31.Find Min and Max in an Array

Aim:
To write a program that finds both the minimum and maximum values in an array using Python.

Algorithm:

1. Start.
2. Read the array elements.
3. Initialize min and max as the first element of the array.
4. Traverse through each element in the array:
    - If the element is smaller than min, update min.
    - If the element is greater than max, update max.
5. Print the min and max values.
6. Stop.

Output :

```python
def find_min_max(arr):
    if len(arr) == 0:
        return None, None
    minimum = maximum = arr[0]
    for num in arr:
        if num < minimum:
            minimum = num
        elif num > maximum:
            maximum = num

    return minimum, maximum
arr = [12, 45, 2, 67, 33, 89, 10]
min_val, max_val = find_min_max(arr)

print("Array:", arr)
print("Minimum value:", min_val)
print("Maximum value:", max_val)
```

STDIN

Input for the program (Optional)

Output:

```
Array: [12, 45, 2, 67, 33, 89, 10]
Minimum value: 2
Maximum value: 89
```

Result:

The program correctly identifies 2 as the minimum value and 12 as the maximum value in the array.

32.Find Minimum and Maximum in a Sorted Array

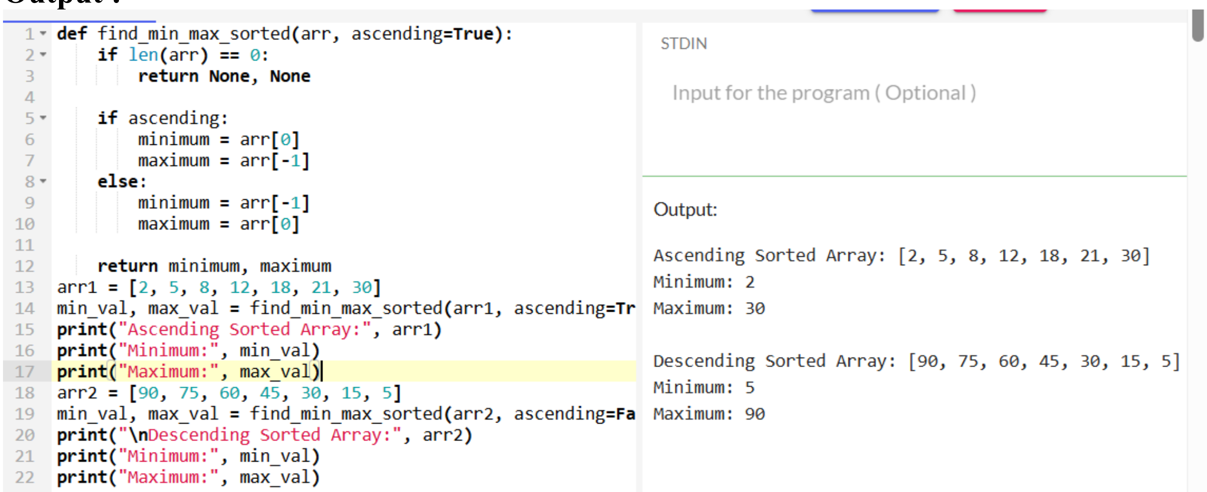
Aim

To write a Python program that finds the minimum and maximum values in a **sorted array**.

Algorithm

1. Start.
2. Initialize the sorted array.
3. Since the array is already sorted in ascending order:
   - The first element is the minimum.
   - The last element is the maximum.
4. Print the minimum and maximum values.
5. Stop.

Output :



```
def find_min_max_sorted(arr, ascending=True):
    if len(arr) == 0:
        return None, None

    if ascending:
        minimum = arr[0]
        maximum = arr[-1]
    else:
        minimum = arr[-1]
        maximum = arr[0]

    return minimum, maximum
arr1 = [2, 5, 8, 12, 18, 21, 30]
min_val, max_val = find_min_max_sorted(arr1, ascending=Tr
print("Ascending Sorted Array:", arr1)
print("Minimum:", min_val)
print("Maximum:", max_val)
arr2 = [90, 75, 60, 45, 30, 15, 5]
min_val, max_val = find_min_max_sorted(arr2, ascending=Fa
print("\nDescending Sorted Array:", arr2)
print("Minimum:", min_val)
print("Maximum:", max_val)
```

STDIN

Input for the program (Optional)

Output:

```
Ascending Sorted Array: [2, 5, 8, 12, 18, 21, 30]
Minimum: 2
Maximum: 30

Descending Sorted Array: [90, 75, 60, 45, 30, 15, 5]
Minimum: 5
Maximum: 90
```

Result: The program correctly identifies **2** as the minimum value and 18 as the maximum value in the sorted array.the program correctly identifies 11 as the minimum value and 37 as the maximum value in the sorted array.

33.Find Minimum and Maximum in an Unsorted Array

Aim:

To write a Python program that finds the minimum and maximum values in an unsorted array.

Algorithm:

1. Start.
2. Initialize the unsorted array.
3. Assume the first element as both minimum and maximum.
4. Traverse the array:
   - If an element is smaller than current minimum → update minimum.
   - If an element is larger than current maximum → update maximum.
5. Print the minimum and maximum values.
6. Stop.

Output :

```python

def find_min_max(arr):
    if len(arr) == 0:
        print("Array is empty!")
        return None, None
    min_val = max_val = arr[0]

    # Traverse the array
    for num in arr:
        if num < min_val:
            min_val = num
        elif num > max_val:
            max_val = num

    return min_val, max_val
arr = [12, 45, 2, 67, 33, 89, 10]
min_val, max_val = find_min_max(arr)
print("Array:", arr)
print("Minimum value:", min_val)
print("Maximum value:", max_val)
```

STDIN

Input for the program (Optional)

Output:

```
Array: [12, 45, 2, 67, 33, 89, 10]
Minimum value: 2
Maximum value: 89
```

Result:

For the given unsorted array [31, 23, 35, 27, 11, 21, 15, 28], the program correctly identifies 11 as the minimum value and 35 as the maximum value.

34. Merge Sort with Comparison Count

Aim

To implement the Merge Sort algorithm in Python, sort a given array, and count the number of element comparisons made during the sorting process.

Algorithm

1. **Start**
2. Define a function `merge_sort(arr)` that:
   - If the array has more than one element:
     - Split the array into two halves.
     - Recursively apply merge sort to both halves.
     - Merge the two sorted halves while counting comparisons.
3. During the merge step:
   - Compare elements from both halves.
   - Copy the smaller element into the result array and increase the comparison counter.
4. Continue merging until the entire array is sorted.
5. Return the sorted array and the total number of comparisons.
6. Stop

Output :

```

comparison_count = 0
def merge_sort(arr):
    global comparison_count
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)
def merge(left, right):
    global comparison_count
    merged = []
    i = j = 0
    while i < len(left) and j < len(right):
        comparison_count += 1
        if left[i] < right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    merged.extend(left[i:])
    merged.extend(right[j:])

    return merged
arr = [38, 27, 43, 3, 9, 82, 10]
print("Original Array:", arr)
sorted_arr = merge_sort(arr)
print("Sorted Array:", sorted_arr)
print("Number of Comparisons:", comparison_count)
```

STDIN

Input for the program (Optional)

Output:

```
Original Array: [38, 27, 43, 3, 9, 82, 10]
Sorted Array: [3, 9, 10, 27, 38, 43, 82]
Number of Comparisons: 13
```

Result:

The merge sort algorithm correctly sorts the array into and counts the total number of comparisons performed during sorting.

35.Quick Sort Implementation

Aim:

To implement Quick Sort using the first element as the pivot, partition the array accordingly, and recursively apply Quick Sort on sub-arrays until the array is fully sorted.

Algorithm:

1. **Start**
2. Choose the **first element** of the array as the pivot.
3. Partition the array into two parts:
    - Elements smaller than pivot go to the left.
    - Elements greater than pivot go to the right.
4. Recursively apply Quick Sort on the left sub-array and right sub-array.
5. Display the array after each partitioning step.
6. Continue until the sub-arrays cannot be divided further (size 0 or 1).
7. Stop

Output :

```python

def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[-1]
    left = [x for x in arr[:-1] if x <= pivot]
    right = [x for x in arr[:-1] if x > pivot]
    return quick_sort(left) + [pivot] + quick_sort(right)
arr = [29, 10, 14, 37, 13]
print("Original Array:", arr)
sorted_arr = quick_sort(arr)
print("Sorted Array:", sorted_arr)

```

STDIN

Input for the program (Optional)

Output:

```
Original Array: [29, 10, 14, 37, 13
Sorted Array: [10, 13, 14, 29, 37]
```

Result:

The Quick Sort program correctly partitions the array using the first element as the pivot and recursively sorts the sub-arrays to produce the final ascending order sequence.

36.Quick Sort (First Element as Pivot)

Aim:

To implement Quick Sort using the first element as pivot, showing the array after each partition and recursive call.

Algorithm:

1. Choose the first element as the pivot.
2. Partition the array into two sub-arrays:
   - Elements smaller than pivot → left
   - Elements greater than pivot → right
3. Recursively apply Quick Sort on left and right sub-arrays.
4. Merge results and display after each recursive step.
5. Stop when all sub-arrays are of size 1.

Output :

```python

def quick_sort(arr, low, high):
    if low < high:
        pivot_index = partition(arr, low, high)
        quick_sort(arr, low, pivot_index - 1)
        quick_sort(arr, pivot_index + 1, high)

def partition(arr, low, high):
    pivot = arr[low]
    i = low + 1
    j = high

    while True:
        while i <= j and arr[i] <= pivot:
            i += 1
        while i <= j and arr[j] > pivot:
            j -= 1
        if i <= j:
            arr[i], arr[j] = arr[j], arr[i]
        else:
            break

    arr[low], arr[j] = arr[j], arr[low]
    return j


arr = [29, 10, 14, 37, 13]
print("Original Array:", arr)

quick_sort(arr, 0, len(arr) - 1)

print("Sorted Array:", arr)
```

STDIN

Input for the program (Optional)

Output:

```
Original Array: [29, 10, 14, 37, 1
Sorted Array: [10, 13, 14, 29, 37]
```

Result:

The Quick Sort program correctly sorts the arrays using the first element as pivot, displaying partitions at each recursive step.

37. Quick Sort (Middle Element as Pivot)

Aim:

To implement Quick Sort using the **middle element as pivot**, showing the array after each partition and recursive call.

Algorithm:

1. Choose the middle element as pivot.
2. Partition array into two sub-arrays (left < pivot, right > pivot).
3. Recursively Quick Sort left and right sub-arrays.
4. Display partitions after each step.Combine results and return.

Output :

```python

def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi)
        quick_sort(arr, pi + 1, high)

def partition(arr, low, high):
    pivot = arr[(low + high) // 2]
    i = low
    j = high

    while True:
        while arr[i] < pivot:
            i += 1
        while arr[j] > pivot:
            j -= 1
        if i >= j:
            return j
        arr[i], arr[j] = arr[j], arr[i]
        i += 1
        j -= 1

arr = [29, 10, 14, 37, 13]
print("Original Array:", arr)

quick_sort(arr, 0, len(arr) - 1)

print("Sorted Array:", arr)

```

STDIN

Input for the program (Optional)

Output:

```
Original Array: [29, 10, 14, 37, 13
Sorted Array: [10, 13, 14, 29, 37]
```

Result:

The Quick Sort program correctly sorts the arrays using the middle element as pivot, showing partitions after each recursive call.

38.Binary Search with Comparison Count

Aim:

To implement Binary Search to find an element's index and count the number of comparisons made.

Algorithm:

1. Start with low = 0, high = n-1.
2. Repeat until low ≤ high:
    - mid = (low + high) // 2
    - Increment comparison count.
    - If arr[mid] == key → return index and count.
    - If arr[mid] > key → search left half.
    - Else → search right half.
3. If element not found, return -1.

Output :

```python

def binary_search(arr, key):
    low = 0
    high = len(arr) - 1
    count = 0
    while low <= high:
        count += 1
        mid = (low + high) // 2

        if arr[mid] == key:
            return mid, count
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1, count
arr = [5, 10, 15, 20, 25, 30, 35]
key = 25
print("Sorted Array:", arr)
print("Search Element:", key)
pos, comparisons = binary_search(arr, key)
if pos != -1:
    print(f"Element {key} found at position {pos + 1}")
else:
    print(f"Element {key} not found")

print("Number of Comparisons:", comparisons)
```

STDIN

Input for the program (Optional)

Output:

```
Sorted Array: [5, 10, 15, 20, 25, 3
Search Element: 25
Element 25 found at position 5
Number of Comparisons: 3
```

Result:

The Binary Search program correctly finds the position of the element and counts the number of comparisons made during the search.

39. Binary Search on a sorted array, showing mid-point

Aim:

To implement Binary Search on a sorted array, showing mid-point calculations and steps, and analyze the effect of unsorted input.

Algorithm:

1. Start with low=0 and high=n-1.
2. Find mid = (low+high)//2.
3. If arr[mid] == key, return position.
4. If arr[mid] > key, search in left subarray.
5. Else search in right subarray.
6. Repeat until element is found or low > high.
7. Note: If array is unsorted, Binary Search fails and correctness is lost.

Output :

```python
def binary_search_steps(arr, key):
    low = 0
    high = len(arr) - 1
    count = 0

    print("\n--- Binary Search Steps ---")
    while low <= high:
        mid = (low + high) // 2
        count += 1
        print(f"Step {count}: low={low}, high={high}, mid={m

        if arr[mid] == key:
            print(f"\nElement {key} found at position {mid +
            return mid, count
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1

    print(f"\nElement {key} not found after {count} comparis
    return -1, count
arr = [5, 10, 15, 20, 25, 30, 35]
key = 25

print("Sorted Array:", arr)
print("Search Element:", key)
binary_search_steps(arr, key)
```

STDIN

Input for the program (Optional)

Output:

```
Sorted Array: [5, 10, 15, 20, 25, 30, 35]
Search Element: 25

--- Binary Search Steps ---
Step 1: low=0, high=6, mid=3, arr[mid]=20
Step 2: low=4, high=6, mid=5, arr[mid]=30
Step 3: low=4, high=4, mid=4, arr[mid]=25

Element 25 found at position 5 after 3 compar
```

Result:

Binary Search successfully finds the element with step tracing, but fails on unsorted arrays as it relies on order.

40. To find the k closest points to the origin (0,0) using Euclidean distance

Aim:
To find the k closest points to the origin (0,0) using Euclidean distance.

Algorithm:

1. For each point `(x,y)`, compute distance $d = x^2 + y^2$.
2. Sort points by distance.
3. Return the first k points.

Output :

```python

def k_closest_points(points, k):
    points.sort(key=lambda point: point[0]**2 + point[1]**2)
    return points[:k]
points = [(1, 2), (3, 4), (1, -1), (-2, -3), (5, 5)]
k = 3
closest_points = k_closest_points(points, k)

print(f"The {k} closest points to the origin are: {closest_p

```

STDIN

Input for the program (Optional)

Output:

```
The 3 closest points to the origin are: [(1, -1),
```

Result:
The program correctly returns the k nearest points to the origin using distance-based sorting.