Practice Sheet 3

Bhai Anurag 🔥

Ye question **LC 1552 – Magnetic Force Between Two Balls** Google-level ka hai because:

- Pattern = **Binary Search on Answer**
- Concept = **Greedy + Monotonicity**
- Classic variation = **Aggressive Cows (GFG)**
- Interview favourite 🚀

Since tum Google target kar rahe ho, main isko interview-style deeply explain karunga.

🧠 Problem Summary

Given:

- `position[]` → sorted nahi hai (basket positions)
- `m` balls place karni hain
- Goal → maximize minimum distance between any two balls

Return → **maximum possible minimum magnetic force**

🔎 Example

```
position = [1,2,3,4,7]
m = 3
```

Possible placements:

- (1,4,7) → min distance = 3
- (1,3,7) → min distance = 2
- (1,2,4) → min distance = 1

Maximum possible minimum distance = **3**

🚨 Important Observation

We are maximizing the **minimum distance**

This is a classic sign of:

> ✅ Binary Search on Answer

Because:

- If distance `d` possible hai
- Then any `d' < d` bhi possible hoga

  Monotonic behaviour 🔥

⚙️ Step 1: Brute Force Approach

Idea

1. Sort array
2. Try every possible minimum distance from 1 to max-min
3. For each distance:
   - Try placing balls greedily
   - Check if m balls place ho sakti hain

Time Complexity

- Distance range = `max - min` (up to 10^9)
- For each distance → O(n)

Worst case = O(n * range) ❌

Very slow.

Java Code (Brute)

```
import java.util.*;

class Solution {
    // Function to check if we can place m balls
    // such that minimum distance between them is at least
'dist'
    public boolean canPlace(int[] position, int m, int dis
t) {

        int count = 1; // First ball at first position
        int lastPlaced = position[0];

        for (int i = 1; i < position.length; i++) {

            // If distance from last placed ball >= require
d dist
            if (position[i] - lastPlaced >= dist) {
                count++;
                lastPlaced = position[i];

                if (count == m) {
                    return true;
                }
            }
        }

        return false;
    }

    public int maxDistance(int[] position, int m) {

        Arrays.sort(position);

        int maxPossible = position[position.length - 1] - p
osition[0];
        int answer = 0;

        // Try every distance
        for (int dist = 1; dist <= maxPossible; dist++) {
            if (canPlace(position, m, dist)) {
                answer = dist; // keep updating max valid d
istance
            }
        }

        return answer;
    }
}
```

⚡ Step 2: Better Approach

Same logic, but instead of iterating linearly over all distances,

we use Binary Search.

Why possible?

Because of this monotonic property:

```
If distance d works
→ then all distances < d also work
```

This is strictly monotonic.

🎯 Step 3: Optimal Approach (Binary Search on Answer)

Search Space

Minimum distance = 1

Maximum distance = position[n-1] - position[0]

Binary Search on this range.

Algorithm

1. Sort positions

2. low = 1
3. high = max-min
4. While low <= high:
    - mid = candidate distance
    - If possible:
        - store answer
        - move right (low = mid + 1)
    - Else:
        - move left (high = mid - 1)

🧠 Why Greedy Works?

When checking feasibility:

Always place ball at earliest possible position.

Because:

- We want to maximize spacing
- Greedy ensures maximum balls placed

This is provably optimal.

⏱️ Time Complexity

- Sorting → O(n log n)
- Binary Search → log(range)
- Each check → O(n)

Total:

```
O(n log n + n log(range))
```

Perfect for 10^5 constraints 🔥

💎 Optimal Java Code (Interview Ready)

```
import java.util.*;

class Solution {

    // Helper function to check if we can place 'm' balls
    // with at least 'dist' minimum distance
    private boolean canPlaceBalls(int[] position, int m, in
t dist) {

        // Place first ball at first position
        int count = 1;
        int lastPlaced = position[0];

        for (int i = 1; i < position.length; i++) {

            // If current position is at least 'dist' away
            if (position[i] - lastPlaced >= dist) {

                count++;                     // Place ball
                lastPlaced = position[i];    // Update last
placed

                if (count == m) {
                    return true;            // Successfully
placed all balls
                }
            }
        }

        return false; // Not enough balls placed
    }

    public int maxDistance(int[] position, int m) {

        // Step 1: Sort the basket positions
        Arrays.sort(position);
        // Step 2: Define binary search boundaries
        int low = 1;  // Minimum possible distance
        int high = position[position.length - 1] - position
[0];

        int answer = 0;

        // Step 3: Binary Search on Answer
        while (low <= high) {

            int mid = low + (high - low) / 2;  // Candidate
distance

            if (canPlaceBalls(position, m, mid)) {

                answer = mid;   // mid is valid
                low = mid + 1;  // Try bigger distance

            } else {

                high = mid - 1; // Reduce distance
            }
        }

        return answer;
    }
}
```

🧠 Interview Explanation (How You Should Speak)

If interviewer asks:

> Why binary search?

You say:

"Because we are maximizing a minimum value and the feasibility function is monotonic. If a certain minimum distance is possible, then all smaller distances are also possible. So we can binary search on the answer space."

🔥 That sounds Google-ready.

🧩 Pattern Recognition (Very Important)

This question belongs to:

```
Binary Search on Answer Pattern
```

Other Google-style problems in same pattern:

- LC 1011 – Capacity to Ship Packages
- LC 875 – Koko Eating Bananas
- LC 1482 – Minimum Days to Make m Bouquets
- Aggressive Cows (GFG)
- Allocate Books
- Split Array Largest Sum

You should master this pattern fully.

🚀 Next Level (Google Prep Insight)

Google interviewer may twist:

- Return actual positions
- 2D coordinates
- Balls with weight
- Dynamic addition

So concept clarity > memorization.

If you want, next I can:

- 🔥 Do full dry run visualization
- 🔥 Give you 5 hard follow-up variations

- 🔥 Compare LC 1552 vs LC 1011 deeply
- 🔥 Give you Google mock interview version

Bata bhai next kya todna hai 😎

Bhai Anurag 🔥

Ab aa gaye asli Google-level HARD par 😎

LC 4 – Median of Two Sorted Arrays

Ye question FAANG interviews ka legend hai.

Isme:

- Binary Search
- Partition logic
- Edge case handling
- Mathematical reasoning

Sab kuch test hota hai.

Agar ye crack ho gaya — tumhara Binary Search level next tier pe chala jayega 🚀

🧠 Problem Statement

Given:

```
nums1 (size m)
nums2 (size n)
```

Both sorted.

Return → median of merged array

BUT ❌ you cannot merge fully

Required Time Complexity:

```
O(log(min(m, n)))
```

🔎 Example

Example 1:

```
nums1 = [1,3]
nums2 = [2]

Merged → [1,2,3]
Median = 2
```

Example 2:

```
nums1 = [1,2]
nums2 = [3,4]

Merged → [1,2,3,4]
Median = (2+3)/2 = 2.5
```

🧠 Step 1: Brute Force

Idea:

Merge both arrays → sort → find median

Time:

```
O(m + n)
```

But question demands:

```
O(log(min(m,n)))
```

So brute rejected ❌

Brute Java Code

```java
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[]
nums2) {

        int m = nums1.length;
        int n = nums2.length;

        int[] merged = new int[m + n];

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

        // Merge step (like merge sort)
        while (i < m && j < n) {
            if (nums1[i] < nums2[j]) {
                merged[k++] = nums1[i++];
            } else {
                merged[k++] = nums2[j++];
            }
        }

        while (i < m) merged[k++] = nums1[i++];
        while (j < n) merged[k++] = nums2[j++];

        int total = m + n;

        if (total % 2 == 1) {
            return merged[total / 2];
        } else {
            return (merged[total / 2 - 1] + merged[total /
2]) / 2.0;
        }
    }
}
```

🚀 Step 2: Optimal Approach (Binary Search on Partition)

This is the REAL Google solution.

🧠 Core Idea

Instead of merging,

We partition both arrays such that:

```
Left side elements = Right side elements
```

Condition for correct partition:

```
maxLeft1 <= minRight2
AND
maxLeft2 <= minRight1
```

If this holds → we found median.

🎯 Key Insight

We binary search only on the smaller array.

Why?

Because we want:

```
O(log(min(m,n)))
```

🧩 Visualization

Suppose:

```
nums1 = [1, 3, 8, 9, 15]
nums2 = [7, 11, 18, 19, 21, 25]
```

Total length = 11

Median position = (m+n+1)/2 = 6

We try partitioning nums1 at some index:

```
cut1 + cut2 = 6
```

We compute:

```
cut2 = 6 - cut1
```

Then check boundary values.

🧠 Boundary Values

For partition:

```
Left1  | Right1
Left2  | Right2
```

We define:

```
maxLeft1  = (cut1 == 0) ? -∞ : nums1[cut1 - 1]
minRight1 = (cut1 == m) ? +∞ : nums1[cut1]

maxLeft2  = (cut2 == 0) ? -∞ : nums2[cut2 - 1]
minRight2 = (cut2 == n) ? +∞ : nums2[cut2]
```

✅ Correct Partition Condition

```
maxLeft1 <= minRight2
AND
maxLeft2 <= minRight1
```

If true:

If total length odd:

```
median = max(maxLeft1, maxLeft2)
```

If even:

```
median = (max(maxLeft1, maxLeft2) + min(minRight1, minRight
2)) / 2
```

❌ If Not Valid?

If:

```
maxLeft1 > minRight2
```

→ Move left (high = cut1 - 1)

Else:

→ Move right (low = cut1 + 1)

⏱️ Time Complexity

```
O(log(min(m,n)))
```

This is genius-level optimization.

💎 Optimal Java Code (Google Interview Ready)

```
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[]
nums2) {

        // Always binary search on smaller array
        if (nums1.length > nums2.length) {
            return findMedianSortedArrays(nums2, nums1);
        }

        int m = nums1.length;
        int n = nums2.length;

        int low = 0, high = m;

        while (low <= high) {

            int cut1 = (low + high) / 2;
            int cut2 = (m + n + 1) / 2 - cut1;

            int maxLeft1  = (cut1 == 0) ? Integer.MIN_VALUE
: nums1[cut1 - 1];
            int minRight1 = (cut1 == m) ? Integer.MAX_VALUE
: nums1[cut1];

            int maxLeft2  = (cut2 == 0) ? Integer.MIN_VALUE
: nums2[cut2 - 1];
            int minRight2 = (cut2 == n) ? Integer.MAX_VALUE
: nums2[cut2];

            // Correct partition found
            if (maxLeft1 <= minRight2 && maxLeft2 <= minRig
ht1) {

                if ((m + n) % 2 == 0) {
                    return (Math.max(maxLeft1, maxLeft2) +
                            Math.min(minRight1, minRight2))
/ 2.0;
                } else {
                    return Math.max(maxLeft1, maxLeft2);
                }
            }

            // Move left
            else if (maxLeft1 > minRight2) {
                high = cut1 - 1;
            }

            // Move right
            else {
                low = cut1 + 1;
            }
        }

        throw new IllegalArgumentException();
    }
}
```

🧠 Interview Explanation (How YOU Should Say It)

If interviewer asks:

"Why does this work?"

You say:

"We are partitioning both arrays such that left half contains exactly half of total elements. Since arrays are sorted, we just ensure that the maximum element on the left is less than or equal to minimum element on the right. This guarantees a correct median without merging."

🔥 That sounds very senior-level.

⚠️ Common Mistakes

- Not searching on smaller array
- Wrong cut2 formula
- Integer division error in even case
- Not handling edge partitions (0 or full)

🔥 Why This Is HARD?

Because:

- It's not obvious
- Requires partition thinking
- Requires understanding median mathematically
- Requires handling edge cases perfectly

🚀 Google Preparation Insight

If you master:

- LC 4
- LC 1552
- LC 1011
- LC 875
- LC 1482

You become dangerous in Binary Search pattern 😎

If you want next:

- 🔥 Full dry run visual diagram
- 🔥 Intuition building from scratch (how someone discovered this)
- 🔥 Edge cases deep dive
- 🔥 Similar Google follow-up twists

Bata bhai next kya phodna hai? 🔥

Bhai Anurag 🔥

Ab aa gaye ek aur **Google-level Binary Search on Answer monster** par 😎

LC 719 – Find K-th Smallest Pair Distance

Ye question directly test karta hai:

- Sorting
- Two Pointers

- Binary Search on Answer
- Counting technique (VERY IMPORTANT)

Ye Google, Meta, Amazon sab jagah favourite pattern hai 🚀

🧠 Problem Statement

Given:

```
nums[]  (unsorted)
k
```

Define:

Pair distance = |nums[i] - nums[j]| (i < j)

Return:

k-th smallest pair distance

🔎 Example

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

All pair distances:

(1,3) → 2

(1,1) → 0

(3,1) → 2

Sorted distances:

```
[0,2,2]
```

k=1 → Answer = 0

🚨 Key Insight

Constraints:

- n up to 10^4
- All pairs = n*(n-1)/2 ≈ 50 million

We cannot generate all pairs ❌

So brute force will TLE.

🧠 Step 1: Brute Force

Idea

1. Generate all pair distances
2. Store in list
3. Sort
4. Return k-1 index

Time Complexity

```
O(n^2 log n^2)
```

Too slow ❌

Brute Code (For Understanding Only)

```
class Solution {
    public int smallestDistancePair(int[] nums, int k) {

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

        for (int i = 0; i < nums.length; i++) {
            for (int j = i + 1; j < nums.length; j++) {
                list.add(Math.abs(nums[i] - nums[j]));
            }
        }

        Collections.sort(list);
        return list.get(k - 1);
    }
}
```

Rejected.

🚀 Step 2: Optimal Approach (Binary Search on Answer)

🔥 This is the real Google solution.

🧠 Core Idea

We don't search for pair.

We search for:

```
distance
```

Search space:

```
min distance = 0
max distance = max(nums) - min(nums)
```

Binary Search on distance.

For each candidate distance `mid`:

👉 Count how many pairs have distance ≤ mid

If count >= k

→ mid might be answer

→ search left

Else

→ search right

🧠 Why This Works?

Because:

If distance d has X valid pairs

Then any distance > d will have ≥ X pairs

Monotonic increasing property ✅

Perfect for Binary Search.

🧩 Most Important Part: Counting Pairs Efficiently

We need to count:

```
# of pairs with distance <= mid
```

Naively = O(n^2) ❌

Better approach:

👉 Sort the array first

Then use **Two Pointer Sliding Window**

🎯 Counting Logic

If array sorted:

For each right pointer:

Move left pointer such that:

```
nums[right] - nums[left] <= mid
```

Then:

```
all elements between left and right form valid pairs
count += (right - left)
```

This gives O(n) counting.

⏱️ Final Complexity

Sorting → O(n log n)

Binary Search → log(range)

Counting → O(n)

Total:

```
O(n log n + n log(range))
```

Google-ready 🚀

💎 Optimal Java Code (Interview Ready)

```
import java.util.*;

class Solution {

    // Function to count pairs with distance <= target
    private int countPairs(int[] nums, int target) {

        int count = 0;
        int left = 0;

        // Right pointer expands window
        for (int right = 0; right < nums.length; right++) {

            // Shrink window until valid
            while (nums[right] - nums[left] > target) {
                left++;
            }

            // All elements between left and right form val
id pairs
            count += (right - left);
        }

        return count;
    }

    public int smallestDistancePair(int[] nums, int k) {

        Arrays.sort(nums);  // Step 1: Sort array

        int low = 0;
        int high = nums[nums.length - 1] - nums[0];

        // Binary Search on answer
        while (low < high) {

            int mid = low + (high - low) / 2;

            int count = countPairs(nums, mid);

            if (count >= k) {
                // Enough pairs -> try smaller distance
                high = mid;
            } else {
                // Not enough -> increase distance
                low = mid + 1;
            }
        }

        return low;
    }
}
```

🧠 Interview Explanation (How YOU Should Say It)

If interviewer asks:

"Why binary search works here?"

You say:

"We binary search on distance instead of pairs because the number of valid pairs is monotonic with respect to distance. If distance d has at least k pairs, then any distance greater than d will also have at least k pairs. So we can binary search the smallest distance satisfying count >= k."

🔥 That sounds like a senior engineer.

🔥 Pattern Recognition

LC 719 belongs to:

```
Binary Search on Answer + Counting
```

Same family as:

- LC 1552 – Magnetic Force
- LC 1011 – Capacity to Ship
- LC 875 – Koko Eating Bananas
- LC 1482 – Minimum Days
- Aggressive Cows

⚠️ Common Mistakes

- Using <= instead of < in binary search loop
- Forgetting to sort
- Using O(n^2) counting
- Wrong count formula

🚀 Google-Level Twist

Interviewer may modify:

- Return actual pair
- K-th largest pair distance
- 3D distance

- Dynamic updates

So concept clarity > memorization.

🧠 Important Difference vs LC 1552

LC 1552	LC 719
Check feasibility	Count valid pairs
Greedy placement	Sliding window count
max min distance	k-th smallest distance

Different thinking, same pattern.

If you want next:

- 🔥 Full dry run step-by-step
- 🔥 Visual explanation of counting trick
- 🔥 Harder variation
- 🔥 10-question Google binary search mastery roadmap

Bata bhai next kya todna hai 😎🔥

Bhai Anurag 🔥

Ab aa gaye ek aur **Google-style HARD Binary Search + Two Pointer + Heap** problem par 😎

LC 786 – K-th Smallest Prime Fraction

Ye question test karta hai:

- Sorted array properties
- Fractions comparison
- Binary Search on Answer (double)
- OR Min Heap (priority queue)
- Mathematical thinking

Isko properly samajh liya → tumhara level serious FAANG tier 🚀

🧠 Problem Statement

Given:

```
arr[]  (sorted)
arr[0] = 1
rest are primes
k
```

Return:

```
k-th smallest fraction arr[i] / arr[j]
where i < j
```

🔎 Example

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

All fractions:

```
1/2 = 0.5
1/3 = 0.33
1/5 = 0.2
2/3 = 0.66
2/5 = 0.4
3/5 = 0.6
```

Sorted:

```
0.2 (1/5)
0.33 (1/3)
0.4 (2/5)
0.5 (1/2)
0.6 (3/5)
0.66 (2/3)
```

k=3 → answer = [2,5]

🚨 Observations

Total fractions:

```
n * (n - 1) / 2
```

Cannot generate all for large n ❌

Need better approach.

🧠 Approach 1: Brute Force

Generate all fractions → store → sort → return kth.

Time:

```
O(n^2 log n)
```

Rejected ❌

🚀 Approach 2: Min Heap (Better & Intuitive)

🧠 Key Insight

Since array sorted:

For each denominator `j`,

fractions:

```
arr[0]/arr[j]
arr[1]/arr[j]
arr[2]/arr[j]
...
```

are increasing.

So we treat it like merging sorted lists.

💡 Idea

Push smallest fractions into heap:

```
(arr[0]/arr[1]),
(arr[0]/arr[2]),
(arr[0]/arr[3]),
...
```

Then:

- Pop smallest
- Push next fraction in that column

Repeat k times.

⏱️ Complexity

```
O(k log n)
```

Good if k small.

💎 Heap Java Code

```
import java.util.*;

class Solution {
    public int[] kthSmallestPrimeFraction(int[] arr, int k)
{

        int n = arr.length;

        // Min heap storing {numerator index, denominator i
ndex}
        PriorityQueue<int[]> pq = new PriorityQueue<>(
            (a, b) -> arr[a[0]] * arr[b[1]] - arr[b[0]] * a
rr[a[1]]
        );

        // Initialize heap with fractions having numerator
index 0
        for (int j = 1; j < n; j++) {
            pq.offer(new int[]{0, j});
        }

        // Remove k-1 smallest fractions
        for (int i = 1; i < k; i++) {
            int[] curr = pq.poll();

            int numIdx = curr[0];
            int denIdx = curr[1];

            // Move numerator forward if possible
            if (numIdx + 1 < denIdx) {
                pq.offer(new int[]{numIdx + 1, denIdx});
            }
        }

        int[] result = pq.peek();

        return new int[]{arr[result[0]], arr[result[1]]};
    }
}
```

🚀 Approach 3: Binary Search on Answer (Google Favourite 🔥)

This is the most impressive solution.

🧠 Core Idea

Fractions range between:

```
0 to 1
```

Binary search on fraction value.

For each mid:

Count how many fractions ≤ mid.

If count >= k

→ go left

Else

→ go right

🎯 Key Challenge

How to count efficiently?

🧠 Counting Trick (Very Important)

Array sorted.

For each numerator i:

We find smallest denominator j such that:

```
arr[i] / arr[j] <= mid
```

Rearrange:

```
arr[i] <= mid * arr[j]
```

Use two pointer technique.

Also track:

👉 largest fraction ≤ mid

(because at the end we need exact fraction)

⏱️ Complexity

Binary search iterations ≈ 40–50 (double precision)

Each iteration → O(n)

Total:

```
O(n log precision)
```

Very powerful solution 🔥

💎 Optimal Binary Search Code

```
class Solution {
    public int[] kthSmallestPrimeFraction(int[] arr, int k)
{

        int n = arr.length;

        double low = 0.0;
        double high = 1.0;

        int p = 0, q = 1;  // store best fraction

        while (true) {

            double mid = (low + high) / 2;

            int count = 0;
            double maxFraction = 0.0;

            int j = 1;
            p = 0;
            q = 1;

            // Count fractions <= mid
            for (int i = 0; i < n - 1; i++) {

                while (j < n && arr[i] > mid * arr[j]) {
                    j++;
                }

                if (j == n) break;

                count += (n - j);

                // Track maximum fraction <= mid
                double fraction = (double) arr[i] / arr[j];

                if (fraction > maxFraction) {
                    maxFraction = fraction;
                    p = arr[i];
                    q = arr[j];
                }
            }

            if (count == k) {
                return new int[]{p, q};
            }

            if (count < k) {
                low = mid;
            } else {
                high = mid;
            }
        }
    }
}
```

🧠 Why This Is Hard?

Because:

- You binary search on floating value
- Counting needs two pointer logic
- Must track best fraction
- Precision handling

This is serious interview question.

🔥 Comparison with LC 719

LC 719	LC 786
Search integer distance	Search double fraction
Count pairs	Count fractions
Sliding window	Two pointer
Return value	Return actual fraction

Same pattern family 🔥

🚀 Google Interview Explanation

If interviewer asks:

"Why does binary search work?"

You say:

"The number of fractions less than or equal to a value x is monotonic with respect to x. So we can binary search on fraction value space between 0 and 1 and use a two-pointer technique to count valid fractions efficiently."

🔥 That sounds elite.

⚠️ Common Mistakes

- Using floating comparison carelessly
- Not handling precision
- Comparator overflow in heap
- Not tracking best fraction

🎯 When To Use Which Approach?

If:

- k small → heap simpler

- Want impressive solution → binary search
- Interviewer pushes optimization → binary search

🧠 Mastery Roadmap

LC 4

LC 1552

LC 719

LC 786

LC 1011

LC 875

LC 1482

Ye sab aagaye control me →

Binary Search on Answer = mastered 🔥

Agar chaaho next:

- 🔥 Full dry run visual explanation
- 🔥 Precision handling deep dive
- 🔥 Hard follow-up variation
- 🔥 Binary Search mastery sheet for Google London prep

Bata bhai next kya todna hai 😎🔥

Bhai Anurag 🔥

Ab aa gaye **Voting Algorithm level problem** par 😎

LC 229 – Majority Element II

Ye Google/Meta/Amazon me bahut pucha gaya hai.

Ye simple lagta hai but isme hidden logic hai.

🧠 Problem Statement

Given:

```
nums[]
```

Return all elements that appear **more than ⌊ n/3 ⌋ times**

🔎 Example

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

3 appears 2 times → answer = [3]
```

Example 2:

```
nums = [1,1,1,3,3,2,2,2]
n = 8
n/3 = 2

1 appears 3 times
2 appears 3 times

Answer = [1,2]
```

🚨 VERY IMPORTANT OBSERVATION

For n/3 threshold:

👉 At most **2 elements** can exist.

Why?

If there were 3 elements each appearing > n/3:

```
> n/3 + n/3 + n/3 = > n
```

Impossible.

🔥 This is the key insight.

🧠 Step 1: Brute Force

Count frequency using HashMap.

Return elements > n/3.

Time:

```
O(n)
```

Space:

```
O(n)
```

Works, but interviewer expects O(1) space.

Brute Code

```
import java.util.*;

class Solution {
    public List<Integer> majorityElement(int[] nums) {

        Map<Integer, Integer> map = new HashMap<>();
        List<Integer> result = new ArrayList<>();

        for (int num : nums) {
            map.put(num, map.getOrDefault(num, 0) + 1);
        }

        int n = nums.length;

        for (int key : map.keySet()) {
            if (map.get(key) > n / 3) {
                result.add(key);
            }
        }
        return result;
    }
}
```

🚀 Step 2: Optimal Approach

👉 Boyer-Moore Voting Algorithm (Extended Version)

Original LC 169 (Majority > n/2) me ek candidate hota hai.

Yaha:

```
threshold = n/3
max candidates = 2
```

So we track:

```
candidate1, count1
candidate2, count2
```

🧠 Algorithm Logic

For each number:

1. If equals candidate1 → count1++
2. Else if equals candidate2 → count2++
3. Else if count1 == 0 → candidate1 = num
4. Else if count2 == 0 → candidate2 = num
5. Else → count1-- and count2--

🎯 Why This Works?

Think like:

Each time we see a different number,

we cancel one vote from both candidates.

Only elements with strong frequency survive.

⚠️ Important:

After first pass, candidates are potential.

We must verify in second pass.

💎 Optimal Java Code (Interview Ready)

```java
import java.util.*;

class Solution {
    public List<Integer> majorityElement(int[] nums) {

        int candidate1 = 0, candidate2 = 0;
        int count1 = 0, count2 = 0;

        // First pass: Find potential candidates
        for (int num : nums) {

            if (num == candidate1) {
                count1++;
            }
            else if (num == candidate2) {
                count2++;
            }
            else if (count1 == 0) {
                candidate1 = num;
                count1 = 1;
            }
            else if (count2 == 0) {
                candidate2 = num;
                count2 = 1;
            }
            else {
                count1--;
                count2--;
            }
        }

        // Second pass: Verify counts
        count1 = 0;
        count2 = 0;

        for (int num : nums) {
            if (num == candidate1) count1++;
            else if (num == candidate2) count2++;
        }

        List<Integer> result = new ArrayList<>();
        int n = nums.length;

        if (count1 > n / 3) result.add(candidate1);
        if (count2 > n / 3) result.add(candidate2);

        return result;
    }
}
```

🧠 Dry Run Example

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

Process:

Eventually:

candidate1 = 1

candidate2 = 2

Verification confirms both > n/3.

⏱️ Complexity

Time:

```
O(n)
```

Space:

```
O(1)
```

Google loves this.

🔥 Why This Is Powerful?

Because:

- No sorting
- No hashmap
- Constant space
- Pure mathematical reasoning

This shows strong algorithm thinking.

🧠 Pattern Recognition

LC 169 → n/2 → 1 candidate

LC 229 → n/3 → 2 candidates

General case → n/k → (k-1) candidates

This is generalized Boyer-Moore.

🚀 Interview Explanation (What YOU Should Say)

If interviewer asks:

"Why maximum 2 elements?"

You say:

"If three elements appeared more than n/3 times, their total frequency would exceed n, which is impossible. So at most two such elements can exist."

🔥 Clean, mathematical, confident.

⚠️ Common Mistakes

- Forgetting second verification pass
- Not handling candidate initialization properly
- Using == incorrectly with objects (if Integer)

🎯 Google-Level Follow Up

Interviewer may ask:

- What about > n/4?
- Can you generalize?
- What if stream of numbers?
- Can you do in distributed system?

If you want, next I can:

- 🔥 Generalized n/k version
- 🔥 Compare LC 169 vs LC 229 deeply
- 🔥 Give you Google-style follow-up twist
- 🔥 Give you next hard array problem

Bata bhai next kya todna hai 😎🔥

Bhai Anurag 🔥

Ab aa gaye ek **pure permutation logic + greedy + pattern recognition** wale FAANG classic par 😎

LC 31 – Next Permutation

Ye problem har serious interview me aa sakta hai.

Google, Meta, Amazon — sab jagah.

Isme test hota hai:

- Lexicographical order
- Right-to-left thinking
- In-place array manipulation
- Greedy intuition

🧠 Problem Statement

Given an array `nums`:

Return the **next lexicographically greater permutation.**

If not possible (array descending order),

return the smallest permutation (ascending order).

Must be **in-place** and O(1) space.

🔎 Example 1

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

Example 2:

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

Example 3:

```
nums = [1,1,5]
Next = [1,5,1]
```

🧠 Intuition Building (Very Important)

Permutation order example:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

We want the **next immediate larger arrangement**.

🚀 Key Insight (Golden Observation)

Traverse from right to left.

Find the first index `i` such that:

```
nums[i] < nums[i + 1]
```

This is called the **breakpoint**.

Why?

Because right side is in descending order.

🧩 Why Right to Left?

Because:

To make next permutation,

we want:

- Smallest possible increase
- Change as right as possible

So we modify from end.

🎯 Algorithm Steps

Step 1: Find breakpoint

Find first index `i` from right:

```
nums[i] < nums[i+1]
```

If not found:

→ Entire array descending

→ Reverse whole array

Step 2: Find just larger element

From right side, find first `j` such that:

```
nums[j] > nums[i]
```

Swap nums[i] and nums[j].

Step 3: Reverse right portion

Reverse array from `i+1` to end.

Because it was descending,

after swap we want smallest order → ascending.

🧠 Why Reverse?

Right side originally descending.

To get smallest lexicographical order after swap,

we make it ascending.

🔥 Dry Run Example

```
nums = [1,2,7,4,3,1]
```

Step 1:

Find breakpoint:

```
2 < 7 → i = 1
```

Step 2:

From right, find element > 2 → 3

Swap:

```
[1,3,7,4,2,1]
```

Step 3:

Reverse right part:

```
[1,3,1,2,4,7]
```

Done ✅

💎 Optimal Java Code (Interview Ready)

```
class Solution {
    public void nextPermutation(int[] nums) {

        int n = nums.length;

        int i = n - 2;

        // Step 1: Find first decreasing element from right
        while (i >= 0 && nums[i] >= nums[i + 1]) {
            i--;
        }

        // If breakpoint found
        if (i >= 0) {

            int j = n - 1;

            // Step 2: Find element just larger than nums
[i]
            while (nums[j] <= nums[i]) {
                j--;
            }

            swap(nums, i, j);
        }

        // Step 3: Reverse right portion
        reverse(nums, i + 1, n - 1);
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }

    private void reverse(int[] nums, int left, int right) {
        while (left < right) {
            swap(nums, left++, right--);
        }
    }
}
```

⏱️ Complexity

Time:

```
O(n)
```

Space:

```
O(1)
```

Perfect.

🧠 Mathematical Explanation

The suffix after breakpoint is always non-increasing.

Why?

Because if there was any increasing pair,

we would have found breakpoint earlier.

So suffix is descending.

Reversing makes it smallest possible order.

🔥 Why This Is Important for Google?

Because:

- Tests permutation thinking
- Tests lexicographical reasoning
- Tests array manipulation
- Tests boundary handling

Also base for:

- K-th permutation
- Previous permutation
- Permutation rank
- Backtracking optimizations

⚠️ Common Mistakes

- Traversing left-to-right
- Not reversing suffix
- Picking wrong swap candidate
- Edge case: already largest permutation

🚀 Interview Explanation (What YOU Should Say)

If interviewer asks:

"Why reverse instead of sort?"

You say:

"Because the suffix is already in descending order. Reversing it makes it ascending, which gives the smallest lexicographical order for that suffix in O(n) time."

🔥 Clean. Efficient. Confident.

🧠 Pattern Category

Belongs to:

```
Greedy + Lexicographical Order
```

Similar problems:

- LC 556 – Next Greater Element III
- K-th Permutation Sequence
- Previous Permutation
- Permutation Rank

Anurag bhai 😎

Tum Binary Search already master kar rahe ho.

Next Permutation jaisi problems:

- Improve greedy thinking
- Improve right-to-left intuition
- Build strong permutation understanding

Agar next chaho:

- 🔥 Previous Permutation
- 🔥 K-th Permutation (Math + factorial)
- 🔥 Permutation rank explanation
- 🔥 Hard Google permutation follow-up

Bata bhai next kya todna hai 🚀🔥

Bhai Anurag 🔥

Ab aa gaye ek **true Google-level HARD** problem par 😎

LC 493 – Reverse Pairs

Ye normal inversion count nahi hai.

Ye **modified inversion + merge sort counting** problem hai.

Google, Meta, Uber — sab jagah favourite.

🧠 Problem Statement

Given:

```
nums[]
```

Count pairs (i, j) such that:

```
i < j
nums[i] > 2 * nums[j]
```

🔎 Example

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

Valid pairs:

(3,1)

(3,1)

Answer = 2

🚨 Important Difference from Inversion Count

Normal inversion:

```
nums[i] > nums[j]
```

Here:

```
nums[i] > 2 * nums[j]
```

So simple inversion logic won't directly work.

🧠 Step 1: Brute Force

Check all pairs:

```
for (i)
   for (j > i)
      if nums[i] > 2 * nums[j]
```

Time:

```
O(n^2)
```

TLE ❌

🚀 Step 2: Optimal Approach → Modified Merge Sort

🔥 This is the key.

Just like inversion count:

- Split array
- Count in left half
- Count in right half
- Count cross pairs
- Merge

🧠 Core Insight

When merging two sorted halves:

Left = sorted

Right = sorted

We need count:

```
nums[i] > 2 * nums[j]
```

Because both halves sorted:

For each element in left,

we can move pointer in right only forward.

Two pointer trick inside merge.

🎯 Algorithm

In merge step:

For each i in left:

Move j in right while:

```
nums[i] > 2 * nums[j]
```

Count += (j - (mid+1))

Important:

Do counting BEFORE merging.

⚠️ Important Edge Case

Use:

```
(long) nums[i] > 2L * nums[j]
```

To avoid overflow.

💎 Optimal Java Code (Google Ready)

```
class Solution {

    public int reversePairs(int[] nums) {
        return mergeSort(nums, 0, nums.length - 1);
    }

    private int mergeSort(int[] nums, int left, int right)
{

        if (left >= right) return 0;

        int mid = left + (right - left) / 2;

        int count = 0;

        count += mergeSort(nums, left, mid);
        count += mergeSort(nums, mid + 1, right);
        count += countPairs(nums, left, mid, right);

        merge(nums, left, mid, right);

        return count;
    }

    private int countPairs(int[] nums, int left, int mid, i
nt right) {

        int count = 0;
        int j = mid + 1;

        for (int i = left; i <= mid; i++) {

            while (j <= right && (long) nums[i] > 2L * nums
[j]) {
                j++;
            }
            count += (j - (mid + 1));
        }

        return count;
    }

    private void merge(int[] nums, int left, int mid, int r
ight) {

        int[] temp = new int[right - left + 1];
        int i = left;
        int j = mid + 1;
        int k = 0;

        while (i <= mid && j <= right) {
            if (nums[i] <= nums[j]) {
                temp[k++] = nums[i++];
            } else {
                temp[k++] = nums[j++];
            }
        }

        while (i <= mid) {
            temp[k++] = nums[i++];
        }

        while (j <= right) {
            temp[k++] = nums[j++];
        }

        System.arraycopy(temp, 0, nums, left, temp.length);
    }
}
```

⏱️ Complexity

Time:

```
O(n log n)
```

Space:

```
O(n)
```

🧠 Why This Works?

Because:

During merge step:

Left half sorted

Right half sorted

So once:

```
nums[i] > 2 * nums[j]
```

holds for some j,

it will hold for all elements before that j.

We move pointer only forward.

Total movement = O(n) per level.

🔥 Dry Run (Short)

```
[1,3,2,3,1]
```

Divide:

[1,3,2] and [3,1]

Eventually:

When merging [2,3] and [1]

We check:

3 > 2*1 → true

Count increments.

🚀 Why Google Loves This?

Because:

- Requires divide and conquer understanding
- Requires pointer movement reasoning
- Requires overflow awareness
- Requires modifying classical algorithm

Not trivial at all.

🧠 Pattern Recognition

This belongs to:

```
Modified Merge Sort Counting
```

Similar problems:

- Count Inversions
- Count Smaller After Self (LC 315)
- Range Sum Count
- Count of Smaller Numbers After Self
- Reverse Pairs

🔥 Common Mistakes

- Counting after merge ❌
- Resetting j inside loop ❌
- Integer overflow ❌
- Forgetting long casting ❌

🧠 Interview Explanation (What YOU Should Say)

If interviewer asks:

"Why can we use merge sort?"

You say:

"Because we need to count cross pairs efficiently. Once both halves are sorted, we can use a two-pointer technique to count how many elements in the right half satisfy nums[i] > 2 * nums[j] for each element in the left half, in linear time per merge."

🔥 Clean. Confident. Senior-level.

📈 Difficulty Level

LC Hard

True Google-level thinking.

If tum:

- LC 4
- LC 493
- LC 719
- LC 1552

comfortably solve kar sakte ho

→ You're entering strong candidate zone 😎

Agar next chaho:

- 🔥 LC 315 deep dive
- 🔥 Segment Tree version
- 🔥 BIT (Fenwick Tree) version
- 🔥 Complete Divide & Conquer mastery roadmap

Bata bhai next kya todna hai 🚀🔥

Bhai Anurag 🔥

Ye **LC 164 – Maximum Gap** Google-level pattern problem hai.

Isme trick sorting nahi hai... **Pigeonhole Principle + Bucket Sort intuition** lagta hai.

Agar tu Google London target kar raha hai, to ye question conceptual depth check karta hai — especially **why sorting is not required**.

🧠 Problem Statement

Given an integer array `nums`, return:

> Maximum difference between successive elements in its sorted form.

⚠️ Constraint:

You must solve it in **O(n)** time and use **O(n)** space.

Example:

```
Input: [3,6,9,1]
Sorted: [1,3,6,9]
Gaps: 2,3,3
Answer: 3
```

🚫 Brute Force Approach

Idea:

1. Sort the array
2. Traverse and compute max difference

Time Complexity:

- Sorting → O(n log n)
- Traversal → O(n)

❌ Not allowed (problem demands O(n))

✅ Brute Java Code

```
import java.util.*;

class Solution {
    public int maximumGap(int[] nums) {

        // If less than 2 elements, no gap possible
        if (nums == null || nums.length < 2)
            return 0;

        // Sort the array
        Arrays.sort(nums);

        int maxGap = 0;

        // Traverse and find maximum adjacent difference
        for (int i = 1; i < nums.length; i++) {
            maxGap = Math.max(maxGap, nums[i] - nums[i -
1]);
        }

        return maxGap;
    }
}
```

⚡ Optimal Approach (O(n)) – Bucket Strategy

🔥 Ye hi Google wala approach hai.

🧠 Core Insight

Instead of fully sorting:

👉 Maximum gap will NOT be inside a bucket

👉 Maximum gap will always be BETWEEN buckets

Why?

Because if we divide numbers into uniform ranges,

then biggest gap will occur where a bucket is empty.

🧮 Mathematical Insight

Let:

```
n = nums.length
min = minimum element
max = maximum element
```

Minimum possible maximum gap:

```
gap = ceil((max - min) / (n - 1))
```

This is based on **Pigeonhole Principle**.

We create:

```
n - 1 buckets
```

Each bucket stores:

- bucketMin
- bucketMax

🪣 Algorithm Steps

1. Find global min and max
2. Compute bucket size
3. Create buckets
4. Place each element in correct bucket
5. Traverse buckets and compute max gap between adjacent buckets

---

💻 Optimal Java Code (With Detailed Comments)

```
class Solution {
    public int maximumGap(int[] nums) {

        // Edge case: If less than 2 elements
        if (nums == null || nums.length < 2)
            return 0;

        int n = nums.length;

        // Step 1: Find global minimum and maximum
        int min = Integer.MAX_VALUE;
        int max = Integer.MIN_VALUE;

        for (int num : nums) {
            min = Math.min(min, num);
            max = Math.max(max, num);
        }

        // If all numbers are same
        if (min == max)
            return 0;

        // Step 2: Calculate bucket size
        int bucketSize = (int) Math.ceil((double)(max - mi
n) / (n - 1));

        // Step 3: Number of buckets
        int bucketCount = (max - min) / bucketSize + 1;

        // Bucket arrays
        int[] bucketMin = new int[bucketCount];
        int[] bucketMax = new int[bucketCount];
        boolean[] used = new boolean[bucketCount];

        // Initialize buckets
        for (int i = 0; i < bucketCount; i++) {
            bucketMin[i] = Integer.MAX_VALUE;
            bucketMax[i] = Integer.MIN_VALUE;
        }

        // Step 4: Place elements into buckets
        for (int num : nums) {

            int index = (num - min) / bucketSize;

            bucketMin[index] = Math.min(bucketMin[index], n
um);
            bucketMax[index] = Math.max(bucketMax[index], n
um);

            used[index] = true;
        }

        // Step 5: Compute maximum gap between buckets
        int maxGap = 0;
        int prevMax = min;

        for (int i = 0; i < bucketCount; i++) {

            // Skip empty buckets
            if (!used[i])
                continue;

            // Gap is difference between current bucket's m
in
            // and previous bucket's max
            maxGap = Math.max(maxGap, bucketMin[i] - prevMa
x);

            prevMax = bucketMax[i];
        }

        return maxGap;
    }
}
```

🧠 Why This Works

Because:

Within a bucket:

```
max difference < bucketSize
```

But between buckets:

```
difference >= bucketSize
```

So real answer always lies BETWEEN buckets.

📊 Complexity

Approach	Time	Space
Brute	O(n log n)	O(1)
Optimal	O(n)	O(n)

🎯 Google Interview Insight

Interviewer expects:

1. Start with sorting
2. Realize constraint O(n)
3. Talk about:
    - Pigeonhole principle
    - Uniform bucket distribution
    - Why gap not inside bucket
4. Code cleanly

🧠 When To Use This Pattern?

- Maximum gap problems

- Linear time sorting requirement
- When full sorting not required
- When only relative order matters

If tu chahe to next hum:

- Iska **dry run step by step**
- Ya iska **mathematical proof**
- Ya iska **related problem (Radix Sort based solution)**

Bata bhai next kya todna hai? 🔥

Bhai LC 18 – **4Sum** 🔥

Ye basically **3Sum ka upgraded version** hai.

Pattern yaad rakh:

```
Sort + Fix 2 + Two Pointer
```

🧠 Problem Statement

Given:

- `nums[]`
- `target`

Find all unique quadruplets:

```
a + b + c + d = target
```

No duplicate answers allowed.

❌ Brute Force

4 nested loops:

```
O(n^4)
```

TLE pakka ❌

✅ Optimal Approach – O(n³)

💡 Core Idea

1. Sort array
2. First element fix karo (i)
3. Second element fix karo (j)
4. Remaining ke liye Two Pointer

🔥 Java Code (Interview Safe Version)

```
import java.util.*;

class Solution {
    public List<List<Integer>> fourSum(int[] nums, int targ
et) {

        List<List<Integer>> result = new ArrayList<>();
        int n = nums.length;

        if (n < 4) return result;

        Arrays.sort(nums);

        for (int i = 0; i < n - 3; i++) {

            // Skip duplicate for first element
            if (i > 0 && nums[i] == nums[i - 1])
                continue;

            for (int j = i + 1; j < n - 2; j++) {

                // Skip duplicate for second element
                if (j > i + 1 && nums[j] == nums[j - 1])
                    continue;

                int left = j + 1;
                int right = n - 1;

                while (left < right) {

                    long sum = (long) nums[i] + nums[j] + n
ums[left] + nums[right];

                    if (sum == target) {

                        result.add(Arrays.asList(
                                nums[i],
                                nums[j],
                                nums[left],
                                nums[right]
                        ));

                        left++;
                        right--;

                        // Skip duplicates
                        while (left < right && nums[left] =
= nums[left - 1])
                            left++;

                        while (left < right && nums[right]
== nums[right + 1])
                            right--;

                    } else if (sum < target) {
                        left++;
                    } else {
                        right--;
                    }
                }
            }
        }

        return result;
    }
}
```

🧠 Why Use `long`?

Integer overflow avoid karne ke liye.

Example:

```
nums = [1000000000,1000000000,1000000000,1000000000]
```

⏱️ Complexity

- Sorting → O(n log n)
- Main loops → $O(n^3)$
- Total → $O(n^3)$

Space → O(1) (excluding result)

🚀 Optimization Trick (Important for Interviews)

Early pruning laga sakte ho:

```
Minimum possible sum > target → break
Maximum possible sum < target → continue
```

Ye performance improve karta hai.

🎯 Pattern Recognition

Ye problem teach karta hai:

- Sorting
- Duplicate handling
- Two pointer inside nested loops
- k-Sum template

🧩 General k-Sum Template

Recursive bana sakte ho:

```
kSum → reduce to (k-1)Sum
Base case → 2Sum
```

Ye Google level follow-up me puch sakte hain 😈

🔥 Interview Follow-ups

1. Can you generalize to k-sum?
2. How to optimize further?
3. What if array very large?

Agar tu Google London prep kar raha hai (jo tu kar raha hai 😎),

to 4Sum ke baad:

- 3Sum
- 3Sum Closest
- k-Sum template
- Meet in the Middle concept

karna hi padega.

Bol bhai next kya attack karein 🚀

Bhai mast question uthaya 🔥

LC 16 – **3Sum Closest** interview ka favourite hai.

Aaj isko **Brute** → **Better** → **Optimal** teenon approach me todte hain,

har step ka logic clear karte hue.

🧠 Problem Statement

Given:

```
nums[]
target
```

Return karo:

👉 3 numbers ka sum jo **target ke sabse closest** ho.

⚠️ Exact equal hona zaroori nahi.

Example

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

Possible sums:

```
-1 + 2 + 1 = 2   (diff = 1)
-1 + 2 + -4 = -3
-1 + 1 + -4 = -4
2 + 1 + -4 = -1
```

Closest = **2**

Answer = 2

🥵 1. Brute Force – O(n^3)

💡 Idea

Har possible 3 combination check karo.

Difference store karo:

```
abs(sum - target)
```

Jo minimum hoga, usko return karo.

🔥 Code (Brute)

```
class Solution {
    public int threeSumClosest(int[] nums, int target) {

        int n = nums.length;
        int closestSum = nums[0] + nums[1] + nums[2];

        for (int i = 0; i < n - 2; i++) {
            for (int j = i + 1; j < n - 1; j++) {
                for (int k = j + 1; k < n; k++) {

                    int sum = nums[i] + nums[j] + nums[k];

                    if (Math.abs(sum - target) < Math.abs(c
losestSum - target)) {
                        closestSum = sum;
                    }
                }
            }
        }

        return closestSum;
    }
}
```

⏱️ Time Complexity

$O(n^3)$

Large input → TLE ❌

😎 2. Better Approach – Sorting + 2 loops + Binary Search

💡 Idea

1. Sort array
2. i fix karo
3. j fix karo
4. Third element ko binary search se approx dhundo

Time:

```
O(n² log n)
```

🔥 Code (Better)

```
import java.util.*;

class Solution {
    public int threeSumClosest(int[] nums, int target) {

        Arrays.sort(nums);
        int n = nums.length;

        int closestSum = nums[0] + nums[1] + nums[2];

        for (int i = 0; i < n - 2; i++) {

            for (int j = i + 1; j < n - 1; j++) {

                int remaining = target - nums[i] - nums[j];

                int left = j + 1;
                int right = n - 1;

                // Binary search style
                while (left <= right) {

                    int mid = left + (right - left) / 2;
                    int sum = nums[i] + nums[j] + nums[mi
d];

                    if (Math.abs(sum - target) < Math.abs(c
losestSum - target)) {
                        closestSum = sum;
                    }

                    if (sum < target) {
                        left = mid + 1;
                    } else {
                        right = mid - 1;
                    }
                }
            }
        }

        return closestSum;
    }
}
```

🚀 3. Optimal Approach – Sorting + Two Pointer (O(n²))

Ye interview ka real solution hai 🔥

💡 Core Idea

1. Sort array
2. i fix karo
3. left = i+1
4. right = n-1

5. Two pointer move karo

🤔 Why Two Pointer Works?

Because sorted array hai.

Agar sum chhota:

```
left++
```

Agar sum bada:

```
right--
```

Hum target ke paas pahunchte rahenge.

🔥 Step-by-Step Example

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

Sorted already.

---

i = 0 (-4)

left = 1 (-1)

right = 3 (2)

sum = -4 + -1 + 2 = -3

diff = 4

sum < target → left++

---

left = 2 (1)

sum = -4 + 1 + 2 = -1

closer → update

left++

Stop.

---

i = 1 (-1)

left = 2 (1)

right = 3 (2)

sum = 2

diff = 1

Closest so far = 2

🔥 Final Optimal Code

```
import java.util.*;

class Solution {
    public int threeSumClosest(int[] nums, int target) {

        Arrays.sort(nums); // Step 1: sort
        int n = nums.length;

        int closestSum = nums[0] + nums[1] + nums[2];

        for (int i = 0; i < n - 2; i++) {

            int left = i + 1;
            int right = n - 1;

            while (left < right) {

                int sum = nums[i] + nums[left] + nums[righ
t];

                // Update closest
                if (Math.abs(sum - target) < Math.abs(close
stSum - target)) {
                    closestSum = sum;
                }

                // Move pointers
                if (sum < target) {
                    left++;  // need bigger sum
                }
                else if (sum > target) {
                    right--; // need smaller sum
                }
                else {
                    return sum; // exact match mil gaya
                }
            }
        }

        return closestSum;
    }
}
```

⏱️ Complexity

Sorting → O(n log n)

Main loop → O(n^2)

Total → **O(n^2)** ✅

Space → O(1)

🧠 Important Interview Points

1. Duplicate skip karna zaroori nahi (kyunki we return single value).
2. Exact match mil jaaye → immediately return.
3. Always maintain `closestSum`.

🔥 Pattern Connection

Ye problem connect hoti hai:

- 2Sum
- 3Sum

- 4Sum
- kSum
- Two Pointer Template

🏆 Interview Me Kaise Bolna

"After sorting the array, I fix one element and use two pointers to approach the target greedily, updating the closest sum whenever I find a smaller difference."

Bhai tu Google prep kar raha hai 😎

To ye pattern 100% master karna hai.

Next kya karen?

- 3Sum (LC 15)
- 4Sum II
- kSum generic template
- ya Two Pointer pattern deep dive?

Bol 🚀

Bhai LC 11 – **Container With Most Water** 🔥

Ye Two Pointer ka sabse classic aur important problem hai.

Isko interviewer 100 baar pooch sakta hai 😎

Aaj isko **Brute** → **Better thinking** → **Optimal** full depth me samjhte hain.

🧠 Problem Statement (Simple)

Tumhe ek array diya hai:

```
height[i]
```

Har index pe ek vertical line hai.

Tumhe do lines choose karni hain jisse:

```
Area = min(height[i], height[j]) * (j - i)
```

maximum ho.

📦 Visualization

Socho:

```
Index:    0  1  2  3  4
Height: [1, 8, 6, 2, 5]
```

Do lines choose karoge:

```
Water height = chhoti wall
Width = distance between them
Area = height * width
```

🥵 1. Brute Force – O(n^2)

💡 Idea

Har pair try karo:

```
for i
   for j
      area calculate karo
```

🔥 Code (Brute)

```
class Solution {
    public int maxArea(int[] height) {

        int n = height.length;
        int maxArea = 0;

        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {

                int width = j - i;
                int h = Math.min(height[i], height[j]);
                int area = width * h;

                maxArea = Math.max(maxArea, area);
            }
        }

        return maxArea;
    }
}
```

⏱️ Time Complexity

$O(n^2)$ ❌

Large input → TLE

🤯 Important Observation

Area formula:

```
area = min(height[i], height[j]) * (j - i)
```

Area depend karta hai:

- Width (distance)
- Chhoti height

🚀 2. Optimal Approach – Two Pointer (O(n))

💡 Core Idea

Start:

```
left = 0
right = n - 1
```

Maximum width se start karte hain.

Ab question:

👉 Kaunsa pointer move karein?

🧠 Most Important Logic

Maan lo:

```
height[left] < height[right]
```

To area depend karega height[left] pe.

Agar hum right ko move karenge:

```
Width kam hoga
Height min same ya kam hoga
Area kabhi improve nahi hoga ❌
```

Isliye:

👉 **Chhoti height wala pointer move karo**

🔥 Step-by-Step Example

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

Start:

```
left = 0 (1)
right = 8 (7)
```

Area:

```
min(1,7) * 8 = 8
```

Left chhota hai → left++

```
left = 1 (8)
right = 8 (7)
```

Area:

```
min(8,7) * 7 = 49
```

Max = 49

Right chhota hai → right--

Is tarah greedy move karte rehte hain.

🔥 Optimal Code (Interview Version)

```
class Solution {
    public int maxArea(int[] height) {

        int left = 0;
        int right = height.length - 1;

        int maxArea = 0;

        while (left < right) {

            int width = right - left;
            int h = Math.min(height[left], height[right]);
            int area = width * h;

            maxArea = Math.max(maxArea, area);

            // Move smaller height pointer
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }

        return maxArea;
    }
}
```

⏱️ Complexity

Time → O(n)

Space → O(1)

🧠 Why This Greedy Works?

Because:

- Area limited by smaller height
- Width har step me kam ho raha hai
- Sirf height increase karke hi compensate kar sakte ho
- Height increase tab possible hai jab smaller side move karo

🎯 Interview Me Kaise Explain Kare

Interviewer pooche:

“Why do you move the smaller height?”

Answer:

> Because area is limited by the smaller height. Moving the taller height cannot increase the area since width decreases and limiting height remains the same or smaller.

🔥 Pattern Recognition

Ye problem sikhata hai:

- Greedy reasoning
- Two pointer shrinking window
- Mathematical observation

⚡ Related Problems

- Trapping Rain Water
- 3Sum
- 3Sum Closest
- Max Consecutive Ones
- Sliding Window Maximum

Bhai ye Two Pointer ka foundation problem hai.

Isko agar samajh liya, to 20 aur problems asaan lagengi.

Next kya karein?

- Trapping Rain Water?
- 3Sum deep dive?
- Sliding Window pattern?
- Ya Two Pointer master sheet bana du?

Bol bhai 😎🚀

Bhai LC 845 – **Longest Mountain in Array**

Isko aaj bilkul interview level clarity se todte hain 🔥

Main tujhe:

- Brute
- Better
- Optimal (O(1) space)
- Intuition

- Edge cases

sab detail me samjhaunga.

🧠 Problem Samajh

Mountain ka matlab:

```
Strictly Increasing
        +
Strictly Decreasing
```

Aur length ≥ 3 honi chahiye.

Valid Mountain:

```
1 3 5 4 2
```

Invalid:

```
1 2 3        (no decreasing)
5 4 3        (no increasing)
1 2 2 1      (equal allowed nahi)
```

🥵 1. Brute Force – O(n²)

💡 Idea

Har index ko peak maan lo.

Fir:

- Left side me check karo kitna increasing hai
- Right side me check karo kitna decreasing hai

🔥 Code (Brute)

```
class Solution {
    public int longestMountain(int[] arr) {

        int n = arr.length;
        int maxLen = 0;

        for (int i = 1; i < n - 1; i++) {

            int left = i;
            int right = i;

            // Move left while increasing
            while (left > 0 && arr[left - 1] < arr[left]) {
                left--;
            }

            // Move right while decreasing
            while (right < n - 1 && arr[right] > arr[right
+ 1]) {
                right++;
            }

            if (left < i && right > i) {
                maxLen = Math.max(maxLen, right - left +
1);
            }
        }

        return maxLen;
    }
}
```

⏱️ Complexity

Worst case → O(n²) ❌

Har index pe expand kar rahe hain.

😎 2. Better Approach – Two Arrays (O(n) time, O(n) space)

💡 Idea

Banate hain:

```
inc[i] → left se increasing count
dec[i] → right se decreasing count
```

🔍 Step 1 – Fill inc[]

Agar:

```
arr[i] > arr[i-1]
```

to:

```
inc[i] = inc[i-1] + 1
```

🔍 Step 2 – Fill dec[]

Agar:

```
arr[i] > arr[i+1]
```

to:

```
dec[i] = dec[i+1] + 1
```

🔥 Code (Better)

```
class Solution {
    public int longestMountain(int[] arr) {
        int n = arr.length;
        if (n < 3) return 0;

        int[] inc = new int[n];
        int[] dec = new int[n];

        // increasing from left
        for (int i = 1; i < n; i++) {
            if (arr[i] > arr[i - 1]) {
                inc[i] = inc[i - 1] + 1;
            }
        }

        // decreasing from right
        for (int i = n - 2; i >= 0; i--) {
            if (arr[i] > arr[i + 1]) {
                dec[i] = dec[i + 1] + 1;
            }
        }

        int maxLen = 0;

        for (int i = 0; i < n; i++) {
            if (inc[i] > 0 && dec[i] > 0) {
                maxLen = Math.max(maxLen, inc[i] + dec[i] +
1);
            }
        }

        return maxLen;
    }
}
```

⏱️ Complexity

Time → O(n)

Space → O(n)

🚀 3. Optimal – Single Pass O(1) Space

Ye real interview solution hai 🔥

💡 Core Idea

Hum do counters maintain karte hain:

```
up
down
```

Logic

- Agar increasing → up++
- Agar decreasing → down++
- Agar equal → reset
- Agar decreasing ke baad fir increasing → reset

Valid mountain tab hai jab:

```
up > 0 AND down > 0
```

🔥 Step-by-Step Example

```
arr = [2,1,4,7,3,2,5]
```

Walk through:

2 → 1 → down++

1 → 4 → reset (kyunki decreasing ke baad increasing)

4 → 7 → up++

7 → 3 → down++

3 → 2 → down++

Mountain length = up + down + 1

🔥 Final Optimal Code

```
class Solution {
    public int longestMountain(int[] arr) {

        int n = arr.length;
        int up = 0;
        int down = 0;
        int maxLen = 0;

        for (int i = 1; i < n; i++) {

            // Reset condition
            if ((down > 0 && arr[i] > arr[i - 1]) ||
                 arr[i] == arr[i - 1]) {

                up = 0;
                down = 0;
            }

            if (arr[i] > arr[i - 1]) {
                up++;
            }

            else if (arr[i] < arr[i - 1]) {
                down++;
            }

            if (up > 0 && down > 0) {
                maxLen = Math.max(maxLen, up + down + 1);
            }
        }

        return maxLen;
    }
}
```

🧠 Why Reset Needed?

Case:

```
1 2 3 2 1 2 3
```

After finishing first mountain:

If we don’t reset,

Second mountain count galat ho jayega.

🎯 Edge Cases

1. All increasing → return 0
2. All decreasing → return 0
3. Equal elements → break mountain
4. n < 3 → return 0

🔥 Interview Explanation

“Maintain two counters for increasing and decreasing slopes. Reset when pattern breaks. Valid mountain requires both slopes.”

🧩 Pattern Learning

Ye problem sikhata hai:

- State tracking
- Strict comparisons
- Reset logic
- Peak detection

Bhai agar tu Google London prep mode me hai 😎

To iske baad kar:

- LC 941 – Valid Mountain

- LC 376 – Wiggle Subsequence
- Longest Bitonic Subarray
- Trapping Rain Water

Bol next kya todna hai 🚀

Bhai LC 881 – **Boats to Save People** 🚤🔥

Ye ek **pure greedy + two pointer** problem hai. Interviewers ko bahut pasand hai.

Aaj isko:

- Brute
- Better thinking
- Optimal (Greedy Two Pointer)
- Proof intuition
- Dry run

sab detail me samjhte hain.

🧠 Problem Statement

Given:

```
people[]  → weights of people
limit     → boat weight capacity
```

Rules:

- Har boat me max 2 log baith sakte hain
- Unka total weight ≤ limit hona chahiye

Goal:

👉 Minimum number of boats return karo.

📌 Example

```
people = [3,2,2,1]
limit = 3
```

Sorted:

```
[1,2,2,3]
```

Possible pairing:

(1,2)

(2)

(3)

Answer = 3 boats

🥵 1. Brute Force Idea

Try all pair combinations.

Har baar:

- 2 log try karo
- Unfit wale alag bhejo

But:

- Har pairing check → O(n^2)
- Removal + marking complexity high

Practically messy & slow ❌

🤯 Key Observation (Very Important)

Boat me max 2 log allowed.

To best strategy kya ho sakta hai?

👉 **Heaviest aadmi ko lightest ke saath pair karo.**

Why?

Because:

- Heaviest banda sabse restrictive hai
- Agar woh lightest ke saath bhi fit nahi hota

  → woh kisi ke saath fit nahi hoga

Ye greedy proof ka base hai.

🚀 Optimal Approach – Sort + Two Pointer

💡 Steps

1. Sort array
2. left = 0 (lightest)
3. right = n-1 (heaviest)
4. Har iteration:
   - Agar dono fit ho jaate hain:

     ```
     left++
     right--
     ```

   - Nahi to:

     ```
     sirf right--
     ```

5. Har step pe boat count++

🔥 Dry Run

Example:

```
people = [3,2,2,1]
limit = 3
```

Sorted:

```
[1,2,2,3]
```

left=0 (1)

right=3 (3)

1 + 3 = 4 > 3 ❌

→ 3 alone jayega

boats = 1

right--

---

left=0 (1)

right=2 (2)

1 + 2 = 3 ✅

→ pair

boats = 2

left++

right--

---

left=1 (2)

right=1 (2)

2 alone jayega

boats = 3

Done.

🔥 Optimal Code (Interview Safe)

```
import java.util.*;

class Solution {
    public int numRescueBoats(int[] people, int limit) {

        Arrays.sort(people); // Step 1

        int left = 0;
        int right = people.length - 1;
        int boats = 0;

        while (left <= right) {

            // If lightest + heaviest fit together
            if (people[left] + people[right] <= limit) {
                left++;
            }

            // Heaviest always goes
            right--;
            boats++;
        }

        return boats;
    }
}
```

⏱️ Complexity

Sorting → O(n log n)

Two pointer → O(n)

Total → O(n log n) ✅

Space → O(1)

🧠 Why Greedy Correct Hai?

Case 1:

Heaviest + Lightest ≤ limit

→ Best pairing possible

Case 2:

Heaviest + Lightest > limit

→ Heaviest kisi ke saath fit nahi hoga

→ Alone hi jayega

Isliye greedy optimal hai.

🎯 Interview Explanation

If interviewer asks:

“Why pairing heaviest with lightest works?”

Answer:

> Because if the heaviest person cannot pair with the lightest, they cannot pair with anyone. So sending them alone is optimal.

🧩 Pattern Recognition

Ye same pattern use hota hai:

- Container With Most Water
- 3Sum
- Two Sum Sorted
- Pair problems
- Greedy interval problems

⚡ Edge Cases

1. All same weights
2. Every person weight = limit
3. Only 1 person
4. Odd number of people

🏆 Real Insight

Ye problem sikhata hai:

- Greedy thinking
- Sorting first

- Two pointer shrinking window
- Proving greedy correctness

Bhai agar tu Google prep mode me hai 😎

To Boats, Container, 3Sum, 4Sum

ye sab same family ke problems hain.

Next kya karein?

- Trapping Rain Water
- 3Sum full depth
- Two Pointer master template
- Sliding Window master sheet

Bol bhai 🚀

Bhai LC 360 – **Sort Transformed Array** 🔥

Ye ek **hidden two pointer + math observation** problem hai.

Thoda tricky hai, but samajh gaya to mast lagega 😎

🧠 Problem Statement

Tumhe diya hai:

```
nums[] → sorted array
a, b, c
```

Har element pe function apply karna hai:

[

f(x) = ax^2 + bx + c

]

Aur result ko sorted order me return karna hai.

⚠️ Important:

- Input array already sorted hai.
- Output sorted chahiye.

📌 Example

```
nums = [-4,-2,2,4]
a = 1, b = 3, c = 5
```

Function:

[

f(x) = x^2 + 3x + 5

]

Compute manually:

```
f(-4) = 16 -12 +5 = 9
f(-2) = 4 -6 +5 = 3
f(2) = 4 +6 +5 = 15
f(4) = 16 +12 +5 = 33
```

Result:

```
[9,3,15,33]
```

Sorted:

```
[3,9,15,33]
```

🥵 1. Brute Force

1. Har element pe function apply karo
2. Result array me store karo
3. Sort karo

```
class Solution {
    public int[] sortTransformedArray(int[] nums, int a, int b, int c) {
        int n = nums.length;
        int[] result = new int[n];

        for (int i = 0; i < n; i++) {
            result[i] = a * nums[i] * nums[i] + b * nums[i]
+ c;
        }

        Arrays.sort(result);
        return result;
    }
}
```

⏱️ Time = O(n log n)

🤯 Important Mathematical Insight

Function:

[

f(x) = ax^2 + bx + c

]

Ye quadratic function hai.

Graph shape depend karta hai `a` pe:

Case 1️⃣ : a > 0

Parabola opens upward (U shape)

👉 Maximum values edges pe milenge.

Example:

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

|x| bada → x^2 bada → value bada

To largest values array ke leftmost aur rightmost pe milenge.

Case 2️⃣ : a < 0

Parabola opens downward (∩ shape)

👉 Smallest values edges pe milenge.

🚀 Optimal Approach – Two Pointer

Since input sorted hai:

- Extreme values largest ya smallest banenge (depending on a)

Use:

```
left = 0
right = n-1
```

Case a > 0

- Largest values edges pe
- Result array ko end se fill karo

Case a < 0

- Smallest values edges pe
- Result array ko start se fill karo

🔥 Step-by-Step (a > 0)

Example:

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

Compare:

```
f(-4) vs f(4)
```

Jo bada hoga → result[end] me daalo

🔥 Final Optimal Code

```
class Solution {
    public int[] sortTransformedArray(int[] nums, int a, in
t b, int c) {

        int n = nums.length;
        int[] result = new int[n];

        int left = 0;
        int right = n - 1;

        int index = (a >= 0) ? n - 1 : 0;

        while (left <= right) {

            int leftVal = transform(nums[left], a, b, c);
            int rightVal = transform(nums[right], a, b, c);

            if (a >= 0) {
                // fill from end
                if (leftVal > rightVal) {
                    result[index--] = leftVal;
                    left++;
                } else {
                    result[index--] = rightVal;
                    right--;
                }
            } else {
                // fill from start
                if (leftVal < rightVal) {
                    result[index++] = leftVal;
                    left++;
                } else {
                    result[index++] = rightVal;
```

```
                        right--;
                    }
                }
            }

            return result;
        }

        private int transform(int x, int a, int b, int c) {
            return a * x * x + b * x + c;
        }
    }
```

⏱️ Complexity

Time → O(n)

Space → O(n)

🧠 Why This Works?

Because:

- Input sorted hai.
- Quadratic function ka maximum/minimum always edges pe milta hai.
- Isliye hum two pointer use karke greedily build kar sakte hain.

🎯 Interview Explanation

If interviewer asks:

“Why two pointer works?”

Answer:

> Since the input array is sorted and quadratic transformation preserves a predictable shape, the extreme values after transformation will lie at the ends

> of the array. So we can compare transformed values from both ends and build the result greedily.

🔥 Pattern Recognition

Ye problem connect hoti hai:

- Sorted Squares
- Container With Most Water
- Two Pointer Greedy
- Monotonic transformation

Bhai ye problem thoda brain use karata hai 😎

Google level interview me aa sakta hai.

Next kya karein?

- Trapping Rain Water
- Sliding Window master
- k-Sum template
- Greedy master sheet

Bol bhai 🚀

Bhai LC 1498 – **Number of Subsequences That Satisfy the Given Sum Condition** 🔥

Ye problem dekhne me tough lagti hai, but once pattern samajh gaya → mast greedy + two pointer + math power trick hai 😎

Aaj isko:

- Brute
- Better thinking
- Optimal (Two Pointer + Powers)
- Dry run
- Mathematical proof intuition

sab detail me samjhte hain.

🧠 Problem Statement (Simple Words)

Given:

```
nums[]
target
```

Count karo number of subsequences (non-empty) such that:

```
min(subsequence) + max(subsequence) <= target
```

⚠️ Order maintain karna zaroori nahi (subsequence = choose any elements in order).

Answer large ho sakta hai → return % (10^9 + 7)

📌 Example

```
nums = [3,5,6,7]
target = 9
```

Valid subsequences:

```
[3]
[3,5]
[3,6]
[3,5,6]
```

Answer = 4

🥵 1. Brute Force – O(2^n)

Har subsequence generate karo (bitmasking).

Check:

```
min + max <= target
```

Impossible for n=10^5 ❌

🤯 Key Observation

Condition sirf depend karta hai:

```
minimum element
maximum element
```

Beech ke elements irrelevant hain for condition.

🚀 Step 1: Sort Array

Sorting ke baad:

```
nums sorted ascending
```

To:

```
left → minimum
right → maximum
```

🔥 Core Idea (Very Important)

Fix karo:

```
left pointer = smallest element
right pointer = largest element
```

If:

```
nums[left] + nums[right] <= target
```

To:

👉 Left fixed hai minimum

👉 Right is maximum allowed

Ab beech ke elements (left+1 ... right-1)

kuch bhi choose karo ya na karo.

🧠 Math Trick

Elements count between:

```
right - left
```

Total combinations:

[

2^{(right - left)}

]

Because:

Har element:

- include
- exclude

🚀 Two Pointer Strategy

1. Sort array
2. Precompute powers of 2
3. left = 0
4. right = n-1

Loop:

- Agar valid:
  - add `2^(right-left)` to answer
  - left++
- Else:

    - right--

🔥 Dry Run

Example:

```
nums = [3,5,6,7]
target = 9
```

Sorted:

```
[3,5,6,7]
```

---

left=0 (3), right=3 (7)

3+7 = 10 > 9 ❌

→ right--

---

left=0 (3), right=2 (6)

3+6 = 9 ✅

Add:

[

2^{(2-0)} = 2^2 = 4

]

Answer = 4

left++

---

left=1 (5), right=2 (6)

5+6=11 >9 ❌

→ right--

Stop.

Answer = 4

🔥 Optimal Code (Interview Ready)

```
import java.util.*;

class Solution {
    public int numSubseq(int[] nums, int target) {

        int mod = 1_000_000_007;
        int n = nums.length;

        Arrays.sort(nums);

        // Precompute powers of 2
        int[] pow = new int[n];
        pow[0] = 1;
        for (int i = 1; i < n; i++) {
            pow[i] = (pow[i - 1] * 2) % mod;
        }

        int left = 0;
        int right = n - 1;

        long count = 0;

        while (left <= right) {

            if (nums[left] + nums[right] <= target) {

                count = (count + pow[right - left]) % mod;
                left++;

            } else {
                right--;
            }
        }

        return (int) count;
    }
}
```

⏱️ Complexity

Sorting → O(n log n)

Two pointer → O(n)

Total → O(n log n)

Space → O(n) for powers

🧠 Why Greedy Works?

Because:

If:

```
nums[left] + nums[right] <= target
```

Then any subset between them will also satisfy condition, since:

- Minimum fixed = nums[left]
- Maximum ≤ nums[right]

So hum ek hi baar me pura block count kar sakte hain.

🎯 Interview Explanation

If interviewer asks:

"How did you get 2^(right-left)?"

Answer:

> For a fixed minimum and maximum, all elements between them can be chosen or not chosen independently, giving 2^(number of elements in between) combinations.

🧩 Pattern Recognition

Ye problem mix hai:

- Two Pointer
- Greedy
- Sorting
- Combinatorics
- Modular arithmetic

🔥 Very Important Insight

Ye problem basically:

"Count valid ranges using greedy window expansion."

Same thinking use hoti hai:

- Sliding window counting problems
- Two pointer counting
- Subarray sum counting

Bhai ye Google level problem hai 😎

Agar ye samajh liya to tu strong zone me hai.

Next kya karein?

- Trapping Rain Water
- Sliding Window master
- Binary Search on Answer sheet
- kSum generic template

Bol bhai 🚀

Bhai LC 948 – **Bag of Tokens** 🔥

Ye ek **Greedy + Two Pointer + Resource Management** type problem hai.

Interview me agar greedy reasoning clear nahi hui to confuse kar deta hai 😄

Aaj isko full depth me samjhte hain:

- Problem intuition
- Brute thinking
- Correct greedy observation
- Optimal solution
- Proof intuition
- Dry run

🧠 Problem Statement (Simple Words)

Given:

```
tokens[]  → har token ka power cost
power     → initial power
```

Rules:

1. Agar tumhare paas enough power hai:
    - Tum token face up khel sakte ho
    - power -= token[i]
    - score++
2. Agar tumhare paas kam se kam 1 score hai:
    - Tum token face down khel sakte ho
    - power += token[i]
    - score--

Goal:

👉 Maximum score return karo.

📌 Example

```
tokens = [100]
power = 50
```

50 < 100

Kuch nahi kar sakte

Answer = 0

Example 2:

```
tokens = [100,200,300,400]
power = 200
```

Answer = 2

Kaise? Dekhenge ab.

🥵 Brute Force Thinking

Har possible sequence try karo:

- Kab face up
- Kab face down

But order important hai.

Combinations exponential ho sakte hain ❌

So brute not practical.

🤯 Important Observation (KEY)

Goal kya hai?

👉 **Score maximize karna**

Score badhta hai sirf jab:

```
token face up
```

Face down sirf tab karte hain jab:

- future me aur score banana ho

💡 Greedy Strategy

1. Sabse sasta token pehle face up karo (kam power lagega)
2. Sabse mehenga token face down karo (zyada power milega)

Isliye:

👉 **Sort tokens**

🚀 Optimal Approach – Two Pointer

Steps:

1. Sort array
2. left = 0 (smallest token)
3. right = n-1 (largest token)
4. Loop:
    - Agar power >= tokens[left]:
        - power -= tokens[left]
        - score++
        - left++
        - maxScore update
    - Else if score > 0:
        - power += tokens[right]
        - score--
        - right--
    - Else:
        - break

🔥 Dry Run

```
tokens = [100,200,300,400]
power = 200
```

Sorted:

```
[100,200,300,400]
```

Step 1

power=200

100 face up

power=100

score=1

Step 2

power=100

200 afford nahi

But score=1

→ 400 face down

power=500

score=0

Step 3

power=500

200 face up

power=300

score=1

Step 4

300 face up

power=0

score=2

Done.

Max score = 2

🔥 Optimal Code (Interview Ready)

```java
import java.util.*;

class Solution {
    public int bagOfTokensScore(int[] tokens, int power) {

        Arrays.sort(tokens);

        int left = 0;
        int right = tokens.length - 1;

        int score = 0;
        int maxScore = 0;

        while (left <= right) {

            // Play face up
            if (power >= tokens[left]) {
                power -= tokens[left];
                score++;
                left++;

                maxScore = Math.max(maxScore, score);
            }

            // Play face down
            else if (score > 0) {
                power += tokens[right];
                score--;
                right--;
            }

            else {
                break;
            }
        }
        return maxScore;
    }
}
```

⏱️ Complexity

Sorting → O(n log n)

Two pointer → O(n)

Total → O(n log n)

Space → O(1)

🧠 Why Greedy Correct Hai?

Goal:

👉 Maximize score

To:

- Sabse sasta token pehle kharch karo → zyada tokens afford kar paoge
- Sabse mehenga token sacrifice karo → maximum power milega

Agar random choose karoge → optimal nahi hoga.

🎯 Interview Explanation

If interviewer asks:

"Why smallest face up and largest face down?"

Answer:

> To maximize the number of tokens played face up, we spend minimum power first. When we need more power, we regain the maximum possible power by sacrificing the largest remaining token.

🧩 Pattern Recognition

Ye problem sikhata hai:

- Greedy resource allocation
- Two pointer on sorted array
- Decision making under constraints
- Maximization strategy

🔥 Similar Problems

- Boats to Save People
- Minimum Cost to Hire K Workers
- Gas Station
- Jump Game

Bhai ye pure greedy brain test hai 😎

Google me aa sakta hai.

Next kya karein?

- Trapping Rain Water
- Jump Game series
- Greedy master sheet
- Binary Search on Answer

Bol bhai 🚀