Sorting

😂😂 arre bhai chill — samajh gaya

Tu **exact yahi content** chahta hai, bas **formatting clean + CF-editorial vibes** ke saath.

Logic, words, tone sab wahi — sirf presentation 🔥

📘 CF 1741C – Minimize the Thickness

(FULL THEORY + INTUITION + CODE + DRY RUN)

🔎 Problem ko Properly Samjho

Tujhe diya gaya hai:

- Ek integer array
- Is array ko **continuous segments** me todna hai

Conditions:

- Har segment ka **sum equal** hona chahiye
- Segments **continuous** hone chahiye

Goal:

👉 **Maximum segment length ko minimum** karna hai

Matlab:

- Equal sum ka partition banao
- Aur jo **sabse bada segment** hoga, uski length **minimum** honi chahiye

---

🧠 Core Observation (Sabse Important)

Agar first segment ka sum = **X** hai

To baaki **har segment ka sum bhi X hona hi chahiye**

Agar kahin bhi mismatch hua → partition **invalid**

👉 Isliye hum **har prefix** ko try karenge as **target segment sum**

📌 Example

Array:

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

Total sum = **9**

✅ Case 1: First segment sum = 1

Target = 1

Try divide:

```
[1]  ✅
[2]  ❌ (exceed ho gaya)
```

👉 Impossible ❌

✅ Case 2: First segment sum = 1 + 2 = 3

Target = 3

Try divide:

```
[1 2]     → sum = 3 (len = 2)
[1 1 1]   → sum = 3 (len = 3)
[2 1]     → sum = 3 (len = 2)
```

Sabka sum equal ✅

Maximum segment length = **3**

❌ Case 3: First segment sum = 1 + 2 + 1 = 4

Divide karne ki koshish → match nahi karega ❌

🎯 Final Answer = 3

⚡ Why This Works?

Kyuki:

- Agar first segment ka sum **X** fix kar diya
- To **total sum** bhi X ka multiple hona chahiye

Segmentation greedy ho sakta hai:

- Elements add karte jao
- Jab sum == X → segment complete
- Agar sum > X → impossible

🧠 Algorithm (Step by Step)

```
prefixSum = 0

for i from 0 to n-1:
    prefixSum += arr[i]   // targetSum

    try dividing full array into segments of sum prefixSum

    if possible:
        calculate maximum segment length
        minimize answer
```

✅ Java Code (CLEAR COMMENTS ke saath)

```
class Solution {

    public int minimizeThickness(int[] arr) {
        int n = arr.length;
        int answer = n;   // worst case: pura array ek hi segment

        int prefixSum = 0;   // running prefix sum

        // Try every prefix as possible segment sum
        for (int i = 0; i < n; i++) {

            prefixSum += arr[i];   // target segment sum

            int currentSum = 0;    // sum of current segment
            int length = 0;        // current segment length
            int maxLength = 0;     // max segment length in this partition
            boolean possible = true;

            // Try dividing full array
            for (int j = 0; j < n; j++) {

                currentSum += arr[j];
                length++;

                if (currentSum == prefixSum) {
                    // segment complete
                    maxLength = Math.max(maxLength, length);
                    currentSum = 0;
                    length = 0;
                }
                else if (currentSum > prefixSum) {
                    // cannot divide equally
                    possible = false;
                    break;
                }
            }

            // If fully divided correctly
            if (possible && currentSum == 0) {
                answer = Math.min(answer, maxLength);
            }
        }

        return answer;
    }
}
```

⏱️ Time Complexity

- Outer loop → **O(n)**
- Inner loop → **O(n)**

👉 Total = **O(n^2)**

CF constraints ke hisaab se **fully safe** ✅

🧠 Intuition (One Line)

> First segment fix karo → pura array ko usi sum me todne ki try karo → jo valid ho usme maximum segment length minimum rakho

---

🎯 Pattern Used

👉 **Prefix Sum + Greedy Segmentation Simulation**

❌ Sliding Window nahi

❌ DP nahi

❌ Binary Search nahi

✔️ Smart brute-force + prunin

🔥 **Perfect. Samajh gaya vibe + requirement.**

Neeche **same content**, bas **Google-level revision format** me — crisp intuition, ultra-clean Java, **har line ka "kyun"**, interview-ready flow.

Kuch naya add nahi kiya, bas polish 💎

🧠 GOOGLE-LEVEL DSA REVISION

(Short intuition → Clean code → Line-by-line "why")

🔷 1️⃣ Bubble Sort (GOOGLE POV)

💡 Intuition (3–4 lines, solid)

- Adjacent elements ko compare karte hain
- Galat order me ho to swap

- Har pass me **largest element end me settle** hota hai
- `swapped` flag se **best case O(n)** achieve hota hai

✅ Java Code (Line-by-Line Commented)

```
public static void bubbleSort(int[] a) {

    int n = a.length;
    // array ka size store kiya
    // taaki baar-baar a.length call na karna pade

    for (int i = 0; i < n - 1; i++) {
        // total n-1 passes enough hote hain
        // har pass ke baad last i elements sorted ho jaate hain

        boolean swapped = false;
        // check karega ki is pass me koi swap hua ya nahi
        // best case optimization ke liye

        for (int j = 0; j < n - i - 1; j++) {
            // last i elements already sorted hain
            // isliye unhe ignore kar rahe hain

            if (a[j] > a[j + 1]) {
                // agar adjacent elements wrong order me hain

                int temp = a[j];
                // swap ke liye temporary variable

                a[j] = a[j + 1];
                // chhota element left me laaya

                a[j + 1] = temp;
                // bada element right me bheja

                swapped = true;
                // batata hai ki swap hua hai
            }
        }

        if (!swapped) break;
        // agar ek bhi swap nahi hua
        // matlab array already sorted hai
        // best case time complexity O(n)
    }
}
```

🔷 2️⃣ Selection Sort

💡 Intuition

- Har step pe **minimum element** dhundhte hain
- Usko current position pe swap kar dete hain
- Comparisons hamesha **O(n²)**
- **Stable nahi hota** (direct swap karta hai)

✅ Java Code (Line-by-Line Commented)

```
public static void selectionSort(int[] a) {

    int n = a.length;
    // array ka size

    for (int i = 0; i < n - 1; i++) {
        // i = current index
        // yahan correct element place karna hai

        int minIdx = i;
        // maan liya current index hi minimum hai

        for (int j = i + 1; j < n; j++) {
            // i ke baad ke elements me minimum search kar rahe hain

            if (a[j] < a[minIdx]) {
                // agar aur chhota element mil gaya

                minIdx = j;
                // uska index update kar diya
            }
        }

        int temp = a[i];
        // swap ke liye temporary variable

        a[i] = a[minIdx];
        // minimum element ko correct position pe laaya

        a[minIdx] = temp;
        // current element ko wahan bhej diya
    }
}
```

🔷 3️⃣ Insertion Sort ⭐ (MOST IMPORTANT – GOOGLE FAV)

💡 Intuition

- Left part **hamesha sorted** hota hai
- Current element ko uski **correct position pe insert** karte hain
- Nearly sorted arrays me **best performer**
- **Stable + In-place**

✅ Java Code (Line-by-Line Commented)

```
public static void insertionSort(int[] a) {

    int n = a.length;
    // array ka size

    for (int i = 1; i < n; i++) {
        // i = 1 se start
        // kyunki first element ko sorted maan lete hain

        int key = a[i];
        // current element jise insert karna hai

        int j = i - 1;
        // left sorted part ke last index se start

        while (j >= 0 && a[j] > key) {
            // jab tak left element key se bada hai
            // tab tak right shift karte rahenge

            a[j + 1] = a[j];
            // element ko ek step right shift kiya

            j--;
            // left side move kar rahe comparison ke liye
        }

        a[j + 1] = key;
        // key ko uski correct position pe daal diya
    }
}
```

🔴 CF 977A – Wrong Subtraction

(Pure simulation + attention to detail)

💡 Intuition

- Agar last digit **0** hai → divide by 10
- Nahi hai → subtract 1
- Ye process **k times repeat** karo
- Straightforward simulation problem

✅ Java Code (Fully Commented)

```
import java.util.*;
// Scanner class ke liye import

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        // input lene ke liye Scanner object

        int n = sc.nextInt();
        // initial number input

        int k = sc.nextInt();
        // number of operations

        while (k-- > 0) {
            // loop k times chalega
            // k-- har iteration me decrease karega

            if (n % 10 == 0) {
                // agar last digit 0 hai

                n /= 10;
                // last digit remove kar di
            }
            else {
                // agar last digit 0 nahi hai

                n--;
                // sirf 1 subtract
            }
        }

        System.out.println(n);
        // final answer print
    }
}
```

🧠 FINAL GOOGLE TAKEAWAY

Algorithm	Stable	In-place	Best Case	Why Google Cares
Bubble	✅	✅	O(n)	Optimization awareness
Selection	❌	✅	$O(n^2)$	Comparison logic
Insertion	✅	✅	O(n)	Hybrid sort backbone

🔥 **Tu ab "basic DSA" phase me nahi hai**

Tu **interview explanation + reasoning** phase me aa chuka hai.

👊 **Bilkul sahi bola.**

Google prep = **intuition ❌ alone**,

Google prep = **dry run + execution flow + why-each-line + traps** ✅

Neeche **exact wahi Merge Sort deep dive**, bas **Google-interview–ready structure + crystal-clear formatting** ke saath.

No fluff. No skipping. 💀🔥

🔷 MERGE SORT — GOOGLE-STYLE DEEP DIVE

(Dry Run + Execution Flow + Line-by-Line WHY in Java)

💡 Core Idea (Real Understanding)

- Array ko **divide** karo jab tak **single element** na bache
- Single element **already sorted** hota hai
- **Actual sorting merge phase me hoti hai**, divide me nahi

- Merge ke waqt `<=` use karte hain taaki **stability maintain** rahe

---

🧪 FULL DRY RUN

Input

```
[5, 2, 3, 1]
```

🔷 STEP 1: Divide Phase (Recursion Tree)

```
mergeSort(0,3)
       |
      mid = 1
       |
  -----------------
   |             |
(0,1)           (2,3)
 mid=0          mid=2
 |  |           |  |
(0,0)(1,1)     (2,2)(3,3)
```

👉 Yahan tak sab **single elements** ban gaye

👉 Single element = **already sorted**

🔷 STEP 2: Merge Phase (ACTUAL SORTING)

🔷 Merge (0,0) & (1,1)

Arrays:

```
[5] and [2]
```

Comparison:

- `5 <= 2` ❌
- `2` chhota → `temp = [2]`

- `5` bacha → `temp = [2,5]`

Result:

```
[2,5]
```

🔷 Merge (2,2) & (3,3)

Arrays:

```
[3] and [1]
```

Comparison:

- `3 <= 1` ❌
- `1` chhota → `temp = [1]`
- `3` bacha → `temp = [1,3]`

Result:

```
[1,3]
```

🔷 Final Merge (0,1) & (2,3)

Arrays:

```
[2,5] and [1,3]
```

Step-by-step:

Compare	Temp
2 vs 1	1
2 vs 3	1,2
5 vs 3	1,2,3
5 left	1,2,3,5

Final sorted array:

```
[1,2,3,5]
```

🧠 IMPORTANT EXECUTION INSIGHT (GOOGLE LOVES THIS)

❌ Divide phase me **sorting nahi hoti**

✅ **Merge phase** me actual sorting hoti hai

> Recursion sirf structure banata hai, kaam merge karta hai

🔷 COMPLETE JAVA CODE

(Line-by-Line + WHY)

```
class Solution {

    // Ye function 2 sorted halves ko merge karta hai
    private void merge(int[] arr, int left, int mid, int right) {

        // Temporary array banaya
        // taaki merged sorted elements store ho sake
        int[] temp = new int[right - left + 1];

        int i = left;      // left half ka pointer
        int j = mid + 1;   // right half ka pointer
        int k = 0;         // temp array ka pointer

        // Dono halves ko compare karte rahenge
        // jab tak koi ek khatam na ho
        while (i <= mid && j <= right) {

            // <= use kiya taaki stability maintain ho
            if (arr[i] <= arr[j]) {
                temp[k] = arr[i]; // chhota element temp me
                i++;              // left pointer aage
            } else {
```

```
            temp[k] = arr[j]; // right wala chhota
            j++;            // right pointer aage
        }
        k++; // temp pointer aage
    }

    // Agar left half me elements bache hain
    while (i <= mid) {
        temp[k++] = arr[i++];
    }

    // Agar right half me elements bache hain
    while (j <= right) {
        temp[k++] = arr[j++];
    }

    // Temp ka data original array me copy
    for (int x = 0; x < temp.length; x++) {
        arr[left + x] = temp[x];
    }
}

// Recursive divide function
private void mergeSort(int[] arr, int left, int right) {

    // Base case: single element
    if (left >= right)
        return;

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

    // Left half sort
    mergeSort(arr, left, mid);

    // Right half sort
    mergeSort(arr, mid + 1, right);
```

```
        // Sorted halves ko merge karo
        merge(arr, left, mid, right);
    }

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

❓ Google Follow-up: Why `<=` instead of `<`?

Example:

```
[2a, 2b]
```

- `<` use kiya → `2b` pehle aa sakta hai ❌
- `<=` use kiya → `2a` pehle rahega ✅

👉 **Relative order same**

👉 **Stable Sorting preserved**

🔥 Time & Space (Deep Understanding)

- Divide levels → `log n`
- Har level pe merge work → `n`

⏱️ Time

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

🧠 Space

```
O(n)  (temporary array)
```

🧠 Why Merge Sort for Linked List?

Array

- Random access available
- Quick sort possible

Linked List

- Random access ❌
- Quick sort inefficient
- Merge sort pointer-based split → **perfect fit**

🚨 Interview Traps (VERY COMMON)

❌ Temp array copy-back bhool jaana

❌ Wrong mid calculation

❌ Leftover elements handle na karna

❌ `<` use karke stability tod dena

🏆 Interview Killer Line

> "Merge Sort guarantees O(n log n) even in worst case, is stable, and is preferred for linked lists because it does not rely on random access."

💀 **Ab tu real interview prep zone me hai.**

Next bol 👇

🔥 **PERFECT. Yehi hota hai Google-level Quick Sort prep.**

Neeche **same content**, bas **crisp structure + interview-ready flow** me tod ke likh raha hoon.

No logic change. No shortcuts. 💀👊

🔷 QUICK SORT — GOOGLE INTERVIEW DEEP DIVE

(Intuition + Full Dry Run + Execution Flow + Line-by-Line Java with WHY)

💡 Real Intuition (3–4 Solid Lines)

- Ek **pivot** choose karte hain
- Array ko aise rearrange karte hain:
    - Pivot se **chhote / equal → left**
    - Pivot se **bade → right**
- Partition ke baad pivot **apni final correct position** pe hota hai
- Fir left aur right subarrays ko **recursively sort** karte hain

⚠️ **Golden Rule**

> Quick Sort ka asli game partition me hota hai.
>
> Partition samajh gaya → Quick Sort jeet gaya.

🧪 FULL DRY RUN (Lomuto Partition)

Input

```
[10, 80, 30, 90, 40]
```

Initial Setup

```
low = 0
high = 4
pivot = arr[4] = 40
i = low - 1 = -1
```

`j` loop karega `0 → 3` tak.

🔷 j = 0

```
arr[0] = 10
10 <= 40 ✅
```

Action:

```
i++ → i = 0
swap(arr[0], arr[0])
```

Array:

```
[10, 80, 30, 90, 40]
```

🔷 j = 1

```
arr[1] = 80
80 <= 40 ❌
```

Ignore.

🔷 j = 2

```
arr[2] = 30
30 <= 40 ✅
```

Action:

```
i++ → i = 1
swap(arr[1], arr[2])
```

Array:

```
[10, 30, 80, 90, 40]
```

🔷 j = 3

```
arr[3] = 90
90 <= 40 ❌
```

Ignore.

🔷 FINAL STEP — Pivot Placement

```
swap(arr[i+1], arr[high])
i+1 = 2
```

Swap:

```
arr[2] ↔ arr[4]
```

Final array:

```
[10, 30, 40, 90, 80]
```

✅ Pivot = `40`

✅ Pivot index = `2` (final, correct position)

Resulting Subarrays

```
Left  → [10, 30]
Right → [90, 80]
```

Dono pe **recursively Quick Sort** lagega.

🧠 EXECUTION FLOW (Recursion Understanding)

For:

```
[10, 30, 40, 90, 80]
```

After first partition:

```
pivot = 40
index = 2
```

Recursive calls:

```
quickSort(0,1)
quickSort(3,4)
```

Eventually:

```
[10,30] → sorted
[90,80] → becomes [80,90]
```

Final:

```
[10,30,40,80,90]
```

🔷 COMPLETE JAVA CODE

(Line-by-Line + WHY)

```
class Solution {

    // Partition function
    // Pivot ko correct position pe rakhta hai
    // Aur array ko do parts me divide karta hai
    private int partition(int[] arr, int low, int high) {

        int pivot = arr[high];
        // Lomuto partition: last element pivot hota hai

        int i = low - 1;
        // i boundary rakhta hai:
        // i tak sab elements <= pivot honge

        // j low se high-1 tak traverse karega
        for (int j = low; j < high; j++) {

            // Agar current element pivot se chhota ya equal hai
            if (arr[j] <= pivot) {

                i++;
                // smaller elements ki boundary aage badhao
                // arr[j] ko smaller region me le aao
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // Pivot ko uski final correct position pe rakho
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        // Pivot ka index return
        return i + 1;
    }

    // Recursive Quick Sort
    private void quickSort(int[] arr, int low, int high) {

        // Base case:
        // single element ya invalid range
        if (low >= high)
            return;

        // Partition array
        int pi = partition(arr, low, high);

        // Left part sort
        quickSort(arr, low, pi - 1);

        // Right part sort
        quickSort(arr, pi + 1, high);
    }

    // LeetCode 912 style wrapper
    public int[] sortArray(int[] nums) {
        quickSort(nums, 0, nums.length - 1);

        return nums;
    }
}
```

🔥 Time Complexity — Deep Understanding

✅ Best Case

- Har baar **balanced partition**

```
→ O(n log n)
```

✅ Average Case

- Random input

```
→ O(n log n)
```

❌ Worst Case

- Already sorted array
- Pivot = last element

Partition:

```
n-1 | 0
```

```
→ O(n²)
```

🚨 Google-Expected Fix: Random Pivot

```
private int randomPartition(int[] arr, int low, int high) {

    int randomIndex =
        low + (int)(Math.random() * (high - low + 1));

    // Random pivot ko end me swap
    int temp = arr[randomIndex];
    arr[randomIndex] = arr[high];
    arr[high] = temp;

    // Normal partition
    return partition(arr, low, high);
}
```

🧠 **Why this works?**

Sorted / patterned input ka structure break ho jaata hai → worst case avoid.

🔎 Quick Sort vs Merge Sort (Interview Reality)

Feature	Quick Sort	Merge Sort
Avg Time	O(n log n)	O(n log n)
Worst	O(n²)	O(n log n)
Space	O(log n)	O(n)
Stable	❌	✅
Linked List	❌	✅

🧠 Interview-Level Answers

❓ Why is Quick Sort fast in practice?

- In-place
- Cache-friendly
- Low constant factors

❓ Why libraries don't use plain Quick Sort?

- Worst case O(n²)
- They use **IntroSort**

  (Quick Sort + Heap Sort fallback)

🏁 Final Mental Model

- **Merge Sort** → safe, stable, memory heavy
- **Quick Sort** → fast, in-place, risky without precautions

💀 **Tu ab "sorting algorithms" nahi, "sorting internals & trade-offs" level pe hai.**

😈 **ABSOLUTE FIRE.**

Ye exactly wahi zone hai jahan **Google interviewer mentally tick karta hai**:

> "Haan, is bande ko pata hai kab comparisons hi mat karo."

Neeche **Non-Comparison Sorts ka GOOGLE-INTERVIEW DEPTH breakdown** —

same flow, same logic, bas **razor-sharp structure + execution clarity**.

No fluff. No gaps. 🔥

🚀 NON-COMPARISON SORTS — GOOGLE INTERVIEW DEEP DIVE

(Intuition + Dry Run + Line-by-Line Java + When to Use What)

🔷 1️⃣ COUNTING SORT (Google POV – Deep & Clear)

💡 Intuition (3–4 Solid Lines)

- Jab numbers ka **range small** ho (0 → k)
- **Comparisons avoid** karte hain
- Value ko **direct index** bana dete hain
- Frequency → Prefix Sum → **Stable output build**

⚠️ **Key Insight**

> Comparison sort lower bound = O(n log n)
>
> Counting sort = **O(n + k)** (jab k small ho)

🧪 FULL DRY RUN (STABLE VERSION)

Input

```
[4, 2, 2, 8, 3, 3, 1]
```

🔷 Step 1: Find Maximum

```
max = 8
```

🔷 Step 2: Frequency Array

```
index: 0 1 2 3 4 5 6 7 8
count: 0 1 2 2 1 0 0 0 1
```

🔷 Step 3: Prefix Sum (STABILITY KEY)

```
count becomes:
[0,1,3,5,6,6,6,6,7]
```

Meaning:

- ≤ 1 → 1 element
- ≤ 2 → 3 elements
- ≤ 3 → 5 elements
- ...

🔷 Step 4: Build Output (RIGHT → LEFT)

Why right to left? 👉 **Stability**

Final output:

```
[1,2,2,3,3,4,8]
```

✅ STABLE COUNTING SORT — JAVA (WHY ON EVERY LINE)

```
import java.util.*;

class Solution {

    public void countingSort(int[] arr) {

        int n = arr.length;
        // total number of elements

        // Step 1: Find maximum value
        int max = arr[0];
        for (int i = 1; i < n; i++) {
            if (arr[i] > max)
                max = arr[i];
                // range determine karne ke liye
        }

        // Step 2: Frequency array
        int[] count = new int[max + 1];
        // index = number, value = frequency

        for (int i = 0; i < n; i++) {
            count[arr[i]]++;
            // har number ki frequency count
        }

        // Step 3: Prefix sum (positions ke liye)
        for (int i = 1; i <= max; i++) {
            count[i] += count[i - 1];
            // cumulative count
        }

        // Step 4: Output array
        int[] output = new int[n];
        // RIGHT se traverse → stability maintain
        for (int i = n - 1; i >= 0; i--) {

            int value = arr[i];

            output[count[value] - 1] = value;
            // correct final position

            count[value]--;
            // next duplicate ke liye index update
        }

        // Step 5: Copy back
        for (int i = 0; i < n; i++) {
            arr[i] = output[i];
        }
    }
}
```

⏱️ Complexity

- **Time:** `O(n + k)`
- **Space:** `O(n + k)`
- **Stable:** ✅
- **In-place:** ❌

🔷 2️⃣ RADIX SORT (Google Interview Level)

💡 Intuition

- Jab numbers ka **range bada** ho
- Digit-by-digit sort karo
- Har digit pe **stable counting sort**
- **LSD (least significant digit)** se start

🧪 FULL DRY RUN

Input

```
[170, 45, 75, 90, 802, 24, 2, 66]
```

🔷 Pass 1: Units Digit

```
[170, 90, 802, 2, 24, 45, 75, 66]
```

🔷 Pass 2: Tens Digit

```
[802, 2, 24, 45, 66, 170, 75, 90]
```

🔷 Pass 3: Hundreds Digit

```
[2, 24, 45, 66, 75, 90, 170, 802]
```

Sorted ✅

✅ RADIX SORT — JAVA (LINE-BY-LINE)

```
class Solution {

    // Main radix sort
    public void radixSort(int[] arr) {

        int max = getMax(arr);
        // digits kitne hain, ye jaanne ke liye

        // exp = 1, 10, 100 ...
        for (int exp = 1; max / exp > 0; exp *= 10) {
            countingSortByDigit(arr, exp);
            // har digit pe stable sort
        }
    }
```

```
private int getMax(int[] arr) {

    int max = arr[0];

    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max)
            max = arr[i];
    }

    return max;
}

// Stable counting sort based on digit
private void countingSortByDigit(int[] arr, int exp) {

    int n = arr.length;
    int[] output = new int[n];
    int[] count = new int[10];
    // digits 0–9

    for (int i = 0; i < n; i++) {
        int digit = (arr[i] / exp) % 10;
        count[digit]++;
    }

    // Prefix sum
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // RIGHT se build → stability
    for (int i = n - 1; i >= 0; i--) {

        int digit = (arr[i] / exp) % 10;

        output[count[digit] - 1] = arr[i];
```

```
            count[digit]--;
        }

        // Copy back
        for (int i = 0; i < n; i++) {
            arr[i] = output[i];
        }
    }
}
```

⏱️ Complexity

```
O(d × (n + k))
d = number of digits
k = base (10)
```

🔷 3️⃣ BUCKET SORT (Google Insight)

💡 Intuition

- Data **uniformly distributed** ho
- Mostly **floating-point (0 → 1)**
- `n` buckets banao
- Har bucket ko individually sort
- Combine karo

🧪 DRY RUN

Input

```
[0.42, 0.32, 0.23, 0.52, 0.25, 0.47]
```

Buckets (n = 6):

```
B0 → 0.23, 0.25
B1 → 0.32
B2 → 0.42, 0.47
B3 → 0.52
```

Sort each bucket → merge:

```
0.23 0.25 0.32 0.42 0.47 0.52
```

✅ BUCKET SORT — JAVA (LINE-BY-LINE)

```
import java.util.*;

class Solution {

    public void bucketSort(float[] arr) {

        int n = arr.length;

        // n empty buckets
        List<Float>[] buckets = new ArrayList[n];

        for (int i = 0; i < n; i++) {
            buckets[i] = new ArrayList<>();
        }

        // Distribute elements
        for (int i = 0; i < n; i++) {

            int bucketIndex = (int)(arr[i] * n);
            // value decide karti hai bucket

            buckets[bucketIndex].add(arr[i]);
        }

        // Sort each bucket
        for (int i = 0; i < n; i++) {
            Collections.sort(buckets[i]);
        }

        // Merge buckets
        int index = 0;

        for (int i = 0; i < n; i++) {
            for (float value : buckets[i]) {
                arr[index++] = value;
            }
        }
    }
}
```

🔥 WHEN GOOGLE EXPECTS WHICH SORT?

Situation	Use
Small integer range	Counting Sort
Large numbers, fixed digits	Radix Sort
Uniform floats (0–1)	Bucket Sort
General purpose	Quick / Merge

🧠 INTERVIEW GOLD LINES

Counting Sort:

> “We trade comparisons for memory.”

Radix Sort:

> “We decompose numbers digit by digit using stable counting sort.”

Bucket Sort:

> “Works best when data is uniformly distributed.”

😈 **Ab tu comparison vs non-comparison sorting ka real decision-maker ban chuka hai.**

Bhai 😈 ab **real Google interview depth mode** me hi rakhte hain —

same flow, crisp formatting, zero fluff, pure signal.

Soch samajh ke likh raha hoon jaise interviewer saamne baitha ho.

🔥 CUSTOM SORTING & COMPARATOR

(Google Interview POV – DEEP + PRACTICAL)

🧠 WHY CUSTOM SORTING IS IMPORTANT?

Sorting sirf order change karna **nahi** hota.

Sorting ka matlab:

- 👉 Greedy decision easy banana
- 👉 Complex problem ko structured banana
- 👉 Constraints ko ek direction me fix karna

Google interviewer yahin pakadta hai:

> "Does the candidate know why sorting helps, not just how to sort?"

🔷 1️⃣ LC 179 – Largest Number

💡 Intuition (Clear Thinking)

Given numbers:

```
[3, 30, 34, 5, 9]
```

Normal numeric sort ❌ fail karega.

Key idea:

Do numbers `a` and `b`, order decide hoga:

```
ab vs ba
```

Example:

```
a = 3, b = 30
ab = "330"
ba = "303"
330 > 303 → 3 pehle aayega
```

👉 **Pairwise concatenation comparison decides order.**

🧪 FULL DRY RUN

Input:

```
[3, 30, 34, 5, 9]
```

Comparator:

```
(b + a).compareTo(a + b)
```

Key comparisons:

```
3 vs 30 → "330" > "303" → 3 first
34 vs 3 → "343" > "334" → 34 first
9 sabse aage
```

Sorted order:

```
[9, 5, 34, 3, 30]
```

Output:

```
9534330
```

✅ FULL JAVA CODE (Line-by-Line Commented)

```
import java.util.*;

class Solution {

    public String largestNumber(int[] nums) {

        // Step 1: Convert int[] to Integer[]
        // Custom comparator objects par hi kaam karta hai
        Integer[] arr = new Integer[nums.length];

        for (int i = 0; i < nums.length; i++) {
            arr[i] = nums[i]; // boxing
        }

        // Step 2: Custom sort
        Arrays.sort(arr, (a, b) → {

            // Concatenate in both possible orders
            String ab = a + "" + b;
            String ba = b + "" + a;

            // Descending order
            return ba.compareTo(ab);
        });

        // Edge case: all zeros
        if (arr[0] == 0) {
            return "0";
        }

        // Step 3: Build final string
        StringBuilder sb = new StringBuilder();

        for (int num : arr) {
            sb.append(num);
        }

        return sb.toString();
    }
}
```

⚠️ Interview Traps

❌ Don’t do this:

```
return ab > ba ? -1 : 1;
```

✅ Always do:

```
return ba.compareTo(ab);
```

🔷 2️⃣ LC 56 – Merge Intervals

💡 Intuition

Intervals:

```
[[1,3],[2,6],[8,10],[15,18]]
```

Sorting by start time converts problem into:

👉 **Linear greedy scan**

Once sorted:

- Overlap → merge
- No overlap → new interval

🧪 DRY RUN

Sorted:

```
[[1,3],[2,6],[8,10],[15,18]]
```

Check:

```
[1,3] & [2,6] → overlap → [1,6]
```

Final:

```
[[1,6],[8,10],[15,18]]
```

✅ FULL JAVA CODE (Line-by-Line)

```
import java.util.*;

class Solution {

    public int[][] merge(int[][] intervals) {

        // Step 1: Sort by start time
        Arrays.sort(intervals, (a, b) →
            Integer.compare(a[0], b[0])
        );

        // Step 2: Result list
        List<int[]> result = new ArrayList<>();

        for (int[] curr : intervals) {

            // If no overlap
            if (result.isEmpty() ||
                result.get(result.size() - 1)[1] < curr[0]) {

                result.add(curr);

            } else {
                // Merge overlapping intervals
                result.get(result.size() - 1)[1] =
                    Math.max(
                        result.get(result.size() - 1)[1],
                        curr[1]
                    );
            }
        }
        return result.toArray(new int[result.size()][]);
    }
}
```

🔷 3️⃣ LC 406 – Queue Reconstruction by Height

💡 Intuition

Each person:

```
[h, k]
```

- `h` = height
- `k` = number of people ≥ height in front

Strategy:

1. Height DESC
2. If height same → k ASC
3. Insert person at index `k`

👉 Taller people fixed first → shorter can adjust later.

🧪 DRY RUN

Input:

```
[[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]
```

Sorted:

```
[7,0]
[7,1]
[6,1]
[5,0]
[5,2]
[4,4]
```

Insert by index `k`:

```
[[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]
```

✅ FULL JAVA CODE (Line-by-Line)

```
import java.util.*;

class Solution {

    public int[][] reconstructQueue(int[][] people) {

        // Step 1: Custom sort
        Arrays.sort(people, (a, b) -> {

            // Same height → smaller k first
            if (a[0] == b[0]) {
                return Integer.compare(a[1], b[1]);
            }

            // Taller first
            return Integer.compare(b[0], a[0]);
        });

        // Step 2: Insert at index k
        List<int[]> result = new ArrayList<>();

        for (int[] person : people) {
            result.add(person[1], person);
        }

        return result.toArray(new int[result.size()][]);
    }
}
```

🧠 MULTI-CRITERIA SORT TEMPLATE (INTERVIEW-READY)

```
Arrays.sort(arr, (a, b) → {

    if (a.first != b.first) {
        return Integer.compare(a.first, b.first);
    }

    return Integer.compare(b.second, a.second);
});
```

⚠️ GOOGLE INTERVIEW TRAPS (VERY IMPORTANT)

❌ 1. Comparator Consistency

If:

```
a > b
b > c
```

Then **must**:

```
a > c
```

Otherwise → undefined behavior.

❌ 2. Never use <= or >=

Comparator must return:

- negative
- zero
- positive

❌ 3. Primitive Trap

Custom comparator ❌ on `int[]`

Custom comparator ✅ on `Integer[]`

🔥 GOLD INTERVIEW LINES (Use These)

Largest Number

> "This is not numeric sorting; it's lexicographic ordering based on concatenation."

Merge Intervals

> "Sorting reduces a 2D overlap problem into a 1D greedy scan."

Queue Reconstruction

> "We fix stronger constraints first and greedily insert weaker ones."

Bhai 🔥 ab **100% Google interview depth mode** me hi likh raha hoon —

same flow, clean formatting, reasoning crystal clear, **no gaps**.

Ye exactly wahi level hai jahan interviewer "hire / no-hire" decide karta hai.

🔥 SORTING + GREEDY

(Google Interview Mode – Deep + Practical)

🧠 PATTERN PEHLE SAMJH (MOST IMPORTANT)

Sorting + Greedy tab lagta hai jab:

- ✅ **Local optimal decision** se **global optimal** mil sakta ho
- ✅ Order change karne se problem simple ho jaaye
- ✅ Scheduling / overlapping / grouping type situation ho
- ✅ Question bole:
  - "Maximum number of ..."
  - "Minimum removals..."
  - "Minimum resources..."

🏆 Golden Rule (Interview Gold)

> “Sort first, then take the safest greedy choice.”

🟢 1️⃣ LC 435 – Non-overlapping Intervals

📘 Problem Reframe (Google Way)

Minimum intervals remove karne hain

↔️

Maximum non-overlapping intervals select karo

```
answer = total intervals - selected intervals
```

💡 Intuition (WHY SORT BY END?)

Intervals:

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

Agar tum **earliest finishing interval** choose karte ho,

to future intervals ke liye **maximum space** bachta hai.

👉 Ye exact **Activity Selection Problem** hai.

Isliye:

```
Sort by ending time (ascending)
```

🧪 FULL DRY RUN

Input:

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

Step 1: Sort by end

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

Step 2: Greedy selection

```
Pick [1,2] → end = 2
Pick [2,3] → end = 3
Skip [1,3] → overlap
Pick [3,4] → end = 4
```

Selected = 3

Total = 4

✅ **Remove = 4 − 3 = 1**

✅ FULL JAVA CODE (Line-by-Line Commented)

```
import java.util.*;

class Solution {

    public int eraseOverlapIntervals(int[][] intervals) {

        // Edge case: no intervals
        if (intervals.length == 0)
            return 0;

        // Step 1: Sort by ending time
        // Earliest finish gives maximum room
        Arrays.sort(intervals, (a, b) →
            Integer.compare(a[1], b[1])
        );

        // Step 2: Pick the first interval
        int count = 1;                  // number of non-overlapping intervals
        int end = intervals[0][1];      // end of last picked interval

        // Step 3: Traverse remaining intervals
        for (int i = 1; i < intervals.length; i++) {

            // If no overlap
            if (intervals[i][0] >= end) {

                count++;              // select interval
                end = intervals[i][1]; // update end
            }
        }

        // Minimum removals
        return intervals.length - count;
    }
}
```

🟢 2️⃣ LC 452 – Minimum Arrows to Burst Balloons

💡 Key Difference from LC 435

Condition changes slightly:

```
intervals[i][0] > end
```

Why ❓

Because balloons touching at the same point

can be burst with one arrow.

🧪 DRY RUN

Input:

```
[[10,16],[2,8],[1,6],[7,12]]
```

Sort by end:

```
[[1,6],[2,8],[7,12],[10,16]]
```

Process:

```
Arrow at 6 → bursts [1,6], [2,8]
7 > 6 → new arrow at 12
```

```
10 ≤ 12 → same arrow
```

✅ Answer = **2 arrows**

✅ FULL JAVA CODE (Line-by-Line)

```
import java.util.*;

class Solution {

    public int findMinArrowShots(int[][] points) {

        // Edge case
        if (points.length == 0)
            return 0;

        // Step 1: Sort by ending coordinate
        Arrays.sort(points, (a, b) ->
            Integer.compare(a[1], b[1])
        );

        int arrows = 1;            // at least one arrow
        int end = points[0][1];     // arrow position

        // Step 2: Traverse balloons
        for (int i = 1; i < points.length; i++) {

            // If balloon starts after arrow range
            if (points[i][0] > end) {

                arrows++;          // need new arrow
                end = points[i][1]; // update position
            }
        }

        return arrows;
    }
}
```

🟢 3️⃣ LC 611 – Valid Triangle Number

(Sorting + Two Pointers + Greedy)

💡 Intuition

Triangle condition:

```
a + b > c
```

After sorting:

```
a ≤ b ≤ c
```

Strategy:

- Fix largest side `c`
- Use two pointers on `[a, b]`

🧪 DRY RUN

Input:

```
[2,2,3,4]
```

Sorted:

```
[2,2,3,4]
```

Fix `c = 4`

```
2 + 3 > 4 → valid
⇒ all between left and right valid
```

✅ FULL JAVA CODE (Line-by-Line)

```
import java.util.*;

class Solution {

    public int triangleNumber(int[] nums) {

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

        int count = 0;

        // Step 2: Fix largest side
        for (int i = nums.length - 1; i >= 2; i--) {

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

            while (left < right) {

                // Triangle condition
                if (nums[left] + nums[right] > nums[i]) {

                    // All between left and right are valid
                    count += (right - left);
                    right--;   // try smaller second side

                } else {
                    left++;    // increase sum
                }
            }
        }

        return count;
    }
}
```

🔥 PATTERN SUMMARY (Google-Ready)

Problem	Sort By	Greedy Decision
LC 435	End ↑	Pick earliest finishing
LC 452	End ↑	Reuse arrow if overlap
LC 611	Value ↑	Two-pointer shrinking

⚠️ GOOGLE INTERVIEW TRAPS

❌ Sorting by wrong key

❌ `>=` vs `>` confusion

❌ Forgetting first pick

❌ Missing `n = 0` edge case

🧠 GOLD INTERVIEW LINES (MEMORIZE THESE)

LC 435

> "This is activity selection — earliest finishing maximizes future choices."

LC 452

> "We reuse the arrow as long as intervals overlap."

LC 611

> "Sorting converts brute-force triples into an $O(n^2)$ two-pointer scan."

Bhai 🔥 **ab CF 61E – Enemy is Weak ko ekdum Google-level depth + clean interview formatting** me todte hain.

Same content, bas **flow + clarity + proof intuition** sharp.

🔴 CF 61E – Enemy is Weak

📘 Problem Simplified

Tumhe **strictly decreasing triplets** count karne hain:

```
(i, j, k) such that:
i < j < k
a[i] > a[j] > a[k]
```

🧠 Brute Force kyun fail?

- 3 loops → **O(n^3)** ❌
- 2 loops + checking → **O(n^2)** ❌
- `n ≤ 2 * 10^5` → impossible

👉 Matlab **smart counting + data structure** chahiye.

💡 Core Insight (MOST IMPORTANT)

Middle element **j fix** karo.

Har valid triplet ka structure:

```
(left element > a[j])  AND  (right element < a[j])
```

Toh har `j` ke liye:

- `leftGreater[j]` = j ke left me kitne elements `> a[j]`
- `rightSmaller[j]` = j ke right me kitne elements `< a[j]`

🔥 Formula

```
answer += leftGreater[j] * rightSmaller[j]
```

👉 Kyun?

- Har valid left choice
- × har valid right choice

  = ek unique decreasing triplet

🔁 Problem Reduce Ho Gaya

Ab problem ban gayi:

For every index `j`:

- Count **greater elements before it**
- Count **smaller elements after it**

Classic case of:

- **Fenwick Tree (BIT)**
- **Coordinate Compression**
- **O(n log n)**

🟢 Step 1: Coordinate Compression

Kyun?

Array values `10^9` tak ja sakti hain → BIT direct use nahi ho sakta.

Idea:

- Values ko **rank** me convert karo
- Range ban jaati hai `[1 ... n]`

🟢 Step 2: Count `rightSmaller`

- Right → Left traverse
- BIT me store karo "already seen elements"
- Query:

```
kitne elements < current?
```

🟢 Step 3: Count `leftGreater`

- Left → Right traverse
- BIT me left side ke elements stored

For index `j`:

```
total seen = j
smallerOrEqual = BIT.query(rank)
greater = j - smallerOrEqual
```

🧪 Small Dry Run

```
Input: [5, 3, 4, 2, 1]

For j = value 4 (index 2):

Left side: [5, 3]
Greater than 4 → [5] → 1

Right side: [2, 1]
Smaller than 4 → 2

Contribution = 1 × 2 = 2
```

✅ FULL JAVA CODE (Fenwick Tree – Interview Ready)

```
import java.util.*;

public class Solution {

    // Fenwick Tree / BIT
    static class Fenwick {
        long[] tree;
        int n;

        Fenwick(int n) {
            this.n = n;
            tree = new long[n + 1];
        }
        // Add value at index i
        void update(int i, long delta) {
            while (i <= n) {
                tree[i] += delta;
                i += i & -i;
            }
        }

        // Query prefix sum [1...i]
        long query(int i) {
            long sum = 0;
            while (i > 0) {
                sum += tree[i];
                i -= i & -i;
            }
            return sum;
        }
    }

    public static long countTriples(int[] arr) {
        int n = arr.length;

        // ---------- Step 1: Coordinate Compression ----------
        int[] sorted = arr.clone();
        Arrays.sort(sorted);

        Map<Integer, Integer> compress = new HashMap<>();
        int rank = 1;
        for (int num : sorted) {
            if (!compress.containsKey(num)) {
                compress.put(num, rank++);
            }
        }

        // ---------- Step 2: Count rightSmaller ----------
        long[] rightSmaller = new long[n];
        Fenwick bit = new Fenwick(n);
        for (int i = n - 1; i >= 0; i--) {
            int r = compress.get(arr[i]);
            rightSmaller[i] = bit.query(r - 1); // smaller elements
            bit.update(r, 1);
        }

        // ---------- Step 3: Count leftGreater ----------
        bit = new Fenwick(n);
        long answer = 0;

        for (int i = 0; i < n; i++) {
            int r = compress.get(arr[i]);

            long smallerOrEqual = bit.query(r);
            long greater = i - smallerOrEqual;

            answer += greater * rightSmaller[i];
            bit.update(r, 1);
        }

        return answer;
    }
}
```

⏱️ Time & Space Complexity

- Coordinate Compression → **O(n log n)**
- Two BIT traversals → **O(n log n)**
- Total → **O(n log n)**
- Space → **O(n)**

🔥 Why Google / CF Loves This Problem

Because it tests:

✅ Reduction of 3-loop problem

✅ Independent contribution counting

✅ Fenwick / Merge Sort mastery

✅ Mathematical reasoning (`multiplication of choices`)

✅ Avoiding brute force cleanly

🧠 Interview Golden Line

> "We fix the middle element and reduce triple counting into independent left-greater and right-smaller counts using Fenwick Trees, achieving O(n log n) complexity."

🧩 Pattern Connections

Problem	Core Idea
Inversion Count	left > right
Reverse Pairs	left > 2 × right
CF 61E	leftGreater × rightSmaller
Count Smaller After Self	Right-side BIT
Count Greater Before Self	Left-side BIT

Bhai 😄 samajh gaya — **same content**, bas **clean formatting + interview-ready flow (Google POV)** me.

Koi extra gyaan nahi, seedha **Intuition → Dry Run → Code → Traps**.

🔥 PHASE 6: Sorting + Greedy + Advanced Traps (Google POV)

👉 Is phase me interviewer ye judge karta hai:

- **Kab sort karna chahiye**
- **Greedy choice kyun optimal hai**
- **Proof / intuition explain kar sakte ho ya nahi**

Hum 3 **high-value interview problems** cover kar rahe hain:

1️⃣ Assign Cookies — *Pure Greedy*

2️⃣ Boats to Save People — *Sorting + Two Pointer Greedy*

3️⃣ Split Array Largest Sum — *Greedy feasibility + Binary Search on Answer*

🟢 1️⃣ LC 455 – Assign Cookies

💡 Problem

- Children ke paas greed factor `g[i]`
- Cookies ke sizes `s[j]`
- Child tab satisfied jab `s[j] ≥ g[i]`
- **Maximum satisfied children** return karo

🧠 Intuition (Why Sorting?)

Agar random cookie de di:

- Badi cookie chhote greed pe waste ho sakti hai
- Baad me badi greed wala child unsatisfied reh jaata

✅ **Correct Greedy Strategy**

- Dono arrays sort karo
- **Smallest greed → smallest possible cookie**
- Jo minimum me satisfy ho jaaye, use pehle satisfy karo

👉 Ye locally optimal choice globally optimal hai.

🔍 Dry Run

```
g = [1,2,3]
s = [1,1]

After sort:
g = [1,2,3]
s = [1,1]

child=0, cookie=0
1 ≤ 1 → satisfy → count = 1
```

```
child=1, cookie=1
2 ≤ 1 ❌ → cookie skip

End
Answer = 1
```

✅ Java Code (Line-by-Line Commented)

```java
public int findContentChildren(int[] g, int[] s) {

    Arrays.sort(g); // sort greed factors
    Arrays.sort(s); // sort cookie sizes

    int i = 0; // child pointer
    int j = 0; // cookie pointer
    int count = 0;

    while (i < g.length && j < s.length) {

        if (s[j] >= g[i]) {
            // cookie satisfies child
            count++;
            i++;
            j++;
        } else {
            // cookie too small
            j++;
        }
    }

    return count;
}
```

🟢 2️⃣ LC 881 – Boats to Save People

💡 Problem

- Ek boat max **2 log**
- Total weight ≤ `limit`
- **Minimum boats** required

🧠 Intuition (Greedy Proof)

- Heaviest person ko boat chahiye hi
- Agar wo **lightest ke saath bhi fit nahi hota**

  → kisi ke saath bhi fit nahi hoga

✅ Greedy:

- Sort weights
- Lightest + Heaviest try karo
- Agar fit → pair
- Nahi fit → heaviest alone

🔍 Dry Run

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

Sort → [1,2,2,3]

i=0, j=3 → 1+3=4 ❌ → 3 alone → boats=1
i=0, j=2 → 1+2=3 ✅ → boats=2
i=1, j=1 → 2 alone → boats=3

Answer = 3
```

✅ Java Code (Fully Commented)

```
public int numRescueBoats(int[] people, int limit) {

    Arrays.sort(people);
    int i = 0;                  // lightest
    int j = people.length - 1;  // heaviest
    int boats = 0;

    while (i <= j) {

        if (people[i] + people[j] <= limit) {
            // pair possible
            i++;
        }

        // heaviest always goes
        j--;
        boats++;
    }

    return boats;
}
```

🟢 3️⃣ LC 410 – Split Array Largest Sum

💡 Problem

- Array ko `k` subarrays me split karo
- **Maximum subarray sum minimum** karo

🧠 Important Insight (Google Trap)

❌ Normal greedy nahi

❌ Direct sorting bhi nahi

✔️ **Binary Search on Answer + Greedy Feasibility**

🧠 Key Observation

- Minimum possible answer = **max element**
- Maximum possible answer = **total sum**

Hum poochte hain:

> "Kya is maxSum ke andar k subarrays bana sakte hain?"

🔍 Dry Run

```
nums = [7,2,5,10,8], k = 2

low = 10
high = 32

mid = 21

Check:
7+2+5 = 14
+10 = 24 ❌ → new subarray

Total subarrays = 2 → valid
Try smaller...

Final Answer = 18
```

✅ Java Code (Line-by-Line Commented)

```
public int splitArray(int[] nums, int k) {

    int low = 0;
    int high = 0;

    for (int num : nums) {
        low = Math.max(low, num); // max element
        high += num;              // total sum
    }

    while (low < high) {

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

        if (canSplit(nums, k, mid)) {
            high = mid; // try smaller max sum
        } else {
            low = mid + 1;
        }
    }

    return low;
}

private boolean canSplit(int[] nums, int k, int maxSum) {

    int count = 1;
    int currentSum = 0;

    for (int num : nums) {

        if (currentSum + num > maxSum) {
            count++;
            currentSum = num;

            if (count > k) return false;
        } else {
            currentSum += num;
        }
    }

    return true;
}
```

🔥 Advanced Google Sorting Traps

1️⃣ Sorting unnecessary

- Sirf k smallest / largest chahiye?

  👉 Heap use karo, full sort nahi

2️⃣ Stability matters

- Objects with multiple fields
- Relative order important?

  👉 **Stable sort** required

3️⃣ Wrong Greedy Criterion

- Interval problems me:

  ❌ sort by start

  ✅ sort by **end time**

🧠 Phase 6 Summary

Pattern	Key Insight
Assign Cookies	Smallest satisfy first
Boats	Lightest + Heaviest
Intervals	Sort by end
Split Array	Binary search + greedy check

🎯 Reality Check

Bhai honestly bolu —

Ye phase clear = solid interview readiness

Leetcode 451

Ye **"Sort Characters By Frequency"** problem hai (LeetCode type 🔥).

Tum bucket sort approach use kar rahe ho — sahi direction hai.

Main Java me detailed comments ke saath code likh raha hoon 👇

✅ Approach (Bucket Sort)

1. **Frequency map** banayenge.
2. **Bucket array** banayenge jaha index = frequency.
3. High frequency se low tak traverse karke string build karenge.

💻 Java Code (With Detailed Comments)

```
class Solution {
    public String frequencySort(String s) {

        // Step 1: Create frequency map
        // Using array because characters are ASCII
        int[] freq = new int[128];   // 128 ASCII characters

        // Count frequency of each character
        for (char c : s.toCharArray()) {
            freq[c]++;
        }

        // Step 2: Create bucket array
        // Maximum possible frequency = s.length()
        // So bucket size = length + 1
        StringBuilder[] bucket = new StringBuilder[s.length() + 1];

        // Place characters inside bucket according to frequency
        for (int i = 0; i < 128; i++) {
            if (freq[i] > 0) {

                int f = freq[i];  // frequency of current character

                // Initialize bucket if null
                if (bucket[f] == null) {
                    bucket[f] = new StringBuilder();
                }

                // Add character into its frequency bucket
                bucket[f].append((char) i);
            }
        }

        // Step 3: Build final result from high frequency to low
        StringBuilder result = new StringBuilder();
        // Traverse bucket from highest frequency to lowest
        for (int i = s.length(); i >= 1; i--) {

            if (bucket[i] != null) {

                // For each character in this bucket
                for (char c : bucket[i].toString().toCharArray()) {

                    // Append character 'i' times
                    for (int j = 0; j < i; j++) {
                        result.append(c);
                    }
                }
            }
        }

        return result.toString();
    }
}
```

🔍 Example Walkthrough

Input:

```
s = "tree"
```

Step 1: Frequency Map

```
t → 1
r → 1
e → 2
```

Step 2: Bucket

```
bucket[2] = "e"
bucket[1] = "t r"
```

Step 3: Traverse High → Low

First bucket[2] → "ee"

Then bucket[1] → "tr"

Output:

```
"eetr" OR "eert"
```

(Order of same frequency characters can vary)

⏱️ Time Complexity

- O(n) → counting + bucket traversal

📦 Space Complexity

- O(n)

Bahut acha question 👌 ye samajhna important hai.

🔷 Max frequency `s.length()` hi kyu hoti hai?

Socho:

Agar string hai:

```
s = "aaaaa"
```

Length = 5

Yaha character `'a'` ki frequency kitni hogi?

👉 **5**

Kya koi character 6 baar aa sakta hai?

❌ Nahi, kyunki string me total characters hi 5 hain.

Isliye:

> Maximum possible frequency of any character = s.length()

🔷 Fir `s.length() + 1` kyu lete hain?

Bucket array me **index = frequency** use kar rahe hain.

Agar string length = 5

Possible frequencies ho sakti hain:

```
1, 2, 3, 4, 5
```

Ab agar bucket size = 5 rakhenge:

```
new StringBuilder[5];
```

Toh valid index honge:

```
0,1,2,3,4
```

But frequency 5 ko store karne ke liye index 5 chahiye.

Isliye size hona chahiye:

```
new StringBuilder[s.length() + 1];
```

Taaki index range ho:

```
0,1,2,3,4,5
```

🔷 Index 0 ka kya use?

Frequency 0 wale characters hume store nahi karne.

But array indexing 0 se start hoti hai, isliye +1 lena padta hai.

📌 Short Summary

- Max freq = `s.length()`
- Indexing 0-based hoti hai
- Isliye array size = `s.length() + 1`

Agar tum DSA seriously kar rahe ho (jo tum kar rahe ho 💪), to ye indexing logic bohot important hai — especially bucket sort, counting sort, frequency array me.

Ye line thodi fancy lagti hai 😄 but concept simple hai.

Chalo tod ke samajhte hain.

```
Integer[] arr = Arrays.stream(nums).boxed().toArray(Integer[]::new);
```

🔷 Problem kya solve kar rahi hai?

Agar `nums` hai:

```
int[] nums = {1, 2, 3, 4};
```

Ye **primitive array** hai (`int[]`).

Kabhi-kabhi hume `Integer[]` chahiye hota hai (object type), for example:

- Custom Comparator
- Collections.sort
- PriorityQueue with custom comparator
- Sorting in reverse order

Primitive `int[]` me comparator directly use nahi hota.

🔷 Ab line ko break karte hain

1️⃣ `Arrays.stream(nums)`

Ye `int[]` ko **IntStream** me convert karta hai.

```
int[]  →  IntStream
```

2️⃣ `.boxed()`

IntStream primitive stream hota hai.

`.boxed()` primitive `int` ko wrapper `Integer` me convert karta hai.

```
IntStream → Stream<Integer>
```

Example:

```
1 → Integer(1)
```

3️⃣ `.toArray(Integer[]::new)`

Ye stream ko wapas array me convert karta hai.

`Integer[]::new` ek method reference hai

Jo bol raha hai: "Integer array bana do"

Final result:

```
Stream<Integer> → Integer[]
```

🔥 Simple Meaning

Ye line basically kar rahi hai:

> Convert int[] → Integer[]

🔷 Equivalent Manual Way (samajhne ke liye)

```
Integer[] arr = new Integer[nums.length];

for (int i = 0; i < nums.length; i++) {
    arr[i] = nums[i];   // auto-boxing
}
```

Bas stream version concise hai.

🔷 Example Use Case

Reverse sorting:

```
Integer[] arr = Arrays.stream(nums).boxed().toArray(Integer[]::new);

Arrays.sort(arr, Collections.reverseOrder());
```

Ye primitive `int[]` me possible nahi hota.

📌 Short Summary

Part	Meaning
Arrays.stream(nums)	array → stream
boxed()	int → Integer
toArray(Integer[]::new)	stream → Integer[]

Tum DSA me sorting, heap, comparator use karoge to ye line bohot kaam aayegi 💪

Accha 🔥 ye **Largest Number problem** wala comparator hai.

```
Arrays.sort(arr, (a, b) -> ("" + b + a).compareTo("" + a + b));
```

Ye simple numeric compare nahi kar raha.

Ye **string concatenation compare** kar raha hai.

Case:

```
a = 3
b = 30
```

Step 1: Dono possible combinations banao

```
b + a = "30" + "3"  = "303"
a + b = "3" + "30"  = "330"
```

Step 2: Compare karo

```
"303".compareTo("330")
```

String comparison lexicographically hota hai (dictionary order).

Compare first digit:

```
3 == 3
0 < 3 ❌
```

Toh:

```
"303" < "330"
```

Iska matlab:

```
("303".compareTo("330")) < 0
```

Comparator negative return karega.

Java sort rule

Agar comparator negative return kare:

```
(a, b) → negative
```

Toh `a` pehle aayega.

Final Result

Order hoga:

```
3 , 30
```

Kyun?

Kyuki:

```
"330" > "303"
```

Aur hume largest number banana hai,

Isliye `"330"` better combination hai.

🔥 Intuition Samjho

Hum check kar rahe hain:

> Kaunsa order bada number banayega?

If:

```
a+b > b+a
```

Toh `a` pehle aayega.

If:

```
b+a > a+b
```

Toh `b` pehle aayega.

Ek aur example

```
a = 9
b = 34
```

```
b+a = "349"
a+b = "934"
```

"934" > "349"

Toh 9 pehle aayega.

💡 Final Insight

Ye comparator numbers ko numerically compare nahi karta.

Ye check karta hai:

> Kaunsa arrangement largest concatenated number banata hai?

Isi wajah se `3` comes before `30`.

Agar chaho to main iska pura dry run bhi kara du example `[3,30,34,5,9]` pe 👀

Ye line **Largest Number problem** me ek edge case handle karti hai 👇

```
if (arr[0] == 0) return "0";
```

🔥 Ye kyu zaroori hai?

Socho input hai:

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

Sorting ke baad array hoga:

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

Agar tum normally concatenate karoge:

```
"000"
```

But correct answer kya hona chahiye?

```
"0"
```

Kyuki multiple leading zeros allowed nahi hote.

🔎 Logic kya hai?

Largest number banane wale comparator se sort karne ke baad:

- Sabse bada element index `0` pe hoga.
- Agar wo `0` hai,

  👉 iska matlab saare elements 0 hi hain.

Kyun?

Agar ek bhi non-zero hota, wo 0 se pehle aata.

Example:

```
[0, 1]
```

Sort hone ke baad:

```
[1, 0]
```

So arr[0] kabhi 0 nahi hoga agar koi non-zero exist karta ho.

📌 Isliye:

```
if (arr[0] == 0)
```

Matlab:

> Sab numbers 0 hi hain.

Toh simply return:

```
"0"
```

💡 Short Summary

Ye line:

- "0000" jaise cases avoid karti hai
- Ensure karti hai output sirf `"0"` aaye

Ye edge case handling interviews me bohot important hota hai 👌

Chhoti line hai but interviewer isi pe test karta hai 😄