K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: E-2	**Roll No.: 16010123325**
Experiment No.___3____	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title: Study, Implementation, and Comparative Analysis of Merge Sort and Quick Sort.

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

---

CO to be achieved:

CO 2 Describe various algorithm design strategies to solve different problems and analyze Complexity.

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 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 AT[], 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;

}
```

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

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 $p \le q \le r$ and sub array *A*[*p* .. *q*] is sorted and sub array //*A*[*q* + 1 .. *r*] is sorted. By restrictions on *p*, *q*, *r*, neither sub array is empty.

//OUTPUT: The two sub arrays are merged into a single sorted sub array 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 Quick Sort:

O(1)

The space complexity of Merge sort:

O(n)

Code (Quicksort)-

```
#include <bits/stdc++.h>
using namespace std;

int partition(vector<int> &arr, int low, int high) {
    int pivot = arr[low];
    int i = low, j = high;
    while (i < j) {
        while (arr[i] <= pivot && i <= high) {
            i++;
        }
        while (arr[j] > pivot && j >= low) {
            j--;
        }
        if (i < j) {
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[low], arr[j]);
    return j;
}

void QuickSort(vector<int> &arr, int low, int high) {
    if (low < high) {
        int pIndex = partition(arr, low, high);
        QuickSort(arr, low, pIndex-1);
        QuickSort(arr, pIndex+1, high);
    }
}

int main() {
    int n; cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; ++i) cin >> arr[i];
    QuickSort(arr, 0, n-1);
```

```
    for (int i = 0; i < n; ++i) cout << arr[i] << " ";
}
```

Output-

```
PS C:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-4\AOA Lab\Lab-3> cd "c:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-4\AOA Lab\Lab-3\" ; if ($?) { g++ QuickSort.cpp -o QuickSort } ; if ($?) { .\QuickSort }
5
9 8 21 33 12
8 9 12 21 33
PS C:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-4\AOA Lab\Lab-3>
```

Code (Merge Sort)-

```
#include <bits/stdc++.h>
using namespace std;

void merge(vector<int> &arr, int low, int mid, int high) {
    vector<int> temp;
    int l = low, r = mid + 1;
    while (l <= mid && r <= high) {
        if (arr[l] < arr[r]) {
            temp.push_back(arr[l++]);
        } else {
            temp.push_back(arr[r++]);
        }
    }
    while (l <= mid) {
        temp.push_back(arr[l++]);
    }
    while (r <= high) {
        temp.push_back(arr[r++]);
    }
    for (int i = low; i <= high; i++) {
        arr[i] = temp[i - low];
    }
}

void mergeSort(vector<int> &arr, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}

int main() {
    int n;
```

```
    cout << "Enter the elements of the array: ";
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; ++i) cin >> arr[i];
    mergeSort(arr, 0, arr.size() - 1);
    for (int i = 0; i < arr.size(); i++) cout << arr[i] << " ";
}
```

Output-

```
PS C:\Users\Shrey\OneDrive\Desktop\KJSCE> cd "c:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-4\AOA Lab\Lab-3\" ; if ($?) { g++ mergesort
.cpp -o mergesort } ; if ($?) { .\mergesort }
Enter the elements of the array: 5
4 1 2 3 8
1 2 3 4 8
PS C:\Users\Shrey\OneDrive\Desktop\KJSCE\SEM-4\AOA Lab\Lab-3>
```

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

Merge Sort Algorithm

```
mergeSort (arr[], low, high) {
if (low < high) {
    mid = (low + high)/2
    mergeSort(arr, low, mid);
    mergeSort (arr, mid+1, high);
    merge (arr, low, mid, high);
  }
}

void merge (int arr[], int low, int mid, int high) {
    int i = low,
    int j = mid+1;
    int k = low,

  int temp[5];

  while (i <= mid && j <= high) {
      if (arr[i] <= arr[j] {
          temp [k] = arr [i];
          i++;
          k++;
    }
  else {
      temp[k] = arr[j];
        j++;
        k++;
    }
  }
```

```
// Copy remaining 1st Half
while (i <= mid) {
    temp [k] = arr [i] ;
    i++;
    k++;
}

// Copy remaining of 2nd half
while (j <= high) {
    temp [k] = arr [j]
    j++;
    k++;
}

// Copy temp arr to original
for ( int k = low ; k <= high ; k++) {
    arr [k] = temp [k];
}
}
```

Space Complexity: O(n

Time Complexity

$T(n) = aT(n/b) + O(n^d) + \log n$

$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

$a = 2,\ b = 2,\ d = 1,\ p = 0$

$\log_b a = \log_2 2 = 1 = d$

∴ Master theorem Case 2: $(p > -1)$

$n^d = n^1 = n$

$n^{\log_b a} = n^1 = n$

As $n^d = n^{\log_b a}$

$T(n) = O(n \log n)$

Best case is also $O(n \log n)$ when already sorted it still divides sub-array

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

Quick Sort Algorithm

```
Algo QuickSort (A[0...n], low, high) {
    if (low < high) {
    mid = partition (A[low...high]);
    QuickSort (A[low... mid-1]);
    QuickSort (A[mid+1 .... high);
}

Algo Partition (A[0...n], low, high) {
    pivot ← A[low]
    i ← low
    j ← high
    while (i ≤ j) do
    {
        while (A[i] ≤ pivot) do i++;
        while (A[j] > pivot do j--;
        if (i ≤ j) then
        swap (A[i], A[j])
    }
    swap (A[low], A[j]);
    return j
}
```

Time Complexity

Worst case : Sorted array

$T(n) = T(n-1) + O(n)$

$= T(n-2) + O(n-1) + O(n)$

$= O(n + (n-1) + (n-2) + \dots + 1)$

$= 1 + 2 + 3 + \dots n \leftarrow \frac{n(n-1)}{2}$

$T(n) = O(n^2)$ or $a = 1$ $k = 2$ $T(n) = O(n^{k+1})$

$= O(n^2)$

Best case: PIVOT is middle element

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$a = 2$, $b = 2$, $d = 1$, $p = 0$

$a = b^k$, $p > -1$

$\therefore \log_b a = \log_2 2 = 1$

$\therefore$ Case 2

$$T(n) = O(n^{\log_b a} * \log^{p+1} n) = O(n \log n)$$

Space complexity $(\underset{\text{array}}{n}, a, \underbrace{\text{pivot}, k, i, j}_{\text{var, constt}})$

$O(n)$

Example for quicksort/Merge tree for merge sort:

Merge Sort

[8, 4, 7, 3, 5]

[8, 4, 7] [3, 5]

[8, 4] [7] [3] [5]

[8] [4]

[8] · [4] → [4, 8]

[4, 8] [7] → [4, 7, 8]

[3] [5] → [3, 5]

[4, 7, 8] [3, 5] → [3, 4, 5, 7, 8]

Quick Sort

pivot

4 3 1 2 5 9 7 10 6

1 2 4 3   6 7 9 10

pivot   pivot

pivot pivot

1 3 4   7 9 10

pivot pivot

4   7 10

↳ [1 2 3 4 5 6 7 9 10]

CONCLUSION:

The above experiment highlights better sorting techniques like Quick Sort, Merge Sort give complexity O(n log n) compared to the previous experiment