**DATA STRUCTURES AND ALGORITHMS (DSA)**

# Introduction

## **Data Structures and Algorithms (DSA)** are fundamental in computer science that help us to organize and process data efficiently. They are used in solving common software challenges, from managing large data sets to Boosting the speed of tasks.

****DSA (Data Structures and Algorithms)**** helps store and organise data to make it easy to find and use whenever needed.

DSA play an important role to solve the problems. By understanding DSA, you can design systems more efficiently, which is very important for some places like web applications, databases, machine learning etc.

* **Data Structures** is about how data can be stored in different structures.
* **Algorithms** are step-by-step plans for solving problems. They form the foundation of writing a program. For writing any programs, the following has to be known: Input. Tasks to be preformed.
* **Data Structures and Algorithms (DSA)** help us to use large amounts of data to solve problems efficiently.

## How to learn DSA?

To learn DSA (Data Structures and Algorithms) step by step, break the process into small and simple parts:

1. Learn at least one programming language
2. Learn Data Structures
3. Learn Algorithms
4. Learn about Time and Space complexities
5. Practice Problems on DSA

## The topic of DSA consists of two parts:

* Data Structures
* Algorithms

### Data Structures

Data structures are the fundamental building blocks of computer programming. They define how data is organized, stored, and manipulated within a program. Understanding data structures is very important for developing efficient and effective algorithms.

In this document, we will explore the most commonly used data structures, including arrays, linked lists, stacks, queues, trees, and graphs.

* A data structure is a way to store data.
* It manages large amounts of data efficiently for uses such as large databases and internet indexing services.
* It is important for creating fast and powerful algorithms. It also reduces complexity, increases efficiency, etc…

#### Data Structure Types, Classifications, and Applications:

The data structure has many different uses in our daily life. There are many different data structures that are used to solve different mathematical and logical problems. By using a data structure, one can organize and process a very large amount of data.

* **Linear data structure:** A linear data structure stores data in a straight line, one after another. **Examples** are lists and queues. You can go through the data in order, step by step.
* **Static data structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure. An **example** of static data structure is array.
* **Dynamic data structure:**In the dynamic data structure, the size is not fixed. It can change the size while running the program. It can grow or smaller as needed, like adding or removing data. **Examples** are linked lists and dynamic arrays.
* **Non-linear data structure:** Data structures where data elements are not linearly are called non-linear data structures. Non-linear data structures do not store data in a straight line. **Examples** of non-linear data structures are trees and graphs.

#### Need for Data Structures:

We need data structures to organise and store data in a way that makes it easy to use. They help us save time and solve problems faster, especially when working with large amounts of data.

* Helps to store data in a systematic manner.
* Easy access to the large database.
* Helps to solve complex problems effectively.
* Manages big amounts of data easily.
* Saves processing time and memory space

### Algorithms

**The algorithm** is a step-by-step procedure for solving a problem. It is a set of well-defined instructions for performing a specific computer task. Good algorithms help computers solve problems **faster** and **better**. They save time, use fewer resources, and give accurate results. It provides a step-by-step procedure that converts an input into a desired output.

#### How do Algorithms Work?

* **Algorithms typically follow a logical structure:**
* **Problem**: A problem is anything from real life that needs a solution, and we can solve it by creating algorithms.
* **Input**: The algorithm receives input data.
* **Processing**: The algorithm performs a series of operations on the input data.
* **Output**: The algorithm produces the desired output.

#### Characteristics of an Algorithm:

* **Well-defined Inputs:** It may have some input valve.
* **Well-defined Outputs:** we will get 1 or more outputs at the end of algorithm.
* **Clear:** It means that the instructions in an algorithm should be clear and simple.
* **Finiteness:** It means that the algorithm should contain a limited number of instructions.
* **Effectiveness:** An algorithm should be effective as each instruction in the algorithm affects the overall process.
* **Language Independent:** Algorithm must be language-independent.

#### What is the Need for Algorithms?

Algorithms are essential for solving complex computational problems efficiently and effectively. They provide a systematic approach to:

* Solving problems: Algorithms break down problems into smaller steps.
* Perfect solutions: Algorithms find the best solutions to problems.
* Automating tasks: Algorithms can automate complex tasks, saving time and effort.

#### Examples of Algorithms

Below are some examples of algorithms:

* **Sorting algorithms:** Merge sort, Quick sort, Heap sort
* **Searching algorithms:** Linear search, Binary search, Hashing
* **Graph algorithms:** Dijkstra's algorithm, Prim's algorithm, Floyd-Warshall algorithm
* **String matching algorithms:** Knuth-Morris-Pratt algorithm, Boyer-Moore algorithm

#### How to write an algorithm?

1. **Understand the problem:** Clearly know what you need to solve.
2. **Identify inputs and outputs:** Decide what data you need (inputs) and what results you want (outputs).
3. **Break into steps:** Divide the solution into small, clear steps in the order they should happen.
4. **Write step-by-step instructions:** Use simple and clear language for each step.
5. **Check your algorithm:** Test it to make sure it works and gives the right result.

Keep it simple and easy to follow!

## Asymptotic Notations

*Asymptotic Analysis is the big idea that handles the above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don’t measure the actual running time). We calculate, the* ***order of growth*** *of time taken (or space) by an algorithm in terms of input size. For example, linear search grows linearly and Binary Search grows logarithmically in terms of input size.*

It is a mathematical tool used to analyze the performance of the algorithm.

### Type of Asymptotic Notations

1. Big-O (Worst case).
2. Omega (Best case).
3. Theta (Average case).

#### Big-O (Worst case)

**Big-O**, commonly referred to as “**Order of**”, is a way to express the **upper bound**of an algorithm’s time complexity, since it analyses the**worst-case** situation of the algorithm. It provides an**upper limit** on the time taken by an algorithm in terms of the size of the input. It’s denoted as**O(f(n))**, where**f(n)** is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size **n**.

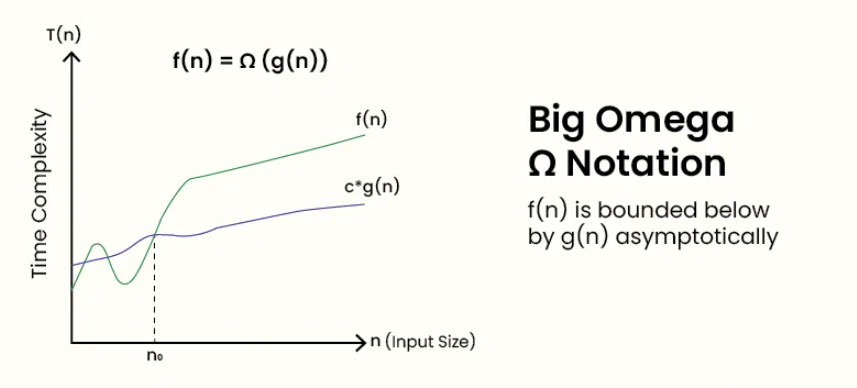
**Important Point:**

* **Big O notation** only describes the asymptotic behavior of a function, not its exact value.
* The **Big O notation** can be used to compare the efficiency of different algorithms or data structures.
* f(n)=5n2+6n+5 **:[{f(n) is a function} and any mathematical number is called constant. which are going to ignore.]**
* f(n)=n2+n+1**:Chose the largest one which is our answer.**
* f(n)=n2  **:This is the largest.**
* O(n2): **This is the answer to this equation.**

#### Omega (Best case)

Big-Omega Ω Notation is a way to express the asymptotic lower bound of an algorithm’s time complexity since it analyses the best-case situation of an algorithm. It provides a lower limit on the time taken by an algorithm in terms of the size of the input. It’s denoted as Ω(f(n)), where f(n) is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size n.

Big-Omega Ω Notation is used when we need to find the asymptotic lower bound of a function. In other words, we use Big-Omega Ω when we want to represent that the algorithm will take at least a certain amount of time or space.



##### How to Determine Big-Omega Ω Notation?

In simple language, Big-Omega Ω notation specifies the asymptotic lower bound for a function f(n). It bounds the growth of the function from below as the input grows infinitely large.

##### Steps to Determine Big-Omega Ω Notation:

1. Break the program into smaller segments:

* Break the algorithm into smaller segments such that each segment has a certain runtime complexity.

1. Find the complexity of each segment:

* Find the number of operations performed for each segment(in terms of the input size) assuming the given input is such that the program takes the least amount of time.

1. Add the complexity of all segments:

* Add up all the operations and simplify it, let’s say it is f(n).

1. Remove all the constants:

* Remove all the constants and choose the term having the least order or any other function that is always less than f(n) when n tends to infinity.

Let’s say the least order function is g(n) then, Big-Omega (Ω) of f(n) is Ω(g(n)).

#### Theta (Average case)

In the analysis of algorithms, asymptotic notations are used to evaluate the performance of an algorithm by providing an exact order of growth.

It provides the tight bound means the upper and lower bound of the running time algorithm.

##### Follow the steps below to find the average time complexity of any program:

1. Break the program into smaller segments.
2. Find all types and numbers of inputs and calculate the number of operations they take to be executed. Make sure that the input cases are equally distributed.
3. Find the sum of all the calculated values and divide the sum by the total number of inputs let say the function of n obtained is g(n) after removing all the constants, then in Θ notation it is represented as Θ(g(n))

## Learn about Complexities

In data structure, complexity means how much time and space a program needs to run, with respect to input size.

* It is used for comparing different algorithms on different input sizes.
* Complexity helps to determine the difficulty of a problem.
* often measured by how much time and space (memory) it takes to solve a particular problem.

### How to measure complexity?

There are two such methods used, time complexity and space complexity which are discussed below:

1. **Time Complexity**: How fast a program runs (time it takes).
2. **Space Complexity**: How much memory a program uses.

#### Time Complexity

Not the actual time taken but the amount of time taken as a function of input size (n).

**Definition–**

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve a given problem is called the time complexity of the algorithm. Time complexity is a very useful measure in algorithm analysis.

* Y=X
* F(x)=x
* F(n)=n
* [O(n)] -Big of n OR Big O of n
* f(n)=5 -O(5)

**Example 1: Linear search in array.**

Int arr[];

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

N-number of input **Here** n =number of input means if n=10 then the number of operations is also 10 if n=100 number of operations will also 100.

X- number of input valve

For(i=0; i<n; i++){

If(arr[i]==target);

Return i;

}

Return False;

**Example 2: Addition of two variables.**

Algorithm ADD variable (A, B)

* **Description:** Perform arithmetic addition of two numbers
* **Input:** Two variables A and B
* **Output:** variable C, which holds the addition of A and B
* C = A + B
* return C

The addition of two scalar numbers requires one addition operation. The time complexity of this algorithm is constant, so T(n) = O(1).

X- Number of input value

#### Space complexity

Problem-solving using a computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve a given problem is called the space complexity of the algorithm.

This refers to how much space or amount of memory an algorithm needs in terms of input size.

Amount of space taken by an algorithm as a function of input size (n).

**Example 1:Linear space complexity.** Take an input in an array[]. Return output in the form of a square of input value in the array[].

**Int arr[];**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

**N-number of input Note:**

* Input ->arr[].
* Output ->Square [].
* Input does not count as a space complexity.
* Only auxiliary space means temporary space will count.

Int Square[];

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 4 | 9 | 16 | 25 | 36 | 49 | 64 | 81 | 100 |

**n-number of square**

**Note:**

* **Arr[]** is an original array it is also an input value that is not counted by space complexity.
* **Square[]** is an auxiliary space (extra) that is counted by space complexity.
* **Arr[] and square[]** contain the same size of space, so whenever arr[] size is increase then also square[] size is increase.

X- input value

Linear space complexity O(n)

**Example 2: Constant space complexity.**

* Input->arr[]
* Out->sum

int sun=0;

For(i=0; i<n; i++){

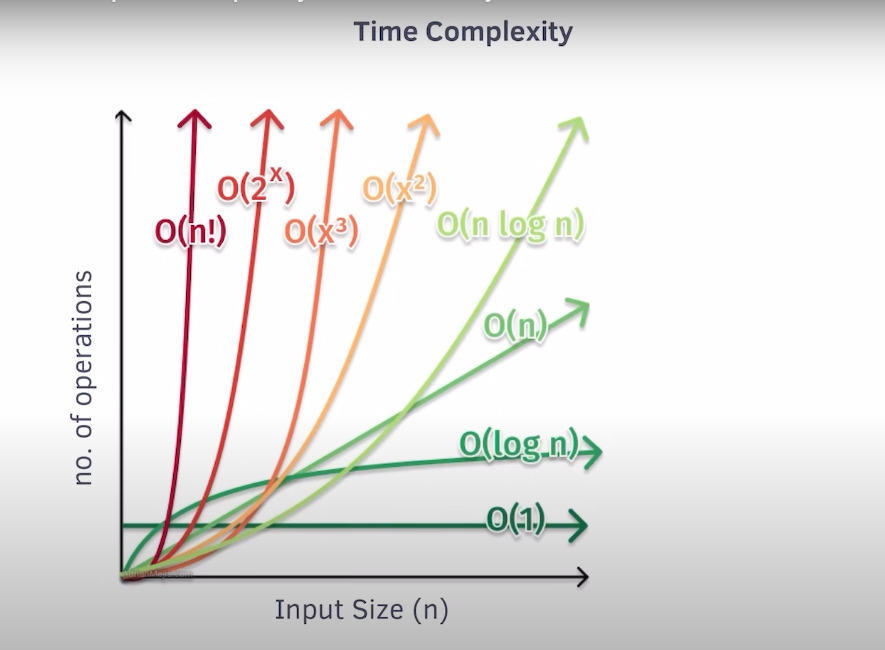
Sun+=arr[i];

}

Printf(sum);

* Hear n=number of input mean. If n =10 then the number of operations. if n=100 then the number of operations will also 100.Y=X
* F(x)=X
* F(n)=n.
* [O(n)]
* F(n)=5. O(5)

##### Some common time complexity.

****

Best time complexity

Pending………………………………………………………………………………………………

# Mathematics.

##### Mathematical Algorithms

For problem-solving, we need to know the basics of Numbers, LCM, GCD, Factorial, Permutation, and Combinations.

1. Write a program in c-language to count the digits.

#include <stdio.h> // Standard input-output header

#include <conio.h>

int main() // Standard main function with return type int

{

    int n = 12367843; // Initialize the number to be processed

    int count = 0;    // Variable to store the count of digits

    // Check if the number is not zero

    if (n != 0)

    {

        // Loop to count the digits

        do

        {

            count++;    // Increment the digit count

            n = n / 10; // Remove the last digit of the number

        } while (n > 0); // Repeat until the number becomes 0

    }

    // Print the count of digits

    printf("The number of digits is: %d\n", count);

    getch(); // Return 0 to indicate successful execution

}

1. write a program in c language to check number is palindrome or not.

#include <stdio.h> // Include standard input-output library

int main()

{

    int n = 123;           // Input number

    int original = n;      // Store the original number to compare later

    int rev = 0;           // Variable to store the reversed number

    int digit;             // Temporary variable to hold individual digits

    // Reverse the number

    while (n != 0)

    {

        digit = n % 10;       // Extract the last digit

        rev = rev \* 10 + digit; // Append the digit to the reversed number

        n = n / 10;           // Remove the last digit from the original number

    }

    // Check if the original number is equal to the reversed number

    if (original == rev)

    {

        printf("%d is a palindrome.\n", original);

    }

    else

    {

        printf("%d is not a palindrome.\n", original);

    }

    return 0; // Return 0 to indicate successful execution

}

1. Write a program in c language to find a factorial number.

#include <stdio.h> // Standard input-output library

int main()

{

    int num, fact = 1; // Initialize fact to 1

    printf("Enter the number: "); // Prompt the user for input

    scanf("%d", &num); // Corrected scanf to include & (address-of operator)

    // Validate input (optional, to ensure non-negative input)

    if (num < 0)

    {

        printf("Factorial is not defined for negative numbers.\n");

        return 0;

    }

    // Calculate factorial using a loop

    for (int i = num; i > 0; i--)

    {

        fact = fact \* i; // Multiply fact with the current value of i

    }

    // Print the result

    printf("The factorial of %d is: %d\n", num, fact);

    return 0; // Return 0 to indicate successful execution

}

# Recursion

**Recursion**is technique used in computer science to solve big problems by breaking them into smaller, similar problems. The process in which a function calls itself directly or indirectly is called **recursion**and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily.

In other word we can say that one complex problem solution depends open same type of one simple problem.

## What is a Recursive Algorithm?

A recursive algorithm is an algorithm that uses recursion to solve a problem. Recursive algorithms typically have two parts:

1. **Base case:** Which is a condition that stops the recursion.
2. **Recursive case:** Which is a call to the function itself with a smaller version of the problem.

### Need of Recursion

Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. A task that can be defined with its similar subtask, recursion is one of the best solutions for it

### Properties of Recursion:

* Performing the same operations multiple times with different inputs.
* In every step, we try smaller inputs to make the problem smaller.
* Base condition is needed to stop the recursion otherwise infinite loop will occur.

### Algorithm: Steps

1. Define a base case: Identify the simplest case for which the solution is known or trivial. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.
2. Define a recursive case: Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.
3. Ensure the recursion terminates: Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.
4. Combine the solutions: Combine the solutions of the subproblems to solve the original problem.

### How are recursive functions stored in memory?

Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished. The recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure.

### What is the base condition in recursion?

In the recursive program, the solution to the base case is provided and the solution to the bigger problem is expressed in terms of smaller problems.

#include <stdio.h>

#include <conio.h>

// Recursive function to calculate factorial

int fact(int n)

{

    // Base case: If n is 0 or 1, return 1

    if (n == 0 || n == 1)

    {

        return 1;

    }

    // Recursive case: Multiply n by the factorial of (n-1)

    return n \* fact(n - 1);

}

void main()

{

    int num = 5; // Input number for which factorial is to be calculated

    // Print the factorial result

    printf("The factorial of %d is: %d", num, fact(num));

    getch(); // Wait for a key press before exiting

}

Why Stack Overflow error occurs in recursion?   
If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

#include <stdio.h>

#include <conio.h>

int fact(int n)

{

    // wrong base case (it may cause stack overflow).

    if (n == 10)

    {

        return 1;

    }

    return n \* fact(n - 1);

}

void main()

{

    int num = 5;

    printf("The the factorial of %d is: %d", num, fact(num));

    getch();

}

If fact(5) is called, it will call fact(4), fact(3), fact(2), and so on but the number will never reach 10. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

## Type of Recursion

1. Direct recursion.
2. Indirect recursion.

### **Direct recursion**

Direct recursion happens when a function calls itself directly.

// It is an example of direct recursion.

#include <stdio.h>

#include <conio.h>

// Recursive function to calculate factorial

int fact(int n)

{

    // Base case: If n is 0 or 1, return 1

    if (n == 0 || n == 1)

    {

        return 1;

    }

    // Recursive case: Multiply n by the factorial of (n-1)

    return n \* fact(n - 1);

}

void main()

{

    int num = 5; // Input number for which factorial is to be calculated

    // Print the factorial result

    printf("The factorial of %d is: %d", num, fact(num));

    getch(); // Wait for a key press before exiting

}

**In this example:**

* The fact function calls itself to calculate the factorial of a number.
* This is **direct recursion** because the function directly uses its own name in the code.

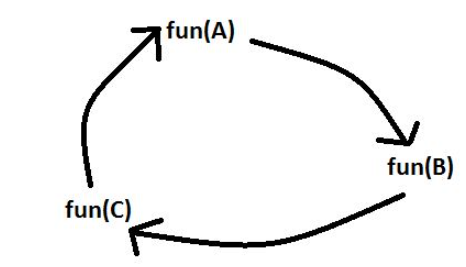
**Key Points:**

* Direct recursion has a **base case** (stopping condition) to prevent infinite calls.
* It's commonly used in problems like factorial, Fibonacci series, or solving mathematical equations.

### **Indirect Recursion**

In this type of recursion, there can be more than one function, and they call each other in a circular manner.

A function fun(A) is called indirect recursive if it calls another function fun(B) and fun(B) called fun(C) then fun(C) call fun(A) is this function is called in circular motion.



#include <stdio.h>

#include <conio.h>

// Function to calculate factorial of a number

int factorial(int n) {

    // Base case: If n is 0 or negative, return 1

    if (n <= 0) {

        return 1;

    } else {

        // Recursive call: n \* factorial of (n - 1)

        return n \* factorial(n - 1);

    }

}

void main() {

    int num = 5;

    // Print the factorial of the given number

    printf("The factorial of %d is: %d", num, factorial(num));

    getch(); // Wait for key press

}

### Tail recursion

If a recursive function calling itself and that recursive call is the last statement in the function then it’s known as **Tail Recursion.** After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling and it does nothing at returning time.

**Example:**

#include <stdio.h>

#include <conio.h>

// Tail recursive function to print numbers from n to 1

void fun(int n)

{

    if (n <= 0)

    { // Base case: stop when n is 0 or negative

        return;

    }

    else

    {

        printf("%d ", n); // Print the current value of n

        fun(n - 1);       // Tail recursive call: last operation in the function

    }

}

void main()

{

    int num = 10; // Starting number

    printf("Numbers from %d to 1: ", num);

    fun(num); // Call the tail-recursive function

    getch();

// The function prints numbers and runs O(n) times

}

### Head Recursion

If a recursive function calling itself and that recursive call is the first statement in the function then it’s known as Head Recursion. There’s no statement, no operation before the call. The function doesn’t have to process or perform any operation at the time of calling and all operations are done at returning time.

**Example:**

#include <stdio.h>

#include <conio.h>

// Function to perform recursion and print numbers

void numb(int n) {

    if (n > 0) { // Base condition: Stop recursion when n <= 0

        numb(n - 1); // Recursive call with n decremented

        printf("%d ", n); // Print the value of n after the recursive call

    }

}

void main() {

    int num = 5; // Initialize the number to be printed

    numb(num);   // Call the recursive function

    getch();     // Wait for user input before closing the program

}

## Applications of Recursion:

Recursion is used in many fields of computer science and mathematics, which includes:

* Searching and sorting algorithms: Recursive algorithms are used to search and sort data structures like trees and graphs.
* Mathematical calculations: Recursive algorithms are used to solve problems such as factorial, Fibonacci sequence, etc.
* Compiler design: Recursion is used in the design of compilers to parse and analyze programming languages.
* Graphics: many computer graphics algorithms, such as fractals and the Mandelbrot set, use recursion to generate complex patterns.
* Artificial intelligence: recursive neural networks are used in natural language processing, computer vision, and other AI applications.

## Writing base cases in Recursion.

#include <stdio.h> // For printf

#include <conio.h>

// Function to print the Fibonacci series up to `num` terms

void fib(int num, int a, int b, int count)

{

    if (count >= num)

    {

        // Termination condition: Stop when `count` reaches `num`

        return;

    }

    printf("%d ", a); // Print the current Fibonacci number

    // Recursive call to calculate the next Fibonacci number

    fib(num, b, a + b, count + 1);

}

void main()

{

    int num = 15;  // The number of Fibonacci terms to print

    int a = 0;     // The first Fibonacci number

    int b = 1;     // The second Fibonacci number

    int count = 0; // Counter to track the number of terms printed

    // Start the Fibonacci sequence

    fib(num, a, b, count);

    getch(); // Wait for a key press before exiting (useful for Windows)

}

## Various Problems on Recursion.

* 1. **Print n to 1 (Tail Recursion).**

#include <stdio.h> // Standard Input/Output library for printf

// Recursive function to print numbers from `n` to 1

void numb(int n)

{

    if (n <= 0)

    {

        // Base case: Stop recursion when `n` is less than or equal to 0.

        return;

    }

    else

    {

        printf("%d ", n);

        // Print the current value of `n`.

        numb(n - 1);

        // Recursive call: Continue recursion with `n-1` to print the next number in descending order.

    }

}

void main()

{

    int num = 5;

    // Initialize the starting number from which the numbers will be printed.

    numb(num);

    // Start the recursive process to print numbers from `num` down to 1.

    printf("\n");

    // Add a newline after the output for better formatting.

}

**The output of the program is as follows:**

* **5 4 3 2 1** 
  1. **Print 1 to n.**

#include <stdio.h> // Standard Input/Output library for printf

// Recursive function to print numbers from 1 to `n`

void numb(int n) {

    if (n <= 0) {

        // Base case: Stop recursion when `n` is less than or equal to 0.

        return;

    } else {

        numb(n - 1);

        // Recursive call: Calls the function with `n-1`.

        printf("%d ", n);

        // Print the current value of `n` after the recursion.

    }

}

int main() {

    int num = 5;

    // Initialize the number up to which the numbers will be printed.

    numb(num);

    // Start the recursive process.

    printf("\n");

    // Add a newline after the output for better formatting.

    return 0;

    // Return 0 to indicate successful program execution.

}

**The output of the program is as follows:**

* **1 2 3 4 5**

1. **Checking Palindrome.**

#include <conio.h>

#include <stdio.h>

// Function to reverse a number using recursion

int pail(int num, int result)

{

    if (num == 0) {

        // Base case: when all digits are processed, return the reversed number

        return result;

    } else {

        // Extract the last digit and build the reversed number

        return pail(num / 10, result \* 10 + (num % 10));

    }

}

int main()

{

    int num = 123;                 // Input number

    int reversed = pail(num, 0);   // Call recursive function to reverse the number

    // Check if the reversed number equals the original number

    if (reversed == num) {

        printf("It is a palindrome: %d", num);

    } else {

        printf("It is not a palindrome: %d", num);

    }

    getch(); // Wait for a key press

    return 0; // Indicate successful program termination

}

**The output of the program is as follows:**

* **It is not a palindrome: 123**

# Array in data structure

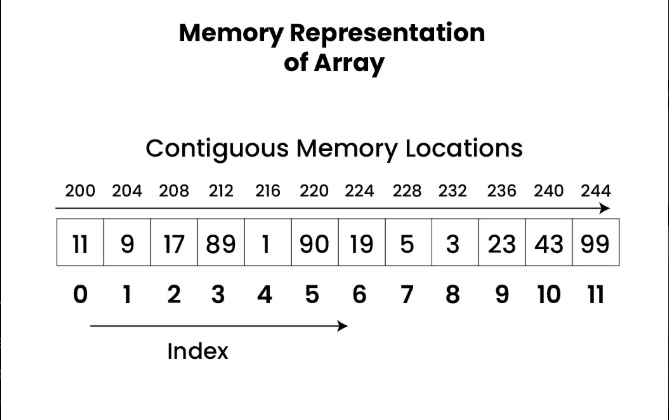
### Array is a collection of items of the same variable type that are stored at contiguous memory locations. It is one of the most popular and simple data structures used in programming.

### Basic terminologies of Array

* **Array Index:** In an array, elements are identified by their indexes. Array index starts from 0.
* **Array element:**Elements are items stored in an array and can be accessed by their index.
* **Array Length:** The length of an array is determined by the number of elements it can contain.

### Memory representation of Array

In an array, all the elements are stored in contiguous memory locations. So, if we initialize an array, the elements will be allocated sequentially in memory. This allows for efficient access and manipulation of elements.



### Declaration of Array

Arrays can be declared in various ways in different languages. In c-language we declare like this:

// This array will store integer type element

int arr[5];

// This array will store char type element

char arr[10];

// This array will store float type element

float arr[20];

### Initialization of Array

Arrays can be initialized in different ways in different languages. Below are some language-specific array initializations:

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

char arr[5] = { 'a', 'b', 'c', 'd', 'e' };

float arr[10] = { 1.4, 2.0, 24, 5.0, 0.0 };

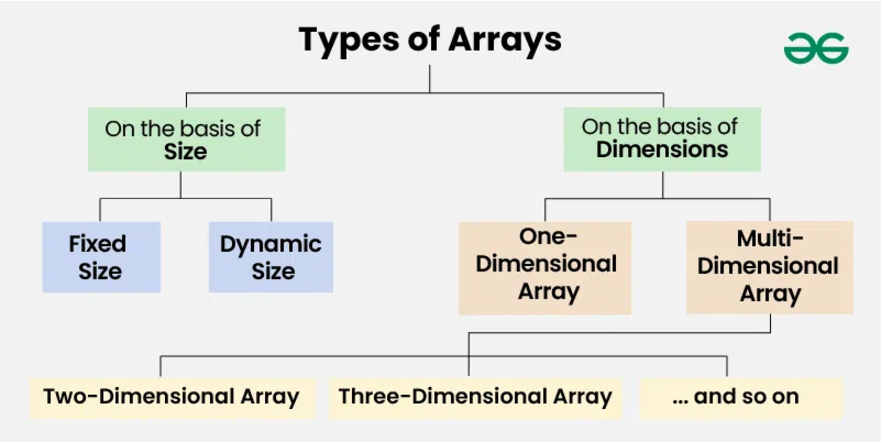
### Need or Applications of Array

* Data Structures Array is a fundamental data structure and many other data structure are implemented using this. Implementing data structures such as stacks and queues.
* Representing data in tables and matrices.
* Creating dynamic data structures such as Hash Tables and Graph.
* When compared to other data structures, arrays have the advantages like random access (we can quickly access i-th item) and cache friendliness (all items are stored at contiguous location).

## Types of Arrays

Arrays can be classified in two ways:

* On the basis of Size.
* On the basis of Dimensions.



## Types of Arrays on the basis of Size:

### Fixed Sized Arrays:

We cannot alter or update the size of this array. Here only a fixed size (i,e. the size that is mentioned in square brackets []) of memory will be allocated for storage.

// Method 1 to create a fixed sized array.

// Here the memory is allocated at compile time.

int arr1[5];

// Another way (creation and initialization both)

int arr2[5] = {1, 2, 3, 4, 5};

// Method 2 to create a fixed sized array

// Here memory is allocated at run time (Also

// known as dynamically allocated arrays)

int \*arr = (int\*)malloc(n \* sizeof(int));

### Dynamic Sized Arrays:

The size of the array changes as per user requirements during execution of code so the coders do not have to worry about sizes. They can add and removed the elements as per the need. The memory is mostly dynamically allocated and de-allocated in these arrays.

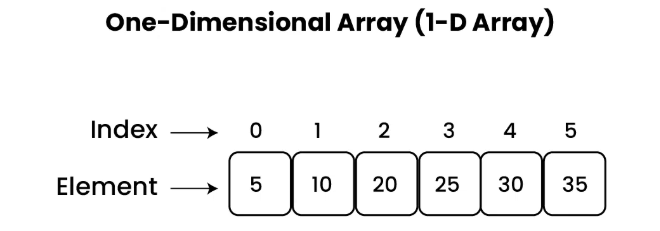
// C does not seem to support

// dynamic sized arrays as of now

## Types of Arrays on the basis of Dimensions:

### One-dimensional Array(1-D Array):

You can imagine a 1D array as a row, where elements are stored one after another.

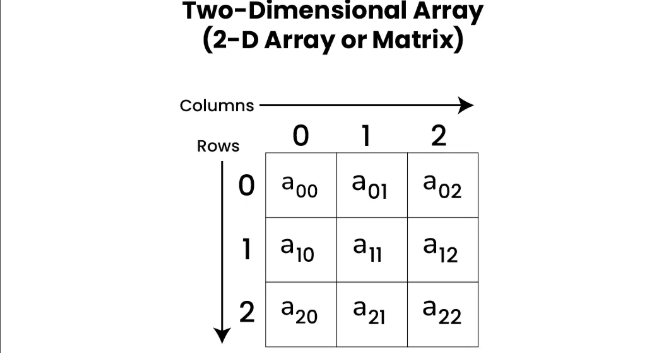


### Multi-dimensional Array:

This array are used to store complex data in the form of a row and column or table. We can have 2-D arrays, 3-D arrays, 4-D arrays, and so on.

* **Two dimensional array(2-D Array or Matrix):**

In this array, data can be stored in the form of a matrix consisting of rows and columns.



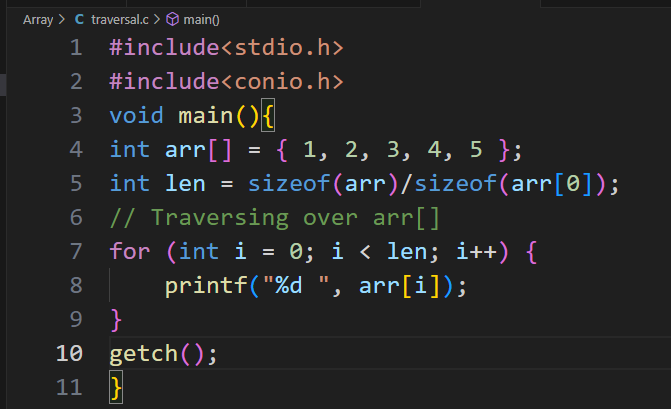
* **Three-Dimensional Array(3-D Array):**

A 3-D Multidimensional array contains three dimensions, so it can be considered an array of two-dimensional arrays.

## Operations on Array

### **Array Traversal**

Array traversal involves visiting all the elements of the array once. Below is the implementation of Array traversal in different Languages:



**Output: 1 2 3 4 5**

### **Insertion in Array**

We can insert one or multiple elements at any position in the array. Below is the implementation of Insertion in Array in different languages:

#include <stdio.h>

#include <conio.h>

void main() {

    int arr[100] = {1, 2, 3, 45, 56, 7, 6, 7, 8, 89, 89, 7};  // Initialized array

    int count = 0, pos, element;

    char dis;

    // Find the capacity of the array

    int len = sizeof(arr) / sizeof(arr[0]);

    printf("The capacity of the array: %d\n", len);

    // Count the number of initialized elements

    for (int i = 0; i < len; i++) {

        if (arr[i] != 0) {

            count++;

        }

    }

    printf("The total number of elements in the array is: %d\n", count);

    // Find the free space in the array

    printf("Total number of free spaces in the array: %d\n\n", len - count);

    printf("How can I help you? If you want to insert an element, press Y or N: ");

    scanf(" %c", &dis);  // Leading space to handle leftover newline

    if (dis == 'Y' || dis == 'y') {

        printf("Which position do you want to insert the element? (0 to %d): ", count);

        scanf("%d", &pos);

        if (pos < 0 || pos > count) {

            printf("Invalid position! Please enter a position between 0 and %d.\n", count);

        } else {

            printf("Enter the element to insert in the array: ");

            scanf("%d", &element);

            // Shift elements to the right if needed

            for (int i = count - 1; i >= pos; i--) {

                arr[i + 1] = arr[i];

            }

            // Insert the element at the desired position

            arr[pos] = element;

            count++;

            // Determine time complexity

            if (pos == count - 1) {

                printf("This is a best-case O(1) time complexity.\n");

            } else {

                printf("This is a worst-case O(n) time complexity.\n");

            }

        }

    }

    // Traverse and display the updated array

    printf("\nArray after insertion: ");

    for (int i = 0; i < count; i++) {

        printf("%d ", arr[i]);

    }

    printf("\nThank you for using the program!");

    getch();

}

### **Deletion in Array**

We can delete an element at any index in an array. Below is the implementation of Deletion of element in an array:

1. **Deleting by element.**

#include <stdio.h>

#include <conio.h>

void main()

{

    int arr[100] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 8, 1};

    int pos, element, count = 0;

    char dis, con;

    // Find the capacity of the array.

    int len = sizeof(arr) / sizeof(arr[0]);

    printf("The capacity of array is: %d.\n", len);

    // Count the number of elements in the array.

    for (int i = 0; i < len; i++)

    {

        if (arr[i] > 0)

        {

            count++;

        }

    }

    printf("The total number of elements in the array is: %d.\n", count);

    // Find the free space in the array.

    printf("Total number of free spaces in the array: %d.\n\n", len - count);

    // Perform delete operation.

    printf("Tell me! How can I help you.\n");

    printf("If you want to delete an element, press Y or N: ");

    scanf(" %c", &dis); // Space before %c to clear buffer

    printf("\n");

    if (dis == 'y' || dis == 'Y')

    {

        // Delete by element

        printf("Enter the element to delete: ");

        scanf("%d", &element);

        int found = 0;

        for (int i = 0; i < count; i++)

        {

            if (arr[i] == element)

            {

                found = 1;

            }

            if (found && i < count - 1)

            {

                arr[i] = arr[i + 1];

            }

        }

        if (found)

        {

            count--;

        }

        else

        {

            printf("This element was not found in the array.\n");

        }

    }

    else if (dis == 'n' || dis == 'N')

    {

        printf("Thank you for using the program!\n");

    }

    else

    {

        printf("Invalid choice!\n");

    }

    // Traverse the array

    printf("Array after operation: ");

    for (int i = 0; i < count; i++)

    {

        printf("%d ", arr[i]);

    }

    getch();

}

1. **Deleting by position.**

#include <stdio.h>

#include <conio.h>

void main()

{

    int arr[100] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 8, 1};

    int pos, element, count = 0;

    char dis, con;

    // Find the capacity of the array.

    int len = sizeof(arr) / sizeof(arr[0]);

    printf("The capacity of array is: %d.\n", len);

    // Count the number of elements in the array.

    for (int i = 0; i < len; i++)

    {

        if (arr[i] > 0)

        {

            count++;

        }

    }

    printf("The total number of elements in the array is: %d.\n", count);

    // Find the free space in the array.

    printf("Total number of free spaces in the array: %d.\n\n", len - count);

    // Perform delete operation.

    printf("Tell me! How can I help you.\n");

    printf("If you want to delete an element, press Y or N: ");

    scanf(" %c", &dis); // Space before %c to clear buffer

    printf("\n");

    if (dis == 'y' || dis == 'Y')

    {

        // Delete by position

        printf("Enter the position of the element: ");

        scanf("%d", &pos);

        if (pos >= count || pos < 0)

        {

            printf("This position is not valid!\n");

        }

        else

        {

            for (int i = pos; i < count - 1; i++)

            {

                arr[i] = arr[i + 1];

            }

            count--;

        }

    }

    else if (dis == 'n' || dis == 'N')

    {

        printf("Thank you for using the program!\n");

    }

    else

    {

        printf("Invalid choice!\n");

    }

    // Traverse the array

    printf("Array after operation: ");

    for (int i = 0; i < count; i++)

    {

        printf("%d ", arr[i]);

    }

    getch();

}

### **Searching in Array**

We can traverse over an array and search for an element. Below is the implementation of Searching of element in an array:

#include <stdio.h>

void main()

{

    // Static array (not dynamic)

    int arr[] = {1, 2, 4, 56, 7, 78, 98, 6, 56};

    int element, found = 0;

    // Find the length of the array

    int count = sizeof(arr) / sizeof(arr[0]);

    printf("Length of the array: %d\n", count);

    // Handle edge case: empty array

    if (count == 0)

    {

        printf("The array is empty. Nothing to search.\n");

        return;

    }

    // Search operation

    printf("Enter the element in the array you want to search: ");

    scanf("%d", &element);

    printf("\n");

    for (int i = 0; i < count; i++)

    {

        if (arr[i] == element)

        {

            printf("Element found at index %d: %d\n", i, arr[i]);

            found = 1;

        }

    }

    // If element is not found

    if (!found)

    {

        printf("The element was not found in the array!\n");

    }

    printf("Thank you for using the program.\n");

    getch();

}

## Advantages of Array

* Arrays allow random access to elements. This makes accessing elements by position faster.
* Arrays have better cache locality which makes a pretty big difference in performance.
* Arrays represent multiple data items of the same type using a single name.
* Arrays are used to implement the other data structures like linked lists, stacks, queues, trees, graphs, etc.

## Disadvantages of Array

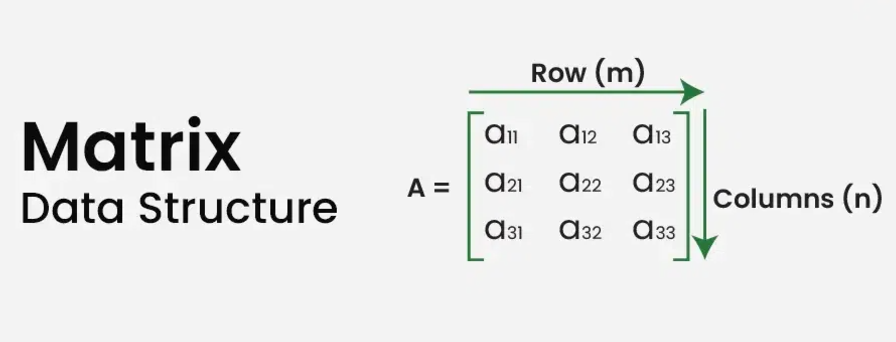
* As arrays have a fixed size, once the memory is allocated to them, it cannot be increased or decreased, making it impossible to store extra data if required. An array of fixed size is referred to as a static array.
* Allocating less memory than required to an array leads to loss of data.
* An array is homogeneous in nature so, a single array cannot store values of different data types.
* Arrays store data in contiguous memory locations, which makes deletion and insertion very difficult to implement. This problem is overcome by implementing linked lists, which allow elements to be accessed sequentially.

## Applications of Array

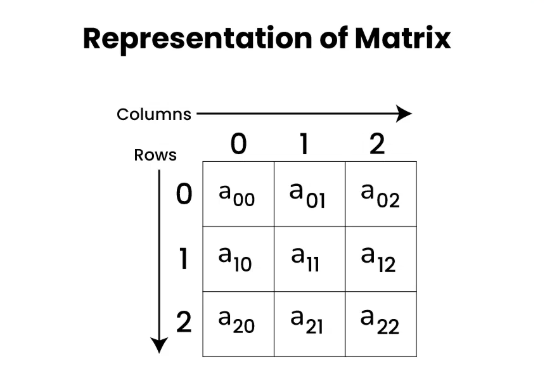
* They are used in the implementation of other data structures such as array lists, heaps, hash tables, vectors, and matrices.
* Database records are usually implemented as arrays.
* It is used in lookup tables by computer.

**2. Matrix/Grid/2D Array.**

**Matrix Data Structure** is a two-dimensional array arranged in rows and columns. It is commonly used to represent mathematical matrices and is fundamental in various fields like mathematics, computer graphics, and data processing. Matrices allow for efficient storage and manipulation of data in a structured format.

****

**Representation of Matrix Data Structure:** The elements are organized in rows and columns. As shown in the above image the cell a[0][0] is the first element of the first row and first column. The value in the first square bracket represents the row number and the value inside the second square bracket represents the column number. (i.e, a[row][column]).

****

**Declaration of Matrix Data Structure :**

Declaration of a Matrix or two-dimensional array is very much similar to that of a one-dimensional array, given as follows.

#include <stdio.h>

#include <conio.h>

int main()

{

    // Defining number of rows and columns in matrix

    int rows = 3, cols = 3;

    // Array Declaration

    int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

    // Print elements of the array

    for (int i = 0; i < rows; i++)

    {

        for (int j = 0; j < cols; j++)

        {

            printf("%d ", arr[i][j]);

        }

        printf("\n");

    }

    return 0;

}

**Initializing Matrix Data Structure:**

In initialization, we assign some initial value to all the cells of the matrix. Below is the implementation to initialize a matrix in different languages.

#include <stdio.h>

int main() {

    // Initializing a 2-D array with values

    int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

    return 0;

}

**Operations on Matrix Data Structure:**

We can perform a variety of operations on the Matrix Data Structure. Some of the most common operations are:

* Access elements of Matrix
* Traversal of a Matrix
* Searching in a Matrix
* Sorting a Matrix

1. **Access elements of Matrix Data Structure:**

Like one-dimensional arrays, matrices can be accessed randomly by using their indices to access the individual elements. A cell has two indices, one for its row number, and the other for its column number. We can use arr[i][j] to access the element which is at the ith row and jth column of the matrix.

#include <stdio.h>

int main()

{

    // Initializing a 2-D array with values

    int arr[3][3]

        = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

    // Accessing elements of 2-D array

    printf("First element of first row: %d\n", arr[0][0]);

    printf("Third element of second row: %d\n", arr[1][2]);

    printf("Second element of third row: %d\n", arr[2][1]);

    return 0;

}

1. **Traversal of a Matrix Data Structure:**

We can traverse all the elements of a matrix or two-dimensional array by using two for-loops.

#include <stdio.h>

int main()

{

    int arr[3][4] = { { 1, 2, 3, 4 },

                      { 5, 6, 7, 8 },

                      { 9, 10, 11, 12 } };

    // Traversing over all the rows

    for (int i = 0; i < 3; i++) {

        // Traversing over all the columns of each row

        for (int j = 0; j < 4; j++) {

            printf("%d ", arr[i][j]);

        }

        printf("\n");

    }

    return 0;

}

1. **Searching in a Matrix Data Structure:**

We can search an element in a matrix by traversing all the elements of the matrix.

Below is the implementation to search an element in a matrix:

#include <stdio.h>

int main()

{

    int arr[3][4] = {{1, 2, 3, 4},

                     {5, 6, 7, 8},

                     {9, 10, 11, 12}};

    int element, found = 0;

    // Get the dimensions of the array

    int rows = sizeof(arr) / sizeof(arr[0]);

    int cols = sizeof(arr[0]) / sizeof(arr[0][0]);

    // Prompt the user to input the element to search

    printf("Enter the element you want to search: ");

    scanf("%d", &element);

    printf("\n");

    // Search the element in the array

    for (int i = 0; i < rows; i++)

    {

        for (int j = 0; j < cols; j++)

        {

            if (arr[i][j] == element)

            {

                printf("Element %d found at position [%d][%d]\n", element, i, j);

                found = 1;

            }

        }

    }

    // If the element was not found

    if (!found)

    {

        printf("Element %d is not present in the array.\n", element);

    }

    return 0;

}

1. **Sorting Matrix Data Structure:**

We can sort a matrix in two-ways:

* Sort the matrix row-wise.
* Sort the matrix column-wise.

1. **Row wise sorting in 2D array:** Given a 2D array, sort each row of this array and print the result.

#include <stdio.h>

#include <conio.h>

void main()

{

    int row, column, temp;

    // Input rows and columns

    printf("Enter the row: ");

    scanf("%d", &row);

    printf("Enter the column: ");

    scanf("%d", &column);

    // Declare the array

    int arr[row][column];

    // Input array elements

    printf("Enter the array elements:\n");

    for (int i = 0; i < row; i++)

    {

        for (int j = 0; j < column; j++)

        {

            scanf("%d", &arr[i][j]);

        }

    }

    // Sort each row of the 2D array

    for (int i = 0; i < row; i++)

    {

        for (int j = 0; j < column; j++)

        {

            for (int k = j + 1; k < column; k++)

            {

                if (arr[i][j] > arr[i][k])

                {

                    temp = arr[i][j];

                    arr[i][j] = arr[i][k];

                    arr[i][k] = temp;

                }

            }

        }

    }

    // Display the sorted 2D array

    printf("Row-wise sorted array:\n");

    for (int i = 0; i < row; i++)

    {

        for (int j = 0; j < column; j++)

        {

            printf("%d ", arr[i][j]);

        }

        printf("\n");

    }

    getch();

}

1. **Colume-wise sorting in 2D array:** Given a 2D array, sort each colume of this array and print the result.

#include <stdio.h>

#include <conio.h>

void main()

{

    int row, column, temp;

    // Input number of rows and columns

    printf("Enter the number of rows: ");

    scanf("%d", &row);

    printf("Enter the number of columns: ");

    scanf("%d", &column);

    // Declare a 2D array

    int arr[row][column];

    // Input elements of the array

    printf("Enter the elements of the array:\n");

    for (int i = 0; i < row; i++)

    {

        for (int j = 0; j < column; j++)

        {

            scanf("%d", &arr[i][j]);

        }

    }

    // Sort the array column-wise

    for (int i = 0; i < column; i++)

    { // Iterate through each column

        for (int j = 0; j < row; j++)

        { // Iterate through rows for the current column

            for (int k = j + 1; k < row; k++)

            { // Compare and sort elements in the column

                if (arr[j][i] > arr[k][i])

                { // If the order is incorrect, swap elements

                    temp = arr[j][i];

                    arr[j][i] = arr[k][i];

                    arr[k][i] = temp;

                }

            }

        }

    }

    // Display the sorted array

    printf("Column-wise sorted array:\n");

    for (int i = 0; i < row; i++)

    {

        for (int j = 0; j < column; j++)

        {

            printf("%d\t", arr[i][j]);

        }

        printf("\n");

    }

    getch(); // Proper program termination

}

**Matrix Data Structure Components, Applications, Advantages and Disadvantages**

Matrix is a two-dimensional array or table consisting of rows and columns. The intersection of a row and column is called a cell. All the data is stored across different cells in the matrix. Matrix data structure is used when we want to store data in the form of table or grid. Each element in a matrix is identified by its row and column indices.

**Components of Matrix Data Structure**

* **Size**: A matrix has a specific size, defined by its number of rows and columns.
* **Element**: A matrix’s row and column indices serve to identify each entry, which is referred to as an element.
* **Operations**: Scalar multiplication and the operations of addition, subtraction, and multiplication on matrices are also supported.
* **Determinant:**A square matrix’s determinant is a scalar number that may be used to solve systems of linear equations and carry out other linear algebraic operations.
* **Inverse**: If a square matrix has an inverse, it may be used to solve linear equation systems and carry out other linear algebraic operations.
* **Transpose**: By flipping a matrix along its main diagonal and switching the rows and columns, you may create the transpose of the matrix.

**Advantages of Matrix Data Structure:**

* It helps in 2D Visualization.
* It stores multiple elements of the same type using the same name.
* It enables access to items at random.
* Any form of data with a fixed size can be stored.
* It is easy to implement.

**Disadvantages of Matrix Data Structure:**

* The matrix size should be needed beforehand.
* Insertion and deletion operations are costly if shifting occurs.
* Resizing a matrix is time-consuming.

1. **String:**

**String** is a sequence of characters, typically used to represent **text**. It is considered a data type that allows for the manipulation and processing of **textual data** in computer programs.

A string is a sequence of characters. The following facts make string an interesting data structure.

* Small set of elements. Unlike normal array, strings typically have smaller set of items. For example, lowercase English alphabet has only 26 characters. ASCII has only 256 characters.
* Strings are immutable in programming languages like Java, Python, JavaScript and C#.
* In programming language, if any word is written in single quote, double quote and triple quote then it is called a string.

Below are some examples of strings:

“geeks” , “for”, “geeks”, “GeeksforGeeks”, “Geeks for Geeks”, “123Geeks”, “@123 Geeks”

In C, a string can be represented in two main ways: as a character pointer or a character array.

1. **Character Array**:
   * A string is stored like a regular array. For example:

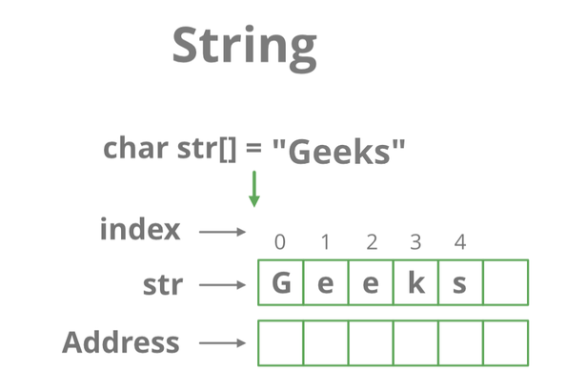
char str[] = "Hello";

* + If the array is a local (automatic) variable, it is stored in the *stack*.
  + If the array is declared as a static or global variable, it is stored in the *data segment* of memory.

1. **Character Pointer**:
   * A pointer points to the starting memory address of the string. For example:

char \*str = "Hello";

* + The string "Hello" here is stored in the *read-only memory segment*, and the pointer simply points to it.

In short, where the string is stored depends on how and where you declare it.

**How to Declare Strings in various languages?**

Below is the representation of strings in various languages:

// C program to illustrate strings

#include <stdio.h>

#include <conio.h>

void main()

{

    // Type: 1 declare and initialize string.

    char string[] = "Pankaj Yadav";

    // Type: 3 declare and initialize string

    char String1[50] = "Pankaj Yadav is a good boy ";

    // print string

    printf("%s\n", String1);

    // print string

    printf("%s\n", string);

    getch();

}

**General Operations performed on String:**

Here we are providing you with some must-know concepts of string:

**1. Concatenation of Strings**

The process of combining more than one string together is known as Concatenation. String Concatenation is the technique of combining two strings.

*Concatenation of Strings*

There are two ways to concatenate two strings:

**a) String concatenation without using any inbuilt methods:**

#include <stdio.h>  // Standard Input/Output library

#include <conio.h>  // Console Input/Output library (for getch())

void main()

{

    // Declare and initialize the first string

    char str1[] = "Hello";

    // Declare and initialize the second string (corrected spelling from "word!" to "world!")

    char str2[] = "world!";

    // Print the concatenated strings with a space in between

    printf("%s %s", str1, str2);

    // Wait for a key press before exiting (only works if conio.h is supported)

    getch();

}

**b) String concatenation using inbuilt methods:**

#include <stdio.h>  // Standard Input/Output library

#include <string.h> // String handling library for strcat()

void main()

{

    // Declare and initialize the first string

    char str1[50] = "Hello"; // Ensure the array is large enough to hold the result

    // Declare and initialize the second string

    char str2[] = " world!";

    // Concatenate str2 to str1 using strcat() function

    strcat(str1, str2);

    // Print the concatenated result

    printf("Concatenated String: %s\n", str1);

    getch(); // Return 0 to indicate successful execution

}

**2. Find in String**

A very basic operation performed on Strings is to find something in the given whole string. Now, this can be to find a given character in a string, or to find a complete string in another string.

**a) Find a character in string:**

Given a string and a character, your task is to find the first position of the character in the string. These types of problems are very competitive programming where you need to locate the position of the character in a string.

#include<stdio.h>  // Standard Input/Output library

#include<conio.h>  // Console Input/Output library for getch()

void main() {

    char str1[] = "Hello Pankaj...."; // Declare and initialize the string

    char word;                        // Variable to store the user-input character

    printf("This data present in program: %s\n", str1);

    // Prompt the user to input the character whose position they want to find

    printf("Enter the alphabet whose position you want to find: ");

    scanf(" %c", &word); // The space before %c ensures it ignores leading whitespaces

    int i = 0; // Initialize index for traversal

    int found = 0; // Flag to track if the character is found

    // Traverse the string to find the character

    while (str1[i] != '\0') {

        if (str1[i] == word) {

            printf("At this position your element is present: %d\n", i + 1);

            found = 1; // Mark that the character was found

        }

        i++;

    }

    // If character is not found, notify the user

    if (!found) {

        printf("The character '%c' is not present in the string.\n", word);

    }

    getch(); // Wait for key press before exiting

}

**b) Find a substring in another string:**

Consider there to be a string of length N and a substring of length M. Then run a nested loop, where the outer loop runs from 0 to (N-M) and the inner loop from 0 to M. For every index check if the sub-string traversed by the inner loop is the given sub-string or not.

#include <stdio.h>

#include <conio.h>

void main()

{

    char str[100], substr[50];

    int str\_len = 0, substr\_len = 0;

    int i, j, found, not\_fount;

    // Input the main string

    printf("Enter the main string: ");

    scanf(" %s",&str); // Read string

    // Input the substring to search

    printf("Enter the substring to search: ");

    scanf(" %s",&substr); // Read substring

    // Calculate the length of the main string

    while (str[str\_len] != '\0')

    {

        str\_len++;

    }

    // Calculate the length of the substring

    while (substr[substr\_len] != '\0')

    {

        substr\_len++;

    }

    // Find the substring manually

    for (i = 0; i <= str\_len - substr\_len; i++)

    {

        found = 1; // Assume substring is found

        for (j = 0; j < substr\_len; j++)

        {

            if (str[i + j] != substr[j])

            {

                found = 0; // Mismatch found

                break;

            }

        }

        if (found)

        {

            printf("Substring found at position: %d\n", i + 1); // 1-based index

            // Exit after finding the first occurrence

            not\_fount=1;

        }

    };

    if (not\_fount==0)

    {

        printf("Substring not found.\n");

    };

    getch();

}

**3. Replace in String**

Many times, it is very important to make corrections in strings. Replacing a character, word or phrase in a String is another very common operation performed on Strings.

**Program pending…..**

**4. Finding the Length of String**

One of the most general operations on String is to find the length/size of a given string. Length is defined as the number of characters in a string is called the length of that string.

**a) Length of string without using any inbuilt methods:**

#include <stdio.h>

#include <conio.h>

void main()

{

    char word[100] = ""; // Declare a character array to store the input string

    printf("Write a program to find the length of a string manually...\n");

    printf("Enter the word: ");

    gets(word); // Takes the input string from the user (consider replacing `gets()` with `fgets()` for safety)

    // Calculate the length of the string manually without using `strlen()`

    int i = 0;

    while (word[i] != '\0') // Loop through the string until the null terminator is reached

    {

        i++; // Increment the counter for each character

    }

    printf("The length of the sentence is: %d\n", i);

    getch();

}

**b) Length of string using inbuilt methods:**

#include <stdio.h>

#include <conio.h>

#include <string.h> // Header file for `strlen`

void main()

{

    char word[100] = ""; // Declare a character array to store the input string

    printf("Write a program to find the length of a string using `strlen()`...\n");

    printf("Enter the word: ");

    gets(word); // Takes the input string from the user (consider replacing `gets()` with `fgets()` for safety)

    // Calculate the length of the string using the built-in `strlen()` function

    int len = strlen(word);

    printf("The length of the sentence is: %d\n", len);

    getch();

}

# Searching Algorithms.

**Searching algorithms** are used to locate specific data within a large set of data. It helps **find a target value** within the data. There are various types of searching algorithms, each with its own approach and efficiency.

**Searching algorithms** are essential tools in computer science used to locate specific items within a collection of data.such as **databases, web search engines**, and more.

## Basics:

### Linear Search Algorithm

Given an array, arr of n integers, and an integer element x, find whether element x is present in the array. Return the index of the first occurrence of x in the array, or -1 if it doesn’t exist.

* ***Input****: arr[] = [1, 2, 3, 4], x = 3****Output****: 2****Explanation****: There is one test case with array as [1, 2, 3 4] and element to be searched as 3. Since 3 is present at index 2, the output is 2.*
* ***Input****: arr[] = [10, 8, 30, 4, 5], x = 5****Output****: 4****Explanation****: For array [1, 2, 3, 4, 5], the element to be searched is 5 and it is at index 4. So, the output is 4.*
* ***Input****: arr[] = [10, 8, 30], x = 6****Output****: -1****Explanation****: The element to be searched is 6 and its not present, so we return -1.*

In Linear Search, We go through each element of the array one by one and check if the current element is equal to the target element. If we find any element to be equal to the target element, then return the index of the current element. Otherwise, if no element is equal to the target element, then return -1 as the element is not found. Linear search is also known as sequential search.

// Consider the array arr[] = {10, 50, 30, 70, 80, 20, 90, 40} and Find key = 30

#include <stdio.h>

int main() {

    int key = 30; // Search element in array

    int arr[] = {10, 50, 30, 70, 80, 20, 90, 40};

    int len = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < len; i++) {

        if (key == arr[i]) {

            printf("Key found at index %d\n", i);

            return 1; // Key found

        }

    }

    // If the loop completes and the key isn't found

    printf("Key not found\n");

    return -1;

}

**Output:**

Key found at index: 2

#### Time and Space Complexity of Linear Search Algorithm:

#### Time Complexity:

* **Best Case:** In the best case, the key might be present at the first index. So the best case complexity is O(1)
* **Worst Case:** In the worst case, the key might be present at the last index. So the worst-case complexity is O(N).
* **Average Case:** O(N)

**Auxiliary Space:** O(1) as except for the variable to iterate through the list, no other variable is used.

#### Applications of Linear Search Algorithm:

* **Unsorted Lists:** When we have an unsorted array or list, linear search is most commonly used to find any element in the collection.
* **Small Data Sets:** Linear Search is preferred over binary search when we have small data sets with
* **Searching Linked Lists:** In linked list implementations, linear search is commonly used to find elements within the list. Each node is checked sequentially until the desired element is found.
* **Simple Implementation:** Linear Search is much easier to understand and implement as compared to Binary Search or Ternary Search.

#### Advantages of Linear Search Algorithm:

* Linear search can be used whether the array is sorted or not. It can be used on arrays of any data type.
* Does not require any additional memory.
* It is a well-suited algorithm for small datasets.

#### Disadvantages of Linear Search Algorithm:

* Linear search has a time complexity of O(N),
* Not suitable for large arrays.
* When to use Linear Search Algorithm?
* When we are dealing with a small dataset.
* When you are searching for a dataset stored in contiguous memory.

### Binary Search Algorithm – Iterative and Recursive Implementation

Binary Search Algorithm is a **searching method** used to find a specific number (element) in a **sorted list**. It works by **repeatedly dividing the search range into half** until the element is found. The idea of binary search is that the **array is already sorted**, so we can use this to **reduce the search time to O(log N)**.

#### What is Binary Search Algorithm?

**Binary search** is a search algorithm used to find the position of a target value within a **sorted**array. It works until the target value is found or the search range becomes empty. To make the search area smaller, the target number is checked against the middle number.

**Example:**

Let's say we have a **sorted list**:  
[10, 20, 30, 40, 50, 60, 70]  
And we want to find **40**.

1. **Find the middle number:**
   * The middle number is **40**.
   * Since 40 is the number we are looking for, we found it! ✅
2. **If we were looking for 50 instead:**
   * The middle number is **40**.
   * 50 is **greater** than 40, so we search in the **right half**: [50, 60, 70].
   * Now, the middle number is **60**.
   * 50 is **smaller** than 60, so we search in the **left half**: [50].
   * We found **50**! ✅

This process continues until we find the number or the list becomes empty.

#### Conditions to apply Binary Search Algorithm in a Data Structure

To apply Binary Search algorithm:

* The data structure must be sorted.
* Access to any element of the data structure should take constant time.

#### Binary Search Algorithm

Below is the step-by-step algorithm for Binary Search:

* **Find the middle position** in the list.
* **Compare the middle number** with the target number.
  + If the middle number **matches** the target, the search is complete. ✅
  + If the target is **smaller**, search in the **left half**.
  + If the target is **larger**, search in the **right half**.
* **Repeat the process** until the number is found or no numbers are left to check.

#### How to Implement Binary Search Algorithm?

The **Binary Search Algorithm** can be implemented in the following two ways

1. Iterative Binary Search Algorithm
2. Recursive Binary Search Algorithm

Given below are the pseudocodes for the approaches.

##### Iterative Binary Search Algorithm:

We use a **while loop** to keep checking the target number and keep dividing the search area into two parts until we find the number or the search ends.

#include <stdio.h>

int search(int arr[], int low, int high, int x)

{

    while (low <= high)

    {

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

        if (arr[mid] == x)

        {

            return mid;  // Found the element, return its index

        }

        if (arr[mid] < x)

        {

            low = mid + 1;  // Search in the right half

        }

        else

        {

            high = mid - 1; // Search in the left half

        }

    }

    return -1; // Return -1 if element is not found (moved outside loop)

}

int main()

{

    int arr[] = {12, 23, 45, 56, 78, 89, 90};

    int len = sizeof(arr) / sizeof(arr[0]);

    int result = search(arr, 0, len - 1, 78);

    if (result != -1)

    {

        printf("Element %d found at position %d\n", arr[result], result);

    }

    else

    {

        printf("Element not found\n"); // Handle case when element is not found

    }

    return 0;

}

**Output:** Element 78 found at position 4

**Time Complexity:** O(log N)  
**Auxiliary Space:** O(1)

##### Recursive Binary Search Algorithm:

We will create a **recursive function** to search for a number (key) in a **sorted array** using **binary search**.

**Steps to Follow:**

1. Start with the full array as our search space.
2. Find the **middle element** of the search space.
3. Compare the middle element with the **key** (the number we are searching for).
   * If they are **equal**, return the index (position).
   * If the key is **smaller**, search in the **left half**.
   * If the key is **greater**, search in the **right half**.
4. Repeat the process until the key is found or the search space is empty.

**Simple Recursive Binary Search in C**

#include <stdio.h>

int Binary\_search\_re(int arr[], int start, int end, int target)

{

    if (start > end)

    {

        return -1; // Target not found

    }

    int mid = start + (end - start) / 2; // Find mid index

    if (arr[mid] == target)

    {

        return mid; // Target found, return index

    }

    else if (arr[mid] > target)

    {

        return Binary\_search\_re(arr, start, mid - 1, target); // Search in left half

    }

    else

    {

        return Binary\_search\_re(arr, mid + 1, end, target); // Search in right half

    }

}

int main()

{

    int arr[] = {12, 34, 56, 78, 90, 99}; // Sorted array

    int len = sizeof(arr) / sizeof(arr[0]);

    int target = 99; // Number to search

    int result = Binary\_search\_re(arr, 0, len - 1, target);

    // Corrected output statement

    if (result != -1)

    {

        printf("Target found at index: %d\n", result);

    }

    else

    {

        printf("Target not found in the array.\n");

    }

    return 0;

}

#### Applications of Binary Search Algorithm

* ***Searching in a Sorted List –*** Quickly finds an element in a sorted array or list.
* ***Finding First or Last Occurrence –*** Helps in locating the first or last position of an element in a sorted list.
* ***Finding Square Root –*** Used to estimate square roots without using a direct formula.
* ***Finding Peak Element –*** Identifies the highest value in a sequence, such as stock prices.
* ***Optimization Problems –*** Helps in problems that require finding the best possible value.

#### Advantages of Binary Search

* Binary search is faster than linear search, especially for large arrays.
* More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
* Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

#### Disadvantages of Binary Search

* The array should be sorted.
* Binary search requires that the data structure being searched be stored in contiguous memory locations.
* Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.
* Index of First Occurence in Sorted Array.

#include <stdio.h>

void main() {

    int arr[] = {1, 2, 3, 4, 5, 9, 9, 9, 10};

    int target = 9;

    int len = sizeof(arr) / sizeof(arr[0]);

    int low = 0, high = len - 1;

    int firstIndex = -1;  // To store the first occurrence

    while (low <= high) {

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

        if (arr[mid] == target) {

            firstIndex = mid;  // Store index

            high = mid - 1;    // Move left to find first occurrence

        } else if (arr[mid] > target) {

            high = mid - 1;

        } else {

            low = mid + 1;

        }

    }

    if (firstIndex != -1) {

        printf("First occurrence of %d is at index %d\n", target, firstIndex);

    } else {

        printf("%d is not present in the array\n", target);

    }

}

**Output:** First occurrence of 9 is at index 5

* Index of Last Occurence in Sorted Array.

#include <stdio.h>

// Function to find the last occurrence of the target using Binary Search

int Last\_Occ(int arr[], int len, int target)

{

    int low = 0, high = len - 1;

    int LastIndex = -1; // Variable to store last occurrence index

    while (low <= high)

    {

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

        if (arr[mid] == target)

        {

            LastIndex = mid; // Store last occurrence index

            low = mid + 1;   // Move right to check for later occurrences

        }

        else if (arr[mid] > target)

        {

            high = mid - 1; // Search in the left half

        }

        else

        {

            low = mid + 1; // Search in the right half

        }

    }

    return LastIndex; // Return last occurrence index

}

int main()

{

    // Sorted array

    int arr[] = {1, 2, 3, 3, 3, 4, 6, 6, 9};

    int len = sizeof(arr) / sizeof(arr[0]); // Calculate array length

    int target = 6; // Number to search for

    int result = Last\_Occ(arr, len, target); // Get last occurrence index

    // Corrected condition to check if target is found

    if (result != -1)

    {

        printf("Last occurrence of %d is at index %d\n", target, result);

    }

    else

    {

        printf("%d is not present in the array\n", target);

    }

    return 0; // Return success

}

**Output:** Last occurrence of 6 is at index 7

* Count of occurrences of x in sorted element.

#include <stdio.h>

// Function to find the first occurrence of the target using Binary Search

int First\_occ(int arr[], int len, int target)

{

    int start = 0, end = len - 1;

    int First\_index = -1; // Variable to store the first occurrence index

    while (start <= end)

    {

        int mid = start + (end - start) / 2; // Calculate mid index

        if (arr[mid] == target)

        {

            First\_index = mid; // Store first occurrence index

            end = mid - 1; // Move left to find earlier occurrences

        }

        else if (arr[mid] < target)

        {

            start = mid + 1; // Search in the right half

        }

        else

        {

            end = mid - 1; // Search in the left half

        }

    }

    return First\_index; // Return the first occurrence index

}

// Function to find the last occurrence of the target using Binary Search

int Last\_occ(int arr[], int len, int target)

{

    int start = 0, end = len - 1;

    int Last\_index = -1; // Variable to store the last occurrence index

    while (start <= end)

    {

        int mid = start + (end - start) / 2; // Calculate mid index

        if (arr[mid] == target)

        {

            Last\_index = mid; // Store last occurrence index

            start = mid + 1; // Move right to find later occurrences

        }

        else if (arr[mid] < target)

        {

            start = mid + 1; // Search in the right half

        }

        else

        {

            end = mid - 1; // Search in the left half

        }

    }

    return Last\_index; // Return the last occurrence index

}

int main()

{

    // Sorted array

    int arr[] = {1, 3, 5, 7, 7, 7, 9, 9, 10, 12, 12, 12, 34};

    int len = sizeof(arr) / sizeof(arr[0]); // Calculate array length

    int target = 9; // Number to search for

    // Find first and last occurrence

    int first = First\_occ(arr, len, target);

    int last = Last\_occ(arr, len, target);

    // Check if target exists before calculating count

    if (first == -1 || last == -1)

    {

        printf("Target not found\n");

    }

    else

    {

        int result = (last - first) + 1; // Count occurrences

        printf("Count of %d: %d\n", target, result);

    }

    return 0; // Return success

}

**Output:** Count of 9: 2

* Count of 1s in a binary sorted array.

#include <stdio.h>

// Function to find the first occurrence of the target in a sorted binary array

int first\_occ(int arr[], int len, int target)

{

    int start = 0, end = len - 1;

    int First\_index = -1; // Variable to store the first occurrence index

    while (start <= end)

    {

        int mid = start + (end - start) / 2; // Calculate mid index

        if (arr[mid] == target)

        {

            First\_index = mid; // Store the first occurrence index

            end = mid - 1; // Move left to check for earlier occurrences

        }

        else if (arr[mid] < target)

        {

            start = mid + 1; // Search in the right half

        }

        else

        {

            end = mid - 1; // Search in the left half

        }

    }

    return First\_index; // Return the first occurrence index

}

// Function to find the last occurrence of the target in a sorted binary array

int Last\_occ(int arr[], int len, int target)

{

    int start = 0, end = len - 1;

    int Last\_index = -1; // Variable to store the last occurrence index

    while (start <= end)

    {

        int mid = start + (end - start) / 2; // Calculate mid index

        if (arr[mid] == target)

        {

            Last\_index = mid; // Store the last occurrence index

            start = mid + 1; // Move right to check for later occurrences

        }

        else if (arr[mid] < target)

        {

            start = mid + 1; // Search in the right half

        }

        else

        {

            end = mid - 1; // Search in the left half

        }

    }

    return Last\_index; // Return the last occurrence index

}

int main()

{

    // Sorted binary array

    int arr[] = {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1};

    int len = sizeof(arr) / sizeof(arr[0]); // Calculate the length of the array

    int target = 1; // Number to search for

    // Get first and last occurrence of target

    int first = first\_occ(arr, len, target);

    int last = Last\_occ(arr, len, target);

    // If the target is not found

    if (first == -1 || last == -1)

    {

        printf("Target not found\n");

    }

    else

    {

        int result = (last - first) + 1; // Calculate count of target

        printf("Count of %d: %d\n", target, result);

    }

    return 0;

}

**Output:** Count of 1: 8

* Find an element in sorted and rotated array.

#include <stdio.h>

// Function to find the target element in a rotated sorted array

int rotated(int arr[], int len, int target)

{

    int start = 0;

    int end = len - 1;

    while (start <= end)

    {

        int mid = start + (end - start) / 2; // Find the middle index

        // Check if mid element is the target

        if (arr[mid] == target)

        {

            return mid;

        }

        // Check if the left half is sorted

        if (arr[start] <= arr[mid])

        {

            // Check if target lies within the sorted left half

            if (target >= arr[start] && target < arr[mid])

            {

                end = mid - 1; // Search in left half

            }

            else

            {

                start = mid + 1; // Search in right half

            }

        }

        // Otherwise, the right half must be sorted

        else

        {

            // Check if target lies within the sorted right half

            if (target > arr[mid] && target <= arr[end])

            {

                start = mid + 1; // Search in right half

            }

            else

            {

                end = mid - 1; // Search in left half

            }

        }

    }

    return -1; // Element not found

}

int main()

{

    int arr[] = {4, 5, 6, 7, 8, 9, 0, 1, 2, 3}; // Rotated sorted array

    int len = sizeof(arr) / sizeof(arr[0]); // Find array length

    int target = 1; // Target element to find

    int result = rotated(arr, len, target); // Function call

    // Print result

    if (result != -1)

        printf("Position of %d is %d\n", target, result);

    else

        printf("Element %d not found\n", target);

    return 0; // Return success

}

**Output:** Position of 1 is 7

# WORD MEANING PAGE

1. **Instead :** बजाय, बदले में, जगह में
2. **Halves :** आधा
3. **Splitting :** बंटवारे
4. **Occurrence :** घटना