# INTRODUCTION TO DSA & ARRAYS

## PRESENTER INFO:

**Rishav Kumar**

*B.Tech. AI&DS*

*Gati Shakti Vishwavidyalaya*

https://www.linkedin.com/in/rishav-kumar-2399241ab/

## KEY TOPICS:

- DSA Basics
  - Problem-solving with data structures & algorithms
  - Real-world applications
- Arrays
  - Fast access, slow inserts
  - Operations on Array
  - Types of Array
  - Fixed vs dynamic size
- Problem Solving
- Strings
- Recursion

# REFERENCES:


1. Dr. Shweta Saharan Ma’am DSA PPT (Assistant Professor GSV)
2. Geeks for Geeks
3. Visual Algo
4. Data Structures using C (Reema Thareja)

# INTRODUCTION

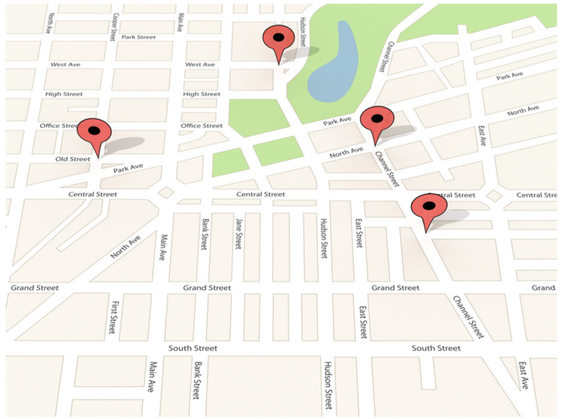
Data structures are the backbone of efficient algorithm design and real-world software development.

**A data structure is an arrangement of data either in computer's memory or on the disk storage.**

- **Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, graphs, and hash tables.**

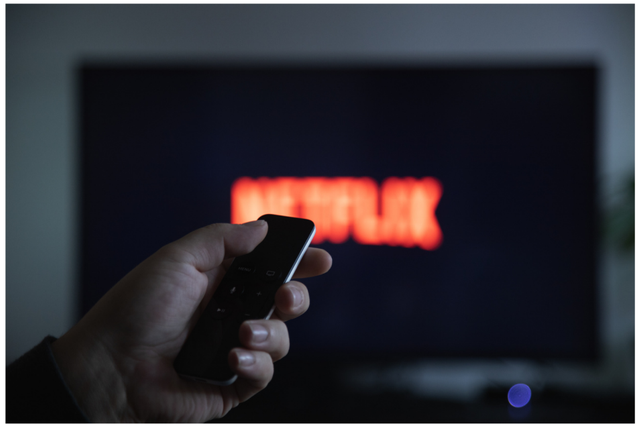
- **Data structures are widely applied in areas like:**
  1. **Compiler design**
  2. **Operating system**
  3. **Statistical analysis package**
  4. **DBMS**
  5. **Numerical analysis**
  6. **Simulation**

# DSA IN REAL-WORLD



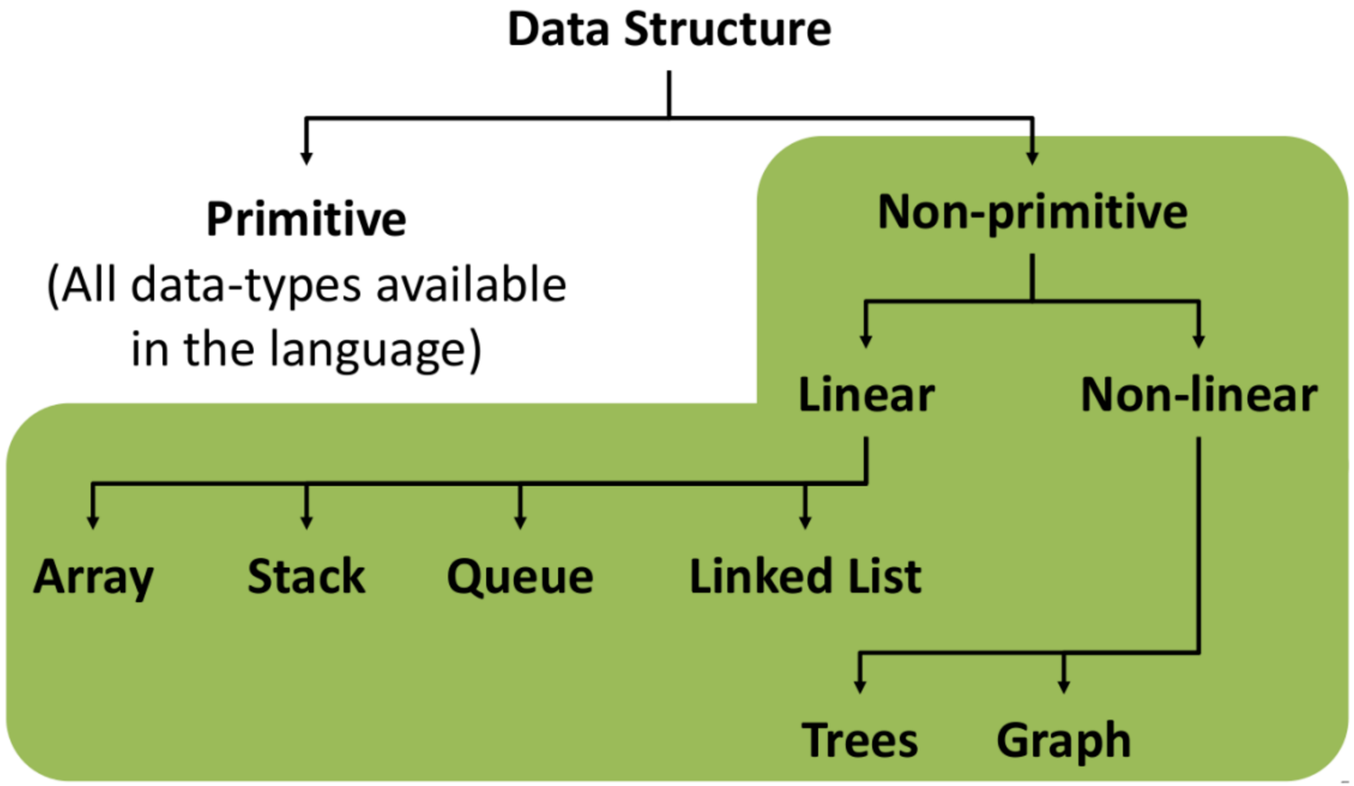

- DSA Used: Graphs

- DSA Used: Graphs (users as nodes, friendships as edges), BFS/DFS
- How:
  - "People You May Know" uses BFS to find 2nd/3rd-degree connections.
  - Newsfeed ranking uses graph traversals + ML.




- DSA Used: Matrices (user-movie ratings), Collaborative Filtering (k-nearest neighbors)
- How: Converts user preferences into a matrix, then finds similar users via dimensionality reduction (SVD).

# Classification Of DS:

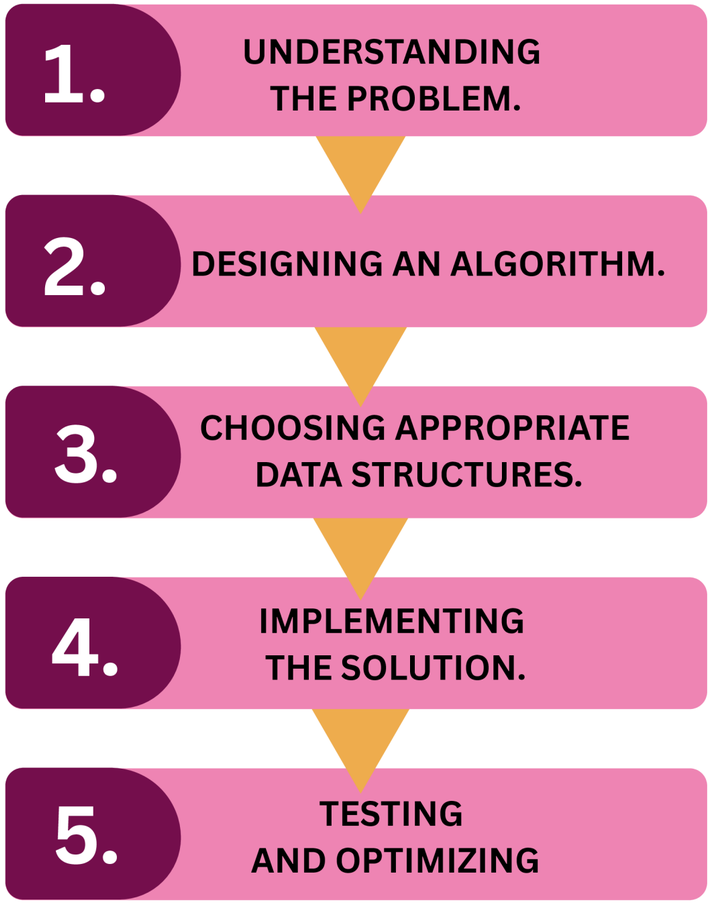



**Linear**: Each element is connected to the one before and after (except the first and last).

**Non-Linear**: In non-linear data structures, elements are arranged in a hierarchical or multi-level manner. One element can be connected to multiple others.

# THE PROBLEM-SOLVING PROCESS IN PROGRAMMING ?

# THE PROBLEM-SOLVING PROCESS IN PROGRAMMING ?



# KEY INSIGHT – THINK BEYOND CODE

- **Algorithms and Data Structures are Language-Agnostic**
  - They define logic and structure, not syntax.
  - The same algorithm can be implemented in any programming language.

- **Why it Matters:**
  - Mastering the process makes you a better problem solver across any language or platform.

# ARRAYS:

## What is an Array?

- An array is a collection of elements of the same data type stored at contiguous memory locations.
- You can access each element using an index (starting from 0).

**Syntax in C:**

```c
int arr[5] = {10, 20, 30, 40, 50};
```

**Memory Representation:**

```makefile
Index:    0    1    2    3    4
Value:  [10,  20,  30,  40,  50]
```

- Element: A single value in the array
- Index: Position number (starts from 0)
- Length: Total number of elements

# BASIC ARRAY OPERATION:

Traversal

Accessing

Insertion

Deletion

```
int arr[5] = {1,2,3,4,5}
arr[0] = 1
arr[1] =  2
arr[2] = 3
arr[3] = 4
arr[4] = 5
```

# BASIC ARRAY OPERATION:

## Traversal:

```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:      Apply Process to A[I]
Step 4:      SET  I = I + 1
        [END OF LOOP]
Step 5: EXIT
```

**Figure 3.12** Algorithm for array traversal

# BASIC ARRAY OPERATION:

## Traversal Example Problems:

1. Write a program to read and display n numbers using array.
2. Write a Program to find the mean of n numbers using arrays
3. *Write a program to find the second largest of n numbers using an array.*
4. *Write a Program to find weather the array of integers contains a duplicate number.*

# BASIC ARRAY OPERATION:

## Insertion at the Beginning of an Array

When the insertion happens at the beginning, it causes all the existing data items to shift one step downward. Here, we design and implement an algorithm to insert an element at the beginning of an array.

### Algorithm

We assume **A** is an array with **N** elements. The maximum numbers of elements it can store is defined by **MAX**. We shall first check if an array has any empty space to store any element and then we proceed with the insertion process.

```
begin

IF N = MAX, return
ELSE
   N = N + 1

   For All Elements in A
      Move to next adjacent location

   A[FIRST] = New_Element

end
```

# BASIC ARRAY OPERATION:

## Insertion at the Given Index of an Array

In this scenario, we are given the exact location (**index**) of an array where a new data element (**value**) needs to be inserted. First we shall check if the array is full, if it is not, then we shall move all data elements from that location one step downward. This will make room for a new data element.

### Algorithm

We assume **A** is an array with **N** elements. The maximum numbers of elements it can store is defined by **MAX**.

```
begin

IF N = MAX, return
ELSE
   N = N + 1

   SEEK Location index

   For All Elements from A[index] to A[N]
      Move to next adjacent location

   A[index] = New_Element

end
```

# BASIC ARRAY OPERATION:

## 🧩 Insertion at the End of an Array

In this scenario, we are inserting a new data element (**value**) at the **end** of an array. This is the most efficient case for insertion because no shifting of elements is required.
We first check whether the array is full. If it is **not full**, we simply add the new element at the end (i.e., at index `N`).

### Algorithm

We assume `A` is an array with `N` elements. The maximum number of elements it can store is defined by `MAX`.

plaintext Copy Edit

```
begin

  IF N = MAX, return    // Array is full, cannot insert
  ELSE
    A[N] = New_Element  // Insert at the end
    N = N + 1           // Increase array size

end
```

# BASIC ARRAY OPERATION:

## 1 Deletion at the Beginning of an Array

We delete the first element, which is at index `0`. All elements must be shifted left.

### Algorithm

We assume `A` is an array with `N` elements.

plaintext Copy Edit

```
begin

  IF N = 0, return    // Array is empty
  ELSE
    For i = 0 to N - 2
        A[i] = A[i + 1]  // Shift elements left

    N = N - 1           // Decrease size

end
```

# BASIC ARRAY OPERATION:

## 2 Deletion in the Middle of an Array

We delete the element at any index between `1` and `N-2`.

### Algorithm

plaintext Copy Edit

```
begin

  IF N = 0, return    // Array is empty
  ELSE
    SEEK location index

    For i = index to N - 2
        A[i] = A[i + 1]  // Shift elements left from index

    N = N - 1           // Decrease size

end
```

# BASIC ARRAY OPERATION:

## 3 Deletion at the End of an Array

The element at index `N - 1` is removed. No shifting is required.

### Algorithm

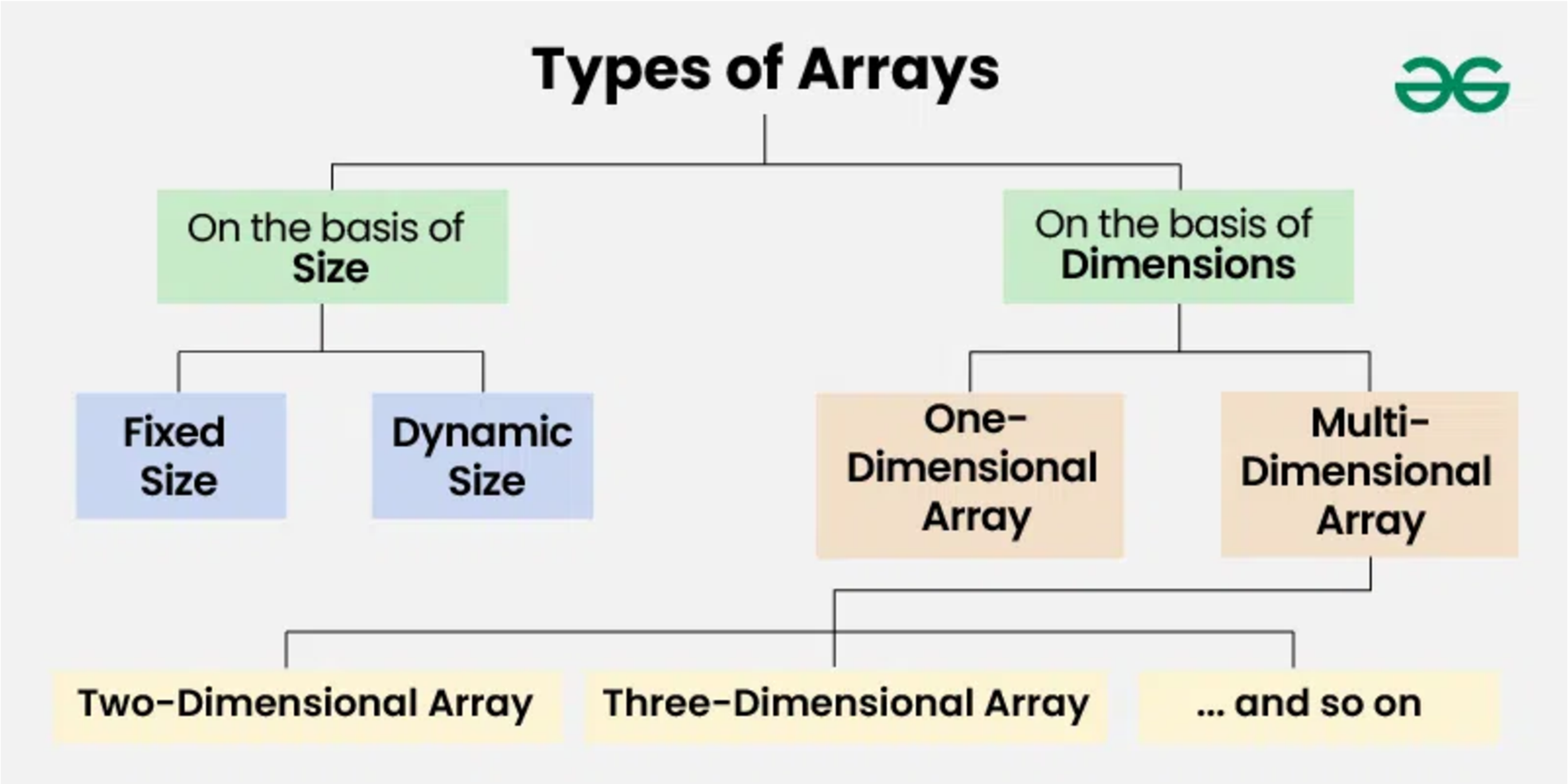
plaintext Copy Edit

```
begin

  IF N = 0, return    // Array is empty
  ELSE
    N = N - 1         // Just reduce size

end
```

# BASIC ARRAY OPERATION:

| Operation | Shifting Required? | Time Complexity |
|---|---|---|
| **Insert at Start** | Yes (right shift) | O(n) |
| **Insert in Middle** | Yes (right shift) | O(n) |
| **Insert at End** | No | O(1) |
| **Delete at Start** | Yes (left shift) | O(n) |
| **Delete in Middle** | Yes (left shift) | O(n) |
| **Delete at End** | No | O(1) |

# Types of Array:



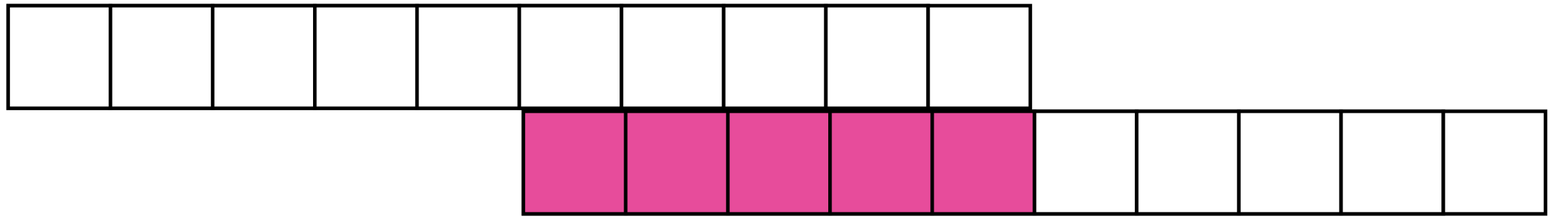

https://www.geeksforgeeks.org/types-of-arrays/

# Static & Dynamic Array:

| Feature | Static Array | Dynamic Array |
|---|---|---|
| **Size** | Fixed at compile-time | Can change at runtime (resizeable) |
| **Memory Allocation** | Allocated on stack (mostly) | Allocated on heap |
| **Flexibility** | Cannot grow or shrink | Can grow or shrink as needed |
| **Memory Efficiency** | May waste space if overestimated | More efficient for unknown data size |
| **Performance** | Faster access and manipulation | Slightly slower due to resizing overhead |
| **Example in C** | int arr[10]; | int *arr = malloc(size * sizeof(int)); |

# Let’s Solve some Problems: ...

- Finding the maximum/minimum element in an array.
- Reversing an array.
- Checking if an element exists in an array.



# What is a String in C?

- **A string in C is a 1D array of characters ending with a '\0' (null character).**
  - Example: char str[] = "Hello";

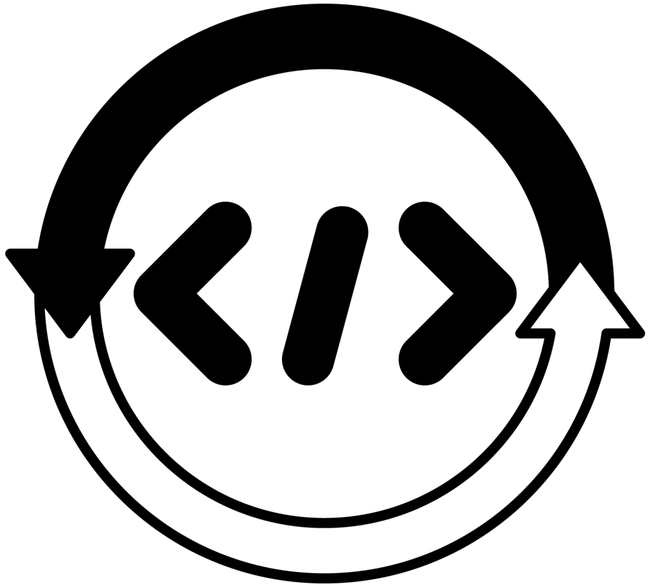

- **Stored in contiguous memory.**
- **C doesn't have a built-in string data type like other high-level languages.**

# Core String Functions in C

| Function | Purpose |
|---|---|
| **strlen(str)** | Finds length |
| **strcpy()** | Copies one string to another |
| **strcmp()** | Compares two strings |
| **strcat()** | Concatenates strings |
| **strchr()** | Finds character in string |
| **strstr()** | Finds substring |

# What is Recursion?

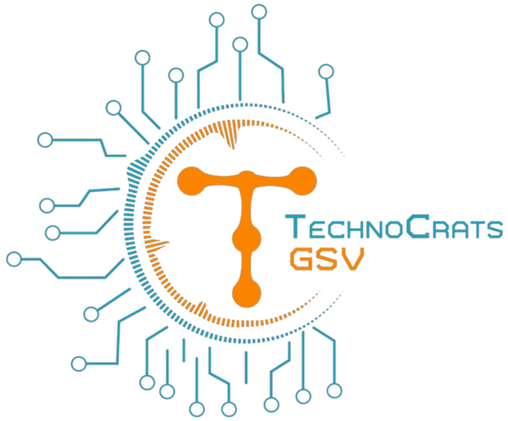
- **Recursion is when a function calls itself to solve smaller instances of a problem.**
  - Must have:
    - Base case – stops recursion
    - Recursive case – reduces the problem



**Basic Syntax:**

```
int factorial(int n) {
    if (n == 0) return 1;  // base case
    return n * factorial(n - 1);  // recursive call
}
```

- Each **recursive call** adds a **new frame** to the call **stack**.
- Stack **grows** until **base case is reached**.




# Thankyou :-)

# Feedback