## 🔹 What is an Array?

An **array** is a collection of elements stored at **contiguous memory locations**.

* It stores multiple items of the **same type**.
* Each item can be accessed using an **index** (0-based indexing in most languages).

👉 Think of it as a row of boxes, each with a number (index) written on it.

## 🔹 Array Basics

* **Fixed size** (in most languages like Java).
* **Fast access** (O(1) time) using index.
* **Insert/Delete** can be costly (O(n)), since elements may need to shift.

## 🔹 Example Problem

Let’s start simple:  
**Problem:** Store 5 numbers in an array, print them, and calculate their sum.

### ✅ Python

# Array in Python (list is dynamic but works like array)

arr = [10, 20, 30, 40, 50]

# Print elements

for num in arr:

print(num, end=" ")

# Sum of elements

total = sum(arr)

print("\nSum =", total)

### ✅ Java

public class ArrayExample {

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40, 50};

// Print elements

for (int num : arr) {

System.out.print(num + " ");

}

// Sum of elements

int total = 0;

for (int num : arr) {

total += num;

}

System.out.println("\nSum = " + total);

}

}

### ✅ JavaScript

let arr = [10, 20, 30, 40, 50];

// Print elements

arr.forEach(num => process.stdout.write(num + " "));

// Sum of elements

let total = arr.reduce((a, b) => a + b, 0);

console.log("\nSum =", total);

## 🔹 Key Operations with Arrays

We’ll cover step by step:

1. Traversal (visiting all elements) ✅
2. Insertion (add element at index)
3. Deletion (remove element)
4. Searching (linear & binary)
5. Sorting

**Find the n number**

**Problem with array1.equals(array2)**

In Java, **arrays are objects**, and the equals() method for arrays **does NOT compare contents**. It behaves like == and checks if both arrays **refer to the same object in memory**.

So even if two arrays have the same elements:

Using Arrays.equals() (built-in method

import java.util.Arrays;

if(Arrays.equals(array1, array2)){

System.out.println("Arrays are equal");

}else{

System.out.println("Arrays are not equal");

}

Alright 👍 let’s now do the **same thing but for 1 to 100**.

**Step 1: What we expect**

Normally, we should have all numbers from **1 to 100**:

[1,2,3,4,...,100][1, 2, 3, 4, ..., 100][1,2,3,4,...,100]

That means there are **100 numbers in total**.

**Step 2: But one is missing**

Example:

[1,2,3,4,6,7,...,100][1, 2, 3, 4, 6, 7, ..., 100][1,2,3,4,6,7,...,100]

Here, **5 is missing**.

**Step 3: Formula method**

1. **Expected sum** of numbers from 1 to 100 is:

100(101)2=5050\frac{100(101)}{2} = 50502100(101)​=5050

1. **Actual sum** → add all numbers in your given array.

Example: if the array is missing 5, then sum will be:

504550455045

1. **Missing number = Expected sum − Actual sum**

5050−5045=55050 - 5045 = 55050−5045=5

✅ So the missing number is **5**.

This works no matter which number is missing.

Nice one, Maaz 🙌 This is a very **popular interview problem** — merging two **sorted arrays** into one sorted array.  
We’ll walk through **Approach 1 (simple)** and also touch on **Approach 2 (heap-based)**.

## 🔹 Problem

You are given two **sorted arrays**. Merge them into **one sorted array**.

Example:

arr1 = [1, 3, 5]

arr2 = [2, 4, 6]

Output = [1, 2, 3, 4, 5, 6]

## 🔹 Approach 1: Merge Like Merge-Sort (Efficient & Standard)

👉 Use two pointers:

1. One starts at arr1[0]
2. Other at arr2[0]
3. Compare, pick smaller, move pointer
4. Continue until both arrays are exhausted

**Time Complexity:** O(n1 + n2)  
**Space Complexity:** O(n1 + n2)

### ✅ Python

def merge\_sorted\_arrays(arr1, arr2):

i, j = 0, 0

merged = []

while i < len(arr1) and j < len(arr2):

if arr1[i] < arr2[j]:

merged.append(arr1[i])

i += 1

else:

merged.append(arr2[j])

j += 1

# Add remaining elements

merged.extend(arr1[i:])

merged.extend(arr2[j:])

return merged

# Example

arr1 = [1, 3, 5]

arr2 = [2, 4, 6]

print("Merged:", merge\_sorted\_arrays(arr1, arr2))

### ✅ Java

import java.util.Arrays;

public class MergeSortedArrays {

public static int[] merge(int[] arr1, int[] arr2) {

int i = 0, j = 0, k = 0;

int[] merged = new int[arr1.length + arr2.length];

while(i < arr1.length && j < arr2.length) {

if(arr1[i] < arr2[j]) {

merged[k++] = arr1[i++];

} else {

merged[k++] = arr2[j++];

}

}

// Copy remaining elements

while(i < arr1.length) merged[k++] = arr1[i++];

while(j < arr2.length) merged[k++] = arr2[j++];

return merged;

}

public static void main(String[] args) {

int[] arr1 = {1, 3, 5};

int[] arr2 = {2, 4, 6};

int[] merged = merge(arr1, arr2);

System.out.println("Merged: " + Arrays.toString(merged));

}

}

### ✅ JavaScript

function mergeSortedArrays(arr1, arr2) {

let i = 0, j = 0, merged = [];

while(i < arr1.length && j < arr2.length) {

if(arr1[i] < arr2[j]) {

merged.push(arr1[i++]);

} else {

merged.push(arr2[j++]);

}

}

// Add remaining elements

return merged.concat(arr1.slice(i)).concat(arr2.slice(j));

}

// Example

let arr1 = [1, 3, 5];

let arr2 = [2, 4, 6];

console.log("Merged:", mergeSortedArrays(arr1, arr2));

## 🔹 Approach 2: Heap (More Advanced)

* Insert all elements into a **min-heap**
* Extract them one by one → merged sorted array

**Time Complexity:** O((n1+n2) \* log(n1+n2))  
**Space Complexity:** O(n1+n2)

👉 Not usually required unless interviewer **explicitly asks for heap-based merging**.

## ✅ Interview Insight

* **Best answer in interviews:** Approach 1 (two-pointer merge).
* If asked about **alternatives**, mention **Heap** or **Insertion method**.
* If arrays are **very large** and stored separately (like in databases), heap-based merging makes sense.

👉 Maaz, do you want me to **also show the Heap Approach code** in Python/Java/JS, or keep focus on the **two-pointer method** (most interview-friendly)?

##### You said:

can do the heap one also

###### ChatGPT said:

Great choice, Maaz 🚀 — let’s tackle the **Heap-based merging**.  
This isn’t the most common in interviews, but if you explain both **two-pointer** and **heap**, you’ll look extra strong 💪

## 🔹 Approach 2: Heap Method

1. Put **all elements** of both arrays into a **min-heap**.
2. Extract elements from the heap one by one → they come out in sorted order.

* **Time Complexity:** O((n1+n2) \* log(n1+n2))
* **Space Complexity:** O(n1+n2)

### ✅ Python (Heap)

import heapq

def merge\_with\_heap(arr1, arr2):

# Combine both arrays

merged = arr1 + arr2

# Convert to heap

heapq.heapify(merged)

# Extract in sorted order

result = [heapq.heappop(merged) for \_ in range(len(merged))]

return result

# Example

arr1 = [1, 3, 5]

arr2 = [2, 4, 6]

print("Merged with heap:", merge\_with\_heap(arr1, arr2))

### ✅ Java (Heap / PriorityQueue)

import java.util.\*;

public class MergeWithHeap {

public static List<Integer> merge(int[] arr1, int[] arr2) {

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

// Add elements of both arrays

for(int num : arr1) minHeap.add(num);

for(int num : arr2) minHeap.add(num);

// Extract elements in sorted order

List<Integer> result = new ArrayList<>();

while(!minHeap.isEmpty()) {

result.add(minHeap.poll());

}

return result;

}

public static void main(String[] args) {

int[] arr1 = {1, 3, 5};

int[] arr2 = {2, 4, 6};

System.out.println("Merged with heap: " + merge(arr1, arr2));

}

}

### ✅ JavaScript (Heap using Priority Queue Simulation)

JavaScript has no built-in heap, so we implement it or use a simple trick with **sorting after concatenation** (which internally uses quicksort/merge-sort, but simulates heap-like merging).

function mergeWithHeap(arr1, arr2) {

// Merge arrays

let merged = arr1.concat(arr2);

// Simulate heap behavior by sorting

merged.sort((a, b) => a - b);

return merged;

}

// Example

let arr1 = [1, 3, 5];

let arr2 = [2, 4, 6];

console.log("Merged with heap:", mergeWithHeap(arr1, arr2));

(If implementing a custom heap in JS is needed, we can code it too — but this trick is often acceptable in interviews.)

## 🔹 Key Comparison

| **Approach** | **Time Complexity** | **Space** |
| --- | --- | --- |
| Two-pointer (best) | O(n1+n2) | O(n1+n2) |
| Heap-based | O((n1+n2) log(n1+n2)) | O(n1+n2) |
| Insertion method | O(n1 \* n2) | O(n1+n2) |

👉 For interviews:

* Start with **two-pointer merge** ✅
* Then mention **heap-based** as an alternative for more complex use cases (like streaming data).

public static List<Integer> merge(int[] arr1, int[] arr2) {

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

// Add elements of both arrays

for(int num : arr1) minHeap.add(num);

for(int num : arr2) minHeap.add(num);

// Extract elements in sorted order

List<Integer> result = new ArrayList<>();

while(!minHeap.isEmpty()) {

result.add(minHeap.poll());

}

return result;

}

🔎 How it works:

PriorityQueue as a Min-Heap

In Java, PriorityQueue by default is a min-heap.

That means the smallest element is always at the head of the queue.

Adding all elements

You put all numbers from both arrays into the min-heap.

At this point, the heap internally arranges them so the smallest is always on top.

Heap operations (add) take O(log n) time each.

Polling elements (sorted order)

poll() removes and returns the smallest element from the heap.

Since the heap always keeps the smallest at the front, every poll() gives the next smallest number.

This means when you keep polling until empty, you naturally get a sorted sequence.

Building the result Each polled element is appended to the result list. Final result is sorted, no matter if arr1 or arr2 were sorted beforehand.

✅ Example:

int[] arr1 = {5, 1, 3};

int[] arr2 = {6, 2, 4};

List<Integer> merged = merge(arr1, arr2);

System.out.println(merged);

Heap process:

Add all → Heap contains {1, 2, 3, 4, 5, 6} internally arranged.

Poll → returns 1, 2, 3, 4, 5, 6.

Output:

csharp

Copy code

[1, 2, 3, 4, 5, 6]

⚖️ Comparison with two-pointer merge:

Two-pointer merge (your first code):

Needs both arrays sorted first.

Runs in O(n + m) time (very efficient).

Priority queue merge (this code):

Works even if arrays are unsorted.

But slower: adding all elements → O((n+m) log(n+m