|  |  |
| --- | --- |
|  | DEPARTMENT OF ARTIFICIAL INTELLIGNECE & DATA SCIENCE |

|  |  |
| --- | --- |
| Subject: Analysis of Algorithm | Course Code: CSC402 |
| Semester: 4 | Course: AI & DS |
| Laboratory No: 205 | Name of Subject Teacher: Pramod Bhavarthe |
| Name of Student: Sahil Shaikh | Roll Id: VU2S2223012 |

|  |  |
| --- | --- |
| Title of Practical | Implement Merge sort. |

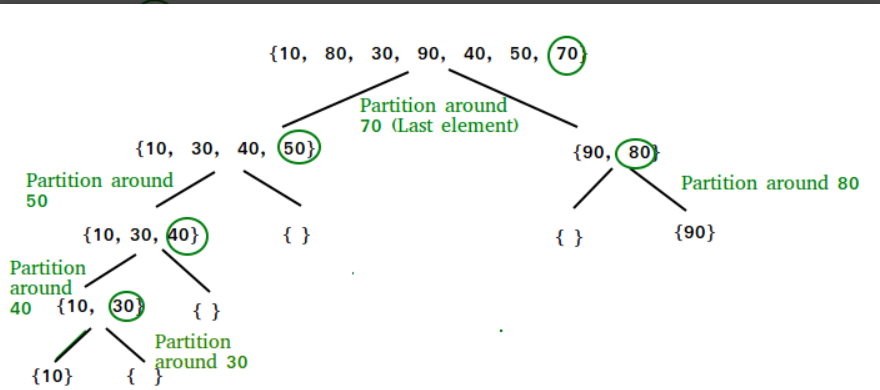
**Theory –**

**Quick Sort**is a[**Divide and Conquer algorithm**](https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/). It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivot in different ways.

* Always pick the first element as a pivot.
* Always pick the last element as a pivot (implemented below)
* Pick a random element as a pivot.
* Pick median as the pivot.

The key process in **quicksort**is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

## **Quick Sort Working Process:**

****

**Program –**

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

    # e 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 quick\_sort(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

        quick\_sort(array, low, pi - 1)

        # Recursive call on the right of pivot

        quick\_sort(array, pi + 1, high)

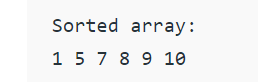
# Driver code

array = [10, 7, 8, 9, 1, 5]

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

print(f'Sorted array: {array}')

**Output –**

****

### ****Analysis of Quick Sort****

**Worst Case:**

**T(n) = T(0) + T(n-1) + (n)which is equivalent to  T(n) = T(n-1) + (n)**

**The solution to the above recurrence is  (n2).**

**Best Case:**

**T(n) = 2T(n/2) + (n)**

**Average Case:**

**T(n) = T(n/9) + T(9n/10) + (n)**

**Is Quick Sort**[**stable**](https://www.geeksforgeeks.org/stability-in-sorting-algorithms/)**?**

* The default implementation is not stable. However, any sorting algorithm can be made stable by considering indexes as comparison parameter.

**Is Quick Sort**[**In-place**](https://www.geeksforgeeks.org/in-place-algorithm/)**?**

* As per the broad definition of in-place algorithm it qualifies as an in-place sorting algorithm as it uses extra space only for storing recursive function calls but not for manipulating the input.

**Conclusion –**

**Therefore, we have successfully understood and Implemented Quick sort.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Grade and Dated Signature of Teacher** | **Total (10)** | **Remark** | **Dated signature of teacher** |
|  |  |  |