
WEEK 1 – DAY 3

Two Pointers, Optimization Thinking, System Trade-offs

Sony evaluates:

- Ability to **reduce brute force**
- Pointer movement logic
- Clean in-place algorithms
- Practical **engineering trade-offs** (not textbook answers)

---

1. Two Pointers – Core Concept

---

1. Two Pointers – Core Concept

What You Should Say (Spoken)

"Two pointers is an optimization technique where we use two indices to traverse a data structure in a controlled way, either moving toward each other or in the same direction, to reduce time and space complexity compared to nested loops."

When Sony Expects Two Pointers (Spoken)

"Sony expects two pointers when the input is sorted, when in-place modification is required, when solving pair or boundary problems, or when memory usage must be minimal."

Warm-Up Interview Questions

1. Why two pointers over nested loops?
2. When does two pointers fail?
3. Difference between two pointers and sliding window?
4. Does it require sorted input?
5. How do you decide pointer movement?

1️⃣ Why two pointers over nested loops?

Spoken (Say This)

"Two pointers reduce time complexity from $O(n^2)$ to $O(n)$ by eliminating redundant comparisons and leveraging order or constraints in the data."

Related Question

Two Sum in Sorted Array

Python (with input)

```
nums = [1, 2, 4, 6, 10]
target = 8

def two_sum_sorted(nums, target):
    left, right = 0, len(nums) - 1
    while left < right:
        s = nums[left] + nums[right]
        if s == target:
            return left, right
        elif s < target:
            left += 1
        else:
            right -= 1
    return -1

print(two_sum_sorted(nums, target))  # Output: (1, 3)
```

JavaScript (with input)

```
const nums = [1, 2, 4, 6, 10];
const target = 8;

function twoSumSorted(nums, target) {
    let left = 0, right = nums.length - 1;
    while (left < right) {
        const sum = nums[left] + nums[right];
        if (sum === target) return [left, right];
        sum < target ? left++ : right--;
    }
```

```
    return -1;
}

console.log(twoSumSorted(nums, target)); // [1, 3]
```

2️⃣ When does two pointers fail?

Spoken

"Two pointers fail when there is no order or monotonic property to guide pointer movement, such as unsorted arrays without constraints."

Related Question

Two Sum in Unsorted Array (Fails → Use Hash Map)

Python

```
nums = [3, 1, 4, 2]
target = 6

def two_sum_unsorted(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        if target - num in seen:
            return seen[target - num], i
        seen[num] = i

print(two_sum_unsorted(nums, target))  # Output: (2, 3)
```

JavaScript

```
const nums = [3, 1, 4, 2];
const target = 6;

function twoSumUnsorted(nums, target) {
    const map = new Map();
    for (let i = 0; i < nums.length; i++) {
```

```
        if (map.has(target - nums[i])) {
            return [map.get(target - nums[i]), i];
        }
        map.set(nums[i], i);
    }
}

console.log(twoSumUnsorted(nums, target)); // [2, 3]
```

---

3 Difference between two pointers and sliding window?

Spoken

"Two pointers is a general technique, while sliding window is a specialized form where both pointers move forward to maintain a fixed or dynamic window."

Related Question

Maximum Sum Subarray of Size K

Python

```
nums = [2, 1, 5, 1, 3, 2]
k = 3

def max_sum_subarray(nums, k):
    window_sum = sum(nums[:k])
    max_sum = window_sum

    for i in range(k, len(nums)):
        window_sum += nums[i] - nums[i - k]
        max_sum = max(max_sum, window_sum)

    return max_sum

print(max_sum_subarray(nums, k))  # Output: 9
```

JavaScript

```
const nums = [2, 1, 5, 1, 3, 2];

const k = 3;


function maxSumSubarray(nums, k) {
   let windowSum = nums.slice(0, k).reduce((a, b) => a + b, 0);
   let maxSum = windowSum;


   for (let i = k; i < nums.length; i++) {
      windowSum += nums[i] - nums[i - k];
      maxSum = Math.max(maxSum, windowSum);
   }
   return maxSum;
}


console.log(maxSumSubarray(nums, k)); // 9
```

---

4 Does two pointers require sorted input?

Spoken

"Not always. Some problems need sorted input, but many in-place partition problems work on unsorted arrays."

Related Question

Move Zeros to End

Python

```
nums = [0, 1, 0, 3, 12]


def move_zeros(nums):
   j = 0
   for i in range(len(nums)):
      if nums[i] != 0:
```

```
            nums[i], nums[j] = nums[j], nums[i]
            j += 1
    return nums


print(move_zeros(nums))  # Output: [1, 3, 12, 0, 0]
```

JavaScript

```
const nums = [0, 1, 0, 3, 12];


function moveZeros(nums) {
    let j = 0;
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] !== 0) {
            [nums[i], nums[j]] = [nums[j], nums[i]];
            j++;
        }
    }
    return nums;
}


console.log(moveZeros(nums)); // [1, 3, 12, 0, 0]
```

---

5 How do you decide pointer movement?

Spoken

"Pointer movement is driven by whether the current state moves us closer to or further from the target condition."

Related Question

Container With Most Water

Python

```
height = [1,8,6,2,5,4,8,3,7]
```

```
def max_area(height):
    left, right = 0, len(height) - 1
    max_area = 0

    while left < right:
        area = (right - left) * min(height[left], height[right])
        max_area = max(max_area, area)

        if height[left] < height[right]:
            left += 1
        else:
            right -= 1

    return max_area

print(max_area(height))  # Output: 49
```

JavaScript

```
const height = [1,8,6,2,5,4,8,3,7];

function maxArea(height) {
    let left = 0, right = height.length - 1;
    let maxArea = 0;

    while (left < right) {
        const area = (right - left) * Math.min(height[left], height[right]);
        maxArea = Math.max(maxArea, area);

        height[left] < height[right] ? left++ : right--;
    }
    return maxArea;
}
```

```
console.log(maxArea(height)); // 49
```

Final Sony One-Liner (Use This)

"I choose two pointers when I can deterministically move indices to reduce the search space while keeping time O(n) and space O(1)."

---

2. Problem 1: Remove Duplicates from Sorted Array

Interview Question

Given a sorted array, remove duplicates in-place and return the number of unique elements.

Constraints (Say This)

"The solution must modify the array in-place, use O(1) extra space, and preserve the relative order of elements."

Optimal Two-Pointer Approach

Key Insight (Say This)

"Because the array is sorted, all duplicates appear next to each other. I maintain a slow pointer that tracks the position of the last unique element and a fast pointer that scans the array."

Spoken Explanation (Step-by-Step)

"I start slow at index 0.
Then I move a fast pointer from index 1 onward.
Whenever nums[fast] is different from nums[slow], I increment slow and copy nums[fast] to nums[slow].
At the end, slow + 1 represents the number of unique elements."

Python Solution (With Input Included)

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


def removeDuplicates(nums):
    if not nums:
        return 0
```

```
    slow = 0
    for fast in range(1, len(nums)):
        if nums[fast] != nums[slow]:
            slow += 1
            nums[slow] = nums[fast]

    return slow + 1

length = removeDuplicates(nums)

print(length)         # Output: 4
print(nums[:length])   # Output: [1, 2, 3, 4]
```

JavaScript Solution (With Input Included)

```
const nums = [1, 1, 2, 2, 3, 3, 4];

function removeDuplicates(nums) {
    if (nums.length === 0) return 0;

    let slow = 0;
    for (let fast = 1; fast < nums.length; fast++) {
        if (nums[fast] !== nums[slow]) {
            slow++;
            nums[slow] = nums[fast];
        }
    }
    return slow + 1;
}

const length = removeDuplicates(nums);
```

```
console.log(length);              // Output: 4
console.log(nums.slice(0, length)); // Output: [1, 2, 3, 4]
```

Complexity (Say This)

"The algorithm runs in O(n) time since each element is visited once, and O(1) extra space because the array is modified in-place."

Sony Follow-Ups

1. What if array is not sorted?
2. Allow duplicates at most twice?
3. Return the actual array?
4. Why not use a Set?
5. How would this behave for large arrays?

---

1 What if the array is not sorted?

Spoken

"This approach relies on sorted order. If the array is unsorted, I would either sort it first or use a hash set, but both change constraints."

Trade-off

- Sorting → O(n log n)
- HashSet → O(n) space (violates constraint)

---

2 Allow duplicates at most twice?

Spoken

"I would allow insertion only if nums[fast] is different from nums[slow−1]."

Python

```
nums = [1,1,1,2,2,3]


def removeDuplicatesTwice(nums):
    if len(nums) <= 2:
        return len(nums)
```

```
    slow = 2
    for fast in range(2, len(nums)):
        if nums[fast] != nums[slow - 2]:
            nums[slow] = nums[fast]
            slow += 1

    return slow

length = removeDuplicatesTwice(nums)
print(nums[:length])  # Output: [1, 1, 2, 2, 3]
```

---

3 Return the actual array?

Spoken

"The array is already modified in-place. The returned length indicates how much of the array is valid."

---

4 Why not use a Set?

Spoken

"A Set uses extra memory, which violates the O(1) space requirement and breaks in-place constraints."

---

5 How would this behave for large arrays?

Spoken

"It scales efficiently because it performs a single linear scan and does not allocate additional memory."

Sony One-Line Summary (Memorize This)

"This solution leverages sorted input to eliminate duplicates in linear time using a two-pointer, in-place strategy."

---

3. Problem 2: Container With Most Water

Interview Question

Given an array where each element represents the height of a vertical line, find two lines that together with the x-axis form a container holding the maximum amount of water.

Brute Force (Say This)

"The brute-force approach is to try all possible pairs of lines and compute the area for each pair, which results in $O(n^2)$ time complexity and is not optimal."

Optimal Two-Pointer Approach

Key Insight (Say This – Very Important)

"The area is limited by the shorter height. To potentially increase area, we must move the pointer pointing to the smaller height, because moving the taller one cannot increase the limiting height."

Spoken Explanation (Step-by-Step)

"I place one pointer at the start and one at the end of the array.
At each step, I calculate the area using the distance between pointers and the minimum of the two heights.
Then, I move the pointer pointing to the shorter line, because that is the only way to possibly increase the area."

Python Solution (With Input Included)

```
height = [1, 8, 6, 2, 5, 4, 8, 3, 7]


def maxArea(height):
    left, right = 0, len(height) - 1
    max_area = 0


    while left < right:
        width = right - left
        area = min(height[left], height[right]) * width
        max_area = max(max_area, area)


        if height[left] < height[right]:
```

```
            left += 1
        else:
            right -= 1

    return max_area

print(maxArea(height))  # Output: 49
```

JavaScript Solution (With Input Included)

```
const height = [1, 8, 6, 2, 5, 4, 8, 3, 7];

function maxArea(height) {
    let left = 0, right = height.length - 1;
    let maxArea = 0;

    while (left < right) {
        const width = right - left;
        const area = Math.min(height[left], height[right]) * width;
        maxArea = Math.max(maxArea, area);

        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }
    return maxArea;
}

console.log(maxArea(height)); // Output: 49
```

Complexity (Say This)

“The algorithm runs in O(n) time because each pointer moves at most n times, and O(1) extra space since no additional memory is used.”

Sony Follow-Ups

1. Why move the smaller height?
2. Can moving the larger height help?
3. What if heights are streamed?
4. Can this be parallelized?
5. Real-world analogy?

---

1 Why move the smaller height?

Spoken

“The container height is limited by the shorter line. Moving the taller line keeps the same limiting height while reducing width, so it cannot produce a better area.”

---

2 Can moving the larger height help?

Spoken

“No. Moving the larger height cannot increase the minimum height, and it always decreases the width, so area can only decrease or stay the same.”

---

3 What if heights are streamed?

Spoken

“This algorithm requires access to both ends, so it cannot be applied directly to streaming data. A streaming version would need approximations or buffering.”

---

4 Can this be parallelized?

Spoken

“Not efficiently, because each pointer decision depends on the previous comparison. However, brute-force checks could be parallelized at the cost of performance.”

5️⃣ Real-world analogy?

Spoken

“It’s similar to choosing two walls to build a water tank—the water level is determined by the shorter wall, not the taller one.”

Sony One-Line Summary (Memorize This)

“This problem demonstrates how two pointers exploit geometric constraints to reduce a quadratic problem to linear time.”

---

4. Sony-Level Variations (Very Likely)

1. Remove element equal to k
2. Pair with target sum in sorted array
3. Squaring sorted array
4. Trapping rain water
5. String palindrome validation

1️⃣ Remove Element Equal to k

Interview Question

Remove all occurrences of k **in-place** and return the new length.

Key Insight (Say This)

“I use a slow pointer to overwrite elements that are not equal to k while scanning the array once.”

Python (with input)

```
nums = [3, 2, 2, 3, 4]
k = 3


def removeElement(nums, k):
    slow = 0
    for fast in range(len(nums)):
        if nums[fast] != k:
            nums[slow] = nums[fast]
```

```
            slow += 1
    return slow

length = removeElement(nums, k)
print(length)          # Output: 3
print(nums[:length])    # Output: [2, 2, 4]
```

JavaScript (with input)

```
const nums = [3, 2, 2, 3, 4];
const k = 3;

function removeElement(nums, k) {
    let slow = 0;
    for (let fast = 0; fast < nums.length; fast++) {
        if (nums[fast] !== k) {
            nums[slow++] = nums[fast];
        }
    }
    return slow;
}

const length = removeElement(nums, k);
console.log(length);            // 3
console.log(nums.slice(0, length)); // [2, 2, 4]
```

---

2 Pair With Target Sum (Sorted Array)

Key Insight (Say This)

"Since the array is sorted, I compare the sum and move pointers inward to reduce the search space."

Python

```
nums = [1, 2, 4, 6, 10]
```

```
target = 8

def pairSum(nums, target):
    left, right = 0, len(nums) - 1
    while left < right:
        s = nums[left] + nums[right]
        if s == target:
            return left, right
        elif s < target:
            left += 1
        else:
            right -= 1
    return -1

print(pairSum(nums, target))  # Output: (1, 3)
```

JavaScript

```
const nums = [1, 2, 4, 6, 10];
const target = 8;

function pairSum(nums, target) {
    let left = 0, right = nums.length - 1;
    while (left < right) {
        const sum = nums[left] + nums[right];
        if (sum === target) return [left, right];
        sum < target ? left++ : right--;
    }
    return -1;
}

console.log(pairSum(nums, target)); // [1, 3]
```

3️⃣ Squaring a Sorted Array

Key Insight (Say This)

“Negative numbers produce larger squares. I compare absolute values from both ends.”

Python

```
nums = [-7, -3, 2, 3, 11]


def sortedSquares(nums):
    n = len(nums)
    res = [0] * n
    left, right = 0, n - 1

    for i in range(n - 1, -1, -1):
        if abs(nums[left]) > abs(nums[right]):
            res[i] = nums[left] ** 2
            left += 1
        else:
            res[i] = nums[right] ** 2
            right -= 1
    return res


print(sortedSquares(nums))  # Output: [4, 9, 9, 49, 121]
```

JavaScript

```
const nums = [-7, -3, 2, 3, 11];


function sortedSquares(nums) {
    const n = nums.length;
    const res = new Array(n);
    let left = 0, right = n - 1;
```

```
    for (let i = n - 1; i >= 0; i--) {
        if (Math.abs(nums[left]) > Math.abs(nums[right])) {
            res[i] = nums[left] ** 2;
            left++;
        } else {
            res[i] = nums[right] ** 2;
            right--;
        }
    }
    return res;
}

console.log(sortedSquares(nums)); // [4, 9, 9, 49, 121]
```

---

4 Trapping Rain Water (Hard – Sony Favorite)

Key Insight (Say This)

“Water trapped depends on the minimum of leftMax and rightMax. I update the smaller side first.”

Python

```
height = [4,2,0,3,2,5]

def trap(height):
    left, right = 0, len(height) - 1
    left_max = right_max = 0
    water = 0

    while left < right:
        if height[left] < height[right]:
            left_max = max(left_max, height[left])
            water += left_max - height[left]
```

```
            left += 1
        else:
            right_max = max(right_max, height[right])
            water += right_max - height[right]
            right -= 1

    return water

print(trap(height))  # Output: 9
```

JavaScript

```
const height = [4,2,0,3,2,5];

function trap(height) {
    let left = 0, right = height.length - 1;
    let leftMax = 0, rightMax = 0, water = 0;

    while (left < right) {
        if (height[left] < height[right]) {
            leftMax = Math.max(leftMax, height[left]);
            water += leftMax - height[left];
            left++;
        } else {
            rightMax = Math.max(rightMax, height[right]);
            water += rightMax - height[right];
            right--;
        }
    }
    return water;
}
```

```
console.log(trap(height)); // 9
```

5️⃣ String Palindrome Validation

Key Insight (Say This)

“I ignore non-alphanumeric characters and compare from both ends.”

Python

```
s = "A man, a plan, a canal: Panama"


def isPalindrome(s):
    left, right = 0, len(s) - 1
    while left < right:
        while left < right and not s[left].isalnum():
            left += 1
        while left < right and not s[right].isalnum():
            right -= 1
        if s[left].lower() != s[right].lower():
            return False
        left += 1
        right -= 1
    return True


print(isPalindrome(s))  # Output: True
```

JavaScript

```
const s = "A man, a plan, a canal: Panama";


function isPalindrome(s) {
    let left = 0, right = s.length - 1;


    while (left < right) {
```

```
    while (left < right && !/[a-zA-Z0-9]/.test(s[left])) left++;
    while (left < right && !/[a-zA-Z0-9]/.test(s[right])) right--;
    if (s[left].toLowerCase() !== s[right].toLowerCase()) return false;
    left++;
    right--;
  }
  return true;
}

console.log(isPalindrome(s)); // true
```

Sony Final Summary (Memorize)

"Two pointers allow linear-time, constant-space solutions by exploiting ordering, symmetry, or constraints in the input."

---

5. Project (Concept): Where to Store Audit Logs

DB vs File vs Hybrid

Interview Framing

"Decide the most appropriate storage for audit logs in ControlHub."

Option 1: Database Storage

Pros

- Easy querying
- Indexing on user/time
- ACID compliance

Cons

- High write volume pressure
- Expensive storage
- Slower inserts

When to Use

- Medium scale

- Compliance-heavy systems
- Frequent queries

Option 2: File-Based Logs

Pros

- Extremely fast writes
- Cheap storage
- Append-only

Cons

- Hard to query
- No indexing
- Manual parsing

When to Use

- Very high volume
- Rare querying
- Backup logs

Option 3: Hybrid (Best Answer for Sony)

Recommended Architecture

App → Kafka → File Storage (S3)

→ DB (Indexed, recent logs)

Why Sony Likes This Answer

- Scalable
- Cost-efficient
- Separation of concerns
- Industry-grade design

How to Explain (Script)

"I'd use a hybrid approach—write logs asynchronously to a queue, store recent logs in a database for quick queries, and archive older logs in file storage."

---

6. Interview Follow-Ups on Audit Storage

1. How do you ensure durability?
2. How to prevent log tampering?
3. How long to retain logs?
4. How to encrypt logs?
5. How to replay logs?

---

7. End-of-Day Sony Checklist

You are ready if you can:

- Identify two-pointer problems instantly
- Code without extra space
- Justify pointer movement logically
- Explain storage trade-offs confidently

Next Step (Choose One)

1. Day-4 (Sliding Window)
2. Full Sony mock interview (DSA + design)
3. Convert Day-1 to Day-3 into a **GitHub prep repo**
4. Create **spoken answer scripts** for all problems

Tell me which one you want next.