Day 1

📘 DSA Math Essentials
🔢 Number Systems
➤ Understanding Bases
A number in base N uses digits from 0 to N-1.
Examples:

- Base 2 → 0, 1 (Binary)
- Base 3 → 0, 1, 2
- Base 8 → Octal
- Base 10 → Decimal
- Base 16 → Hexadecimal (Digits: 0-9 and A-F, where A=10, ..., F=15)

🔄 Conversions
➤ Base-N to Decimal
To convert a number from any base to base 10:

1. Write each digit with its positional power.
2. Multiply each digit by the base raised to its position (starting from right, index 0).
3. Sum the results.

Example: Convert 1011 (base 2)
$= 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
$= 8 + 0 + 2 + 1 = 11$

➤ Decimal to Base-N
To convert a decimal number to any base:

1. Divide the number by the new base.
2. Note the remainder.
3. Replace the number with the quotient.
4. Repeat until the quotient is 0.
5. Read remainders bottom to top for the result.

Example: Convert 11 to base 2
$11 \div 2 = 5$ R1
$5 \div 2 = 2$ R1
$2 \div 2 = 1$ R0
$1 \div 2 = 0$ R1
→ Result: 1011

✅ Even or Odd Check

Use modulus:

if (num % 2 == 0) → even

else → odd

⏫ Exponentiation

➤ Regular Power

- $a^1 = a$
- $a^0 = 1$
- $a^{-n} = 1 / a^n$
- Example: $5^3 = 5 \times 5 \times 5 = 125$

➤ Fast Exponentiation (Binary Exponentiation)

Optimized power calculation in O(log n) time:

- Uses squaring and divides the problem recursively.
- Example:

```
2^7 → 2 * (2^2)^3
    → 2 * (4)^3
    → 2 * 4 * (4^2)^1
    → 2 * 4 * 16 * 1
```

→ So, binary numbers with a single leading 1 and rest zeros are **powers of 2**.

➤ **In Base 3 (ternary):**

- $10_3 = 3_{10} \rightarrow 3^1$
- $100_3 = 9_{10} \rightarrow 3^2$
- $1000_3 = 27_{10} \rightarrow 3^3$

  → So, ternary numbers of this pattern are **powers of 3**.

📌 1. Counting Digits in a Number

To find how many digits are in a number, you divide it repeatedly by 10 until it becomes 0. Each division removes one digit.

Time Complexity Insight:
Since you're dividing by 10 each time, the number of operations is proportional to log base 10 of the number.

Formula:
Number of digits in $n = \text{floor}(\log_{10}(n)) + 1$

📌 2. Armstrong Number

An **Armstrong number** is a number where the sum of its digits raised to the power of the number of digits equals the number itself.

Example:
$153 \rightarrow 1^3 + 5^3 + 3^3 = 153$ ✅

General Rule:
For an n-digit number x:
$x = a_1^n + a_2^n + \dots + a_n^n$

Note:

- All 1-digit numbers are Armstrong.
- Permutations of an Armstrong number are **not** always Armstrong.
- Armstrong numbers are **not necessarily** palindromes.
- The sum of Armstrong numbers isn't guaranteed to be Armstrong.

(⚠ Some properties you noted like permutation or palindrome aren't accurate for all Armstrong numbers.)

📌 3. Palindrome Number

A number is a **palindrome** if it reads the same forward and backward.

How to check:

1. Reverse the number.
2. Compare it with the original.

If both match, it's a palindrome.

📌 4. Square Root using Newton-Raphson Method

Used for approximating square roots efficiently.

Steps:

1. Start with a guess X = N.
2. Use formula:
   root = 0.5 * (X + (N / X))
3. Repeat until |root - X| is within a small margin (tolerance).
4. Return the root.

📌 5. Print All Divisors of a Number

For a number N, divisors are numbers that divide it completely (i.e., remainder is 0).

Optimized Approach:

- Only check from 1 to √N.
- For every divisor i, its pair N/i is also a divisor.

📌 6. Prime Number Check

A **prime number** has only two divisors: 1 and itself.

Efficient Check:

- Try dividing N from 2 to √N.
- If divisible, it's not prime.

📌 7. Sieve of Eratosthenes

Used to find all primes from 0 to N.

Idea:

1. Start with 2 (the first prime).
2. Eliminate all its multiples.
3. Move to the next unmarked number and repeat.
4. Remaining numbers are prime.

📌 8. GCD (Greatest Common Divisor)

GCD of two numbers is the largest number that divides both.

Euclidean Algorithm:

1. If a > b, set a = a % b
2. Repeat until b = 0
3. a is the GCD

Example:

GCD(60, 48) → 60 % 48 = 12 → 48 % 12 = 0 → GCD = 12

📌 9. LCM (Least Common Multiple)

LCM is the smallest number divisible by both a and b.

Formula:
lcm(a, b) = (a * b) / gcd(a, b)

📌 10. Modular Arithmetic Rules

When working with mod (%), these properties are useful:

1. (a + b) % m = ((a % m) + (b % m)) % m
2. (a - b) % m = ((a % m) - (b % m) + m) % m
3. (a * b) % m = ((a % m) * (b % m)) % m
4. (a / b) % m = (a % m) * (b^{-1} % m) % m
   where b^{-1} is the modular inverse of b modulo m

📌 11. Factorial (!N)

N! = N × (N - 1) × ... × 2 × 1

📌 12. Trailing Zeros in a Number

To count zeros at the **end** of a number:

- Extract digits from the end.
- Count how many 0s before a non-zero appears.

📌 13. Trailing Zeros in a Factorial

In N!, trailing zeros are formed by pairs of 2 × 5.
Since 2s are always more than 5s, just count 5s.

Count of 5s:

trailing_zeros = N/5 + N/25 + N/125 + ...

Why?

- Every multiple of 5 contributes one 5.
- Multiples of 25 contribute two 5s, and so on.

Example:

For N = 30
Trailing Zeros = $\lfloor 30/5 \rfloor + \lfloor 30/25 \rfloor$ = 6 + 1 = **7**

Day 2

🧠 Bit Manipulation Notes

🔹 MSB and LSB

1. **MSB (Most Significant Bit)** - This is the **leftmost bit** in a binary number.
   - For **signed numbers**, the MSB shows the **sign**:
     - 0 → positive
     - 1 → negative
2. **LSB (Least Significant Bit)** - This is the **rightmost bit**, affecting the smallest value.
3. **Range of Numbers**:
   - **Signed integers** → From $-(2^{n-1})$ to $(2^{n-1} - 1)$
   - **Unsigned integers** → From 0 to $(2^n - 1)$

➕ Bit Addition Rules

1. 1 + 1 = 10 → results in a carry
2. 1 + 1 + 1 = 11 → carry is generated again
3. Extra bits beyond capacity get **discarded** (overflow is ignored).

➖ Bit Subtraction with 2's Complement

To subtract numbers using bits:

1. Find **2's complement** of the number you want to subtract (i.e., negate).
   - Step 1: Invert all bits (1's complement)
   - Step 2: Add 1 to it
2. Add this complement to the other number.
3. Ignore the carry if it goes beyond n bits.

🧮 Bitwise Operators

1. AND (&)
- Only 1 & 1 = 1, everything else is 0.
- Special rule: x & 1 gives you the **last bit** (useful to check even/odd).

2. OR (|)
- At least one 1 gives result 1. Otherwise, 0.

3. XOR (^)
- 1 ^ 1 = 0, 0 ^ 1 = 1, 0 ^ 0 = 0

- Useful for **toggling bits**:
    - $x \wedge 0 = x$
    - $x \wedge 1 = \sim x$

4. **Left Shift (<<)**
   - Shifts all bits to the **left**, adds 0 on the right.
   - Example: 10110 << 2 → 1011000
   - Mathematically: $a << b = a \times 2^b$
5. **Right Shift (>>)**
   - Shifts bits to the **right**, drops bits from right side.
   - Example: 10110 >> 2 → 101
   - Mathematically: $a >> b = a \div 2^b$
6. **Bitwise NOT (~)**
   - Inverts all bits: 0 ↔ 1
   - Example: ~(1011100) = 0100011

🔎 Check Even or Odd with Bits

- Use: x & 1
    - If result is 0, then **even**
    - If result is 1, then **odd**

🔍 Observing Patterns in Bit Representation (Base Change Method)

When a number has **one 1 followed by only zeros** in a specific base, it's a **power** of that base.

➤ **In Base 2 (binary):**

- $10_2 = 2_{10} \rightarrow 2^1$
- $100_2 = 4_{10} \rightarrow 2^2$
- $1000_2 = 8_{10} \rightarrow 2^3$

✍️ Bit Manipulation Questions - Notes

---

1. ✅ Check if a Number is Even or Odd

Use: num & 1

- If result is 1 → **Odd**
- If result is 0 → **Even**

➤ Because only odd numbers have the **last bit set** (LSB = 1)

---

2. 🔍 Find the i-th Bit of a Number

To extract the bit at position i (0-based):

- Step 1: Create a mask → (1 << i)
- Step 2: Use AND operation → num & (1 << i)

➤ If result is non-zero, bit is **1**, else it's **0**

---

3. ✔ Check if i-th Bit is Set

Same as above, but explicitly check:

```
if ((num & (1 << i)) == 0)
    → bit is NOT set (it's 0)
else
    → bit is set (it's 1)
```

---

4. ✨ Set the i-th Bit

To **turn on** the bit at index i:

- Mask: (1 << i)
- Operation: num = num | (1 << i)

➤ This ensures the bit is set to 1, even if it wasn't before.

---

5. ❌ Unset (Clear) the i-th Bit

To **turn off** the bit at index i:

- Mask: ~(1 << i)
- Operation: num = num & ~(1 << i)

⚠ Original had a small mistake (! instead of ~)
➤ This clears only the i-th bit, rest remain unchanged.

---

6. 🔁 Toggle the i-th Bit

To **flip** the i-th bit (0 → 1 or 1 → 0):

- Mask: (1 << i)
- Operation: num = num ^ (1 << i)

---

7. 🧯 Remove the Last Set Bit

To **clear** the rightmost set bit (i.e., turn off the lowest 1):

- Use: num = num & (num - 1)

➤ Helpful in counting set bits or optimizing space.

---

8. 🔢 Count Set Bits in a Number

Use Brian Kernighan's Algorithm:

```
count = 0
while (n != 0) {
    n = n & (n - 1)
    count++
}
```

➤ Each step removes one 1. Loop continues until n = 0.

---

9. 📐 Check if a Number is a Power of 2

Property: Power of 2 has **only one set bit.**
Use: n & (n - 1)

- If result is 0, **it is a power of 2**
- Also ensure n != 0

---

10. 🧾 Print Bits of a Number

To print each bit from **MSB to LSB** (say 31 to 0 in 32-bit):

```
for (int i = 31; i >= 0; i--) {
    cout << ((num >> i) & 1);
}
```

➤ Shifts the i-th bit to the end and prints it.

Day 3

📘 ARRAY in Java - Notes

📌 What is an Array?

An **array** is a **container object** that holds a **fixed number of values** of the **same type**.

In Java, arrays are **objects** and are stored in **heap memory**.

💡 Useful when you want to store **multiple values under one variable name**.

🛠️ Declaration of Array:

```
int[] arr;        // preferred way
int arr[];        // also valid
```

🏗️ Memory Allocation:

java

```
arr = new int[5];  // allocates memory for 5 integers
```

📃 Declaration + Initialization:

```
int[] arr = new int[5];              // default values: 0
int[] arr = {10, 20, 30, 40, 50};    // initialized directly
```

🧠 Important Points:

- Array index starts from 0
- Java arrays are **fixed in size**
- Array is an **object** → we can use .length to get size
- All elements are stored **contiguously** in memory

🖌️ Sorting an Array:

Using Java's built-in method:

```java
import java.util.Arrays;
Arrays.sort(arr);   // sorts in ascending order
```

🧠 Multidimensional Array:

```java
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6}
};
System.out.println(matrix[1][2]); // Output: 6
```

⏱️ Time Complexity:

Operation	**Time Complexity**
Access	O(1)
Search	O(n)
Insert/Delete	O(n) (worst)

✨ Key Takeaways:

- Arrays are **fixed in size**
- Use .length to get number of elements
- Java provides **utility methods** in Arrays class
- For dynamic arrays → use ArrayList

📘 What is Linear Search?

Linear Search (also called Sequential Search) is a simple technique used to find a particular element in an array.
It checks **each element one by one**, from **start to end**, until the desired element is found or the array ends.

✅ When to Use Linear Search?

- Array is **unsorted**
- Array is **small in size**
- Simpler logic is preferred over efficiency

🔁 Algorithm Steps

1. Start from index 0
2. Compare each element with the target
3. If element matches → return index
4. If no match till the end → return -1

Day 4

💡 What is Binary Search?

Binary Search is an efficient algorithm to find an element in a **sorted array**.
It **repeatedly divides** the search range in half, checking the middle element each time.

✅ Prerequisite:

➡️ Array must be **sorted** (either ascending or descending)

🪜 Steps (For Ascending Order):

1. Set low = 0, high = arr.length - 1
2. While low <= high:
   - Calculate mid = (low + high) / 2
   - If arr[mid] == target, return mid
   - If target < arr[mid], search left → high = mid - 1
   - Else, search right → low = mid + 1
3. If element not found → return -1

📍 1. binarySearch1() - First or Last Occurrence

This method finds:

- The **first** or **last occurrence** of a target value.
- Use the boolean isFirst:
  - true → search for **first**
  - false → search for **last**

➡️ Logic:

- When found, save index and continue left (first) or right (last)

📍 2. binarySearch() – Utility for searchRange

- Same as above, but returns the index instead of printing.
- Used by searchRange() method.

📍 3. searchRange() – First and Last Position

java

CopyEdit

int[] searchRange(int[] arr, int target)

- Returns [firstOccurrence, lastOccurrence]
- Useful to **count occurrences**:
  ➡️ count = last - first + 1
- If element not found → returns [-1, -1]

📍 4. binarySearchFloor()

- Finds the **floor** of the target:
  ➡️ Largest value ≤ target
- If found → return value
- Else → keep storing potential answers

📍 5. binarySearchCeil()

- Finds the **ceil** of the target:
  ➡️ Smallest value ≥ target
- Logic is similar to floor, but we save values on the **right side**

📍 6. findAbsoluteMinimumDif()

- Finds the **closest difference** between the target and array values
- If target exists → difference is 0

- Else → Compare arr[start] and arr[end]
  ➡️ Return minimum difference

📍 7. findRangeOfInfiniteArray()

Used when array size is **unknown** (infinite sorted array):

java

CopyEdit

start = 0, end = 1

While arr[end] < target → keep doubling end

➡️ This gives a range: [start, end] where the element might exist

---

📍 8. binarySearchInRange()

- A simple binary search, but limited to the range [start, end]
- Used after range is determined in an infinite array

Day 5

🌄 What is a Bitonic Array?

A **bitonic array** is:

- First **increasing**, then **decreasing**.
- No need to be strictly increasing/decreasing (but in code, usually we use **strictly** increasing/decreasing).

✅ Example: 2, 4, 8, 12, 9, 5, 1
❌ Not bitonic: 5, 5, 4, 3 (not strictly increasing)

🧭 Monotonic vs Bitonic

- **Monotonic** → Always increases or always decreases
- **Bitonic** → Increases first, then decreases

🔍 Functions Used

✅ findMinElement(int[] arr)

- Returns the **minimum value** from bitonic array
- 🔸 Compares only **first** and **last** elements

📝 Why this works?

- In a bitonic array, either the first or the last will be the smallest.

✅ validateMountainArray(int[] arr)

Checks if the array is a **valid mountain** (bitonic) array

✅ Conditions:

- Length ≥ 3

- Strictly increasing till a peak
- Strictly decreasing after the peak

🧠 Steps:

1. Start from left and move forward until it stops increasing.
2. If we never increased or reached the end → return false.
3. After the peak, make sure it's strictly decreasing till the end.

✅ **findPeakElement(int[] arr)**

- Uses **Binary Search** to find **peak element** (maximum point)

🧠 **Peak** = element which is greater than both left & right neighbors

🔄 Binary Search Logic:

- If arr[mid] > arr[mid-1] and arr[mid] > arr[mid+1] → it's peak
- If arr[mid] < arr[mid+1] → move to **right half**
- Else → move to **left half**

⏱️ Time complexity = **O(log n)**

🔰 **Rotated Sorted Array**

- A **sorted array** that has been **rotated** at some pivot
- Example:
  Original: [0,1,2,3,4,5,6]
  Rotated: [4,5,6,0,1,2,3]

✅ Functions Covered

---

1 search(int[] arr, int target)

🎯 **Goal:**
Find index of target in rotated sorted array (no duplicates)

🔍 **Logic:**

- Use **binary search**
- Check if **left part is sorted**
    - If yes, check if target lies in that part
- Else, right part must be sorted
    - If yes, check if target lies there

🧠 **Why it works:**
Only one part (left or right) is always sorted in a rotated array!

⏱ Time complexity: O(log n)

2 searchWithDuplicate(int[] arr, int target)

🎯 **Goal:**
Search in rotated array **with duplicate values**

🔍 **Extra Handling for Duplicates:**

- If arr[start] == arr[mid] == arr[end] → can't decide sorted part
- So, move start++ and end-- to skip duplicates

🧠 **Why needed?**
Duplicates break the logic of identifying the sorted half, so we reduce size manually.

⏱ Time complexity:

- Worst case: O(n) (when many duplicates)

---

3 findMin(int[] arr)

🎯 **Goal:**
Find the **minimum element** in a rotated sorted array

🔍 **Logic:**

- Binary search + keep track of minElement
- If left part is sorted → min is arr[start]
- Else → min is arr[mid]

✅ At end → return min value

Day 6

📚 Allocate Minimum Pages Problem

🧩 Problem Statement:

You are given:

- An array of books, where each book has some pages
- A number of students

🎯 Goal:

Distribute books **in order** to students so that:

- Every student gets at least one book
- Each book is allocated to exactly one student
- **Minimize the maximum number of pages assigned to a student**

📌 Constraints:

- books.length >= students
- Books must be assigned **contiguously**

✅ Approach: Binary Search on Answer

🧮 Step 1: Define Search Space

- Minimum pages → **max** element in books array
- Maximum pages → **sum** of all pages

Day 7

Problem:
Koko wants to eat all banana piles within h hours.
Each hour, she eats k bananas from a single pile (if less than k bananas remain, she eats them all).
Goal: Find the minimum integer eating speed k to finish all bananas in time.

Key Idea:

Use **Binary Search** on possible eating speeds (k) between 1 and max pile size.

Steps:

1. **Find max pile size:**
   This is the upper bound for speed k.
2. **Initialize search range:**
   - start = 1 (minimum possible speed)
   - end = max pile size
3. **Binary Search loop:**
   - Calculate mid = start + (end - start)/2 (mid speed)
   - Check if Koko can finish all piles with speed = mid in h hours:
     - Use helper function isEatingSpeedValid()
   - If true (can finish), store mid as possible answer and try to find smaller speed (reduce end)
   - Else, increase speed (start = mid + 1)
4. When loop ends, ans holds the minimum valid speed.

Helper function: isEatingSpeedValid(piles, speed, hours)

- Calculate total hours needed to finish all piles at current speed:
  For each pile:
  - hoursSpent += piles[i] / speed (integer division)
  - If remainder exists (piles[i] % speed != 0), add 1 more hour
- If total hours needed > hours → return false (speed too slow)
- Else → return true

Day 8

Problem Statement:

*Given an array nums and a threshold value, find the **smallest divisor** such that when every element in nums is divided by this divisor and rounded up, the sum of these results is **less than or equal to the threshold**.*

Approach:

*Use **Binary Search** on possible divisor values between 1 and max element in the array.*

Steps:

1. ***Find max element in array:***
   This will be the upper limit for divisor search.
2. ***Initialize search boundaries:***
   - *start = 1 (minimum divisor)*
   - *end = max element in nums*
3. ***Binary Search loop:***
   - *Calculate mid = (start + end) / 2 (current divisor)*
   - *Check if this divisor produces a sum of divisions ≤ threshold using helper function isDivisorPossible()*
   - *If yes, record this divisor as possible answer and try smaller divisor by setting end = mid - 1*
   - *Else, increase divisor by setting start = mid + 1*
4. *When loop ends, res holds the smallest valid divisor.*

Helper Function: isDivisorPossible(nums, divisor, threshold)

- *Calculate sum of division results for all elements using current divisor:*

    - For each element,
        - Divide by divisor using integer division
        - If remainder exists (element % divisor != 0), add 1 extra to round up
    - Keep track of total sum
- If sum > threshold, return false (divisor too small)
- Else return true

Key Points:

- The main trick is binary searching the divisor value, not directly iterating.
- Rounding up division results is done by adding 1 when remainder exists.
- Stop early if sum exceeds threshold to optimize

start = max(pages)

end = sum(pages)

🔁 Step 2: Binary Search

- For each mid (pages limit), check:
    - Can we **allocate books** to all students without any student getting more than mid pages?
- If yes → try to minimize further: end = mid - 1
- If no → increase the limit: start = mid + 1

🧪 Helper: isAllocationPossible()

💡 Logic:

- Loop over all books
- Assign to current student until pages > limit (maxPages)
- Then allocate to **next student**
- Count number of students needed
    - If count > given students → ❌ Not possible

Day 9

💡 **Problem: Minimize the Maximum Products in Any Store**

Given:

- quantities[]: array where each element = quantity of a product type
- n: total number of stores

📌 **Goal:**
Distribute all products to n stores such that **no store gets more than X items**, and we want to **minimize X**.

✅ **Idea: Use Binary Search on possible values of X (max products per store)**

📌 **Steps:**

1. **Set range for binary search:**
   - start = 1 (min limit, at least 1 product per store)
   - end = max value in quantities[]
     (since no store will ever receive more than the largest product quantity)

---

2. **Binary Search Logic:**

```
while (start <= end):
    mid = (start + end) / 2
    if distribution is possible with maxProducts = mid:
        save mid as answer
        try to minimize it → end = mid - 1
    else:
        increase mid → start = mid + 1
```

🔍 Helper Function: isDistributionPossible(quantities, maxProducts, n)

🛒 Goal:

Check if we can distribute products such that **no store gets more than maxProducts items**

Logic:

- For each quantity:
  - storeCount += quantity / maxProducts
  - If there's a remainder → need 1 more store (% != 0)
- If total storeCount > n, return **false**
- Else, return **true**

Day 10

🐄 Problem: Aggressive Cows (Binary Search on Answer)

🧩 Problem Statement:

- You are given n stall positions and cows number of cows.
- Place the cows in stalls such that **minimum distance between any two cows is maximized.**
- Find that **maximum possible minimum distance.**

🛠️ Approach: Binary Search on Distance

✨ Steps:

1. Sort the stall positions:

java

CopyEdit

```java
Arrays.sort(stalls);
```

2. Set binary search range:

- start = 1 → minimum possible distance
- end = stalls[n-1] - stalls[0] → maximum possible distance between two farthest stalls

3. Binary Search Logic:

java

CopyEdit

```
while (start <= end):
    mid = (start + end) / 2  → this is current minimum distance to test
    if allocation possible at this distance:
        store mid in ans
        try for more distance → start = mid + 1
    else:
        try smaller distance → end = mid - 1
```

✅ **Function: isAllocationPossible(stalls, minDistance, cows)**

- Start by placing the **first cow at the first stall**
- For every next stall:
  - If distance between current stall and last stall with cow ≥ minDistance, place new cow there
- Keep a count of cows placed
- If placed cows ≥ required cows → return true
- Else return false

```
if (pages > maxPages)
  move to next student
```