**DAY-5**

1.Write a Program to find both the maximum and minimum values in the array. Implement using any programming language of your choice. Execute your code and provide the maximum and minimum values found.

Input : N= 8, a[] = {5,7,3,4,9,12,6,2}

Output : Min = 2, Max = 12

**Program:-**

array = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

if not array:  # Check if the array is empty

    max\_value = None

    min\_value = None

else:

    max\_value = array[0]

    min\_value = array[0]

for num in array:

    if num > max\_value:

        max\_value = num

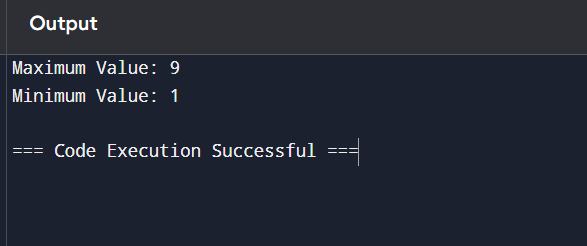
    if num < min\_value:

        min\_value = num

print("Maximum Value:", max\_value)

print("Minimum Value:", min\_value)

**output:-**



2.Consider an array of integers sorted in ascending order: 2,4,6,8,10,12,14,18. Write a Program to find both the maximum and minimum values in the array. Implement using any programming language of your choice. Execute your code and provide the maximum and minimum values found.

Input : N=8, 2,4,6,8,10,12,14,18.

Output : Min = 2, Max =18

**Program:-**

# Given sorted array

array = [2, 4, 6, 8, 10, 12, 14, 18]

# Check if the array is empty

if not array:

    max\_value = None

    min\_value = None

else:

    max\_value = array[-1]  # Last element in a sorted array is the maximum

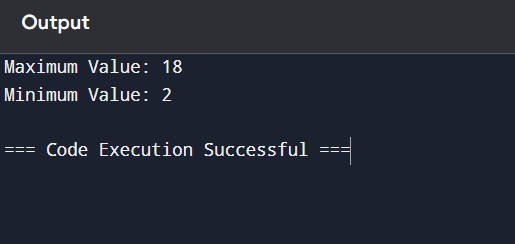
    min\_value = array[0]   # First element in a sorted array is the minimum

# Print the results

print("Maximum Value:", max\_value)

print("Minimum Value:", min\_value)

**output:-**



3.You are given an unsorted array 31,23,35,27,11,21,15,28. Write a program for Merge Sort and implement using any programming language of your choice.

Test Cases :

Input : N= 8, a[] = {31,23,35,27,11,21,15,28}

Output : 11,15,21,23,27,28,31,35

**Program:-**

# Given unsorted array

array = [31, 23, 35, 27, 11, 21, 15, 28]

# Merge Sort algorithm

def merge\_sort(arr):

    if len(arr) > 1:

        mid = len(arr) // 2  # Finding the mid of the array

        left\_half = arr[:mid]  # Dividing the array elements into 2 halves

        right\_half = arr[mid:]

        # Sorting the first half

        merge\_sort(left\_half)

        # Sorting the second half

        merge\_sort(right\_half)

        i = j = k = 0

        # Copy data to temporary arrays left\_half[] and right\_half[]

        while i < len(left\_half) and j < len(right\_half):

            if left\_half[i] < right\_half[j]:

                arr[k] = left\_half[i]

                i += 1

            else:

                arr[k] = right\_half[j]

                j += 1

            k += 1

        # Checking if any element was left

        while i < len(left\_half):

            arr[k] = left\_half[i]

            i += 1

            k += 1

        while j < len(right\_half):

            arr[k] = right\_half[j]

            j += 1

            k += 1

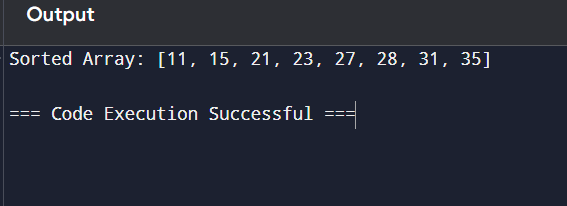
# Call the merge sort algorithm

merge\_sort(array)

# Print the sorted array

print("Sorted Array:", array)

**output:-**



4.Implement the Merge Sort algorithm in a programming language of your choice and test it on the array 12,4,78,23,45,67,89,1. Modify your implementation to count the number of comparisons made during the sorting process. Print this count along with the sorted array.

Test Cases :

Input : N= 8, a[] = {12,4,78,23,45,67,89,1}

Output : 1,4,12,23,45,67,78,89

**Program:-**

# Given unsorted array

array = [12, 4, 78, 23, 45, 67, 89, 1]

# Initialize comparison count

comparison\_count = 0

# Merge Sort algorithm with comparison counting

def merge\_sort(arr):

    global comparison\_count  # Use global variable to track comparisons

    if len(arr) > 1:

        mid = len(arr) // 2  # Finding the mid of the array

        left\_half = arr[:mid]  # Dividing the array elements into 2 halves

        right\_half = arr[mid:]

        # Sorting the first half

        merge\_sort(left\_half)

        # Sorting the second half

        merge\_sort(right\_half)

        i = j = k = 0

        # Copy data to temporary arrays left\_half[] and right\_half[]

        while i < len(left\_half) and j < len(right\_half):

            comparison\_count += 1  # Increment comparison count

            if left\_half[i] < right\_half[j]:

                arr[k] = left\_half[i]

                i += 1

            else:

                arr[k] = right\_half[j]

                j += 1

            k += 1

        while i < len(left\_half):

            arr[k] = left\_half[i]

            i += 1

            k += 1

        while j < len(right\_half):

            arr[k] = right\_half[j]

            j += 1

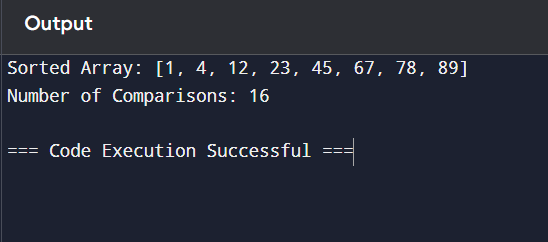
            k += 1

merge\_sort(array)

print("Sorted Array:", array)

print("Number of Comparisons:", comparison\_count)

**output:-**



5.Given an unsorted array 10,16,8,12,15,6,3,9,5 Write a program to perform Quick Sort. Choose the first element as the pivot and partition the array accordingly. Show the array after this partition. Recursively apply Quick Sort on the sub-arrays formed. Display the array after each recursive call until the entire array is sorted.

Input : N= 9, a[]= {10,16,8,12,15,6,3,9,5}

Output : 3,5,6,8,9,10,12,15,16

**Program:-**

# Given unsorted array

array = [10, 16, 8, 12, 15, 6, 3, 9, 5]

# Quick Sort algorithm

def quick\_sort(arr, low, high):

    if low < high:

        # Partition the array and get the pivot index

        pivot\_index = partition(arr, low, high)

        # Display the array after partitioning

        print("Array after partitioning:", arr)

        # Recursively apply Quick Sort on the sub-arrays

        quick\_sort(arr, low, pivot\_index - 1)

        quick\_sort(arr, pivot\_index + 1, high)

def partition(arr, low, high):

    pivot = arr[low]  # Choosing the first element as the pivot

    left = low + 1

    right = high

    while True:

        # Move left pointer to the right as long as the value is less than the pivot

        while left <= right and arr[left] <= pivot:

            left += 1

        # Move right pointer to the left as long as the value is greater than the pivot

        while left <= right and arr[right] >= pivot:

            right -= 1

        # If the left pointer has crossed the right, partitioning is done

        if left > right:

            break

        else:

            # Swap the left and right elements

            arr[left], arr[right] = arr[right], arr[left]

    # Swap the pivot with the right pointer

    arr[low], arr[right] = arr[right], arr[low]

    return right  # Return the pivot index

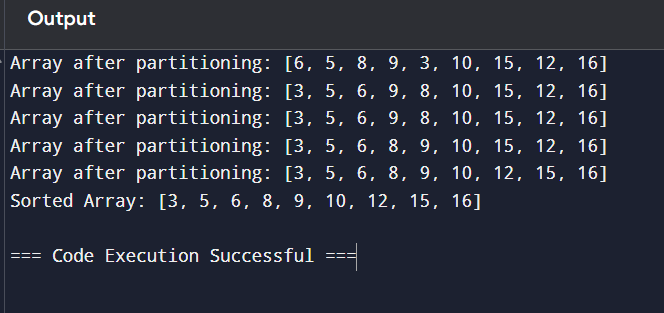
# Call the quick sort algorithm

quick\_sort(array, 0, len(array) - 1)

# Print the final sorted array

print("Sorted Array:", array)

**output:-**



6.Implement the Quick Sort algorithm in a programming language of your choice and test it on the array 19,72,35,46,58,91,22,31. Choose the middle element as the pivot and partition the array accordingly. Show the array after this partition. Recursively apply Quick Sort on the sub-arrays formed. Display the array after each recursive call until the entire array is sorted. Execute your code and show the sorted array.

Input : N= 8, a[] = {19,72,35,46,58,91,22,31}

Output : 19,22,31,35,46,58,72,91

**Program:-**

# Given unsorted array

array = [19, 72, 35, 46, 58, 91, 22, 31]

# Quick Sort algorithm

def quick\_sort(arr, low, high):

    if low < high:

        # Partition the array and get the pivot index

        pivot\_index = partition(arr, low, high)

        # Display the array after partitioning

        print("Array after partitioning:", arr)

        # Recursively apply Quick Sort on the sub-arrays

        quick\_sort(arr, low, pivot\_index - 1)

        quick\_sort(arr, pivot\_index + 1, high)

def partition(arr, low, high):

    # Choose the middle element as the pivot

    mid = (low + high) // 2

    pivot = arr[mid]

    arr[mid], arr[high] = arr[high], arr[mid]

    left = low

    right = high - 1

    while True:

        # Move left pointer to the right as long as the value is less than the pivot

        while left <= right and arr[left] < pivot:

            left += 1

        # Move right pointer to the left as long as the value is greater than the pivot

        while left <= right and arr[right] > pivot:

            right -= 1

        # If the left pointer has crossed the right, partitioning is done

        if left >= right:

            break

        else:

            # Swap the left and right elements

            arr[left], arr[right] = arr[right], arr[left]

            left += 1

            right -= 1

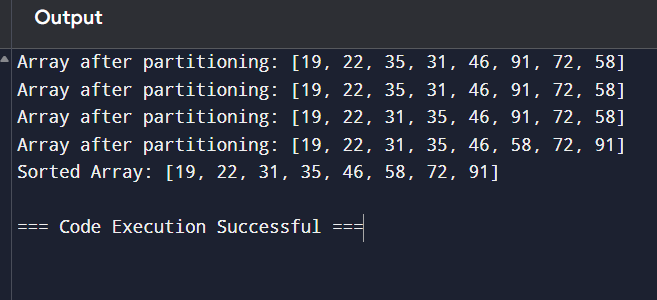
    arr[left], arr[high] = arr[high], arr[left]

    return left  # Return the pivot index

quick\_sort(array, 0, len(array) - 1)

print("Sorted Array:", array)

**output:-**



7.Implement the Binary Search algorithm in a programming language of your choice and test it on the array 5,10,15,20,25,30,35,40,45 to find the position of the element 20. Execute your code and provide the index of the element 20. Modify your implementation to count the number of comparisons made during the search process. Print this count along with the result.

Input : N= 9, a[] = {5,10,15,20,25,30,35,40,45}, search key = 20

Output : 4

**Program:-**

# Given sorted array

array = [5, 10, 15, 20, 25, 30, 35, 40, 45]

target = 20

# Initialize comparison count

comparison\_count = 0

# Binary Search algorithm with comparison counting

def binary\_search(arr, target):

    global comparison\_count

    low = 0

    high = len(arr) - 1

    while low <= high:

        mid = (low + high) // 2

        comparison\_count += 1  # Increment comparison count

        if arr[mid] == target:

            return mid  # Target found

        elif arr[mid] < target:

            low = mid + 1  # Search in the right half

        else:

            high = mid - 1  # Search in the left half

    return -1  # Target not found

# Call the binary search algorithm

index = binary\_search(array, target)

# Print the result and the number of comparisons

if index != -1:

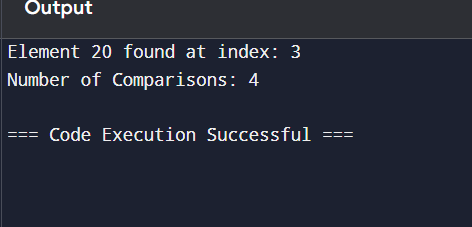
    print(f"Element {target} found at index: {index}")

else:

    print(f"Element {target} not found in the array.")

print("Number of Comparisons:", comparison\_count)

**output:-**



8.You are given a sorted array 3,9,14,19,25,31,42,47,53 and asked to find the position of the element 31 using Binary Search. Show the mid-point calculations and the steps involved in finding the element. Display, what would happen if the array was not sorted, how would this impact the performance and correctness of the Binary Search algorithm?

Input : N= 9, a[] = {3,9,14,19,25,31,42,47,53}, search key = 31

Output : 6

**Program:-**

# Given sorted array

array = [3, 9, 14, 19, 25, 31, 42, 47, 53]

target = 31

# Binary Search algorithm with step-by-step output

def binary\_search(arr, target):

    low = 0

    high = len(arr) - 1

    step = 1  # Step counter for iterations

    while low <= high:

        mid = (low + high) // 2  # Calculate mid-point

        print(f"Step {step}: Low = {low}, High = {high}, Mid = {mid}, Value at Mid = {arr[mid]}")

        if arr[mid] == target:

            print(f"Element {target} found at index: {mid}")

            return mid  # Target found

        elif arr[mid] < target:

            low = mid + 1  # Search in the right half

        else:

            high = mid - 1  # Search in the left half

        step += 1

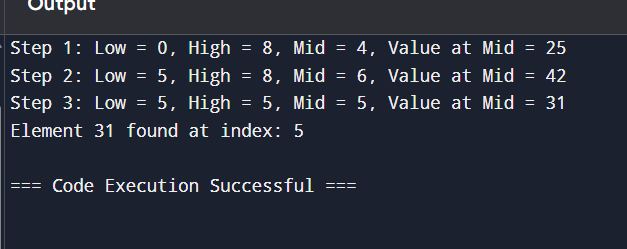
    print(f"Element {target} not found in the array.")

    return -1  # Target not found

# Call the binary search algorithm

binary\_search(array, target)

**output:-**



9.Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the k closest points to the origin (0, 0).

1. Input : points = [[1,3],[-2,2],[5,8],[0,1]],k=2
2. Output:[[-2, 2], [0, 1]]

**Program:-**

import math

import heapq

points = [[1, 3], [-2, 2], [5, 8], [0, 1], [2, -2]]

k = 3

max\_heap = []

for (x, y) in points:

    distance = x \*\* 2 + y \*\* 2  # Calculate squared distance from origin

    # Push the negative distance along with the point to create a max-heap

    heapq.heappush(max\_heap, (-distance, (x, y)))

    # If we have more than k points, remove the farthest point

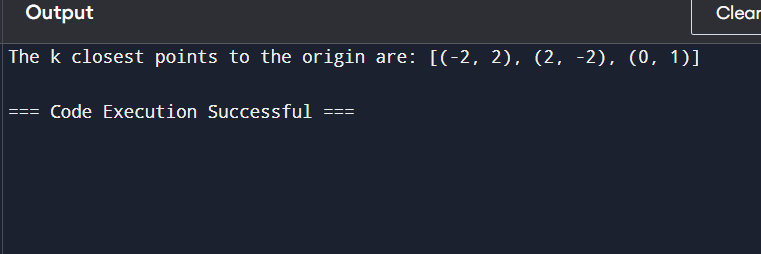
    if len(max\_heap) > k:

        heapq.heappop(max\_heap)

closest\_points = [point for (\_, point) in max\_heap]

print("The k closest points to the origin are:", closest\_points)

**output:-**



10.Given four lists A, B, C, D of integer values,Write a program to compute how many tuples n(i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.

1. Input: A = [1, 2], B = [-2, -1], C = [-1, 2], D = [0, 2]

Output: 2

1. Input: A = [0], B = [0], C = [0], D = [0]

Output: 1

**Program:-**

from collections import defaultdict

# Given lists

A = [1, 2]

B = [-2, -1]

C = [-1, 2]

D = [0, 2]

sum\_ab = defaultdict(int)

for a in A:

    for b in B:

        sum\_ab[a + b] += 1

count = 0

for c in C:

    for d in D:

        target\_sum = -(c + d)  # We want A[i] + B[j] = - (C[k] + D[l])

        count += sum\_ab[target\_sum]  # Count tuples that match

print("Number of tuples (i, j, k, l) such that A[i] + B[j] + C[k] + D[l] = 0 is:", count)

**output:-**

