What are Data Structures and Algorithms Terminology

- **Data structure** — a way to store and organize data so you can use it efficiently (examples: arrays, linked lists, trees, hash maps).
- **Algorithm** — a step-by-step procedure to solve a problem (examples: sorting, searching, shortest path).
- **Interplay:** choosing the right data structure often makes the algorithm simpler/faster (e.g., using a hash map for fast lookup).

What is a Data Structure

- A **data structure** is a way of **organizing, storing, and managing data** so that it can be used efficiently.
- It provides operations like **insertion, deletion, searching, updating, and traversal**.
- Example: Imagine a library – books can be organized in different ways (alphabetically, by subject, by author). The method you choose to organize them is like a data structure.

Why are Data Structures important

- Helps in writing **efficient programs**.
- Saves **time** (faster access, searching, processing).
- Saves **memory** (proper arrangement of data).
- Forms the **base of algorithms** used in real-world applications (databases, OS, AI, compilers, etc.).

Types of Data Structures

(A) Primitive Data Structures

- Basic data types provided by programming languages.
- Examples in Java:
  - int, float, char, boolean, double.
- These store single values (atomic).

(B) Non-Primitive Data Structures

Abstract Data Types (ADT)

- These define **what operations** can be performed but **not how** they are implemented.
- Examples:
  - **List** (ordered collection of items).
  - **Stack** (push, pop).
  - **Queue** (enqueue, dequeue).
  - **Tree**, **Graph**.
- These are more advanced structures built using primitive types.
- They are divided into two categories:

1. Linear Data Structures

- Data elements are arranged **sequentially**, one after another.
- Easy to traverse (using loops).
- Examples:
    1. **Array** – fixed-size, continuous memory, fast access.
    2. **Linked List** – dynamic memory, nodes connected by pointers.
    3. **Stack** – LIFO (Last In First Out), e.g., undo operation in Word.
    4. **Queue** – FIFO (First In First Out), e.g., printer queue, ticket line.

2. Non-Linear Data Structures

- Data elements are arranged **hierarchically or as a network**.
- One element can connect to multiple elements.
- Examples:
    1. **Tree** – hierarchical structure (root → children → leaves).
        - Binary Tree, Binary Search Tree, AVL, Heap, etc.
    2. **Graph** – set of nodes (vertices) and edges (connections).
        - Social networks, maps, routes.

Basic Operations on Data Structures

Almost all DS support these fundamental operations:

Operation	Description
Traversal	Accessing each data element exactly once
Insertion	Adding a new element
Deletion	Removing an element
Searching	Finding a specific element
Sorting	Arranging elements in order (ascending/descending)
Merging	Combining two or more data structures

Some Basics of Data Structures

1. Decimal to Binary

1. Take the decimal number.
2. Divide it by 2.
3. Write down the remainder (0 or 1).
4. Update the number = quotient of division.
5. Repeat steps 2–4 until the number becomes 0.
6. Write all remainders in **reverse order** → That's the binary.

Example:
Decimal = 13

Step	Divide by 2	Quotient	Remainder
1	13 ÷ 2	6	1
2	6 ÷ 2	3	0
3	3 ÷ 2	1	1
4	1 ÷ 2	0	1

Write remainders in reverse → **1101** → Binary

2. Binary to Decimal

1. Take the binary number.
2. Start from the **rightmost bit**, assign position 0.
3. Multiply each bit by **2^position**.
4. Add all the results → That's the decimal number.

Example
Binary = 1101

Bit (from right)	Position	Multiply by 2^position	Value
1	0	1 × 2^0	1
0	1	0 × 2^1	0
1	2	1 × 2^2	4
1	3	1 × 2^3	8

Add all → 1 + 0 + 4 + 8 = **13** → Decimal

Bitwise Operators in Java

1. Overview

- **Definition:** Bitwise operators work **on individual bits** of integer types (`int`, `long`, `byte`, `short`).
- **Purpose:** Efficient manipulation of bits, fast computations, and low-level operations.
- **Common Uses:** Masking, checking bits, setting bits, toggling, swapping, competitive programming, encryption.

List of Bitwise Operators

Operator	Symbol	Description
AND	&	Bitwise AND
OR	|	Bitwise OR
XOR	^	Bitwise Exclusive OR
NOT	~	Bitwise Complement (NOT)
Left Shift	<<	Shift bits left, fill 0 on right
Right Shift	>>	Shift bits right, sign-extended
Unsigned Right Shift	>>>	Shift bits right, fill 0 on left

Bitwise AND (&)

- **Definition:** 1 if **both bits are 1**, else 0
- **Truth Table:**

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

- **Java Example:**

```
int a = 5; // 101
int b = 3; // 011
System.out.println(a & b); // 1
```

Bitwise OR (|)

- **Definition:** 1 if **any bit is 1**, else 0
- **Truth Table:**

A	B	A | B
0	0	0

A	B	A | B
0	1	1
1	0	1
1	1	1

- **Java Example:**

```
int a = 5; // 101
int b = 3; // 011
System.out.println(a | b); // 111 = 7
```

- **Applications:** Setting bits, combining masks, flag operations.

5. Bitwise XOR (^)

- **Definition:** 1 if **bits are different**, else 0
- **Truth Table:**

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

- **Java Example:**

```
int a = 5; // 101
int b = 3; // 011
System.out.println(a ^ b); // 110 = 6
```

- **Applications:** Swapping numbers without temp, finding unique elements.

6. Bitwise NOT (~)

- **Definition:** Flips all bits (0 → 1, 1 → 0)
- **Example:**

```
a = 5 → 00000101
~a = 11111010 → -6 (Two's complement)
```

- **Java Example:**

```
int a = 5;
System.out.println(~a); // -6
```

- **Applications:** Complement operations, negative numbers handling, masking.

7. Shift Operators

Left Shift (<<)

- Shifts bits left by specified positions, **fills 0 on the right**
- Equivalent to multiplying by 2^n
- **Example:**

```
int a = 5; // 101
System.out.println(a << 1); // 1010 = 10
```

Right Shift (>>)

- Shifts bits right, **sign bit (MSB) is preserved**
- Equivalent to dividing by 2^n (floor for positive, ceil for negative)
- **Example:**

```
int a = 10; // 1010
System.out.println(a >> 1); // 0101 = 5
int n = 5; // 101
int pos = 0;
n = n ^ (1 << pos); // 100 = 4
```

Quick Tips / Summary

Operator	Function	Example
&	AND	Masking, checking even
|	OR	Setting bits, flags
^	XOR	Swap without temp, unique element
~	NOT	Bit complement
<<	Left Shift	Multiply by 2^n
>>	Right Shift	Divide by 2^n (signed)
>>>	Unsigned Right Shift	Divide by 2^n (fill 0)

Key Points:

1. Bitwise ops are **fast (O(1))**
2. Widely used in **competitive programming and system-level tasks**
3. Mastery of **masking, toggling, setting/clearing bits** is essential

Time and Space Complexity

Time Complexity

- It measures **how much time an algorithm takes** to run as a function of the input size **n**.
- It helps us **compare algorithms** independent of hardware or programming language.

Example:

```
for(int i=0; i<n; i++) {
   cout << i;
}
```

→ Runs `n` times → **Time Complexity = O(n)**

Space Complexity

- It measures the **amount of extra memory (RAM)** required by an algorithm.
- Includes:
  - **Input Space**
  - **Auxiliary Space** (extra variables, recursion stack, etc.)

Example:

```
int a[n]; // uses n memory cells
```

→ **Space Complexity = O(n)**

Asymptotic Notations

These notations describe **algorithm efficiency** as input size `n → ∞`.

Notation	Meaning	Describes	Example
Big O (O)	**Upper Bound**	Worst-case performance	$O(n^2)$
Big Omega (Ω)	**Lower Bound**	Best-case performance	Ω(n)
Big Theta (Θ)	**Tight Bound**	Average or exact performance	Θ(n log n)

(a) Big O Notation — Worst Case

Represents the **maximum time** an algorithm can take.
It gives an **upper limit**.

Example:

```
for(int i=0; i<n; i++)  // O(n)
```

```
for(int j=0; j<n; j++)  // O(n)
   cout << i+j;
```

→ Total = **O(n²)**

(b) Big Omega (Ω) — Best Case

Represents the **minimum time** an algorithm will take.
Gives a **lower bound**.

Example:
In **Linear Search**,

- Best case → element found at first position → **Ω(1)**

(c) Big Theta (Θ) — Average / Tight Bound

When an algorithm's upper and lower bounds are the same.
It shows the **exact growth rate**.

Example:
If an algorithm is **O(n)** and **Ω(n)** → then it is **Θ(n)**.

Common Time Complexities

Complexity	Name	Example
O(1)	Constant	Accessing array element
O(log n)	Logarithmic	Binary search
O(n)	Linear	Linear search
O(n log n)	Log-linear	Merge sort, Quick sort
$O(n^2)$	Quadratic	Bubble sort
$O(2^n)$	Exponential	Recursive Fibonacci

Arrays in Java

An **array** is a **collection of elements of the same data type**, stored in **contiguous memory locations**.
It allows random access using an **index**.

Declaration and Initialization

```
// Declaration
```

```
int[] arr;

// Memory allocation
arr = new int[5];   // 5 elements, all initialized to 0

// Declaration + Initialization
int[] nums = {10, 20, 30, 40, 50};
```

Accessing Elements

```
System.out.println(nums[0]);  // prints 10
nums[2] = 99;                 // updates 3rd element
```

Length of Array

```
System.out.println(nums.length); // 5
```

Traversal Example

```
for(int i = 0; i < nums.length; i++)
    System.out.println(nums[i]);
```

Enhanced for-loop

```
for(int n : nums)
    System.out.println(n);
```

Multidimensional Arrays

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

Common Operations

Operation	Example	Time Complexity
Access	arr[i]	O(1)
Update	arr[i] = x	O(1)
Traversal	loop through	O(n)
Insertion / Deletion (in middle)	shift elements	O(n)

2. Strings in Java

A **String** in Java is a **sequence of characters** enclosed in double quotes (`" "`).
It is an **object** of the **`String` class** (immutable).

Creation

```
String s1 = "Hello";
String s2 = new String("World");
```

String Immutability

Once created, a `String` **cannot be changed**.
All operations that modify a string create a **new String object**.

Example:

```
String s = "Java";
s.concat(" Rocks");
System.out.println(s);  // Output: Java  (not "Java Rocks")
```

Important String Methods

Method	Description	Example	Output
`length()`	Returns string length	`"Hello".length()`	`5`
`charAt(i)`	Returns character at index	`"Java".charAt(2)`	`'v'`
`toUpperCase()` / `toLowerCase()`	Changes case	`"java".toUpperCase()`	`"JAVA"`
`substring(start, end)`	Extract substring	`"Hello".substring(1,4)`	`"ell"`
`equals()` / `equalsIgnoreCase()`	Compare strings	`"Hi".equals("hi")`	`false`
`compareTo()`	Lexicographic compare	`"A".compareTo("B")`	`-1`
`indexOf(ch)`	Find index of char	`"banana".indexOf('a')`	`1`
`contains(str)`	Checks substring	`"hello".contains("he")`	`true`
`trim()`	Removes extra spaces	`" Java ".trim()`	`"Java"`
`split(" ")`	Splits into array	`"a b c".split(" ")`	`["a","b","c"]`

String Concatenation

```
String s1 = "Hello";
String s2 = "World";
String s3 = s1 + " " + s2;  // "Hello World"
```

Or using:

```
s1.concat(s2);
```

Mutable Strings

If you need modifiable strings, use:

StringBuilder (faster, single-threaded)

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" Java");
System.out.println(sb); // Hello Java
```

StringBuffer (thread-safe, slower)

```
StringBuffer sbf = new StringBuffer("Hi");
sbf.append(" Everyone");
System.out.println(sbf); // Hi Everyone
```

Common StringBuilder Methods

Method	Description
`append()`	Add text
`insert(pos, str)`	Insert text
`delete(start, end)`	Delete range
`reverse()`	Reverse string
`toString()`	Convert to String

Basic Operations on Arrays

```
class MyArray {
    private int[] arr;
    private int size; // current number of elements
    private int capacity; // total capacity

    // Constructor
    public MyArray(int capacity) {
        this.capacity = capacity;
        arr = new int[capacity];
        size = 0;
    }

    // Insert element at specific position
    public void insert(int element, int pos) {
        if (size == capacity) {
            System.out.println("Array is full! Cannot insert.");
            return;
        }
        if (pos < 0 || pos > size) {
            System.out.println("Invalid position!");
            return;
        }
        // Shift elements right
        for (int i = size; i > pos; i--) {
            arr[i] = arr[i - 1];
        }
        arr[pos] = element;
        size++;
        System.out.println("Inserted " + element + " at position " + pos);
```

```
    }

    // Delete element at specific position
    public void delete(int pos) {
        if (pos < 0 || pos >= size) {
            System.out.println("Invalid position!");
            return;
        }
        System.out.println("Deleted element: " + arr[pos]);
        for (int i = pos; i < size - 1; i++) {
            arr[i] = arr[i + 1];
        }
        size--;
    }

    // Update element at specific position
    public void update(int pos, int newValue) {
        if (pos < 0 || pos >= size) {
            System.out.println("Invalid position!");
            return;
        }
        System.out.println("Updated element " + arr[pos] + " → " +
newValue);
        arr[pos] = newValue;
    }

    // Display array elements
    public void display() {
        if (size == 0) {
            System.out.println("Array is empty!");
            return;
        }
        System.out.print("Current Array: ");
        for (int i = 0; i < size; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}
```

Some Basic Problems of Array

1. Find the maximum and minimum element in an array.
2. Find the sum and average of array elements.
3. Count the number of even and odd elements in an array.
4. Print array elements in reverse order.
5. Insert a new element at a given position in an array.
6. Delete an element from a given position in an array.
7. Copy all elements from one array to another array.
8. Find the second largest element in an array.
9. Count duplicate elements in an array.
10. Find the total number of positive and negative elements.
11. Merge two arrays into a single array.
12. Sort an array in ascending order.
13. Count the frequency of each element in an array.
14. Find the index of a given element in an array.
15. Replace all negative numbers with zero in an array.

16. Find the difference between the largest and smallest element.
17. Find the sum of elements at even and odd positions separately.
18. Find the product of all elements in an array.
19. Find how many times a given number appears in the array.
20. Count how many elements are greater than a given number.

ArrayList in Java (Dynamic Array)

Definition

An **ArrayList** is a **resizable array** provided by the **Java Collections Framework**. Unlike normal arrays, **ArrayList size grows and shrinks automatically** when elements are added or removed.

Package:

```
import java.util.ArrayList;
```

Class Definition:

```
ArrayList<Type> list = new ArrayList<Type>();
```

Key Features

- Dynamic resizing (automatic growth and shrink)
- Allows **duplicate elements**
- Maintains **insertion order**
- Provides **random access** (like arrays)
- **Not synchronized** (not thread-safe)

Common Methods of ArrayList

Method	Description	Example
`add(element)`	Adds element at end	`list.add(10)`
`add(index, element)`	Adds element at specific index	`list.add(2, 50)`
`get(index)`	Returns element at index	`list.get(0)`
`set(index, element)`	Updates element	`list.set(1, 99)`
`remove(index)`	Removes element at index	`list.remove(2)`
`size()`	Returns number of elements	`list.size()`
`contains(element)`	Checks existence	`list.contains(10)`
`isEmpty()`	Checks if list is empty	`list.isEmpty()`

Method	Description	Example
`clear()`	Removes all elements	`list.clear()`

Example Program

```
import java.util.ArrayList;

public class Example {
    public static void main(String[] args) {
        // Create ArrayList
        ArrayList<Integer> numbers = new ArrayList<>();

        // Add elements
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);
        numbers.add(1, 15);  // insert at index 1

        // Access elements
        System.out.println("Element at index 2: " + numbers.get(2));

        // Update element
        numbers.set(2, 25);

        // Remove element
        numbers.remove(1);

        // Traverse list
        System.out.println("ArrayList elements:");
        for(int num : numbers)
            System.out.println(num);

        // Size
        System.out.println("Size: " + numbers.size());
    }
}
```

Working of ArrayList (Internal Mechanism)

- Internally uses a **dynamic array** (initial capacity = 10).
- When the array becomes full, a **new array (1.5× old size)** is created, and elements are copied over.
- Hence, adding elements occasionally takes extra time (due to resizing), but **amortized time = O(1)**.

Time Complexity of Operations

Operation	Average Time	Worst Time	Description
`add(element)`	O(1)	O(n)	Amortized constant time
`add(index, element)`	O(n)	O(n)	Need to shift elements

Operation	Average Time	Worst Time	Description
`get(index)`	O(1)	O(1)	Random access
`set(index, element)`	O(1)	O(1)	Update directly
`remove(index)`	O(n)	O(n)	Shift elements left
`contains(element)`	O(n)	O(n)	Linear search

Array vs ArrayList (Comparison)

Feature	Array	ArrayList
Size	Fixed	Dynamic (resizable)
Type	Primitive or Objects	Objects only
Performance	Faster	Slightly slower (resize overhead)
Length Property	`arr.length`	`list.size()`
Add/Delete	Manual shifting	Built-in methods
Part of	Java Language	Java Collections Framework

Conversion

Array → ArrayList

```
String[] arr = {"A", "B", "C"};
ArrayList<String> list = new ArrayList<>(Arrays.asList(arr));
```

ArrayList → Array

```
String[] newArr = list.toArray(new String[0]);
```

Reverse, Palindrome, and Anagram

Reverse a String

Problem:

Given a string, print it in reverse order.
Example: `"hello"` → `"olleh"`

Approach 1: Using Loop

```
String str = "hello";
String rev = "";
```

```
for(int i = str.length() - 1; i >= 0; i--)
    rev += str.charAt(i);
System.out.println("Reversed: " + rev);
```

Approach 2: Using StringBuilder

```
StringBuilder sb = new StringBuilder("hello");
System.out.println(sb.reverse());
```

Time Complexity: O(n)
Space Complexity: O(1) *(StringBuilder modifies in-place)*

Palindrome String

Problem:

A string is a **palindrome** if it reads the same forward and backward.
xample: `"madam"`, `"racecar"`

Approach 1: Using Two Pointers

```
String str = "madam";
boolean isPalindrome = true;

int left = 0, right = str.length() - 1;
while(left < right) {
    if(str.charAt(left) != str.charAt(right)) {
        isPalindrome = false;
        break;
    }
    left++;
    right--;
}

if(isPalindrome)
    System.out.println("Palindrome");
else
    System.out.println("Not Palindrome");
```

Approach 2: Using Reverse

```
String str = "level";
String rev = new StringBuilder(str).reverse().toString();

if(str.equals(rev))
    System.out.println("Palindrome");
else
    System.out.println("Not Palindrome");
```

Time Complexity: O(n)
Space Complexity: O(1)

Anagram Strings

Problem:

Two strings are **anagrams** if they contain the **same characters in any order**.
Example: `"listen"` and `"silent"`

Approach 1: Sort and Compare

```
import java.util.Arrays;

String s1 = "listen";
String s2 = "silent";

if(s1.length() != s2.length())
    System.out.println("Not Anagram");
else {
    char[] a1 = s1.toCharArray();
    char[] a2 = s2.toCharArray();

    Arrays.sort(a1);
    Arrays.sort(a2);

    if(Arrays.equals(a1, a2))
        System.out.println("Anagram");
    else
        System.out.println("Not Anagram");
}
```

Approach 2: Frequency Count (Optimized)

```
boolean isAnagram(String s1, String s2) {
    if(s1.length() != s2.length()) return false;

    int count[] = new int[256]; // for all ASCII characters
    for(int i = 0; i < s1.length(); i++) {
        count[s1.charAt(i)]++;
        count[s2.charAt(i)]--;
    }
    for(int c : count)
        if(c != 0) return false;

    return true;
}
```

Time Complexity: O(n)
Space Complexity: O(1) *(fixed 256 array)*

Summary Table

Problem	Description	Example	Time	Space
Reverse String	Print in reverse order	`"abc"` → `"cba"`	O(n)	O(1)
Palindrome Check	Same forward/backward	`"madam"`	O(n)	O(1)
Anagram Check	Same letters, different order	`"listen"`, `"silent"`	O(n)	O(1)