Work through the guidelines

1st break down the problem into small groups, where each problem can be defined into input and output

The whole idea behind analyzing algorithms using Big O notation—**tweaking and improving your algorithms** to make them more efficient in terms of time (and sometimes space) complexity. When you're designing or choosing an algorithm, you aim for one that can handle larger inputs without becoming too slow.

**Why is efficiency important?**

As input sizes grow (e.g., bigger datasets, more complex tasks), inefficient algorithms can slow down significantly. By optimizing for better time complexity, you ensure that your program runs faster and scales well.

**Example: Sorting Algorithms**

Different sorting algorithms have different time complexities:

* **Bubble Sort**: O(n2n^2n2) – inefficient for large inputs, as it checks pairs repeatedly.
* **Merge Sort**: O(nlog⁡nn \log nnlogn) – much more efficient because it divides the input in half with each recursive step.
* **Quick Sort**: Also O(nlog⁡nn \log nnlogn) on average but can degrade to O(n2n^2n2) in the worst case (though this is rare with optimizations).

So, if you know your input size could be large, you’d want to choose **Merge Sort** or **Quick Sort** over **Bubble Sort** because of their better time complexity.

**How to tweak algorithms for efficiency:**

1. **Reduce unnecessary work**: Avoid repeating operations that don’t need to be done multiple times.
   * Example: Instead of looping over the same list twice, find a way to do it once.
2. **Use better data structures**: Certain data structures allow for faster operations.
   * Example: A hash table (or dictionary in Python) allows O(1) time complexity for lookups compared to a list's O(nnn).
3. **Divide and conquer**: Breaking problems into smaller subproblems can lead to faster algorithms.
   * Example: **Binary search** splits the search space in half with each step, giving O(log⁡n\log nlogn) time, much faster than a linear search O(nnn).
4. **Optimize loops and recursion**: Finding ways to reduce the number of iterations or recursive calls helps reduce the time complexity.

**A practical approach:**

When you're writing code, always think about how it will perform with **larger inputs**. Understanding the time complexity lets you adjust your algorithms to be as efficient as possible, especially when dealing with **real-world data**, where input sizes can be massive.