**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.**

**Code:**

def find\_max\_min(arr):

max\_val = max(arr)

min\_val = min(arr)

return max\_val, min\_val

array = [3, 7, 1, 9, 4, 5]

max\_value, min\_value = find\_max\_min(array)

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

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

**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.**

**Code:**

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

max\_value = max(array)

min\_value = min(array)

print ("Maximum value in the array:", max\_value)

print ("Minimum value in the array:", min\_value)

**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.**

**Code:**

def merge\_sort(arr):

if len(arr) > 1:

mid = len(arr) // 2

left\_half = arr[:mid]

right\_half = arr[mid:]

merge\_sort(left\_half)

merge\_sort(right\_half)

i = j = k = 0

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

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

return arr

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

sorted\_arr = merge\_sort(arr)

print ("Sorted array:", sorted\_arr)

**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.**

**Code:**

def merge\_sort(arr):

comparisons = 0

if len(arr) > 1:

mid = len(arr) // 2

left\_half = arr[:mid]

right\_half = arr[mid:]

comparisons += merge\_sort(left\_half)

comparisons += merge\_sort(right\_half)

i = j = k = 0

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

comparisons += 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

return comparisons

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

comparisons = merge\_sort(arr)

print ("Sorted Array:", arr)

print ("Number of Comparisons:", comparisons)

**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.**

**Code:**

def quick\_sort(arr):

if len(arr) <= 1:

return arr

else:

pivot = arr[0]

less\_than\_pivot = [x for x in arr[1:] if x <= pivot]

greater\_than\_pivot = [x for x in arr[1:] if x > pivot]

return quick\_sort(less\_than\_pivot) + [pivot] + quick\_sort(greater\_than\_pivot)

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

print ("Initial unsorted array:", arr)

sorted\_arr = quick\_sort(arr)

print ("Array after Quick Sort:", sorted\_arr)

**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.**

**Code:**

def quick\_sort(arr):

if len(arr) <= 1:

return arr

else:

pivot = arr[len(arr) // 2]

left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

right = [x for x in arr if x > pivot]

return quick\_sort(left) + middle + quick\_sort(right)

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

sorted\_array = quick\_sort(array)

print ("Sorted Array:", sorted\_array)

**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.**

**Code:**

def binary\_search(arr, target):

low = 0

high = len(arr) - 1

count = 0

while low <= high:

mid = (low + high) // 2

count += 1

if arr[mid] == target:

return mid, count

elif arr[mid] < target:

low = mid + 1

else:

high = mid - 1

return -1, count

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

target = 20

index, comparisons = binary\_search(arr, target)

if index != -1:

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

else:

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

print(f"Number of comparisons made: {comparisons}")

**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?**

**Code:**

def binary\_search(arr, target):

low = 0

high = len(arr) - 1

while low <= high:

mid = (low + high) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

low = mid + 1

else:

high = mid - 1

return -1

sorted\_array = [3, 9, 14, 19, 25, 31, 42, 47, 53]

target\_element = 31

result = binary\_search(sorted\_array, target\_element)

print(f"Position of {target\_element} in the sorted array: {result}")

unsorted\_array = [25, 14, 53, 9, 42, 19, 3, 47, 31]

result\_unsorted = binary\_search(unsorted\_array, target\_element)

print(f"Position of {target\_element} in the unsorted array: {result\_unsorted}")

**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).**

**Code:**

import heapq

def kClosest(points, k):

heap = []

for x, y in points:

dist = -(x\*x + y\*y)

if len(heap) == k:

heapq.heappushpop(heap, (dist, x, y))

else:

heapq.heappush(heap, (dist, x, y))

return [(x, y) for (dist, x, y) in heap]

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

k = 2

print(kClosest(points, k))

**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.**

**Code:**

from collections import defaultdict

def fourSumCount(A, B, C, D):

AB\_sum = defaultdict(int)

count = 0

for a in A:

for b in B:

AB\_sum[a + b] += 1

for c in C:

for d in D:

count += AB\_sum[-c - d]

return count

A = [1, 2]

B = [-2, -1]

C = [-1, 2]

D = [0, 2]

print(fourSumCount(A, B, C, D))