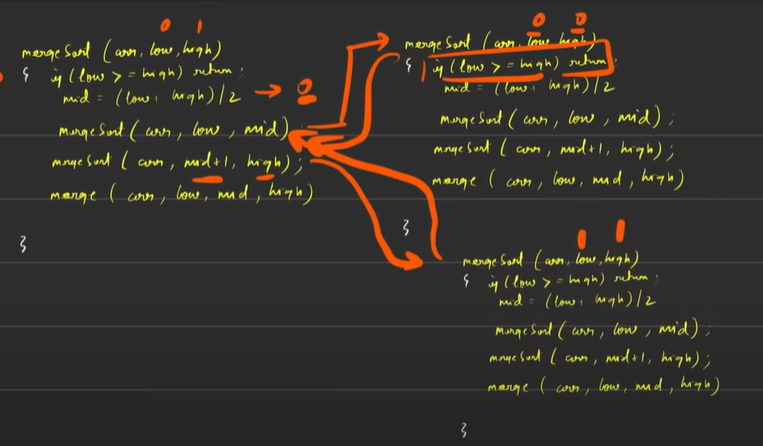
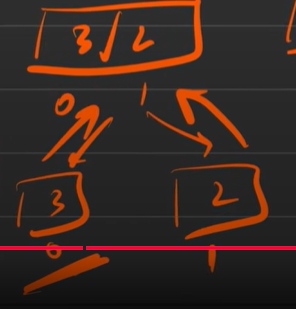
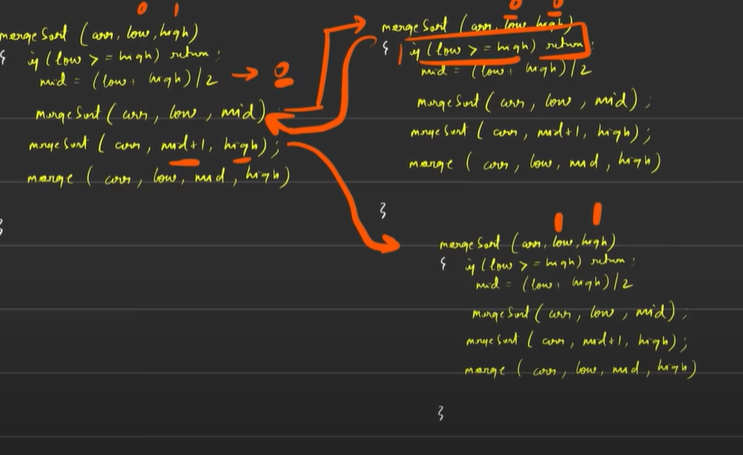
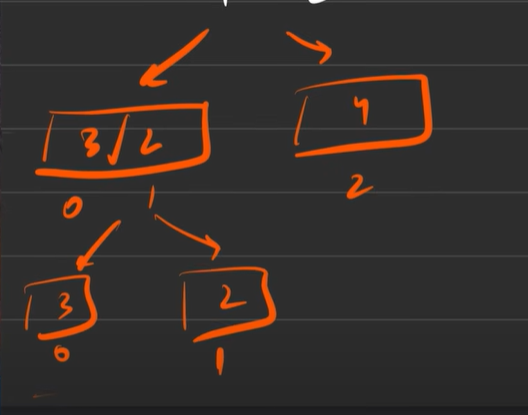
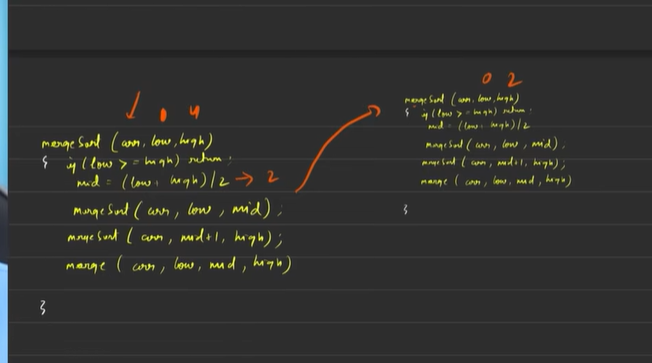
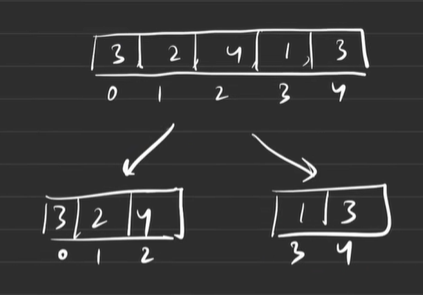
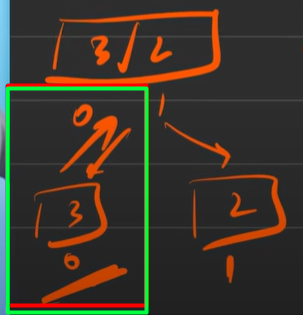
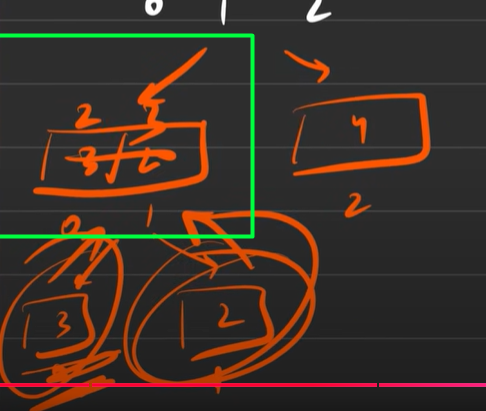
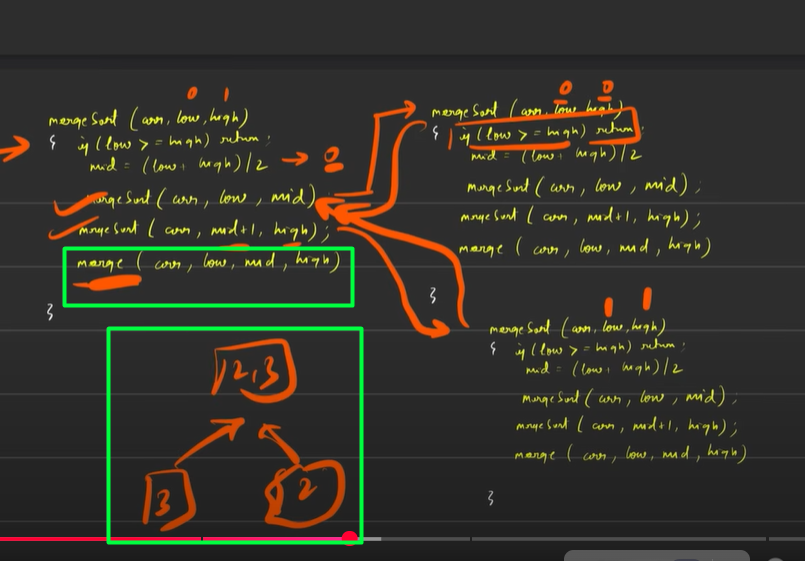
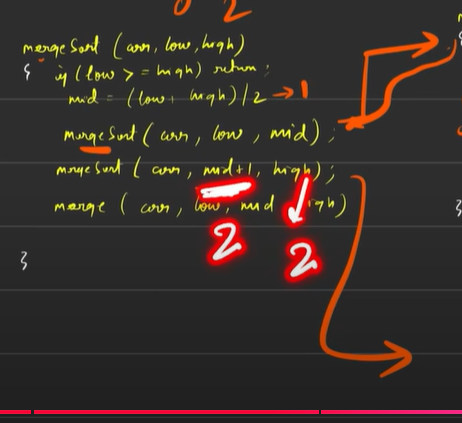
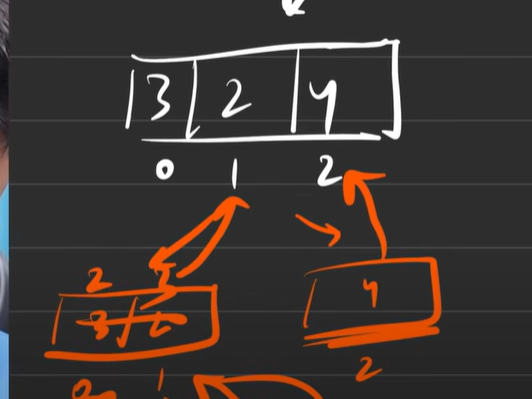
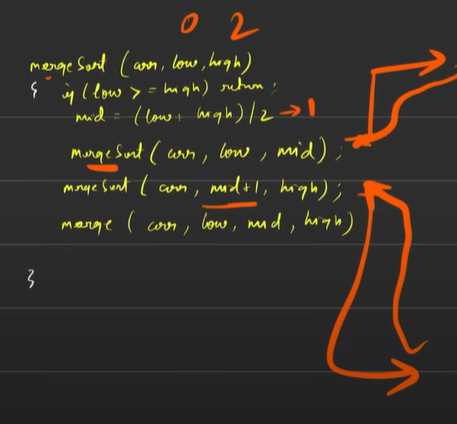
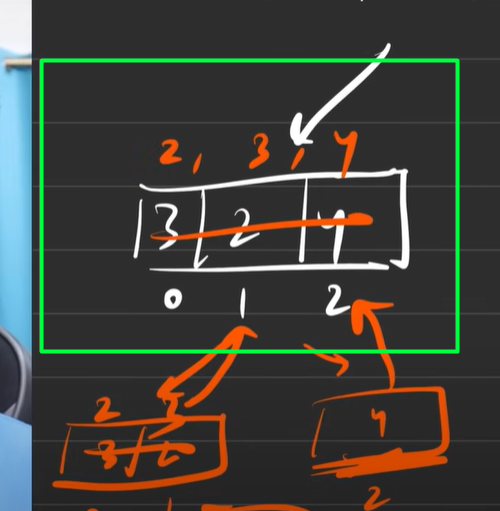
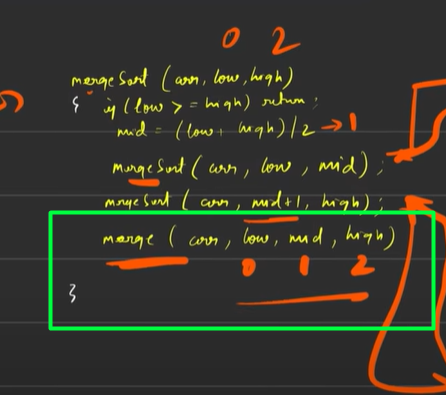
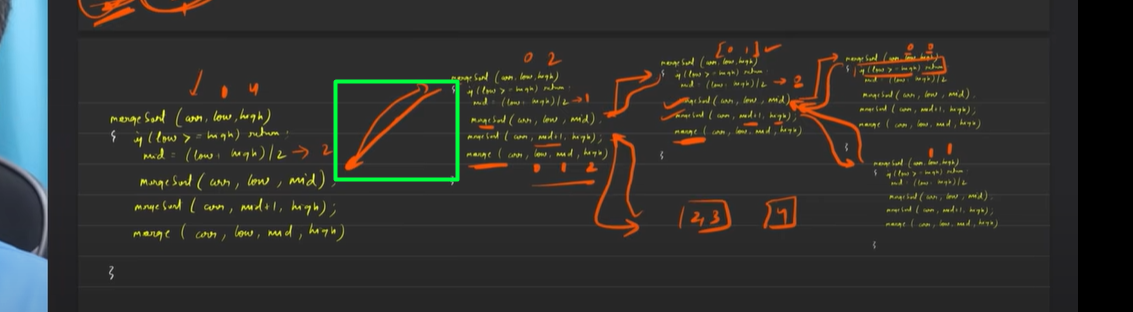
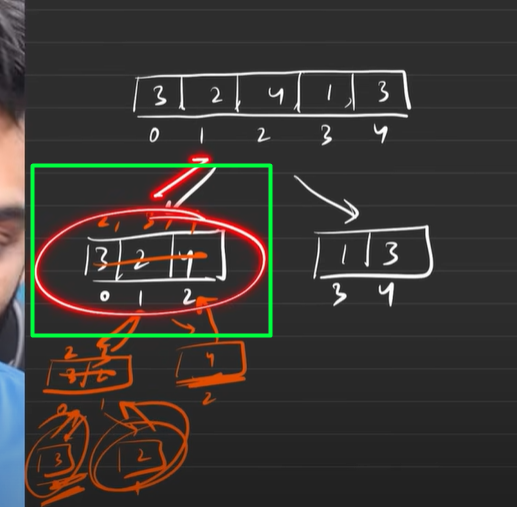
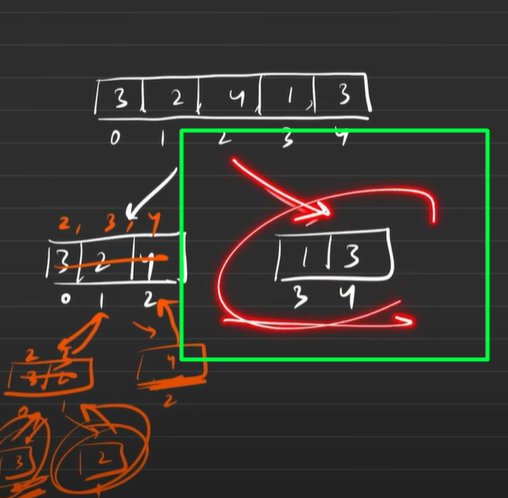
Merge Sort Leftovers

Dry Run

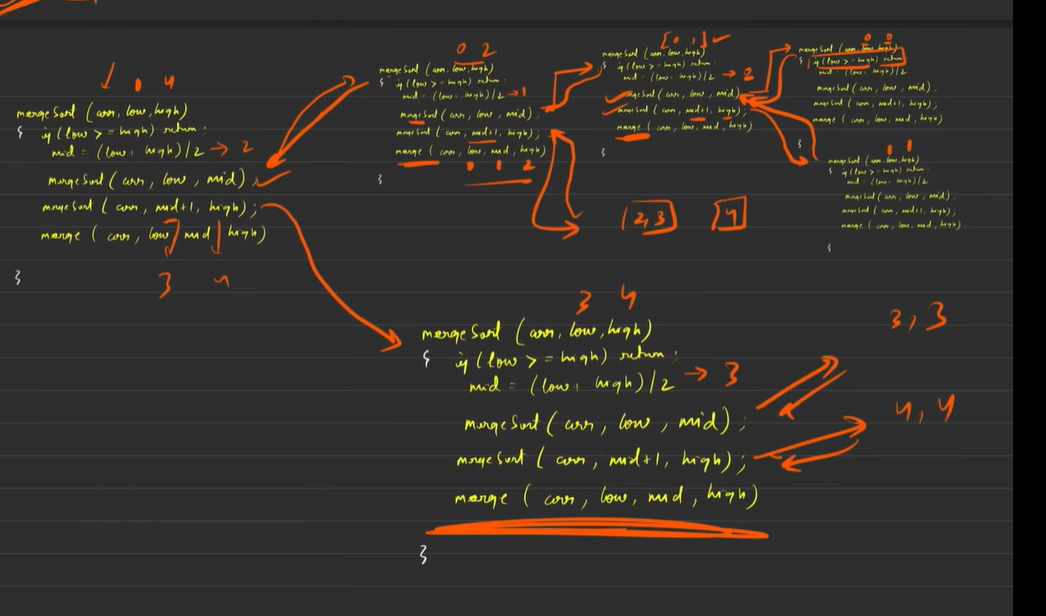
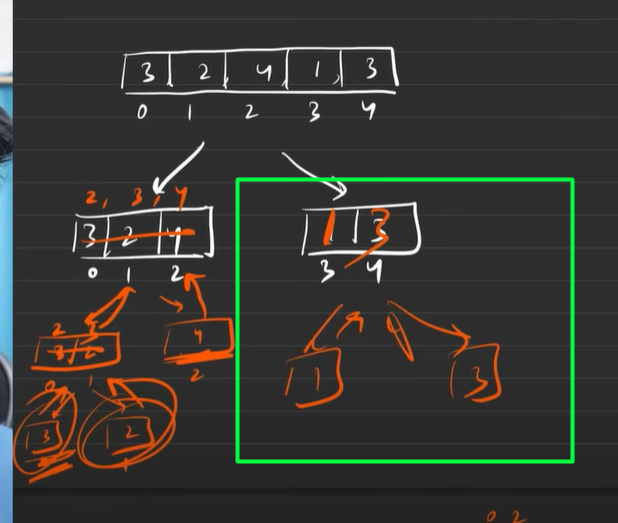


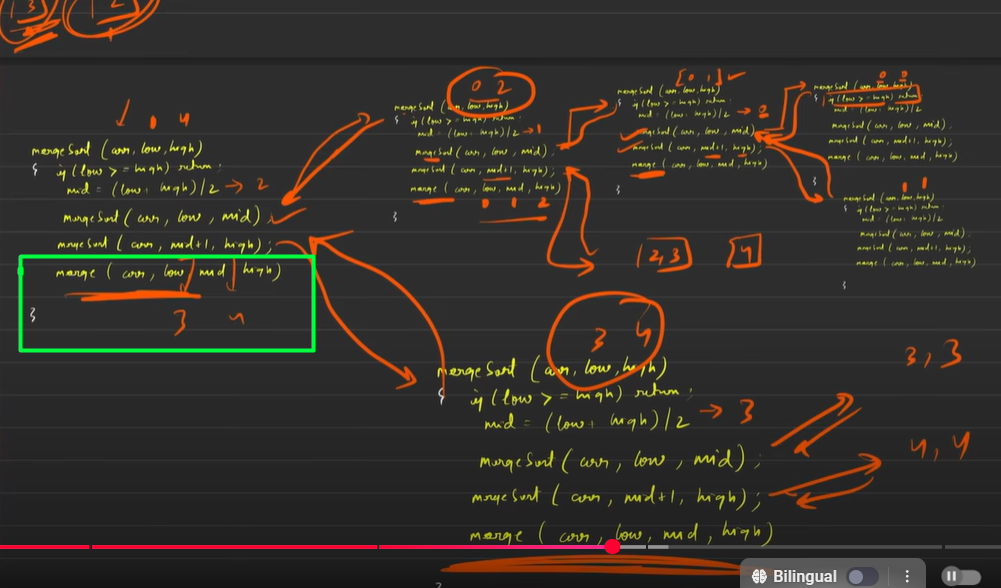
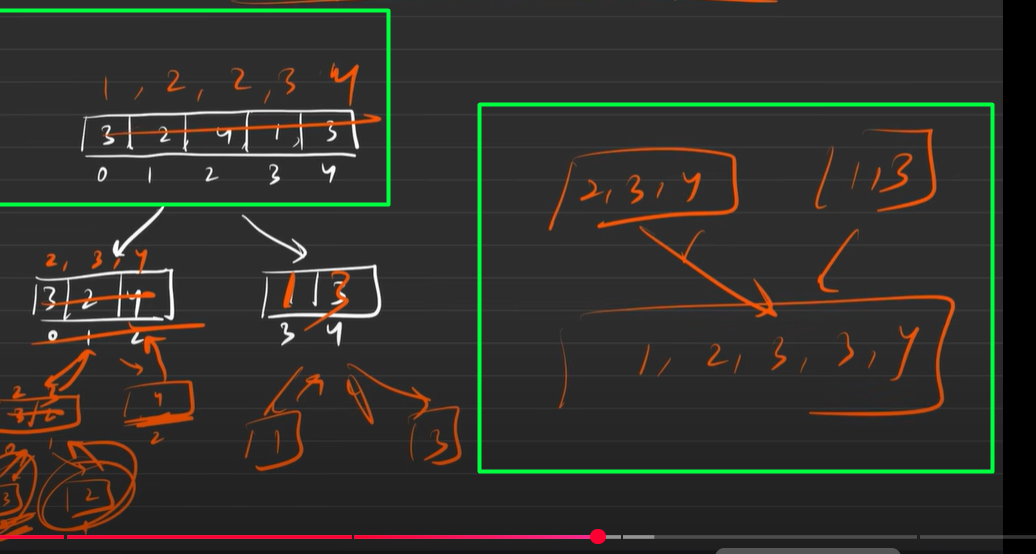


First left half is sorted



Focusing on the right part now



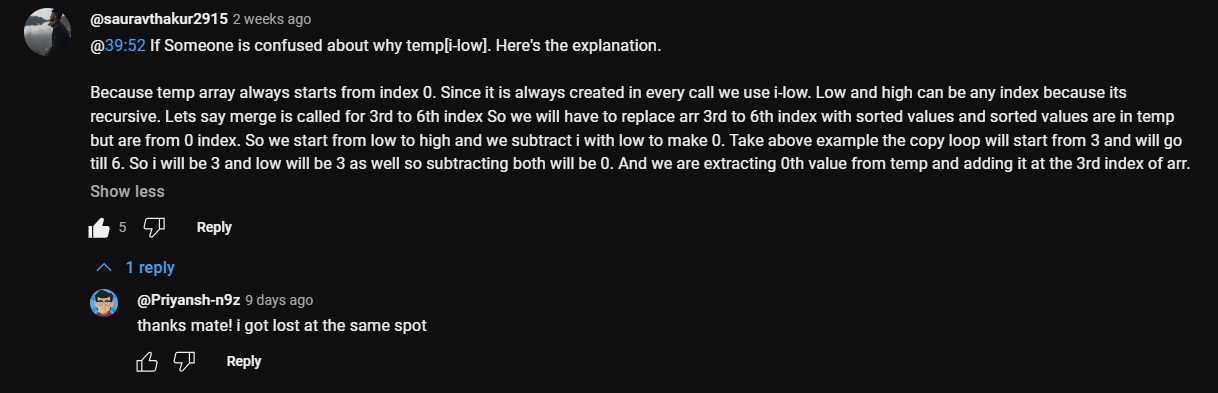


Striver’s advice: To Understand this in depth, take a pen-paper and do it all over again

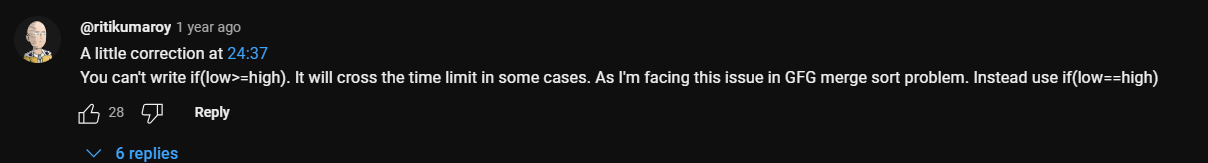
Merging takes place at the last step

We have both the halves sorted now and now the last merge takes place

Merge code doubt



Tips And Extra Concerns



Other doubts

hey I am bit confuesed by the way striver explained about the time complexity of merge sort.....he said that mergeArrays function takes about O(N) , and for merging two given arrays at a time into a single array of size n, we take minimum n steps to compare and combine them, how and why?....besides how is the tc being reduced in mergeArray() during recursion call like n, n/2, n/4...etc...and if its true why the total tc comes to be N X log2N...I know that division takes log2N but what about the merging part...and why does the overal tc comes to be N times of log2N

**ChatGPT said:**

Great question, Vaibhav — and it's a **very important concept** to understand deeply. Let's break it down step by step so you not only **remember** but also **feel** it clearly. Let’s build the intuition first and then confirm it mathematically.

**🧠 Merge Sort Time Complexity — The Intuition**

**📌 1. How mergeArrays() takes O(N)**

Let’s say you are merging two **sorted** subarrays:

* Left: n/2 elements
* Right: n/2 elements
* Total size: n

To merge these into a single sorted array:

* You **compare** the first element of each subarray.
* Pick the smaller one.
* Move forward in that subarray.
* Repeat until both arrays are exhausted.

🧮 So, every element is **visited once** → total operations = n.  
Thus, mergeArrays() is **O(n)** for an array of size n.

**📌 2. Why merging happens at different sizes in recursion (n, n/2, n/4...)**

Imagine your array is split recursively:

less

CopyEdit

Level 0: [n] --> 1 merge of size n

Level 1: [n/2][n/2] --> 2 merges of size n/2

Level 2: [n/4][n/4][n/4][n/4] --> 4 merges of size n/4

...

Level log₂n: [1][1][1]...[1] --> base case

At each level:

* You **double the number of subarrays**
* But the **total size** remains n  
  (Because 2 × n/2 = 4 × n/4 = 8 × n/8 = ... = n)

So, **merging cost per level = O(n)**  
And there are **log₂(n)** levels of recursion.

**📌 3. Why Total Time = O(n × log₂n)**

Now let’s add up the work at **each level**:

| **Level** | **# of subarrays** | **Size of each subarray** | **Total merge cost** |
| --- | --- | --- | --- |
| 0 | 1 | n | n |
| 1 | 2 | n/2 | n |
| 2 | 4 | n/4 | n |
| 3 | 8 | n/8 | n |
| ... | ... | ... | ... |
| log₂n | n | 1 | n |

So, if each level costs O(n), and there are log₂(n) levels:

**🧮 Final Time Complexity = O(n × log₂n)**

**🔁 Analogy for Better Understanding**

Think of Merge Sort like:

* Cutting a big **cake** (array) into smaller equal pieces.
* Each **cut** (divide step) is fast: no actual computation.
* The real work is in **assembling it back** layer by layer in sorted order (merge step).
* At every layer (height in the tree), you're just rearranging n total pieces — that’s why it’s n × log₂n.

**✅ TL;DR Summary:**

* **Divide step:** No cost (just splitting).
* **Merge step:** Cost per level = O(n)
* **Number of levels:** log₂(n)
* **Total work:** O(n × log₂n)

Let me know if you'd like a tree diagram showing how this works visually, or if you want the space complexity explanation too (which is also interesting!) 👀

4o

**You said:**

how is every elemnt visted at leats once?

**ChatGPT said:**

Excellent follow-up, Vaibhav! Let’s zoom into this part:

**"How is every element visited at least once during the merge step?"**

Let me explain with **intuition + an example**, and then back it up logically.

**🧩 Think About the Purpose of mergeArrays()**

The mergeArrays() function's job is to **compare elements** from the left and right halves and put them into a temporary sorted array. While doing so:

* Each element from both halves must be **moved once** into the temp array.
* Whether an element is copied because it's smaller or because the other subarray is exhausted — it’s still moved.
* So, every element is **touched** (i.e., read and placed) exactly **once per merge call**.

**📘 Example: Array of 6 elements**

cpp

CopyEdit

Original array: [4, 2, 1, 5, 6, 3]

Let's say we reach the final merge step:

cpp

CopyEdit

Left: [1, 2, 4]

Right: [3, 5, 6]

Here's what happens:

1. Compare 1 and 3 → pick 1 → put in temp
2. Compare 2 and 3 → pick 2 → put in temp
3. Compare 4 and 3 → pick 3 → put in temp
4. Now 3 elements are taken from left, 1 from right
5. Remaining elements: 4 (left), 5 and 6 (right)
6. All are **copied** over directly

✅ **Total touches = 6**, each element handled **once** (either compared or directly pushed)

**📌 Final Logic**

For any merge step with n total elements:

* You process each of the n elements exactly **once**.
* This includes either:
  + Comparing and choosing the smaller element.
  + Or directly copying leftover elements.

Thus, **mergeArrays() does O(n) work** where n is the total number of elements being merged.

**🧠 Analogy:**

Imagine you’re sorting socks into two piles:

* Left pile: 3 socks
* Right pile: 3 socks

You look at the smallest on top from both piles, pick one and place it into a new row. Even if you don’t compare every sock, you at least **touch/move each sock once**.

Let me know if you want to go deeper into the number of comparisons vs. movements — both are counted as **work done** in complexity analysis.

4o

**You said:**

what about the space complexit, he says that the its the additonal space that we are taking, but that would be only auxlary space right besides, Ig he didn't take in account the array space that was given as parameter, he only to the size of temp array, and why did he say that O(n) is the tc ?

**ChatGPT said:**

Oof yes, good catch Vaibhav — you're asking exactly the right kind of questions now 🔥  
Let’s **break down the confusion** about:

1. Why Striver says the time complexity of merging is **O(n)**
2. What **auxiliary space** really means in this context
3. Why we don’t count the input array’s space

**✅ 1. Why is the time complexity of merge() O(n)?**

When Striver says merging two arrays of size n takes **O(n)** time, here's what he means:

* If you're merging a left array of size n/2 and a right array of size n/2, then the total number of elements = n.
* You will **compare and move each element once** into a temporary array.

So the number of operations = proportional to n → That’s **O(n)**.

Even if you compare some and just push others, you still **touch/move each one once** → still **linear work**.

So in recursion:

* Level 0 → 1 merge of n → O(n)
* Level 1 → 2 merges of size n/2 → O(n)
* Level 2 → 4 merges of size n/4 → O(n)
* ...
* There are log₂n levels → so total work = **O(n × log n)**

**✅ 2. What about space complexity? And why only temp is counted?**

When he says **O(n) auxiliary space**, he means:

"Extra space used apart from the input itself"

In your code:

cpp

CopyEdit

vector<int> temp; // in each merge

This extra space **is not reused across recursion calls**, so **it adds up**.

Even though you pass the array arr around, that is:

* A reference (&arr) → no new space is made for arr
* So, we **don’t count it**

Only the **temporary arrays** made during merge steps are **extra / auxiliary**, and their **total space used** (in worst case) is proportional to n ⇒ **O(n)**

So:

* **Time complexity** = O(n log n)
* **Auxiliary space complexity** = O(n)

👉 If we used **in-place merge** (which is hard), we could reduce space.

**🧠 TL;DR Summary:**

| **Aspect** | **Value** | **Why** |
| --- | --- | --- |
| Time Complexity | O(n log n) | log n levels, each doing O(n) merge work |
| Space Complexity | O(n) auxiliary | Temporary vectors in mergeArrays() at each recursion level |
| Input array’s space | Not counted | Already given, not "extra" space |

If you want, I can also show how the temp arrays stack up in memory at each recursive level. Wanna go visual with it too?

4o

**You said:**

yeah

**ChatGPT said:**

Perfect! Let’s walk through a **visual breakdown** of how both **time and space complexity** behave during Merge Sort — specifically focusing on how:

* Each level of recursion works (log₂n levels)
* How much merging (and hence temp space) is done per level
* Why total space = **O(n)** and total time = **O(n log n)**

**🧠 Imagine the Array:**

Say we start with an array of **8 elements**:  
[7, 3, 1, 9, 4, 2, 6, 5]  
(This makes it easier as 8 is a power of 2)

**🔁 Merge Sort Breakdown (Recursive Tree View):**

mathematica

CopyEdit

Level 0: [7 3 1 9 4 2 6 5] → size = 8 (1 merge) → O(8) merge time

/ \

Level 1: [7 3 1 9] [4 2 6 5] → 2 merges of size 4 → O(8) total

/ \ / \

Level 2: [7 3] [1 9] [4 2] [6 5] → 4 merges of size 2 → O(8) total

/ \ / \ / \ / \

Level 3: [7] [3] [1] [9] [4] [2] [6] [5] → 8 arrays of size 1 (base case)

**🧠 Time per Level:**

No matter the level, if you sum all merges:

* **Each level merges n elements total**
  + Level 0: 1 merge of size 8 → 8 steps
  + Level 1: 2 merges of size 4 → 4+4 = 8 steps
  + Level 2: 4 merges of size 2 → 2+2+2+2 = 8 steps
  + Level 3: No merging (base case)  
    → So per level: **O(n)**  
    → log₂n levels ⇒ **Total time = O(n log n)** ✅

**🧠 Space Usage at Each Merge Call:**

Each mergeArrays() function makes a temp vector for its merge:

At any point in time:

* Only **one path down** the recursion is active
* So **maximum depth** = log₂n
* BUT — the **temp array** at the topmost level = size n
* It does **not** grow larger than n at any moment
* We **don’t reuse** the temp vector between merges, but since merge happens in a depth-first recursive way → **space never exceeds O(n)**

**So:**

* Max temp array size per merge = size of subarray being merged
* Max total extra memory at any given point = **O(n)**

**✅ Final Conclusion:**

| **Metric** | **Value** | **Explanation** |
| --- | --- | --- |
| **Time Complexity** | O(n log n) | log₂n levels × O(n) merge work per level |
| **Auxiliary Space** | O(n) | temp vectors used during merges |
| **Input array space** | Not counted | Already provided, not "extra" |

Wanna test this with a dry-run example on paper? I can help you walk through the steps with actual values if you want too!

4o

**You said:**

I want to know why the inout soace is avoided in sc...is necssesary to do so?

**ChatGPT said:**

Great question, Vaibhav! 🙌  
Let’s break this down clearly and deeply 👇

**💡 Why Input Array Space is *not* counted in Space Complexity:**

When we analyze space complexity, we are trying to **measure the *extra* or *additional* memory** that a program uses **on top of the input**.

**So basically:**

| **Component** | **Counted in Space Complexity?** | **Why?** |
| --- | --- | --- |
| Input array (e.g., arr[]) | ❌ No | It's already given as part of the input. We don't allocate it during execution — we just operate on it. |
| Temporary array (temp[]) | ✅ Yes | This is newly created during execution. It's *auxiliary* memory used to solve the problem. |
| Recursive stack | ✅ Yes | Created dynamically due to function calls — part of execution overhead. |

**🧠 Is this approach necessary?**

Yes, and here's **why we must follow this standard**:

Imagine two sorting algorithms:

* One that sorts **in-place** (like quicksort)
* One that uses **extra space** (like mergesort)

If we include the input array in both:

* Both would show **O(n)** space complexity 🤔
* But that’s misleading! Because quicksort doesn’t use extra memory for data
* So, by **excluding the input**, we make a fair comparison:
  + **Quicksort: O(log n)** (stack only)
  + **Mergesort: O(n)** (temp array + stack)

**✅ Summary:**

We avoid counting the input space in space complexity because:

1. It’s **already available**, not part of what our algorithm adds.
2. It ensures **fair comparison** between algorithms.
3. We only measure what’s **additionally required** to solve the problem.

**So in Merge Sort:**

* Input space: ❌ Not counted
* temp[]: ✅ Counted → O(n)
* Recursion stack: ✅ Counted → O(log n)
* ⇒ Total Space Complexity = **O(n)** (because O(n) dominates O(log n))

Let me know if you want a visual analogy or comparison with quicksort to strengthen this further 💪

4o

**You said:**

so arr given in the parameter is counted in input space

**ChatGPT said:**

Exactly, Vaibhav! 💯  
You're spot on.

**✅ Yes — The arr passed as a parameter is part of the input space, not additional space.**

So when you do something like:

cpp

CopyEdit

vector<int> arr = {5, 2, 9, 1, 6};

mergeSort(arr, 0, arr.size() - 1);

That arr is already provided to your program. It's the input.  
We **don’t** count this space because:

* You **didn't allocate it** during your algorithm.
* You're **just using what's given** to you.
* It exists **regardless of which algorithm** you use.

**In space complexity analysis:**

| **Component** | **Counted?** | **Why?** |
| --- | --- | --- |
| arr (input vector) | ❌ No | Already exists; part of input. |
| temp inside merge() | ✅ Yes | Created during execution — auxiliary space. |
| Recursion call stack | ✅ Yes | Overhead from recursive calls. |

So when someone says "Merge Sort has **O(n)** space complexity", they’re talking about the **temporary vector (temp)** used during merging.

**📌 TL;DR:**

Yes — arr is **input space**, not counted in space complexity.  
Only count the **extra memory** your algorithm creates **during execution**.

Want a real-life analogy to help remember this?

4o

**You said:**

but in a separate video he said that sc = input space + auxiliary space, but whike implemntation, he focuses on the auxilary space only

**ChatGPT said:**

Ahh yes, Vaibhav — great catch 🔍 and a very insightful question.

You're absolutely right. Here's the **clarification**:

**🎯 What Striver (and others) *mean* when they say:**

**"Space Complexity = Input Space + Auxiliary Space"**

That’s **technically true**, and used when you're doing a **complete** or **formal** space complexity analysis (for interviews or theoretical evaluation). So:

* **Input Space** → Space required to store the input (arr in our case) → O(n)
* **Auxiliary Space** → Extra space used by your algorithm (like recursion stack, temp arrays) → Also O(n) for Merge Sort

So **overall**, it's O(n + n) = O(2n) = O(n).

BUT…

**🤔 Why do we *usually* ignore Input Space in analysis?**

Because the input is **already given** to your program, and almost **every algorithm** takes that same input — so it’s **not fair to compare** algorithms based on input space.

So we say:

* "Let’s only compare algorithms based on what **extra** space they need."
* Hence we focus only on **auxiliary space**.

**🧠 When do we *include* input space?**

* In **complete theoretical analysis**
* When explicitly asked: *"Total space required including input?"*
* In **competitive programming**, it’s often ignored to focus on optimization

**✅ Summary:**

| **Type of Space** | **Counted Usually?** | **Why** |
| --- | --- | --- |
| Input Space | ❌ Not in typical TC/SC comparison | It's given to you |
| Auxiliary Space | ✅ Yes | Shows how efficient your algorithm is |
| Total Space (Input + Aux) | ✅ Only when full analysis is required |  |

So yeah — Striver wasn’t contradicting himself — he was just **shifting perspective** depending on whether he was explaining theory vs implementation vs optimization.

Let me know if you want a visual or real-life analogy to pin this down better 🔥