INTRO TO ALGORITHMS (Learn from the ground up, using Merge Sort!)

Presenters: Jesiah Martin & Kesnel Mezinord

IMPORTANCE OF ALGORITHMS

Algorithms are everywhere!

- Imagine you had millions of books in a library and had to find a specific one without any kind of search system. You'd have to go through each book manually, which could take hours or even days. Algorithms are like shortcuts—they make it possible to search, sort, and retrieve information quickly. In library software, they ensure you can find the exact book you need in seconds rather than wasting valuable time sifting through everything yourself. Without algorithms, the process would be slow, inefficient, and overwhelming.

Introduce Yourself

Name

Year

Major

There are thousands of algorithms out there Why Merge Sort?

The Time Complexity

- Time Complexity is the measure of how long an algorithm takes to run in relation to the size of the input
- A time complexity of O(n) implies the length of time an algorithm takes is linear with the input.
- Merge Sort has a time complexity of O(n*log n)

  This is the best a comparison-based sorting algorithm can get

Back to the Basics!

Algorithm – A set of instructions that a computer follows to complete a task or solve a problem. What might take one person years hours to accomplish (sorting 1000 papers by date), an algorithm completes in mere seconds!

Python – A programming language often used to build softwares, automate tasks, and analyze data

Function – A block of code used to perform a specific task. Defined using the 'def' keyword, and can be used multiple times

Variables – Names that refer to values in a program. Used to store and manipulate data. 'Middle' can store '5'

Lists (Arrays) – An ordered collection mainly storing elements of the same type. 'my_list' can store '[1, 2, 3, 4, ...]. Arrays use 0-indexing, meaning the first number is called using my_list[0] and returns '1'

List Methods – Built in methods available for lists performing various operations. We will be using 'append()' to add elements and 'extend()' to 'merge' the two lists. Think of them as mini-functions

Parameters – variables listed in a functions definition, used to pass information into them. merge_sort(arr) means 'arr' is the array that will be sorted

Slicing – Process of accessing a subset of elements from a list using a range. left_half = arr[:mid] puts the left half of the array into left_half

Edge cases – Boundaries or special cases that must be considered for proper function. What if the array is empty?

How to Install Python

Install Python:

1. Head to python.org/downloads/ and download the version of python specific to your system

   If Windows, click 'Add Python to PATH'.

   if macOS, drag the icon into the applications folder

   if Linux, use 'sudo apt-get install python3

   Verify Installation through command prompt/terminal with python –version

2. Head to jetbrains.com/pycharm/ and download the appropriate version

   If Windows, run the .exe file.

   if macOS, open the .dmg file

   if Linux, extract the .tar.gz and run the pycharm.sh script

   Launch PyCharm

Watch Instructor for further instructions

Path to Completion

1. Understand the Problem
2. Write your Initial Thoughts
3. Plan the Solution
4. Implement and Document
5. Test and Analyze
6. Refine and Review

THE PROBLEM

88. Merge Sorted Array

Easy | Topics | Companies | Hint

You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

Merge `nums1` and `nums2` into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to `0` and should be ignored. `nums2` has a length of `n`.

Example 1:

> **Input:** nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
> **Output:** [1,2,2,3,5,6]
> **Explanation:** The arrays we are merging are [1,2,3] and [2,5,6].
> The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1.

Example 2:

> **Input:** nums1 = [1], m = 1, nums2 = [], n = 0
> **Output:** [1]
> **Explanation:** The arrays we are merging are [1] and [].
> The result of the merge is [1].

Example 3:

> **Input:** nums1 = [0], m = 0, nums2 = [1], n = 1
> **Output:** [1]
> **Explanation:** The arrays we are merging are [] and [1].
> The result of the merge is [1].
> Note that because m = 0, there are no elements in nums1. The 0 is only there to ensure the merge result can fit in nums1.

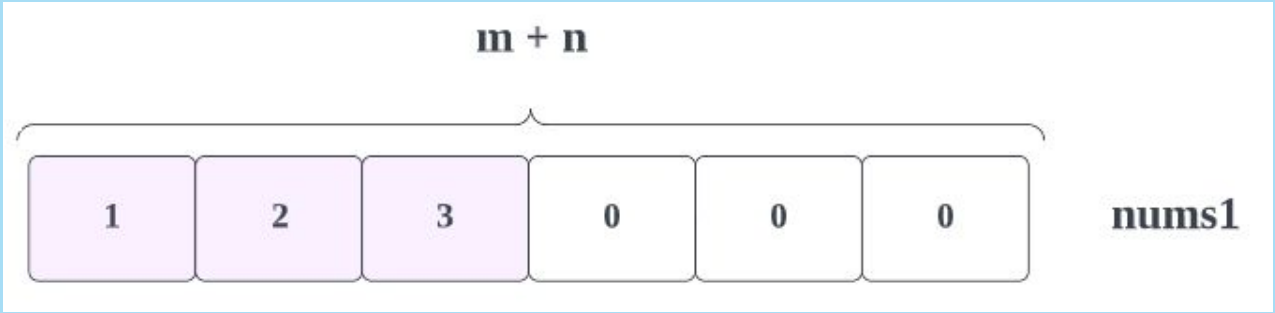
Constraints:

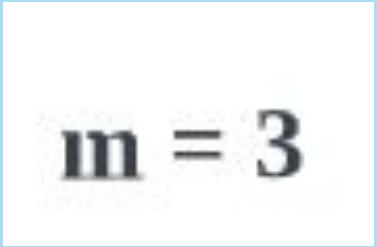
- `nums1.length == m + n`
- `nums2.length == n`
- `0 <= m, n <= 200`
- `1 <= m + n <= 200`
- `-10^9 <= nums1[i], nums2[j] <= 10^9`

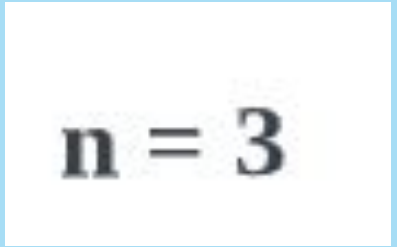
UNDERSTANDING THE PROBLEM

#1: Ensure you understand the problem you're solving.

#2: Identify the input and output.







INITIAL THOUGHTS

- **DO NOT** jump into using data structures or algorithms without first understanding the problem.
- **Complex problems** often have **simple solutions** once you understand the task at hand.
- If I wanted to take a weekend trip to the Everglades, I could book a delta ticket, or road trip with friends and save $500 in the process. Just like this decision, you want to choose the least expensive approach in terms of complexity and resources.

Code

```
def merge_sort(arr):
   if len(arr) > 1:
      mid = len(arr) // 2
      left_half = arr[:mid]
      right_half = arr[mid:]
      merge_sort(left_half)
      merge_sort(right_half)
      i = j = k = 0
      while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
           arr[k] = left_half[i]
           i += 1
        else:
           arr[k] = right_half[j]
           j += 1
        k += 1
      while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1
      while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1
```

Code

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

THE PROBLEM

88. Merge Sorted Array

Easy | Topics | Companies | Hint

You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

Merge `nums1` and `nums2` into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to `0` and should be ignored. `nums2` has a length of `n`.

Example 1:

> **Input:** nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
> **Output:** [1,2,2,3,5,6]
> **Explanation:** The arrays we are merging are [1,2,3] and [2,5,6].
> The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1.

Example 2:

> **Input:** nums1 = [1], m = 1, nums2 = [], n = 0
> **Output:** [1]
> **Explanation:** The arrays we are merging are [1] and [].
> The result of the merge is [1].

Example 3:

> **Input:** nums1 = [0], m = 0, nums2 = [1], n = 1
> **Output:** [1]
> **Explanation:** The arrays we are merging are [] and [1].
> The result of the merge is [1].
> Note that because m = 0, there are no elements in nums1. The 0 is only there to ensure the merge result can fit in nums1.

Constraints:

- `nums1.length == m + n`
- `nums2.length == n`
- `0 <= m, n <= 200`
- `1 <= m + n <= 200`
- `-10^9 <= nums1[i], nums2[j] <= 10^9`

Coding Knowledge Required

If – Checks if a condition is true. If the array has 0 or 1 elements, it is already sorted, so return arr stops the function and gives the sorted array back.

len(arr) – returns the number of elements in the array

// (Floor Division) – Divides two numbers and rounds down to the nearest whole number. So 5 // 2 = 2

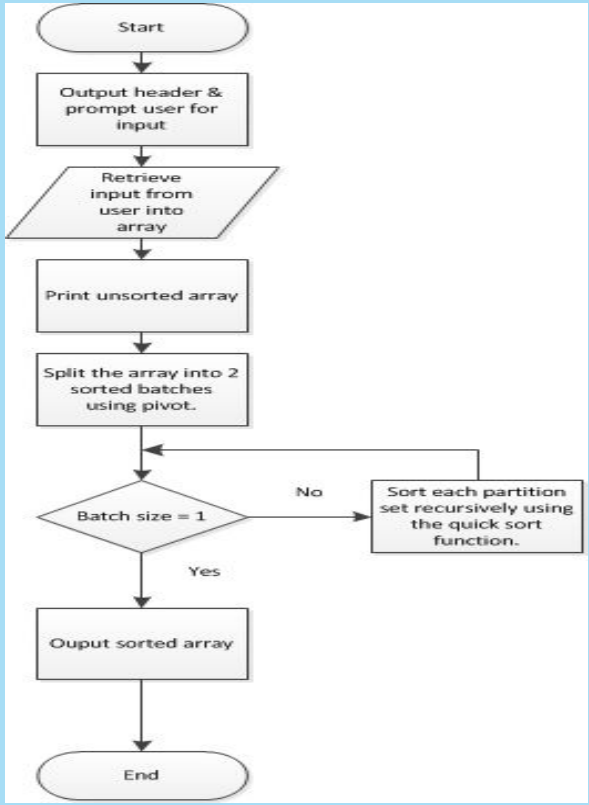
while () – Keeps repeating as long as the condition is true.

else: The counterpart of If

Extend(): adds all remaining elements from one array to another

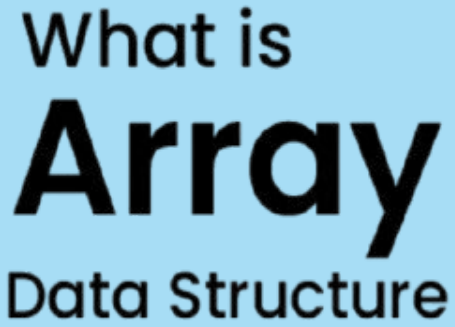
APPROACH

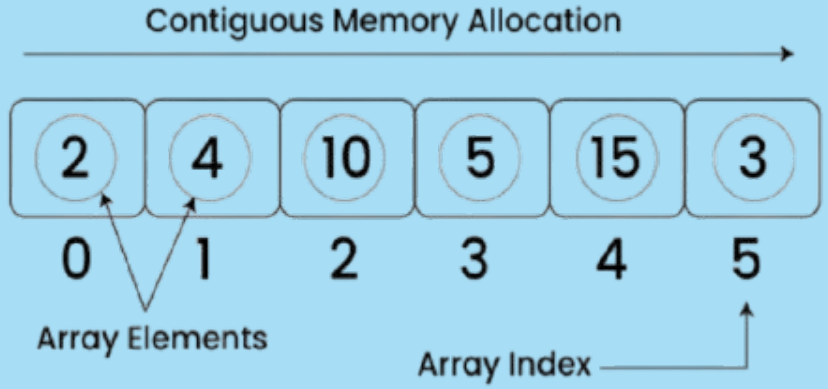
- Without algorithms or code, how would you solve this?
- Breaking down the problem helps simplify it and better translate your thought process into code.



ANALYZING INPUT

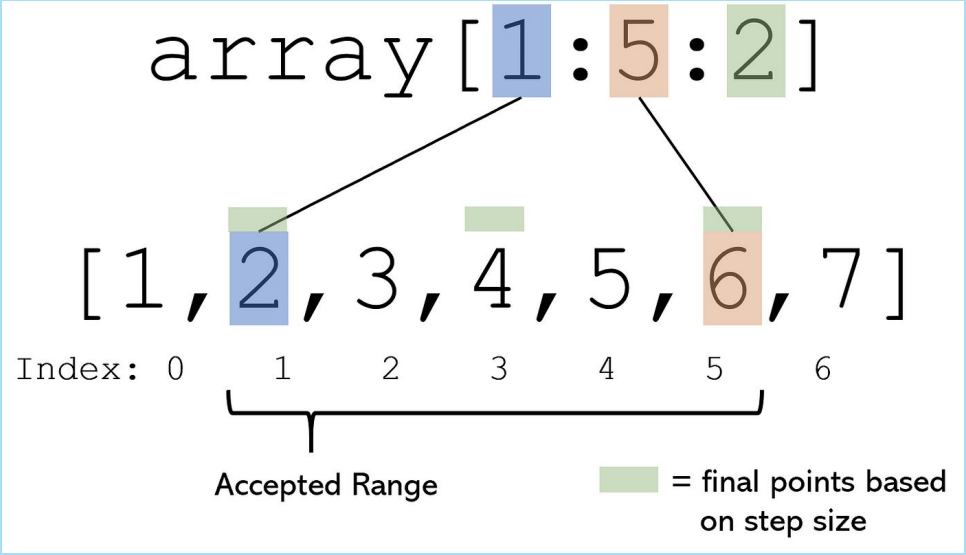
- First, we need to access the input: two lists of numbers.
- The computer needs pointers to "see" the values in the arrays.





POINTERS AND INDEXING

- Introduce **pointers** to traverse the array.
- Explain **indexing**: accessing array elements like nums1[0] = 1, nums2[1] = 5.



TRAVERSING ARRAYS WITH A WHILE LOOP

- Use a **while loop** to traverse nums1 and nums2.
- Set conditions: while m > 0 and n > 0.
- Purpose: to iterate until pointers reach the start of the arrays.

COMPARING ELEMENTS

When you need to compare elements in two different arrays, you can use a **nested for loop**. The concept is simple: the **outer for loop** iterates through the elements of the first array, and the **inner for loop** iterates through the elements of the second array. This allows you to compare each element of the first array with each element of the second array.

Here's how it works:

- The **outer for loop** starts with the first element of the first array and stays on that element until the **inner for loop** has gone through all the elements of the second array.
- Once the inner for loop has completed comparing the first element of the first array with all elements of the second array, the outer for loop moves on to the next element of the first array.
- This process continues until all elements of both arrays have been compared.

The inner loop completes all its iterations for every single iteration of the outer loop, ensuring that every element in the first array gets compared with every element in the second array.

Why does this work for comparing elements in two different arrays?

This approach works well because the nested loop structure allows you to systematically compare each element from one array with each element of another array. It ensures that all combinations of elements between the two arrays are covered. Without the nested structure, you would miss some comparisons.

EDGE CASES AND POTENTIAL ISSUES

How could our program go wrong?

- Exit the loop prematurely
- Leave nums2 unsorted

HANDLING REMAINING ELEMENTS

Key Steps to Handle Remaining Elements:

1. **Merging sorted subarrays:**
    - Compare the current elements of the left and right subarrays.
    - Copy the smaller element into the final array.
    - Continue until one of the subarrays is exhausted.
2. **Handling remaining elements:**
    - Once you've merged all elements from one subarray, the other subarray may still contain elements.
    - Since the elements in the remaining subarray are already sorted, you can copy them directly into the final array without further comparisons.

Next Steps

We're excited to kick off **LeetCode Saturdays**, where we'll be diving into key concepts like **arrays, strings, and various data structures**.

- **Learn essential data structures and algorithms**, starting with arrays, strings, and more advanced topics over time.
- **Ask questions** about anything you're struggling with—our community is here to help!
- Practice **problem-solving** in a supportive, collaborative environment.
- Build confidence in your coding abilities for technical interviews and beyond.

No matter your skill level, there's always something new to learn.

Join Our LinkedIn Group!

Codeducation
LinkedIn Group

This is the perfect space to:**No need to sift through LinkedIn — we'll do it for you!** Stay informed on the best opportunities and insights, all in one place.

This is the perfect space to:

- Access **tech insights** and career advice.
- Share your **accomplishments** and connect with fellow members.

Scan the QR code to join and become a part of our growing community!

SOCIALS

Discord

Instagram

CONCLUSION

At its core, coding is all about problem-solving. No one starts out as an expert, and a big part of learning to code is embracing the process of trial and error. When tackling a problem, the first step is to understand what is being asked. From there, map out a plan for how you'll approach the solution. Keep in mind that it's normal to face setbacks or find that your initial solution isn't perfect. Coding often requires trying different approaches, learning from mistakes, and refining your strategy until you achieve an efficient solution. With persistence, every challenge becomes an opportunity to grow and improve your problem-solving skills.