INTORDUCTION TO ALGORITHMS – EC351

ASSIGNMENT – 3

1. Find out Time complexity for the arrays using Quick Sorting and Merge Sorting Algorithms

Sol :

Merge Sorting Algorithms :

CODE :

```
from datetime import datetime
start_time = datetime.now()
def merge_sort(arr, begin, end):

    if end - begin > 1:
        middle = (begin + end)//2
        merge_sort(arr, begin, middle)
        merge_sort(arr, middle, end)
        merge_list(arr, begin, middle, end)
 def merge_list(arr, begin, middle, end):
    left = arr[begin:middle]
    right = arr[middle:end]
    k = begin
    i = 0
    j = 0
    while (begin + i < middle and middle + j < end):
        if (left[i] <= right[j]):
            arr[k] = left[i]
            i = i + 1
        else:
            arr[k] = right[j]
            j = j + 1
        k = k + 1
    if begin + i < middle:
        while k < end:
            arr[k] = left[i]
            i = i + 1
            k = k + 1
    else:
        while k < end:
            arr[k] = right[j]
            j = j + 1
            k = k + 1
 arr = input('Enter the list of numbers: ').split()
arr = [float(x) for x in arr]
merge_sort(arr, 0, len(arr))
print('Sorted list: ', end=")
print(arr)
end_time = datetime.now()
print('Duration : {}'.format(end_time - start_time))
```

ALGORITHM :

```
MergeSort(arr[], l,  r)
If r > l
     STEP 1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
    STEP 2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
```

STEP 3. Call mergeSort for second half:

```
Call mergeSort(arr, m+1, r)
```

STEP 4. Merge the two halves sorted in step 2 and 3:

```
Call merge(arr, l, m, r)
```

INPUT ARRAY :

1. A [2.5, 4.5, 3.0, 1.2, 6.5, 8.9, 7.4, 6.3]

OUTPUT :

Enter the list of numbers: 2.5 4.5 3.0 1.2 6.5 8.9 7.4 6.3

Sorted list: [1.2, 2.5, 3.0, 4.5, 6.3, 6.5, 7.4, 8.9]

2. B [5 3 6 3 4 5 4 6 4]

Enter the list of numbers: B [5 3 6 3 4 5 4 6 4]

Sorted list: [3.0, 3.0, 4.0, 4.0, 4.0, 5.0, 5.0, 6.0, 6.0]

Time complexity For Merge Sorting Algorithms :

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

T(n) = 2T(n/2) + O(n)

Time complexity of Merge Sort is **O(nlogn)** in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

Quick Sorting Algorithm :

CODE :

```
from datetime import datetime

start_time = datetime.now()

def quicksort(arr, begin, end):


    if end - begin > 1:

        p = partition(arr, begin, end)
        quicksort(arr, begin, p)
        quicksort(arr, p + 1, end)
 def partition(arr, begin, end):
    pivot = arr[begin]
    i = begin + 1
    j = end - 1

    while True:
        while (i <= j and arr[i] <= pivot):
            i = i + 1
        while (i <= j and arr[j] >= pivot):
            j = j - 1

        if i <= j:
           arr[i], arr[j] = arr[j], arr[i]
        else:
            arr[begin], arr[j] = arr[j], arr[begin]
            return j
 arr = input('Enter the list of numbers to be Sorted: ').split()
arr = [float(x) for x in arr]
quicksort(arr, 0, len(arr))
print('Sorted list: ', end='')
print(arr)
end_time = datetime.now()
print('Duration : {}'.format(end_time - start_time))
```

ALGORITHM :

Quick Sort Pivot Algorithm :

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

QUICK SORT ALGORITHM :

Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

INPUT ARRAY :

1. A [2.5, 4.5, 3.0, 1.2, 6.5, 8.9, 7.4, 6.3]

OUTPUT :

Enter the list of numbers to be sorted: 2.5 4.5 3.0 1.2 6.5 8.9 7.4 6.3

Sorted list: [1.2, 2.5, 3.0, 4.5, 6.3, 6.5, 7.4, 8.9]

2. B [5 3 6 3 4 5 4 6 4]

Enter the list of numbers to be sorted: B [5 3 6 3 4 5 4 6 4]

Sorted list: [3.0, 3.0, 4.0, 4.0, 4.0, 5.0, 5.0, 6.0, 6.0]

TIME COMPLEXITY OF QUICK SORT ALGORITHM :

Best case :

To find the location of an element that splits the array into two parts, O(n) operations are required.

- This is because every element in the array is compared to the partitioning element.
- After the division, each section is examined separately.
- If the array is split approximately in half (which is not usually), then there will be logn splits.
- Therefore, total comparisons required are f(n) = n x logn = O(nlogn).
- Order of Quick Sort in best case = O(nlogn).

Worst Case :

Quick Sort is sensitive to the order of input data.

- It gives the worst performance when elements are already in the ascending order.
- It then divides the array into sections of 1 and (n-1) elements in each call.
- Then, there are (n-1) divisions in all.
- Therefore, here total comparisons required are f(n) = n x (n-1) = $O(n^2)$.
- Order of Quick Sort in worst case = $O(n^2)$

2. Find out Arrays Sorting program execution time using python or C++.

Sol : Execution time using Python : -

Quick sort algorithm execution time :

1.

```
Enter the list of numbers to be Sorted: 2.5 4.5 3.0 1.2 6.5 8.9 7.4 6.3
Sorted list: [1.2, 2.5, 3.0, 4.5, 6.3, 6.5, 7.4, 8.9]
Duration : 0:00:23.270743
```

2.

```
Enter the list of numbers to be Sorted: 5 3 6 3 4 5 4 6 4
Sorted list: [3.0, 3.0, 4.0, 4.0, 4.0, 5.0, 5.0, 6.0, 6.0]
Duration : 0:00:37.650884
```

Merge sort algorithm execution time :

1.

```
Enter the list of numbers: 2.5 4.5 3.0 1.2 6.5 8.9 7.4 6.3
Sorted list: [1.2, 2.5, 3.0, 4.5, 6.3, 6.5, 7.4, 8.9]
Duration : 0:00:32.074479
```

2.

```
Enter the list of numbers: 5 3 6 3 4 5 4 6 4
Sorted list: [3.0, 3.0, 4.0, 4.0, 4.0, 5.0, 5.0, 6.0, 6.0]
Duration : 0:00:26.142812
```

BY
Y SANTHI SWARUP
18BEC051