DAY-3/90 CTO BHAIYA

GITHUB -:DAY-3

? LeetCode Problem Solutions – Two Pointer & HashMap Mastery

Complete guide with crystal-clear explanations, intuitions, and step-by-step breakdowns for interview preparation.

? Table of Contents

1? LeetCode 1 – Two Sum

? Problem Link

LeetCode 1 - Two Sum

? Intuition / Approach

Problem: Given an **unsorted array**, find two numbers that add up to the target.

Key Insight: - For each number x, we need to find if `(target - x)` exists in the array - Instead of checking the entire array repeatedly (O(n^2)), we can use a **HashMap** to instantly check if the required partner exists

Think of it like this:

```
nums = [2, 7, 11, 15], target = 9

At index 0: current = 2
  → Need: 9 - 2 = 7
  → Is 7 in map? NO
  → Store: {2: 0}

At index 1: current = 7
  → Need: 9 - 7 = 2
  → Is 2 in map? YES! ✅
  → Return: [0, 1]
```

?Why Use HashMap?

Approach	Time Complexity	Why?
Brute Force (nested loops)	$O(n^2)$	Check every pair - slow!
HashMap	$O(n)$	Instant lookup - one pass only!

HashMap gives us: - ✓O(1) lookup time - ✓Single pass through array - ✓Store number → index mapping

⍰ Complexity Analysis

- **Time Complexity:** O(n) – we traverse array once
- **Space Complexity:** O(n) – HashMap stores at most n elements

⍰ Code with Detailed Comments

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        // HashMap to store: number → its index
        unordered_map<int, int> mp;

        for(int i = 0; i < nums.size(); i++) {
            // Calculate what number we need to reach target
            int diff = target - nums[i];

            // Check if required number already exists in map
            if(mp.find(diff) != mp.end()) {
                // Found! Return both indices
                return {mp[diff], i};
            }

            // Store current number with its index for future lookups
            mp[nums[i]] = i;
        }

        return {}; // No solution found
    }
};
```

⍰ Step-by-Step Execution

Example: `nums = [3, 2, 4], target = 6`

Step	i	nums[i]	diff	HashMap Before	Action	Result
1	0	3	6-3=3	{}	3 not in map	mp = {3:0}
2	1	2	6-2=4	{3:0}	4 not in map	mp = {3:0, 2:1}
3	2	4	6-4=2	{3:0, 2:1}	2 found! ✓	return [1, 2]

🗺 Mind Map / Key Points

```
Two Sum (Unsorted Array)
│
├─ Problem: Find 2 numbers = target
├─ Key Insight: Use HashMap for O(1) lookup
├─ Algorithm:
│  ├─ Calculate: diff = target - current
│  ├─ Check: Is diff in map?
│  │  ├─ YES → Return indices
│  │  └─ NO → Store current in map
│  └─ Continue
└─ Complexity: O(n) time, O(n) space
```

Remember: - HashMap = Fast Partner Finder 🔍 - diff = target - current - Store as you go, check before storing

---

2️⃣ LeetCode 167 – Two Sum II (Sorted Array)

🔗 Problem Link

LeetCode 167 - Two Sum II

💡 Intuition / Approach

Problem: Given a **sorted array**, find two numbers that add up to the target.

Key Insight: - Array is **sorted** → we can use **Two Pointers** technique - Start pointer at beginning, end pointer at end - Move pointers based on sum comparison

Visual Understanding:

```
numbers = [2, 7, 11, 15], target = 9

Start: [2, 7, 11, 15]
        ↑            ↑
      start         end

Sum = 2 + 15 = 17 > 9 (too big!)
→ Move end left

Next:  [2, 7, 11, 15]
        ↑       ↑
      start    end

Sum = 2 + 11 = 13 > 9 (still too big!)
→ Move end left

Next:  [2, 7, 11, 15]
        ↑   ↑
      start end

Sum = 2 + 7 = 9 ✅Found!
```

❓Why Use Two Pointers?

Approach	Time	Space	Why Better?
HashMap	O(n)	O(n)	Works but uses extra space
Two Pointers	**O(n)**	**O(1)**	✅No extra space + sorted advantage!

Two Pointers Logic: - Sum **too small** → increase sum → move **start** right → - Sum **too big** → decrease sum → move **end** left ← - Sum **equals target** → Found! ✅

⏱ Complexity Analysis

- **Time Complexity:** O(n) – single pass with two pointers
- **Space Complexity:** O(1) – no extra data structures

💻 Code with Detailed Comments

```
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        int start = 0;                      // Left pointer
        int end = numbers.size() - 1;      // Right pointer

        while(start < end) {
            int sum = numbers[start] + numbers[end];

            if(sum == target) {
                // Found! Return 1-based indices
                return {start + 1, end + 1};
            }
            else if(sum < target) {
                // Sum too small → need bigger number
                // Move start right to increase sum
                start++;
            }
            else {
                // Sum too big → need smaller number
                // Move end left to decrease sum
                end--;
            }
        }

        return {}; // No solution
    }
};
```

▯ Step-by-Step Execution

Example: `numbers = [2, 3, 4], target = 6`

Step	start	end	numbers[start]	numbers[end]	sum	Action
1	0	2	2	4	6	sum == target ✓
Result						return [1, 3]

Example 2: `numbers = [1, 2, 3, 4, 6], target = 6`

Step	start	end	sum	Comparison	Action
1	0	4	1+6=7	7 > 6	end–
2	0	3	1+4=5	5 < 6	start++
3	1	3	2+4=6	6 == 6 ✓	return [2, 4]

▯ Mind Map / Key Points

```
Two Sum II (Sorted Array)
│
├─ Key: Array is SORTED
├─ Technique: Two Pointers
├─ Pointer Movement:
│   ├─ sum < target → start++ (increase sum)
│   ├─ sum > target → end-- (decrease sum)
│   └─ sum == target → Found!
├─ Why Better?: O(1) space vs HashMap O(n)
└─ Return: 1-based indices
```

Remember: - Sorted array = Two Pointers opportunity ▯ - Compare sum, move accordingly - O(1) space advantage!

3▯ LeetCode 88 – Merge Sorted Array

▯ Problem Link

LeetCode 88 - Merge Sorted Array

▯ Intuition / Approach

Problem: Merge two sorted arrays into `nums1` **in-place** (nums1 has extra space).

Key Insight: - Merging from **start** → would overwrite nums1 elements ✗- Merging from **end** → empty space available ✓- Place largest elements first at the end

Visual Understanding:

```
nums1 = [1, 2, 3, 0, 0, 0], m = 3
nums2 = [2, 5, 6], n = 3
```

```
Step 1: Compare from end
[1, 2, 3, 0, 0, 0]
       ↑        ↑
       i        k
[2, 5, 6]
       ↑
       j

Compare: 3 vs 6 → 6 is bigger
Place 6 at position k

Step 2:
[1, 2, 3, 0, 0, 6]
       ↑     ↑
       i     k
[2, 5, 6]
    ↑
    j

Compare: 3 vs 5 → 5 is bigger
Place 5 at position k

... continue until done
```

❓Why Merge from End?

Approach	Issue	Solution
Start → End	Overwrites nums1 elements	❌Need extra space
End → Start	Empty space at end of nums1	✅No overwriting!

Advantages: - ✅No extra array needed - ✅Truly in-place - ✅Uses empty space smartly

⍰ Complexity Analysis

- **Time Complexity:** O(m + n) – single pass through both arrays
- **Space Complexity:** O(1) – no extra space, merge in-place

⍰ Code with Detailed Comments

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        // Three pointers: i (nums1), j (nums2), k (merged position)
        int i = m - 1;          // Last element of nums1's valid part
        int j = n - 1;          // Last element of nums2
        int k = m + n - 1;      // Last position in nums1 (total array)

        // Merge from end to start
        while(i >= 0 && j >= 0) {
            if(nums1[i] > nums2[j]) {
                // nums1's element is bigger
                nums1[k--] = nums1[i--];
            }
            else {
                // nums2's element is bigger or equal
                nums1[k--] = nums2[j--];
            }
        }

        // If nums2 has remaining elements, copy them
        // (If nums1 has remaining, they're already in place)
        while(j >= 0) {
            nums1[k--] = nums2[j--];
        }
    }
};
```

? Step-by-Step Execution

Example: `nums1 = [1, 2, 3, 0, 0, 0], m = 3, nums2 = [2, 5, 6], n = 3`

Step	i	j	k	nums1[i]	nums2[j]	Compare	Action	nums1 State
Start	2	2	5	3	6	3 < 6	Place nums2[j]	[1,2,3,0,0,6]
1	2	1	4	3	5	3 < 5	Place nums2[j]	[1,2,3,0,5,6]
2	2	0	3	3	2	3 > 2	Place nums1[i]	[1,2,3,3,5,6]
3	1	0	2	2	2	2 == 2	Place nums2[j]	[1,2,2,3,5,6]
4	1	-1	1	-	-	j done	Exit while	[1,2,2,3,5,6]
Final								**[1,2,2,3,5,6]** ✓

? Mind Map / Key Points

```
Merge Sorted Array
│
├─ Challenge: Merge in-place without extra space
├─ Key Insight: Merge from END → avoid overwriting
├─ Three Pointers:
│   ├─ i → last valid element in nums1
│   ├─ j → last element in nums2
│   └─ k → current merge position
├─ Algorithm:
│   ├─ Compare nums1[i] vs nums2[j]
│   ├─ Place LARGER at nums1[k]
│  ├─ Move corresponding pointer
│  └─ Copy remaining nums2 if any
└─ Complexity: O(m+n) time, O(1) space
```

Remember: - Start from END, not beginning! ⍰ - Three pointers: i, j, k - Pick the BIGGER element - Copy remaining nums2 if needed

---

4⍰ LeetCode 2824 – Count Pairs Whose Sum < Target

⍰ Problem Link

LeetCode 2824 - Count Pairs Whose Sum < Target

⍰ Intuition / Approach

Problem: Count all pairs `(i, j)` where `i < j` and `nums[i] + nums[j] < target`.

Key Insight: - Brute force: Check all pairs → $O(n^2)$ with nested loops - **Optimization:** Sort + Two Pointers → count multiple pairs at once! - If `nums[start] + nums[end] < target`, then ALL elements between start and end form valid pairs with start

Visual Understanding:

```
nums = [1, 2, 3, 4], target = 5

After sorting: [1, 2, 3, 4]

Step 1: start=0, end=3
[1, 2, 3, 4]
 ↑        ↑
start    end

sum = 1 + 4 = 5 (NOT < 5)
→ end--

Step 2: start=0, end=2
[1, 2, 3, 4]
 ↑     ↑
start end

sum = 1 + 3 = 4 < 5 ✅
Valid pairs: (1,2), (1,3) → count = 2
→ start++

Step 3: start=1, end=2
[1, 2, 3, 4]
    ↑  ↑
  start end
```

```
sum = 2 + 3 = 5 (NOT < 5)
→ end--

Done! Total count = 2
```

❓Why Sort First?

Approach	Time	Why?
Brute Force (nested loops)	$O(n^2)$	Check every pair one by one
Sort + Two Pointers	**O(n log n)**	✅Count multiple pairs at once!

Magic of Sorted Array: When `nums[start] + nums[end] < target`: - `nums[start] + nums[start+1] < target` (smaller number) - `nums[start] + nums[start+2] < target` (even smaller) - … ALL elements from start+1 to end work! - **Count += (end - start)** pairs in one step! ▯

▯ Complexity Analysis

- **Time Complexity:** O(n log n) – sorting takes O(n log n), two pointers scan is O(n)
- **Space Complexity:** O(1) – only using pointers

▯ Code with Detailed Comments

```
class Solution {
public:
    int countPairs(vector<int>& nums, int target) {
        // Sort to enable two-pointer technique
        sort(nums.begin(), nums.end());

        int start = 0;
        int end = nums.size() - 1;
        int count = 0;

        while(start < end) {
            int sum = nums[start] + nums[end];

            if(sum < target) {
                // KEY INSIGHT: If nums[start] + nums[end] < target,
                // then nums[start] + ALL elements from (start+1 to end)
                // will also be < target (because array is sorted)
                count += (end - start);
                start++;  // Move to next element
            }
            else {
                // Sum >= target, need smaller sum
                end--;    // Move end pointer left
            }
        }
        return count;
    }
};
```

▯ Step-by-Step Execution

Example: `nums = [-1, 1, 2, 3, 1], target = 2`

After sorting: `nums = [-1, 1, 1, 2, 3]`

Step	start	end	nums[start]	nums[end]	sum	Comparison	Action	Count Added	Total Count
1	0	4	-1	3	2	2 == 2	end–	0	0
2	0	3	-1	2	1	1 < 2 ✓	start++	3	3
3	1	3	1	2	3	3 > 2	end–	0	3
4	1	2	1	1	2	2 == 2	end–	0	3
Done	1	1	-	-	-	start==end	Stop	-	**3**

Valid pairs: (-1,1), (-1,1), (-1,2) = **3 pairs** ✓

▯ Mind Map / Key Points

```
Count Pairs (Sum < Target)
│
├─ Step 1: SORT the array
├─ Step 2: Two Pointers (start, end)
├─ Logic:
│   ├─ sum < target:
│   │   ├─ ALL pairs from start to end are valid
│   │   ├─ count += (end - start)
│   │   └─ start++
│   └─ sum >= target:
│       └─ end-- (reduce sum)
├─ Why Sort?: Enables counting multiple pairs at once
└─ Complexity: O(n log n) time, O(1) space
```

Remember: - Sort first! ▯ - sum < target → count += (end - start) - One pointer move can count MULTIPLE pairs - Sorted array = efficiency boost

---

5▯ LeetCode 15 – 3Sum

▯ Problem Link

LeetCode 15 - 3Sum

▯ Intuition / Approach

Problem: Find all **unique triplets** `[a, b, c]` where `a + b + c = 0`.

Key Insight: - Convert 3Sum problem → 2Sum problem! - Fix one element → find two elements that sum to `-fixed_element` - Sort array to easily skip duplicates

Breakdown: 1. **Sort** the array 2. **Fix** first element (loop with `i`) 3. Use **Two Pointers** (left, right) for remaining two elements 4. **Skip duplicates** to ensure unique triplets

Visual Understanding:

```
nums = [-1, 0, 1, 2, -1, -4]

After sorting: [-4, -1, -1, 0, 1, 2]

Fix i=1 (nums[i]=-1):
Need: 0 - (-1) = 1

[-4, -1, -1, 0, 1, 2]
      ↑   ↑        ↑
      i   l        r

sum = -1 + (-1) + 2 = 0 ✅
Found: [-1, -1, 2]

Skip duplicates, continue...
```

❓Why Sort + Two Pointers?

Approach	Time	Issues
Three nested loops	O(n^3)	Too slow + hard to avoid duplicates
Sort + Fix one + 2 Pointers	**O(n^2)**	✅Efficient + easy duplicate handling

Advantages: - ✅Reduces from O(n^3) to O(n^2) - ✅Sorted array helps skip duplicates easily - ✅Two pointer technique for inner elements

▯ Complexity Analysis

- **Time Complexity:** O(n^2) – O(n log n) for sort + O(n^2) for loop with two pointers
- **Space Complexity:** O(1) – excluding output array (sorting is typically O(log n) for space)

▯ Code with Detailed Comments

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> ans;
```

```cpp
        // Step 1: Sort to enable two-pointer and skip duplicates
        sort(nums.begin(), nums.end());

        // Step 2: Fix first element
        for(int i = 0; i < nums.size(); i++) {
            // Skip duplicate first elements
            if(i > 0 && nums[i] == nums[i-1])
                continue;

            // Step 3: Two pointers for remaining two elements
            int l = i + 1;              // Left pointer
            int r = nums.size() - 1;    // Right pointer

            while(l < r) {
                int sum = nums[i] + nums[l] + nums[r];

                if(sum == 0) {
                    // Found a valid triplet!
                    ans.push_back({nums[i], nums[l], nums[r]});

                    // Skip duplicate left elements
                    while(l < r && nums[l] == nums[l+1])
                        l++;

                    // Skip duplicate right elements
                    while(l < r && nums[r] == nums[r-1])
                        r--;

                    // Move both pointers
                    l++;
                    r--;
                }
                else if(sum < 0) {
                    // Sum too small, need bigger number
                    l++;
                }
                else {
                    // Sum too big, need smaller number
                    r--;
                }
            }
        }

        return ans;
    }
};
```

▯ Step-by-Step Execution

Example: `nums = [-1, 0, 1, 2, -1, -4]`

After sorting: `[-4, -1, -1, 0, 1, 2]`

i	nums[i]	l	r	sum	Action	Result
0	-4	1	5	-4+-1+2=-3	sum<0, l++	-
0	-4	2	5	-4+-1+2=-3	sum<0, l++	-
0	-4	3	5	-4+0+2=-2	sum<0, l++	-
0	-4	4	5	-4+1+2=-1	sum<0, l++	-
0	-4	5	5	l==r	Next i	-
1	**-1**	2	5	-1+-1+2=0 ✓	Found!	**[-1,-1,2]**
1	-1	3	4	-1+0+1=0 ✓	Found!	**[-1,0,1]**
2	-1	-	-	Skip (duplicate)	continue	-
3+	-	-	-	No valid triplets	-	-

Final Answer: `[[-1,-1,2], [-1,0,1]]` ✓

▯ Mind Map / Key Points

```
3Sum Problem
│
├─ Goal: Find unique triplets with sum = 0
├─ Strategy: Fix one + Two Pointers for rest
│
├─ Algorithm:
│   ├─ Step 1: SORT array
│   ├─ Step 2: Loop with i (fix first element)
│   │   └─ Skip duplicates: if nums[i]==nums[i-1]
│   ├─ Step 3: Two Pointers (l, r)
│   │   ├─ sum == 0: Add to result, skip duplicates, move both
│   │   ├─ sum < 0: l++ (need bigger)
│   │   └─ sum > 0: r-- (need smaller)
│   └─ Return all unique triplets
│
├─ Duplicate Handling:
│   ├─ Skip duplicate i values
│   ├─ Skip duplicate l values
│   └─ Skip duplicate r values
│
└─ Complexity: O(n²) time, O(1) space
```

Remember: - Sort first! ▯ - Fix i, then 2Sum problem for rest - Skip duplicates at ALL levels (i, l, r) - sum logic: <0 → l++, >0 → r–, ==0 → found!

▯ Final Comparison Table

Problem	Technique	Key Insight	Time	Space
Two Sum	HashMap	Unsorted → instant lookup	O(n)	O(n)
Two Sum II	Two Pointers	Sorted → move based on sum	O(n)	O(1)
Merge Sorted	Three Pointers	Merge from END	O(m+n)	O(1)
Count Pairs	Sort + Two Pointers	count += (end-start)	O(n log n)	O(1)
3Sum	Sort + Fix + Two Pointers	Fix one, 2Sum for rest	O(n^2)	O(1)

▯ Quick Revision Checklist

When to use HashMap: - ✅Unsorted array - ✅Need O(1) lookup - ✅Finding pairs/complements

When to use Two Pointers: - ✅Sorted array (or sort first) - ✅Need O(1) space - ✅Finding pairs/triplets with sum conditions

Golden Rules: 1. **Sorted array** → Think Two Pointers first 2. **Unsorted + need pairs** → Think HashMap 3. **Merge operations** → Work from END 4. **Avoid duplicates** → Sort + skip same values 5. **3Sum/4Sum** → Fix element(s) + reduce to 2Sum