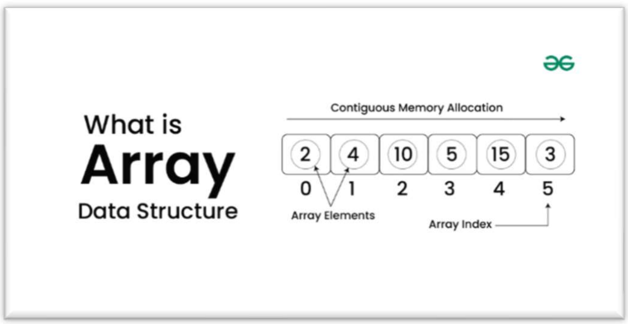
ARRAYS

Prerequisites: None
Language Used: Java
Real-Life Analogy: Think of an array like a row of mailboxes—each box (index) holds a value (element), and they're all side-by-side in memory.

BASICS OF ARRAYS

Definition:

An **array** is a collection of elements stored in contiguous memory locations. All elements are of the **same data type**. Each element is accessed using its **index**, starting from 0.



Key Properties:

- Fixed size (once declared)
- Elements accessed by index
- Homogeneous data type

Declaration and Initialization:

```
int[] numbers = new int[5];            // Declaration with size
int[] scores = {10, 20, 30, 40, 50};   // Declaration with initialization
```

Memory Representation:

If you write int[] scores = {10, 20, 30}, then memory is allocated as:

Index: 0 1 2

Value: 10 20 30

Access the Elements of an Array:

You can access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars[0]);

// Outputs Volvo
```

Important Points to Note about Arrays

- Array is a fundamental data structure and used to implement other data structures like stack, queue, dequeue and heap.
- The main advantages of using array over other data structures are cache friendliness and random access memory.

1D ARRAY

Description:

A **1D array** is a linear collection of data elements of the same type.

Example:

```
public class OneDArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        for(int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

Output:

1 2 3 4 5

2D ARRAY

Description:

A **2D array** is like a table with rows and columns.

```
int[][] matrix = {
    {1, 2, 3},
```

```
    {4, 5, 6}
};
```

Example:

```java
public class TwoDArray {
    public static void main(String[] args) {
        int[][] mat = {
            {1, 2, 3},
            {4, 5, 6}
        };
        for (int i = 0; i < mat.length; i++) {
            for (int j = 0; j < mat[i].length; j++) {
                System.out.print(mat[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Output:

1 2 3

4 5 6

DYNAMIC ARRAYS

Description:

Dynamic arrays can grow or shrink during runtime. In Java, use ArrayList.

```java
import java.util.ArrayList;

public class DynamicArray {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(10);
```

```
        list.add(20);

        list.add(30);

        list.remove(1);

        System.out.println(list); // Output: [10, 30]

    }

}
```

ARRAY REVERSE

Given an array **arr[]**, the task is to **reverse** the array. Reversing an array means **rearranging** the elements such that the **first** element becomes the **last**, the **second** element becomes **second last** and so on.

Examples:

Input: *arr[] = {1, 4, 3, 2, 6, 5}*
Output: *{5, 6, 2, 3, 4, 1}*
Explanation: *The first element* ***1*** *moves to last position, the second element* ***4*** *moves to second-last and so on.*

Input: *arr[] = {4, 5, 1, 2}*
Output: *{2, 1, 5, 4}*
Explanation: *The first element* ***4*** *moves to last position, the second element* ***5*** *moves to second last and so on.*

Using Inbuilt Methods - O(n) Time and O(1) Space

[Naive Approach] Using a temporary array - O(n) Time and O(n) Space

The idea is to use a ***temporary array*** *to store the reverse of the array.*

- *Create a* ***temporary*** *array of same size as the original array.*
- *Now, copy all elements from original array to the temporary array in* ***reverse order****.*
- *Finally, copy all the elements from temporary array back to the original array.*

Working:

Below is the implementation of the algorithm:

```
// Java Program to reverse an array using temporary array


import java.util.Arrays;


class GfG {
```

```
    // function to reverse an array
    static void reverseArray(int[] arr) {
        int n = arr.length;

        // Temporary array to store elements in reversed order
        int[] temp = new int[n];

        // Copy elements from original array to temp in reverse order
        for (int i = 0; i < n; i++)
            temp[i] = arr[n - i - 1];

        // Copy elements back to original array
        for (int i = 0; i < n; i++)
            arr[i] = temp[i];
    }

    public static void main(String[] args) {
        int[] arr = { 1, 4, 3, 2, 6, 5 };

        reverseArray(arr);

        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
    }
}
```

Output

5 6 2 3 4 1

Time Complexity: O(n), Copying elements to a new array is a linear operation.
Auxiliary Space: O(n), as we are using an extra array to store the reversed array.

[Expected Approach - 1] Using Two Pointers - O(n) Time and O(1) Space

The idea is to maintain two pointers: ***left*** *and* ***right****, such that* ***left*** *points at the* ***beginning*** *of the array and* ***right*** *points to the* ***end*** *of the array.*

While left pointer is less than the right pointer, swap the elements at these two positions. After each swap, ***increment*** *the* ***left*** *pointer and* ***decrement*** *the* ***right*** *pointer to move towards the center of array. This will swap all the elements in the first half with their corresponding element in the second half.*

Working:

Below is the implementation of the algorithm:

```
// Java Program to reverse an array using Two Pointers

import java.util.Arrays;

class GfG {

    // function to reverse an array

    static void reverseArray(int[] arr) {


        // Initialize left to the beginning and right to the end

        int left = 0, right = arr.length - 1;


        // Iterate till left is less than right

        while (left < right) {


            // Swap the elements at left and right position

            int temp = arr[left];

            arr[left] = arr[right];

            arr[right] = temp;


            // Increment the left pointer

            left++;


            // Decrement the right pointer

            right--;

        }
```

```java
    }

    public static void main(String[] args) {
        int[] arr = { 1, 4, 3, 2, 6, 5 };

        reverseArray(arr);

        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
    }
}
```

Output

5 6 2 3 4 1

Time Complexity: O(n), as we are visiting each element exactly once.
Auxiliary Space: O(1)

[Expected Approach - 2] By Swapping Elements - O(n) Time and O(1) Space

The idea is to iterate over the ***first half*** *of the array and* ***swap*** *each element with its corresponding element from the* ***end****. So, while iterating over the first half, any element at index* ***i*** *is swapped with the element at index* ***(n - i - 1)****.*

Working:

Below is the implementation of the algorithm:

```java
// Java Program to reverse an array by swapping elements

import java.util.Arrays;

class GfG {

    // function to reverse an array
    static void reverseArray(int[] arr) {
        int n = arr.length;
```

```
        // Iterate over the first half and for every index i,
        // swap arr[i] with arr[n - i - 1]
        for (int i = 0; i < n / 2; i++) {
            int temp = arr[i];
            arr[i] = arr[n - i - 1];
            arr[n - i - 1] = temp;
        }
    }

    public static void main(String[] args) {
        int[] arr = { 1, 4, 3, 2, 6, 5 };

        reverseArray(arr);

        for (int i = 0; i < arr.length; i++)
            System.out.print(arr[i] + " ");
    }
}
```

Output

5 6 2 3 4 1

Time Complexity: O(n), the loop runs through half of the array, so it's linear with respect to the array size.
Auxiliary Space: O(1), no extra space is required, therefore we are reversing the array **in-place**.

[Alternate Approach] Using Recursion - O(n) Time and O(n) Space

The idea is to use recursion and define a ***recursive function*** *that takes a range of array elements as input and reverses it. Inside the recursive function,*

- *Swap the first and last element.*
- *Recursively call the function with the remaining subarray.*

```
// Java Program to reverse an array using Recursion
```

```
import java.util.Arrays;

class GfG {

    // recursive function to reverse an array from l to r
    static void reverseArrayRec(int[] arr, int l, int r) {
        if (l >= r)
            return;

        // Swap the elements at the ends
        int temp = arr[l];
        arr[l] = arr[r];
        arr[r] = temp;

        // Recur for the remaining array
        reverseArrayRec(arr, l + 1, r - 1);
    }

    // function to reverse an array
    static void reverseArray(int[] arr) {
        int n = arr.length;
        reverseArrayRec(arr, 0, n - 1);
    }

    public static void main(String[] args) {
        int[] arr = { 1, 4, 3, 2, 6, 5 };

        reverseArray(arr);

        for (int i = 0; i < arr.length; i++)
```

```java
            System.out.print(arr[i] + " ");
    }
}
```

Output

5 6 2 3 4 1

Time Complexity: O(n), the recurrence relation will be **T(n) = T(n - 2) + O(1)**, which can be simplified to O(n).
Auxiliary Space: O(n), as we are using recursion stack.

Using Inbuilt Methods - O(n) Time and O(1) Space

*The idea is to use inbuilt **reverse** methods available across different languages.*

```java
// Java Program to reverse an array using inbuilt methods


import java.util.*;


class GfG {


    // function to reverse an array
    static void reverseArray(List<Integer> arr) {
        Collections.reverse(arr);
    }


    public static void main(String[] args) {
        List<Integer> arr =
          new ArrayList<>(Arrays.asList(1, 4, 3, 2, 6, 5));


        reverseArray(arr);


        for (int i = 0; i < arr.size(); i++)
            System.out.print(arr.get(i) + " ");
    }
```

```
}
```

Output

5 6 2 3 4 1

Time Complexity: O(n), the reverse method has linear time complexity.
Auxiliary Space: O(1) Additional space is not used to store the reversed array, as the in-built array method swaps the values in-place.

PREFIX SUM

Description:

Prefix sum is the cumulative sum of elements up to a certain index.

Example:

```
public class PrefixSum {
    public static void main(String[] args) {
        int[] arr = {2, 4, 6, 8};
        int[] prefix = new int[arr.length];
        prefix[0] = arr[0];
        for (int i = 1; i < arr.length; i++) {
            prefix[i] = prefix[i - 1] + arr[i];
        }
        for (int sum : prefix) {
            System.out.print(sum + " ");
        }
    }
}
```

Output:

2 6 12 20

SLIDING WINDOW TECHNIQUE

Description:

Used to reduce the time complexity of problems involving subarrays.

Example: Max sum of subarray of size k

```
public class SlidingWindow {
    public static void main(String[] args) {
        int[] arr = {1, 4, 2, 10, 23, 3, 1, 0, 20};
        int k = 4, maxSum = 0, windowSum = 0;

        for (int i = 0; i < k; i++)
            windowSum += arr[i];
        maxSum = windowSum;

        for (int i = k; i < arr.length; i++) {
            windowSum += arr[i] - arr[i - k];
            maxSum = Math.max(maxSum, windowSum);
        }
        System.out.println("Max sum = " + maxSum);
    }
}
```

TWO POINTERS TECHNIQUE

Description:

Use two pointers to solve problems on **sorted arrays** efficiently.

Example: Find if there's a pair with sum = X

```
import java.util.Arrays;

public class TwoPointers {
    public static void main(String[] args) {
        int[] arr = {2, 4, 7, 11, 15};
        int target = 9;
        int left = 0, right = arr.length - 1;
```

```
    while (left < right) {
      int sum = arr[left] + arr[right];
      if (sum == target) {
        System.out.println("Pair found: " + arr[left] + " & " + arr[right]);
        return;
      } else if (sum < target) {
        left++;
      } else {
        right--;
      }
    }
    System.out.println("No pair found.");
  }
}
```

KADANE’S ALGORITHM

Description:

Finds the maximum sum subarray in O(n) time.

```
public class KadaneAlgo {
  public static void main(String[] args) {
    int[] arr = {-2, -3, 4, -1, -2, 1, 5, -3};
    int maxSoFar = arr[0], currMax = arr[0];

    for (int i = 1; i < arr.length; i++) {
      currMax = Math.max(arr[i], currMax + arr[i]);
      maxSoFar = Math.max(maxSoFar, currMax);
    }

    System.out.println("Maximum Subarray Sum is " + maxSoFar);
  }
```

}

ARRAY ROTATION

Description:

Rotate array elements by k positions.

```
import java.util.Arrays;

public class ArrayRotation {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int k = 2;
        rotateArray(arr, k);
        System.out.println(Arrays.toString(arr));
    }

    static void rotateArray(int[] arr, int k) {
        k = k % arr.length;
        reverse(arr, 0, arr.length - 1);
        reverse(arr, 0, k - 1);
        reverse(arr, k, arr.length - 1);
    }

    static void reverse(int[] arr, int l, int r) {
        while (l < r) {
            int temp = arr[l];
            arr[l++] = arr[r];
            arr[r--] = temp;
        }
    }
}
```

LINEAR SEARCH

Description:

Search each element one-by-one.

```
public class LinearSearch {
    public static void main(String[] args) {
        int[] arr = {5, 3, 7, 9};
        int key = 7;
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == key) {
                System.out.println("Found at index " + i);
                return;
            }
        }
        System.out.println("Not found");
    }
}
```

BINARY SEARCH

Description:

Search in a **sorted array** in O(log n) time.

```
public class BinarySearch {
    public static void main(String[] args) {
        int[] arr = {2, 4, 6, 8, 10};
        int key = 8;
        int low = 0, high = arr.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] == key) {
```

```
            System.out.println("Found at index " + mid);
            return;
        } else if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    System.out.println("Not found");
  }
}
```

SORTING ALGORITHMS

Bubble Sort

```
public class BubbleSort {
  public static void main(String[] args) {
    int[] arr = {5, 1, 4, 2, 8};
    for (int i = 0; i < arr.length - 1; i++)
      for (int j = 0; j < arr.length - i - 1; j++)
        if (arr[j] > arr[j + 1]) {
          int temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp;
        }
    System.out.println(java.util.Arrays.toString(arr));
  }
}
```

Insertion Sort

```
public class InsertionSort {
  public static void main(String[] args) {
    int[] arr = {5, 1, 4, 2, 8};
    for (int i = 1; i < arr.length; i++) {
```

```
            int key = arr[i], j = i - 1;
            while (j >= 0 && arr[j] > key) arr[j + 1] = arr[j--];
            arr[j + 1] = key;
        }
        System.out.println(java.util.Arrays.toString(arr));
    }
}
```

Selection Sort

```
public class SelectionSort {
    public static void main(String[] args) {
        int[] arr = {5, 1, 4, 2, 8};
        for (int i = 0; i < arr.length - 1; i++) {
            int min = i;
            for (int j = i + 1; j < arr.length; j++)
                if (arr[j] < arr[min]) min = j;
            int temp = arr[min]; arr[min] = arr[i]; arr[i] = temp;
        }
        System.out.println(java.util.Arrays.toString(arr));
    }
}
```

Quick Sort

```
public class QuickSort {
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
```

```
    public static int partition(int[] arr, int low, int high) {
        int pivot = arr[high], i = low - 1;
        for (int j = low; j < high; j++)
            if (arr[j] < pivot) {
                int temp = arr[++i]; arr[i] = arr[j]; arr[j] = temp;
            }
        int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
        return i + 1;
    }

    public static void main(String[] args) {
        int[] arr = {10, 7, 8, 9, 1, 5};
        quickSort(arr, 0, arr.length - 1);
        System.out.println(java.util.Arrays.toString(arr));
    }
}
```

Merge Sort

```
public class MergeSort {
    public static void mergeSort(int[] arr, int l, int r) {
        if (l < r) {
            int m = (l + r) / 2;
            mergeSort(arr, l, m);
            mergeSort(arr, m + 1, r);
            merge(arr, l, m, r);
        }
    }

    public static void merge(int[] arr, int l, int m, int r) {
        int[] left = java.util.Arrays.copyOfRange(arr, l, m + 1);
        int[] right = java.util.Arrays.copyOfRange(arr, m + 1, r + 1);
```

```
        int i = 0, j = 0, k = l;
        while (i < left.length && j < right.length)
            arr[k++] = (left[i] <= right[j]) ? left[i++] : right[j++];
        while (i < left.length) arr[k++] = left[i++];
        while (j < right.length) arr[k++] = right[j++];
    }

    public static void main(String[] args) {
        int[] arr = {12, 11, 13, 5, 6, 7};
        mergeSort(arr, 0, arr.length - 1);
        System.out.println(java.util.Arrays.toString(arr));
    }
}
```

SUBARRAYS, SUBSEQUENCES, AND SUBSETS IN ARRAY

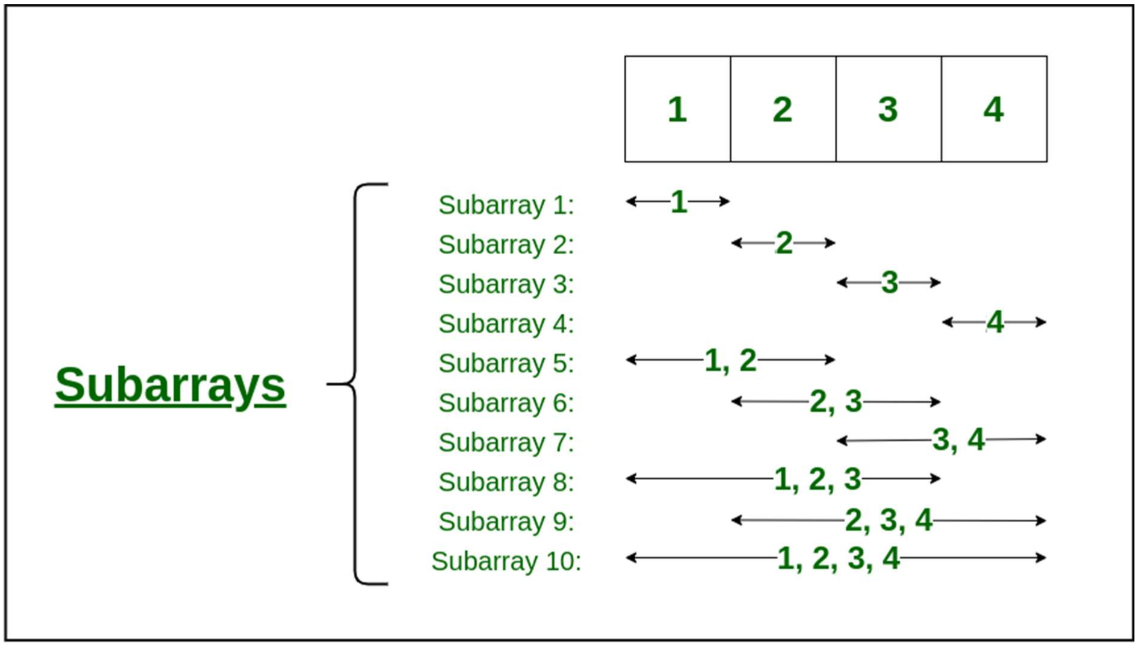
What is a Subarray?

*A **subarray** is a contiguous part of array, i.e., Subarray is an array that is inside another array.*

In general, for an array of size n, there are **n*(n+1)/2** non-empty subarrays.

For example, Consider the array [1, 2, 3, 4], There are 10 non-empty sub-arrays. The subarrays are:

(1), (2), (3), (4), (1,2), (2,3), (3,4), (1,2,3), (2,3,4), and (1,2,3,4)



Subarray

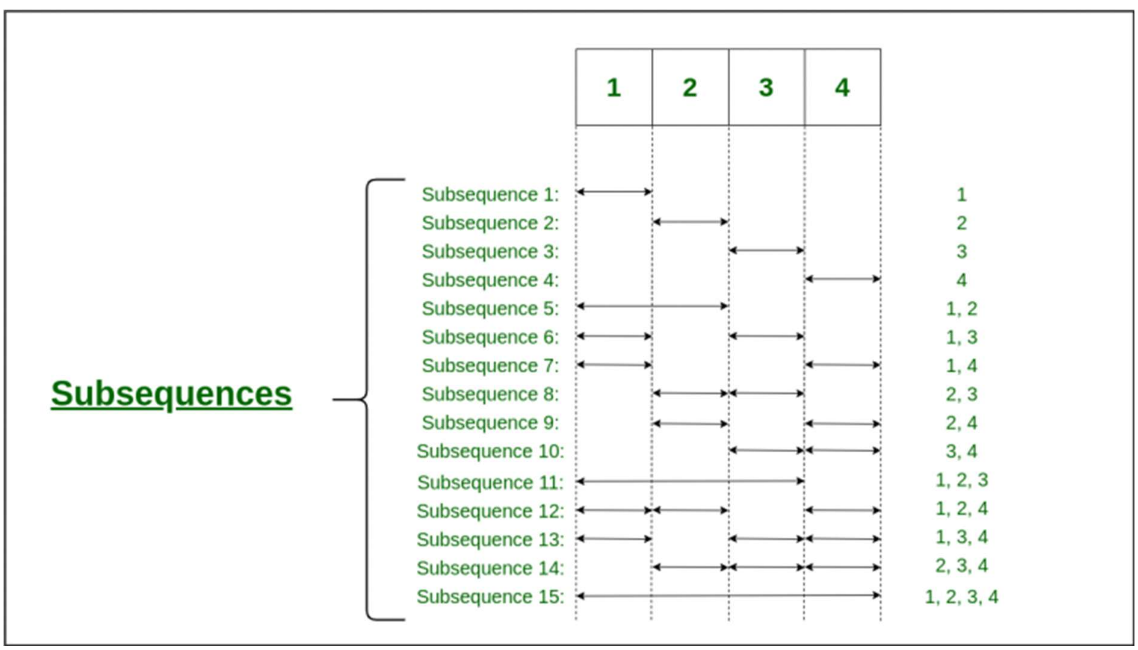
What is a Subsequence?

*A **subsequence** is a sequence that can be derived from another sequence by removing zero or more elements, without changing the order of the remaining elements.*

More generally, we can say that for a sequence of size n, we can have **(2n – 1)** non-empty sub-sequences in total.

For the same above example, there are 15 sub-sequences. They are:

(1), (2), (3), (4), (1,2), (1,3),(1,4), (2,3), (2,4), (3,4), (1,2,3), (1,2,4), (1,3,4), (2,3,4), (1,2,3,4).



Subsequences

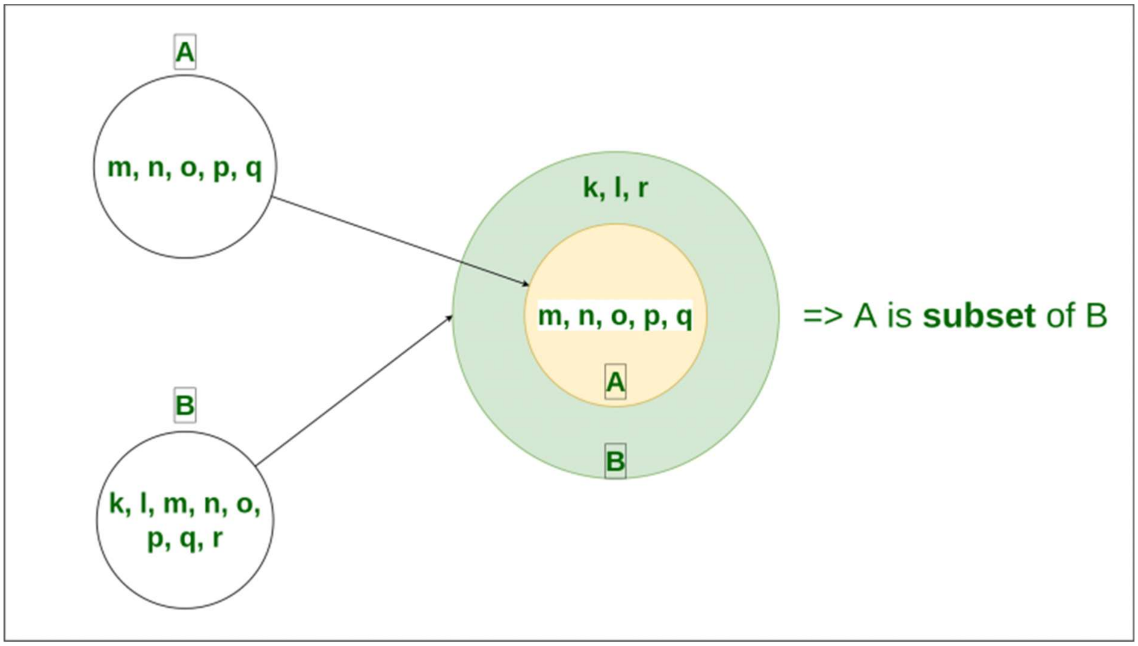
What is a Subset?

*If a Set has all its elements belonging to other sets, this set will be known as a **subset** of the other set.*

A Subset is denoted as "⊆". If set A is a subset of set B, it is represented as **A ⊆ B**.

For example, Let Set_A = {m, n, o, p, q}, Set_ B = {k, l, m, n, o, p, q, r}

Then, A ⊆ B.



Subset

ADVANCED PRACTICE PROBLEMS (ARRAYS)

1. Maximum Product Subarray

Problem: Given an integer array nums, find the contiguous subarray within an array (containing at least one number) which has the largest product.
Difficulty: Medium
Hint: Track both max and min products at each step.

---

2. Subarray Sum Equals K

Problem: Given an array of integers and an integer k, you need to find the total number of continuous subarrays whose sum equals to k.
Difficulty: Medium
Hint: Use prefix sum with HashMap.

---

3. Merge Intervals

Problem: Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals.
Difficulty: Medium
Hint: Sort intervals by start time first.

---

4. Median of Two Sorted Arrays

Problem: Given two sorted arrays nums1 and nums2, return the median of the two sorted arrays.
Difficulty: Hard
Hint: Use binary search and partitioning.

---

5. Maximum Sum of 3 Non-Overlapping Subarrays

Problem: Find three non-overlapping subarrays of length k with maximum total sum.
Difficulty: Hard
Hint: Use prefix sum and dynamic programming.

---

6. Count Inversions

Problem: Count the number of inversions in an array (i.e., pairs i < j such that arr[i] > arr[j]).
Difficulty: Medium to Hard
Hint: Use merge sort for efficient counting.

---

7. Next Greater Element

Problem: For each element, find the next greater element to its right in the array.
Difficulty: Medium
Hint: Use a monotonic stack.

---

8. Maximum of All Subarrays of Size k

Problem: Given an array and a number k, find the maximum for each and every contiguous subarray of size k.
Difficulty: Medium
Hint: Use deque/sliding window technique.

---

9. Rotate Image (90 degrees)

Problem: Rotate a square matrix (2D array) by 90 degrees in-place.
Difficulty: Medium
Hint: Transpose + reverse rows.

---

10. Spiral Order Matrix Traversal

Problem: Given a matrix, return all elements in spiral order.
Difficulty: Medium
Hint: Use four pointers (top, bottom, left, right).

RECOMMENDED LINKS

https://www.geeksforgeeks.org/top-50-array-coding-problems-for-interviews/

SUGGESTED YOUTUBE LINKS:

1. https://youtu.be/239ubH043lI?si=LL7Ids26f42S-MZE
2. https://youtu.be/NTHVTY6w2Co?si=m9LX2NGE6TAUu_Jc
3. https://youtu.be/uidBSlGLUK4?si=oJoSe7nue0UIwPiR
4. https://youtu.be/v4J2bEQF6jk?si=PBw-jJrontV5VOc1
5. https://youtu.be/UPjMnMkwKOQ?si=DvyIaFRV0Gtve_dX
6. https://youtu.be/VkzV642CVdo?si=wAyyKeJ2-ocfUnEd
7. https://youtu.be/FosoYnhpbHA?si=gczv6LBoMeJ_dht4
8. https://youtu.be/hdyGgvBYpII?si=ZGHSK5B1kZzsAJhA
9. https://youtu.be/xWLxhF3b5P8?si=O7AoGW_voqnVRhLx
10. https://youtu.be/oABQlhrhXzg?si=EsY-C2ByUodB4Jpi