Introduction To Algorithms

EC351

Assignment 3

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

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

To do tasks :

1. Find out Time complexity for the arrays using Quick

Sorting and Merge Sorting Algorithms

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

Quick sort algorithm:

Pseudo code:

```
Partition (Arr, low, high)
 {
    pivot = A[low]
    i = low;
    for ( j = low +1,  j < high,  j++)
    {
        if ( Arr[j] <  pivot )
        {
              i++;
            swap (A[i] A[j]);
          }
     }
    swap (A[pivot], A[low] );        # Pivot sorted position
    return i;
}
Quick Sort(Arr[], low, high)
{ if (low ==  high)
       return Arr[low]
  else
        {                  /* K is partitioning index, arr[K] is now at right place */
                i = partition(Arr, low, high);
                QuickSort(Arr, low, i - 1);           // Before i
                QuickSort(Arr, i + 1, high);        // After i
                return Arr
         }
}
```

Time complexity:

Merge sort is a sorting technique based on divide and conquer technique. It is one of the fastest of sorting algorithms. It picks an element as pivot and partitions the given array around the picked pivot by putting the smaller element on left of the pivot and larger element on right of the pivot. Then this process is repeated by picking a new pivot for the sub array on the left of the pivot and the right of the pivot until the whole array is sorted.

Worst case: $\mathbf{O(n^2)}$

Average case: $\mathbf{O(\log_2 n)}$

Best case: $\mathbf{O(\log_2 n)}$

Based on its complexity it is clear that it is an adaptive algorithm.

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

Python Code:

```
import time
def partition(A, low, high):
    i = (low-1)
    pivot = A[high]
    for j in range(low, high):
        if A[j] <= pivot:
            i = i+1
            A[i], A[j] = A[j], A[i]
    A[i+1], A[high] = A[high], A[i+1]
    return (i+1)

def quickSort(A, low, high):
    if len(A) == 1:
        return A
    if low < high:
        pi = partition(A, low, high)
        quickSort(A, low, pi-1)
        quickSort(A, pi+1, high)

A = [2.5, 4.5, 3.0, 1.2, 6.5, 8.9, 7.4, 6.3]
start_time = time.time()
n = len(A)
quickSort(A, 0, n-1)
print("Sorted array is:")
print(A)
end_time = time.time()
print(end_time-start_time)
```

Output:

Sorted array is:
[1.2, 2.5, 3.0, 4.5, 6.3, 6.5, 7.4, 8.9]
Execution time for this program is :
0.000244140625

Observation: The array has been sorted and its execution time is

0.000244140625 second.

2)B[5, 3, 6, 3, 4, 5, 4, 6, 4]

Python Code:

```
import time
def partition(B, low, high):
    i = (low-1)
    pivot = B[high]
    for j in range(low, high):
        if B[j] <= pivot:
            i = i+1
            B[i], B[j] = B[j], B[i]
    B[i+1], B[high] = B[high], B[i+1]
    return (i+1)

def quickSort(B, low, high):
    if len(B) == 1:
        return B
    if low < high:
        pi = partition(B, low, high)
        quickSort(B, low, pi-1)
        quickSort(B, pi+1, high)
```

```
B = [2.5, 4.5, 3.0, 1.2, 6.5, 8.9, 7.4, 6.3]
start_time = time.time()
n = len(B)
quickSort(B, 0, n-1)
print("Sorted array is:")
print(B)
end_time = time.time()
print(end_time-start_time)
```

Output:

```
Sorted array is:
[3, 3, 4, 4, 4, 5, 5, 6, 6]
Execution time for this program is :
0.00023865699768066406
```

Observation: The array has been sorted and its execution time is 0.00023865699768066406 second.

Merge sort algorithm:

Pseudo code:

```
mergeSort(arr)
{
	if (n==1)
		return arr
	l=arr[0 to n/2]
	r=arr[n/2+1 to n]
	l=mergeSort(l1)
	r=mergeSort(l2)
	return merge(l,r)
}
merge(l,r)
{
	while (i<n1 and j<n2)
	{
		if (l[i]>r[j])
			add r[j] to end of array c
			j++
		else
			add l[i] to end of array c
			i++
	}
	while (i<n1)
		add remaining elements of l to end of c
```

```
    while (j<n2)
        add remaining elements of r to end of c
    return c
}
```

Time complexity:

Merge sort is a sorting technique based on divide and conquer technique. In this algorithm the output is divided into 2 sub arrays until the size of the sub array is not equal to 1. Then the elements of the sub array are compared and placed in ascending order in a new array and returned until all the sub arrays are sorted.

Worst case: $\mathbf{O(n * \log_2 n)}$

Average case: $\mathbf{O(n * \log_2 n)}$

Best case: $\mathbf{O(n * \log_2 n)}$

Based on its complexity it is clear that it is a non-adaptive algorithm.

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

Python Code:

```
import time
def merge(A, l, m, r):
    n1 = m - l + 1
    n2 = r- m
    L = [0] * (n1)
    R = [0] * (n2)
    for i in range(0 , n1):
        L[i] = A[l + i]
    for j in range(0 , n2):
        R[j] = A[m + 1 + j]
    i = 0
    j = 0
    k = l
    while i < n1 and j < n2 :
        if L[i] <= R[j]:
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1
        k += 1
    while i < n1:
        A[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        A[k] = R[j]
        j += 1
        k += 1
```

```
def mergeSort(A,l,r):
    if l < r:
        m = (l+(r-1))//2
        mergeSort(A, l, m)
        mergeSort(A, m+1, r)
        merge(A, l, m, r)

A = [2.5, 4.5, 3.0, 1.2, 6.5, 8.9, 7.4, 6.3]
start_time=time.time()
n = len(A)
print ("Given array is")
print(A)
mergeSort(A,0,n-1)
print ("\nSorted array is")
print(A)
end_time=time.time()
print('\nExecution time for this program is:')
print(end_time-start_time)
```

Output:

```
Given array is
[2.5, 4.5, 3.0, 1.2, 6.5, 8.9, 7.4, 6.3]

Sorted array is
[1.2, 2.5, 3.0, 4.5, 6.3, 6.5, 7.4, 8.9]

Execution time for this program is:
0.0005788803100585938
```

Observation: The array has been sorted and its execution time is 0.0005788803100585938 second. Also the sorting of this array using merge sort took a little longer time compared to quick sort.

2)B[5, 3, 6, 3, 4, 5, 4, 6, 4]

Python Code:

```
import time
def merge(B, l, m, r):
    n1 = m - l + 1
    n2 = r- m
    L = [0] * (n1)
    R = [0] * (n2)
    for i in range(0 , n1):
        L[i] = B[l + i]
    for j in range(0 , n2):
        R[j] = B[m + 1 + j]
    i = 0
    j = 0
    k = l
    while i < n1 and j < n2 :
        if L[i] <= R[j]:
            B[k] = L[i]
            i += 1
        else:
            B[k] = R[j]
            j += 1
        k += 1
    while i < n1:
        B[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        B[k] = R[j]
        j += 1
        k += 1
```

```
def mergeSort(B,l,r):
    if l < r:
        m = (l+(r-1))//2
        mergeSort(B, l, m)
        mergeSort(B, m+1, r)
        merge(B, l, m, r)

B = [5, 3, 6, 3, 4, 5, 4, 6, 4]
start_time=time.time()
n = len(B)
print ("Given array is")
print(B)
mergeSort(B,0,n-1)
print ("\nSorted array is")
print(B)
end_time=time.time()
print('\nExecution time for this program is:')
print(end_time-start_time)
```

Output:

```
Given array is
[5, 3, 6, 3, 4, 5, 4, 6, 4]

Sorted array is
[3, 3, 4, 4, 4, 5, 5, 6, 6]

Execution time for this program is:
0.00055575370788574222
```

Observation: The array has been sorted and its execution time is 0.00055575370788574222 second. Also the sorting of this array using merge sort took a little longer time compared to quick sort.