K. J. Somaiya College of Engineering, Mumbai-77

(Autonomous College Affiliated to University of Mumbai)

Batch: B1 **Roll No.: 1711072**

Experiment No. 3

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

Title: Implementation of Quick sort/Merge sort algorithm

Objective: To learn the divide and conquer strategy of solving the problems of different types

CO to be achieved:

Sr. No	Objective
CO 1	Compare and demonstrate the efficiency of algorithms using asymptotic complexity notations.
CO 2	Analyze and solve problems for divide and conquer strategy, greedy method, dynamic programming approach and backtracking and branch & bound policies.
CO 3	Analyze and solve problems for different string matching algorithms.

Books/ Journals/ Websites referred:

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran," Fundamentals of computer algorithm", University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein," Introduction to algorithms",2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://en.wikipedia.org/wiki/Quicksort**
4. **https://www.cs.auckland.ac.nz/~jmor159/PLDS210/qsort.html**
5. **http://www.cs.rochester.edu/~gildea/csc282/slides/C07-quicksort.pdf**
6. **http://www.sorting-algorithms.com/quick-sort**
7. **http://www.cse.ust.hk/~dekai/271/notes/L01a/quickSort.pdf**

8. **http://en.wikipedia.org/wiki/Merge_sort**
9. **http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm**
10. **http://www.sorting-algorithms.com/merge-sort**
11. **http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Merge_sort.html**

Pre-Lab/ Prior Concepts:
Data structures, various sorting techniques

Historical Profile:
Quicksort and merge sort are s a divide-and-conquer sorting algorithm in which division is dynamically carried out. They are one the most efficient sorting algorithms.

New Concepts to be learned:
Number of comparisons, Application of algorithmic design strategy to any problem, Classical problem solving Vs Divide-and-Conquer problem solving.

Algorithm Recursive Quick Sort:

```
void quicksort( Integer A[ ], Integer left, Integer right)
//sorts A[left.. right] by using partition() to partition A[left.. right], and then //calling itself //
twice to sort the two subarrays.
{ IF ( left < right ) then
        {       q = partition( A, left, right);
                quicksort( A, left, q–1);
                quicksort( A, q+1, right);

        }

}
```

Integer *partition(integer A*T[], Integer *left,* Integer *right*)
//This function rarranges *A*[*left*..*right*] and finds and returns an integer *q*, such that *A*[*left*], ..., //*A*[*q*–1] <~ *pivot*, *A*[*q*] = *pivot*, *A*[*q*+1], ..., *A*[*right*] > *pivot*, where *pivot* is the first element of //a[left..right], before partitioning**.**

```
{
pivot = A[left]; lo = left+1; hi = right;
WHILE ( lo ≤ hi )
{       WHILE ( A[hi] > pivot )                         hi = hi – 1;
        WHILE ( lo ≤ hi and A[lo] <~pivot )             lo = lo + 1;
        IF ( lo ≤ hi ) then                     swap( A[lo], A[hi]);
}
swap( pivot, A[hi]);
  RETURN hi;

}
```

The space complexity of QuickSort:

The space complexity is O(logn)

Derivation of best case and worst case time complexity (Quick Sort)
Worst case

n

0 n-1

0 n-2

0 n-3

0 n-4

The comparisons then become kn(n-1)/2 where k is the number of levels.

The complexity for worst case is **$O(n^2)$**

Best/Avg case

n

~n/2 ~n/2

~n/4 ~n/4

~ n/8 ~n/8

The height of the tree is $\log_2 n$.

Complexity is **$O(n\log_2 n)$**.

Algorithm MergeSort

MERGE-SORT (A, p, r)

// To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (A, //1, n). Array A and indices p, q, r such that $p \leq q \leq$ r and subarray $A[p \,..\, q]$ is sorted and subarray //$A[q + 1 \,..\, r]$ is sorted. By restrictions on p, q, r, neither subarray is empty.
//OUTPUT: The two subarrays are merged into a single sorted subarray in $A[p \,..\, r]$.

```
IF p < r                                  // Check for base case
   THEN q = FLOOR[(p + r)/2]              // Divide step
        MERGE (A, p, q)                   // Conquer step.
        MERGE (A, q + 1, r)               // Conquer step.
        MERGE (A, p, q, r)                // Conquer step.
```

```
MERGE (A, p, q, r )
{
    n1 ← q − p + 1
    n2 ← r − q
    Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
    FOR i ← 1 TO n1
        DO L[i] ← A[p + i − 1]
    FOR j ← 1 TO n2
        DO R[j] ← A[q + j ]
    L[n1 + 1] ← ∞
    R[n2 + 1] ← ∞
  i ← 1
  j ← 1
  FOR k ← p TO r
      DO IF L[i ] ≤ R[ j]
          THEN A[k] ← L[i]
               i ← i + 1
          ELSE A[k] ← R[j]
               j ← j + 1

}
```

The space complexity of Merge sort:

Space complexity is always Ω(n) as you have to store the elements somewhere. Additional space complexity can be O(n) in an implementation using arrays and O(1) in linked list implementations. In practice implementations using lists need additional space for list pointers, so unless you already have the list in memory it shouldn't matter.

Iif you count stack frames, then it's O(n)+ O(log n) , so still O(n) in case of arrays. In case of lists it's O(log n) additional memory.

Derivation of best case and worst case time complexity (Merge Sort)

In sorting *n* objects, merge sort has an average and worst-case performance of O(*n* log *n*). If the running time of merge sort for a list of length *n* is *T*(*n*), then the recurrence *T*(*n*) = 2*T*(*n*/2) + *n* follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the *n* steps taken to merge the resulting two lists). The closed form follows from the master theorem.

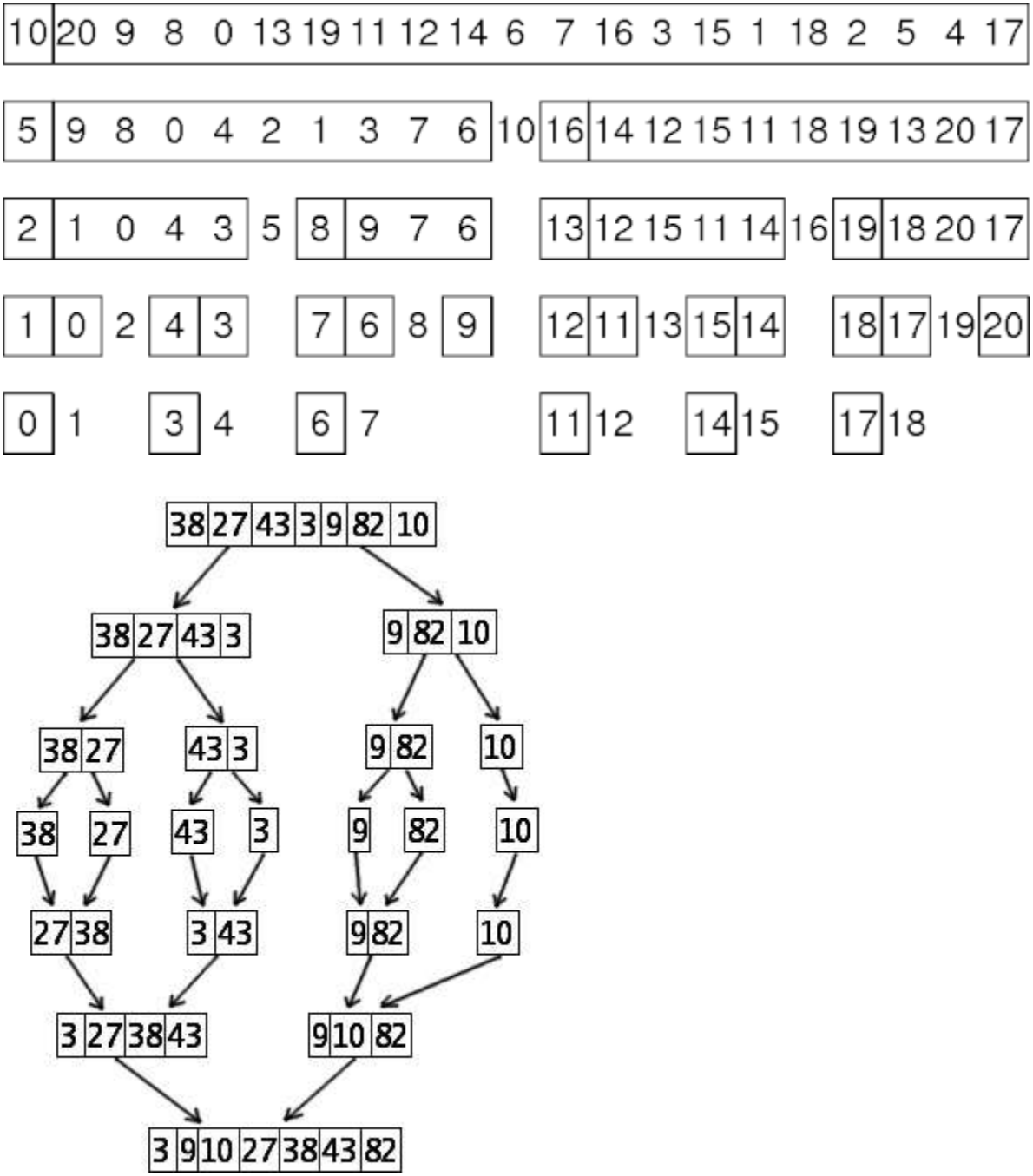
In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than ($n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$), which is between ($n \lg n - n + 1$) and ($n \lg n + n + O(\lg n)$).

For large *n* and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches $\alpha \cdot n$ fewer than the worst case where in the *worst* case, merge sort does about 39% fewer comparisons than quicksort does in the *average* case. In terms of moves, merge sort's worst case complexity is O(*n* log *n*)—the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.

Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort.

Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in (see below for versions that need only *n*/2 extra spaces).

Example for quicksort/Merge tree for merge sort:



Code:

```
import random,time,sys
import matplotlib.pyplot as plt
sys.setrecursionlimit(50000)
inc,n = 20,50
```

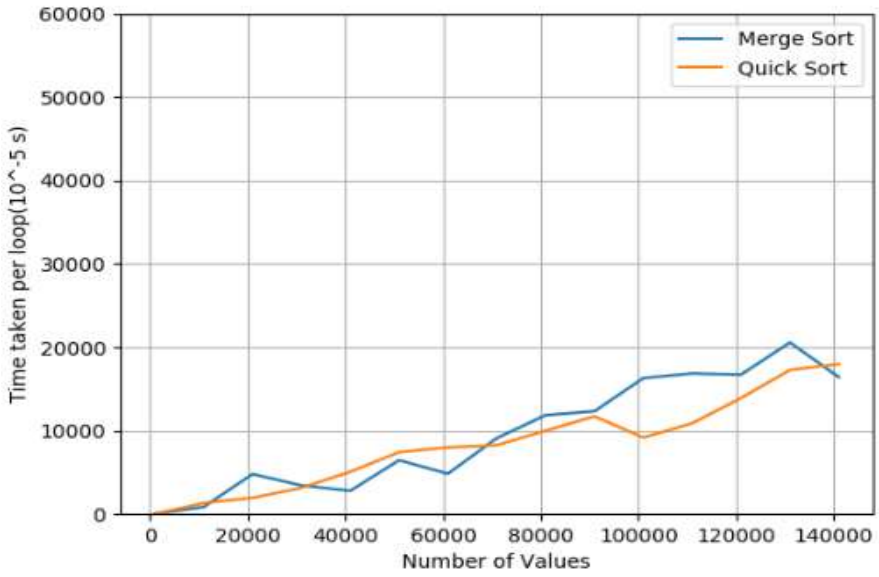
```
x1 = []
y1 = []
x2 = []
y2 = []
for val in range(15):
  arr1=list(random.random() for _ in range(n*inc))
  arr2=arr1[:]

  #Merge Sort
  def MergeSort(arr):
    if len(arr)>1:
      mid=len(arr)//2
      left=arr[:mid]
      right=arr[mid:]
      i=j=k=0
      while(i<len(left) and j<len(right)):
        if(left[i]<right[j]):
          arr1[k]=left[i]
          i+=1
        else:
          arr1[k]=right[j]
          j+=1
        k+=1
      while(i<len(left)):
        arr1[k]=left[i]
        i+=1
        k+=1
      while(j<len(right)):
        arr1[k]=right[j]
        j+=1
        k+=1
  #Quick Sort
  def QuickSort(arr, low, high):
    if low<high:
      p_index=Partition(arr, low, high)
      QuickSort(arr,low,p_index-1)
      QuickSort(arr,p_index+1,high)

  def Partition(arr,low,high):
    pivot=arr[high]
    i=low-1
    for j in range(low,high):
      if arr[j]<=pivot:
        i+=1
        arr[i],arr[j]=arr[j],arr[i]
```

```
    arr[i+1],arr[high]=arr[high],arr[i+1]
    return i+1
  tic=time.time()
  MergeSort(arr1)
  tac=time.time()
  x1.append(len(arr1))
  y1.append((tac-tic)*(10**5))
  tac=time.time()
  QuickSort(arr2,0,len(arr2)-1)
  toe=time.time()
  x2.append(len(arr2))
  y2.append((toe-tac)*(10**4))
  n=n+500
plt.ylim((0,60000))
plt.plot(x1, y1, label = 'Merge Sort')
plt.plot(x2,y2, label='Quick Sort')
plt.legend()
plt.grid()
plt.xlabel(' Number of Values ')
plt.ylabel(' Time taken per loop(10^-5 s) ')
plt.savefig('graph.png')
```

Graph:



CONCLUSION:

The program ran successfully for both merge sort and quick sort and the graph for both were obtained for big range of 'n'.