Quicksort Algorithm Mini Project

A quick-sorting method called quicksort divides a huge data array into smaller sub-arrays. By dividing the input into two parts, sorting them, and then recombining them, it is implied that each iteration operates in this manner. The approach is very effective for large datasets since its average and best-case complexity are both O(n\*logn). Divide-and-conquer is the foundation of Quicksort.

Tony Hoare developed it in 1961, and it is still among the best all-purpose sorting algorithms on the market. It operates by dynamically moving list elements around a specified pivot while recursively sorting the sub-lists on either side of the pivot.

As a result, the quicksort method can be summarized in three steps:

1. Divide: Divides the problem into smaller sub-problems by partitioning the array into two smaller subarrays around pivot
2. Conquer: Solves both sub-problems and sort both smaller arrays recursively.
3. Combine: Combines the arrays that have previously been sorted.

**Benefits of Quicksort**

* It works rapidly and effectively.
* It has the best time complexity when compared to other sorting algorithms.
* Quick sort has a space complexity of O(logn), making it an excellent choice for situations when space is limited.

**Limitations of Quicksort**

* This sorting technique is considered unstable since it does not maintain the key-value pairs initial order.
* When the pivot element is the largest or smallest, or when all the components have the same size. The performance of the quicksort is significantly impacted by these worst-case scenarios.
* It’s difficult to implement since it’s a recursive process, especially if recursion isn’t available.

**How to Implement Quicksort**

1. Choose a pivot (usually highest-index item).
2. Create a left reference, pointing to element at the lowest index.
3. Create a right reference, pointing to the element at highest index (not pivot).
4. While left reference is less than the pivot, move the pointer one element to the right. While right reference is greater than pivot, move the pointer one element to the left.
5. If both left reference is greater than pivot and right reference is smaller than pivot, swap the elements at the two references.
6. Once the index of the left reference. is greater than (or equal to) the index of the right reference, swap the pivot with the element at the left reference.

The following procedure implements quicksort:

quicksort(A, p, r)

1 if p < r

2 q = partition(A, p, r)

3 quicksort(A, p, q-1)

4 quicksort(A, q+1,r)

The key to the algorithm is the PARTITION procedure, which rearranges the subarray A[p..r] in place.

Partition(A, p, r)

1. x=A[r]
2. i=p – 1
3. for j = p to r – 1

if A[j] <= x

i = i+ 1

exchange A[i] with A[j]

1. exchange A[ i + 1] with A[r]
2. return i+1

**Python Program for Quicksort Algorithm**

**# Quick sort in Python**

**# function to find the partition position**

**def partition(array, low, high):**

**# choose the rightmost element as pivot**

**pivot = array[high]**

**# pointer for greater element**

**i = low - 1**

**# traverse through all elements**

**# compare each element with pivot**

**for j in range(low, high):**

**if array[j] <= pivot:**

**# if element smaller than pivot is found**

**# swap it with the greater element pointed by i**

**i = i + 1**

**# swapping element at i with element at j**

**(array[i], array[j]) = (array[j], array[i])**

**# swap the pivot element with the greater element specified by i**

**(array[i + 1], array[high]) = (array[high], array[i + 1])**

**# return the position from where partition is done**

**return i + 1**

**# function to perform quicksort**

**def quickSort(array, low, high):**

**if low < high:**

**# find pivot element such that**

**# element smaller than pivot are on the left**

**# element greater than pivot are on the right**

**pi = partition(array, low, high)**

**# recursive call on the left of pivot**

**quickSort(array, low, pi - 1)**

**# recursive call on the right of pivot**

**quickSort(array, pi + 1, high)**

**data = [8, 7, 2, 1, 0, 9, 6]**

**print("Unsorted Array")**

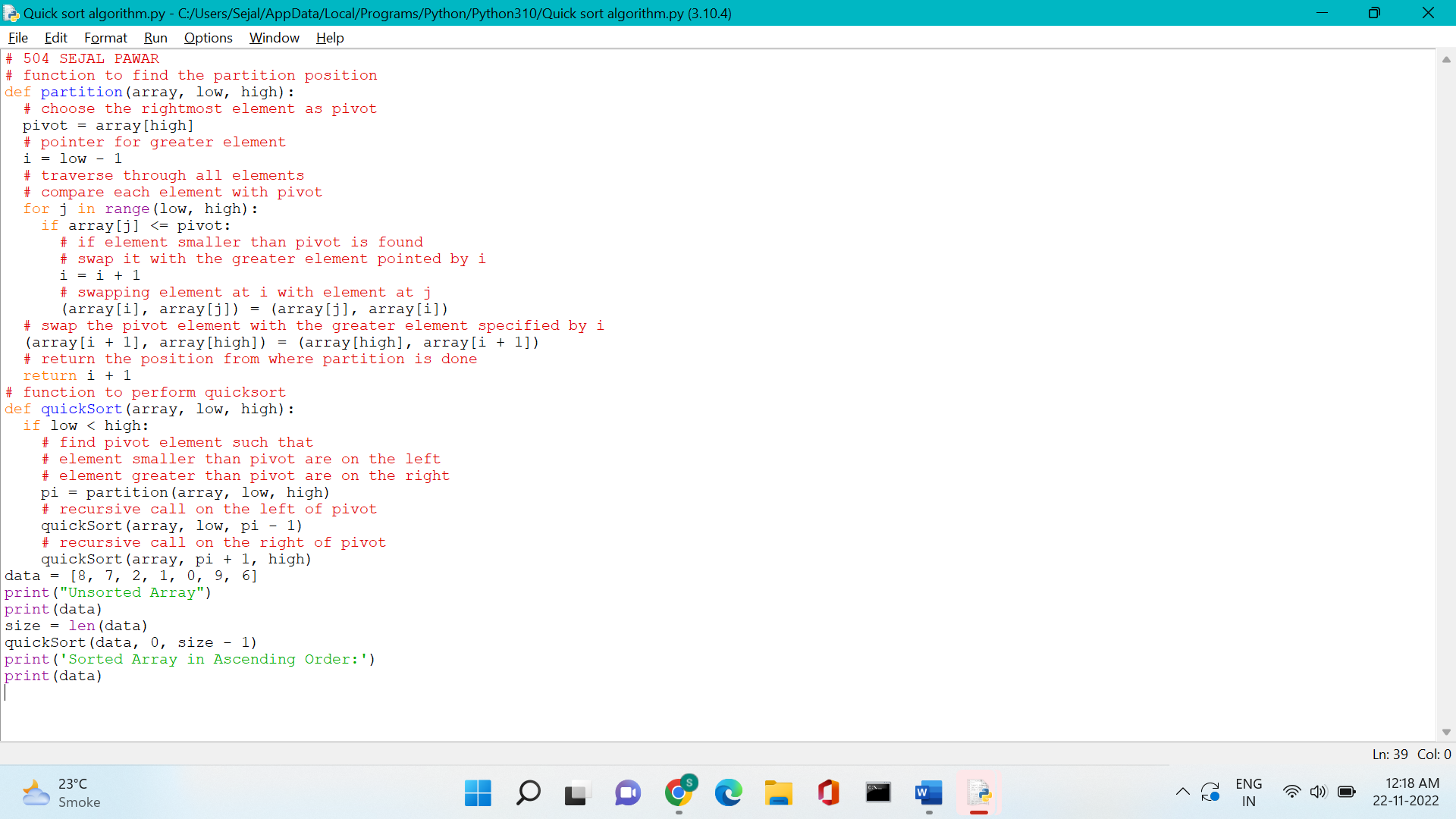
**print(data)**

**size = len(data)**

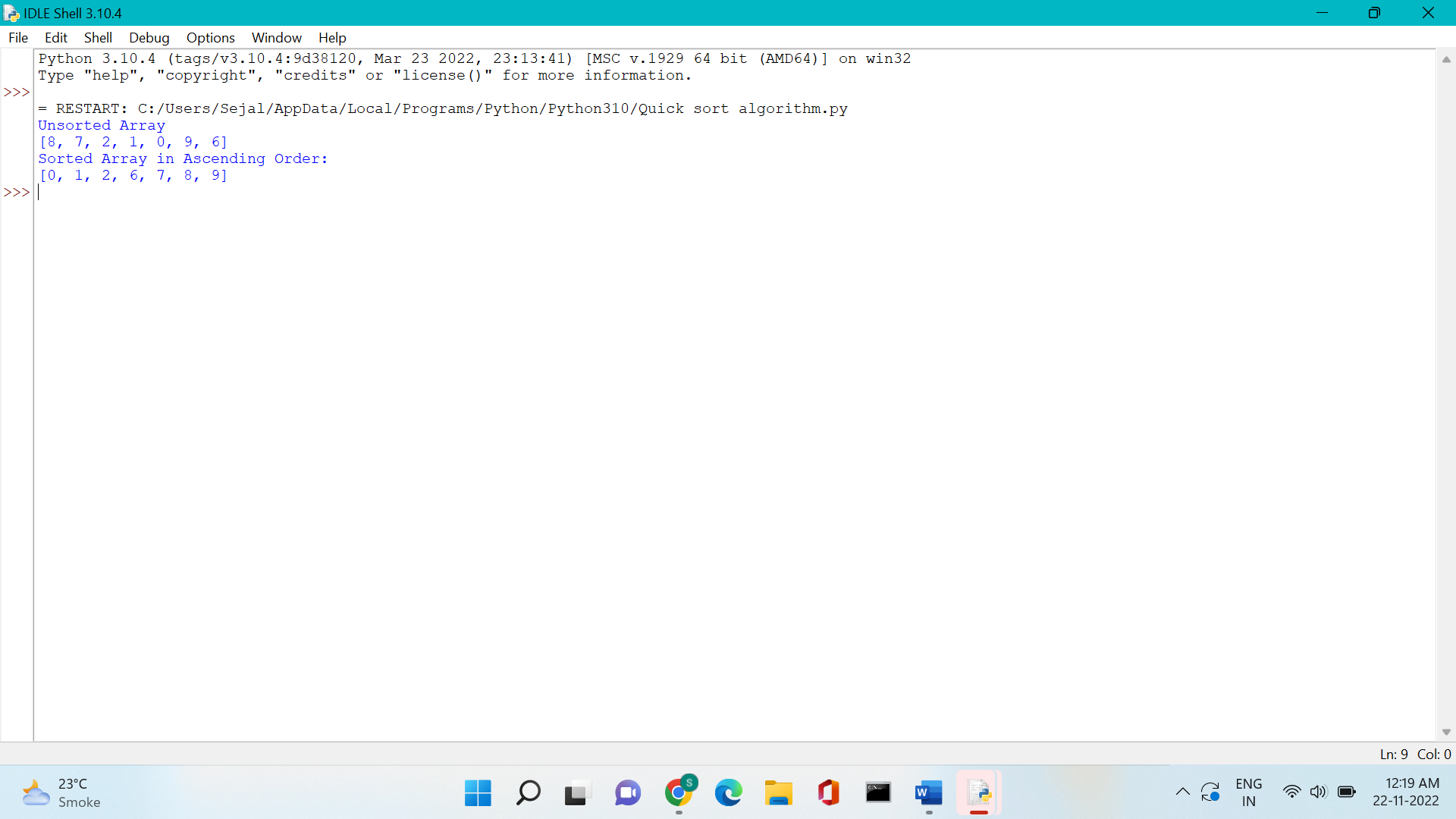
**quickSort(data, 0, size - 1)**

**print('Sorted Array in Ascending Order:')**

**print(data)**



OUTPUT :



## Quicksort Complexity

|  |  |
| --- | --- |
| **Time Complexity** |  |
| Best | O(n\*log n) |
| Worst | O(n2) |
| Average | O(n\*log n) |
| **Space Complexity** | O(log n) |
| **Stability** | No |

* **Worst Case Complexity [Big-O]**: O(n2)  
  It occurs when the pivot element picked is either the greatest or the smallest element.  
    
  This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 elements. Thus, quicksort is called only on this sub-array.  
    
  However, the quicksort algorithm has better performance for scattered pivots.
* **Best Case Complexity [Big-omega]**: O(n\*log n)  
  It occurs when the pivot element is always the middle element or near to the middle element.
* **Average Case Complexity [Big-theta]**: O(n\*log n)  
  It occurs when the above conditions do not occur.

### 2. Space Complexity

The space complexity for quicksort is O(log n).

**Quicksort Applications** :

Quicksort algorithm is used when

* the programming language is good for recursion
* time complexity matters
* space complexity matters