Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.
Department of Master of Computer Applications

Experiment	1
Aim	Understand sorting algorithms on the basis of Divide and Conquer approach
Objective	1) Learn Divide and Conquer strategy in sorting algorithms 2) Learn Merge Sort and Quick Sort 3) Compare the Time complexity of Merge Sort and Quick Sort
Name	Vivek Tiwari
UCID	2023510059
Class	FY MCA
Batch	C
Date of Submission	15/02/24

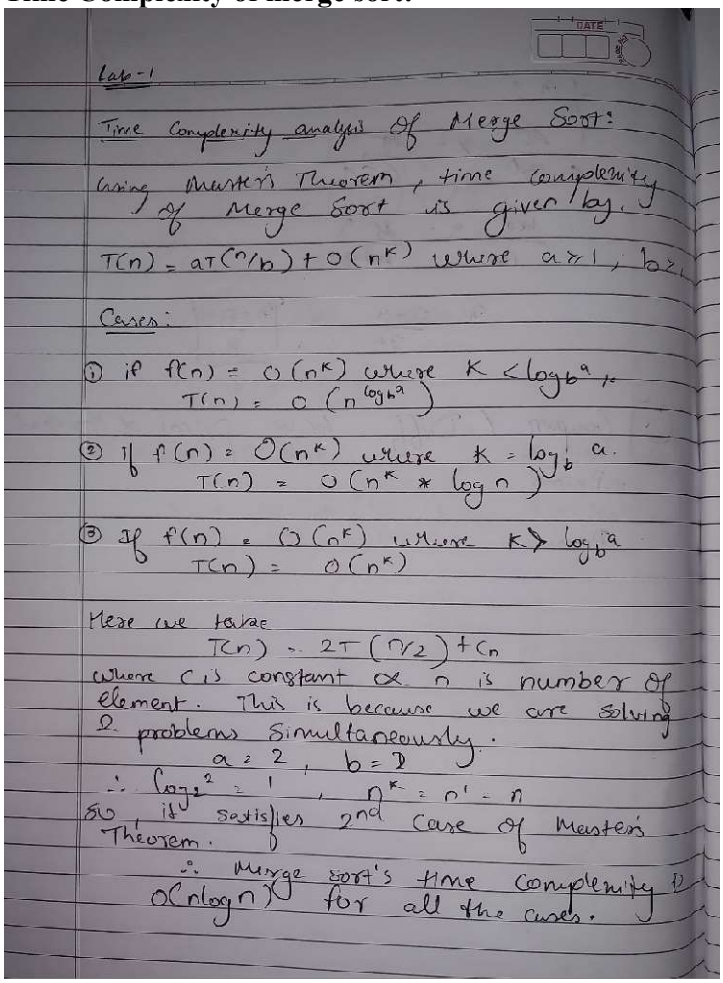
Algorithm and explanation of the technique used

Algorithm of Merge Sort:

MergeSort(arr, left, right)
1. if left < right
    a. Set middle = (left + right)/2
    b. Call MergeSort(arr, left, middle)
    c. Call MergeSort(arr, middle+1, right)
    d. Call Merge(arr, left, middle, right)

Merge(arr, left, middle, right)
1. Set n1 = middle – left + 1
2. Set n2 = right – middle
3. Create temporary array L[1 … n1] and R[1 … n2]
4. Copy data to L[1 … n1] from arr[left ... middle]
5. Copy data to R[1 … n2] from arr[middle + 1 … right]
6. Set i=1, j=1, k=left
7. While i<=n1 and j<=n2
    a. if L[i] <= R[j]
        - Set arr[k] = L[i]
        - Increment i
    b. Else
        - Set arr[k] = R[i]
        - Increment j
    c. Increment k
8. While i<=n1
    a. Set arr[k] = L[i]
    b. Increment i and k
9. While j <= n2
    a. Set arr[k] = R[j]
    Increment j and k

Time Complexity of merge sort:



Lab-1

Time Complexity analysis of Merge Sort:

Using Master's Theorem, time complexity of Merge Sort is given by,

$T(n) = aT(n/b) + O(n^K)$ where $a \geq 1$, $b \geq 1$

Cases:

① if $f(n) = O(n^K)$ where $K < \log_b a$,
$T(n) = O(n^{\log_b a})$

② if $f(n) = O(n^K)$ where $K = \log_b a$.
$T(n) = O(n^K * \log n)$

③ if $f(n) = O(n^K)$ where $K > \log_b a$
$T(n) = O(n^K)$

Here we have
$T(n) = 2T(n/2) + cn$
where c is constant & n is number of element. This is because we are solving 2 problems simultaneously.
$a = 2, \quad b = 2$
$\therefore \log_2 2 = 1, \quad n^K = n^1 = n$
So it satisfies 2nd case of Master's Theorem.
∴ Merge sort's time complexity is $O(n \log n)$ for all the cases.

Algorithm of Quick Sort:

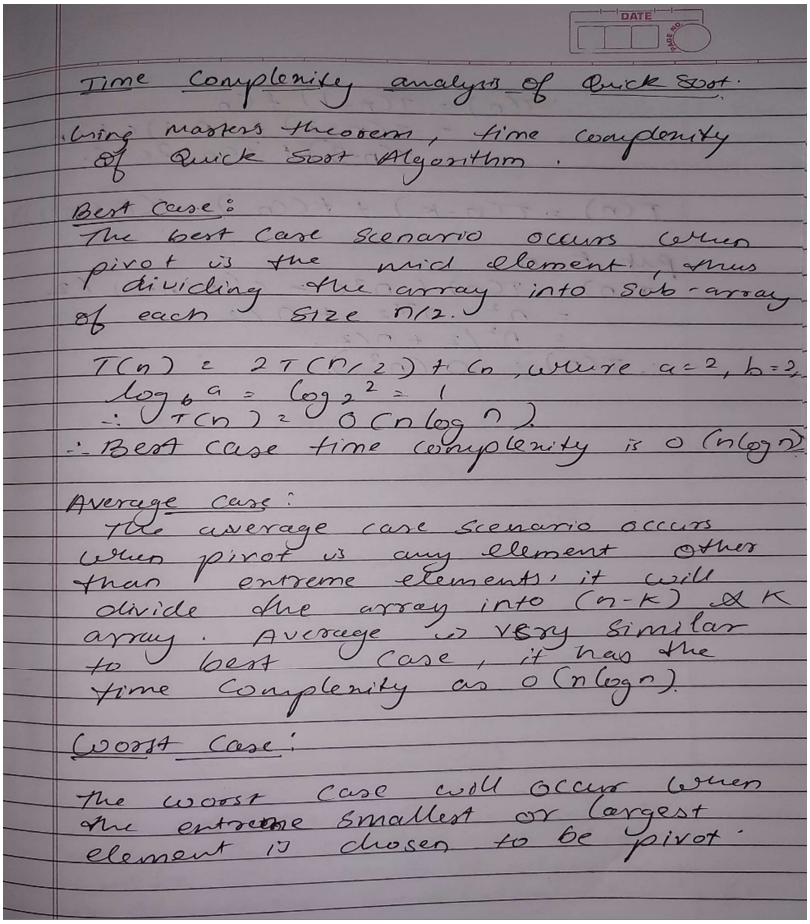
QuickSort(arr, low, high)

1. if low < high
   a. Set pivotIndex = Partition(arr, low, high)
   b. Call QuickSort(arr, low, pivotIndex - 1)
   c. Call QuickSort(arr, pivotIndex + 1, high)

Partition(arr, low, high)

1. Set pivot = arr[high]
2. Set i = low - 1
3. for j from low to high - 1
   a. if arr[j] <= pivot
      i. Increment i
      ii. Swap arr[i] and arr[j]
4. Swap arr[i + 1] and arr[high]
5. Return i + 1

Time Complexity for quick Sort:



Time Complexity analysis of Quick Sort.

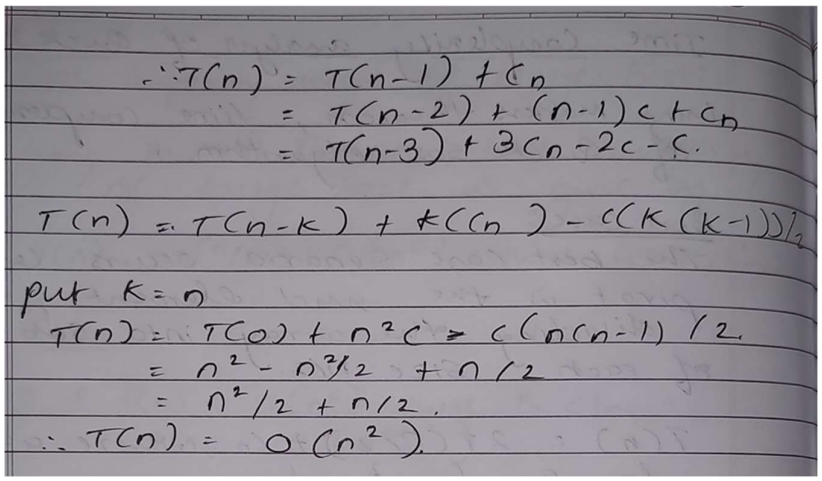
Using Masters theorem, time complexity of Quick Sort Algorithm.

Best case:
The best case scenario occurs when pivot is the mid element, thus dividing the array into sub-array of each size n/2.

$T(n) = 2T(n/2) + cn$, where $a=2, b=2$,
$\log_b a = \log_2 2 = 1$
$\therefore T(n) = O(n \log n)$
$\therefore$ Best case time complexity is $O(n \log n)$

Average case:
The average case scenario occurs when pivot is any element other than extreme elements, it will divide the array into $(n-k)$ & k array. Average is very similar to best case, it has the time complexity as $O(n \log n)$.

Worst case:
The worst case will occur when the extreme smallest or largest element is chosen to be pivot.



$\therefore T(n) = T(n-1) + cn$
$= T(n-2) + (n-1)c + cn$
$= T(n-3) + 3cn - 2c - c.$

$T(n) = T(n-k) + k(cn) - c(k(k-1))/2$

put $k = n$
$T(n) = T(0) + n^2 c = c(n(n-1)/2.$
$= n^2 - n^2/2 + n/2$
$= n^2/2 + n/2.$
$\therefore T(n) = O(n^2).$

Merge Sort

Average Case:

```
public class mergesort {
   public static void mergeSort(int[] arr) {
      if (arr == null || arr.length <= 1) {
         return;
      }
      int[] temp = new int[arr.length];
      mergeSort(arr, temp, 0, arr.length - 1);
   }
   private static void mergeSort(int[] arr, int[] temp, int left, int right) {
      if (left < right) {
         int mid = left + (right - left) / 2;
         mergeSort(arr, temp, left, mid);
         mergeSort(arr, temp, mid + 1, right);
         merge(arr, temp, left, mid, right);
      }
   }
   private static void merge(int[] arr, int[] temp, int left, int mid, int right)
{
      System.arraycopy(arr, left, temp, left, right - left + 1);
      int i = left;
      int j = mid + 1; int k = left;
      while (i <= mid && j <= right) {
         if (temp[i] <= temp[j]) {
            arr[k++] = temp[i++];
         } else {
            arr[k++] = temp[j++];
         }
      }
      while (i <= mid) {
         arr[k++] = temp[i++];
      }
   }
   public static void main(String[] args) {
      int[] arr = { 359, 419, 239, 119, 59, 179, 599, 479, 299, 539 };
      System.out.println("Shuffled array:");
      printArray(arr);
      mergeSort(arr);
      System.out.println("\nSorted Array:");
      printArray(arr);
      System.out.println();
   }
   public static void printArray(int[] arr) {
      for (int num : arr) {
         System.out.print(num + " ");
      }
      System.out.println();
   }
}
```

Worst Case:

```
public class mergesortWorst {
   public static void mergeSort(int[] arr) {
      if (arr == null || arr.length <= 1) {
         return;
      }
      int[] temp = new int[arr.length];
      mergeSort(arr, temp, 0, arr.length - 1);
   }
   private static void mergeSort(int[] arr, int[] temp, int left, int right) {
      if (left < right) {
         int mid = left + (right - left) / 2;
         mergeSort(arr, temp, left, mid);
         mergeSort(arr, temp, mid + 1, right);
         merge(arr, temp, left, mid, right);
      }
   }
   private static void merge(int[] arr, int[] temp, int left, int mid, int right)
{
      System.arraycopy(arr, left, temp, left, right - left + 1);
      int i = left;
      int j = mid + 1; int k = left;
      while (i <= mid && j <= right) {
         if (temp[i] <= temp[j]) {
            arr[k++] = temp[i++];
         } else {
            arr[k++] = temp[j++];
         }
      }
      while (i <= mid) {
           arr[k++] = temp[i++];
      }
   }
   public static void main(String[] args) {
      int[] arr = { 599, 539, 479, 419, 359, 299, 239, 179, 119, 59 };
      System.out.println("Shuffled array:");
      printArray(arr);
      mergeSort(arr);
      System.out.println("\nSorted Array:");
      printArray(arr);
      System.out.println();
   }
   public static void printArray(int[] arr) {
      for (int num : arr) {
         System.out.print(num + " ");
      }
      System.out.println();
   }
}
```

Best Case:

```
public class mergesortWorst {
   public static void mergeSort(int[] arr) {
      if (arr == null || arr.length <= 1) {
         return;
      }
      int[] temp = new int[arr.length];
      mergeSort(arr, temp, 0, arr.length - 1);
   }
   private static void mergeSort(int[] arr, int[] temp, int left, int right) {
      if (left < right) {
         int mid = left + (right - left) / 2;
         mergeSort(arr, temp, left, mid);
         mergeSort(arr, temp, mid + 1, right);
         merge(arr, temp, left, mid, right);
      }
   }
   private static void merge(int[] arr, int[] temp, int left, int mid, int right)
{
      System.arraycopy(arr, left, temp, left, right - left + 1);
      int i = left;
      int j = mid + 1; int k = left;
      while (i <= mid && j <= right) {
         if (temp[i] <= temp[j]) {
            arr[k++] = temp[i++];
         } else {
            arr[k++] = temp[j++];
         }
      }
      while (i <= mid) {
         arr[k++] = temp[i++];
      }
   }
   public static void main(String[] args) {
      int[] arr = { 59, 119, 179, 239, 299, 359, 419, 479, 539, 599  };
      System.out.println("Shuffled array:");
      printArray(arr);
      mergeSort(arr);
      System.out.println("\nSorted Array:");
      printArray(arr);
      System.out.println();
   }
   public static void printArray(int[] arr) {
      for (int num : arr) {
         System.out.print(num + " ");
      }
      System.out.println();
   }
}
```

Quick Sort

Average Case

```
//Quicksort for Average case
public class quicksort {
   public static void main(String[] args) {
      int[] array = { 359, 419, 239, 119, 59, 179, 599, 479, 299, 539 };
      System.out.print("Array before sorting: ");
      printArr(array);
      quickSort(array, 0, array.length - 1);
      System.out.print("Array after sorting (Quick): ");
      printArr(array);
   }
   public static void quickSort(int[] array, int low, int high) {
      if (low < high) {
// Partition the array, and get the index of the pivot
         int pivotIndex = partition(array, low, high);
// Recursively sort the sub-arrays on the left and right of the pivot
         quickSort(array, low, pivotIndex - 1);
         quickSort(array, pivotIndex + 1, high);
      }
   }
   public static int partition(int[] array, int low, int high) {
// Choose the last element as the pivot
      int pivot = array[high];
// Index of the smaller element
      int i = low - 1;
// Traverse the array and rearrange elements
      for (int j = low; j < high; j++) {
         if (array[j] <= pivot) {
            i++;
// Swap array[i] and array[j]
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
         }
      }
// Swap array[i + 1] and the pivot
      int temp = array[i + 1];
      array[i + 1] = array[high];
      array[high] = temp;
// Return the index of the pivot element
      return i + 1;
   }
   public static void printArr(int[] arr){
      System.out.println();
      for(int i=0; i<arr.length; i++){
         System.out.print(arr[i]+" ");
      }
      System.out.println();
   }
}
```

Worst Case

```
public class quicksort {
```

```
    public static void main(String[] args) {
        int[] array = { 599, 539, 479, 419, 359, 299, 239, 179, 119, 59 };
        System.out.print("Array before sorting: ");
        printArr(array);
        quickSort(array, 0, array.length - 1);
        System.out.print("Array after sorting (Quick): ");
        printArr(array);
    }
    public static void quickSort(int[] array, int low, int high) {
        if (low < high) {
// Partition the array, and get the index of the pivot
            int pivotIndex = partition(array, low, high);
// Recursively sort the sub-arrays on the left and right of the pivot
            quickSort(array, low, pivotIndex - 1);
            quickSort(array, pivotIndex + 1, high);
        }
    }
    public static int partition(int[] array, int low, int high) {
// Choose the last element as the pivot
        int pivot = array[high];
// Index of the smaller element
        int i = low - 1;
// Traverse the array and rearrange elements
        for (int j = low; j < high; j++) {
            if (array[j] <= pivot) {
                i++;
// Swap array[i] and array[j]
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
// Swap array[i + 1] and the pivot
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;
// Return the index of the pivot element
        return i + 1;
    }
    public static void printArr(int[] arr){
        System.out.println();
        for(int i=0; i<arr.length; i++){
            System.out.print(arr[i]+" ");
        }
        System.out.println();
    }
}
```

Best Case

```
//Quicksort for Best case
public class quicksort {
    public static void main(String[] args) {
        int[] array = { 59, 119, 179, 239, 299, 359, 419, 479, 539, 599  };
        System.out.print("Array before sorting: ");
        printArr(array);
```

```
        quickSort(array, 0, array.length - 1);
        System.out.print("Array after sorting (Quick): ");
        printArr(array);
    }
    public static void quickSort(int[] array, int low, int high) {
        if (low < high) {
// Partition the array, and get the index of the pivot
            int pivotIndex = partition(array, low, high);
// Recursively sort the sub-arrays on the left and right of the pivot
            quickSort(array, low, pivotIndex - 1);
            quickSort(array, pivotIndex + 1, high);
        }
    }
    public static int partition(int[] array, int low, int high) {
// Choose the last element as the pivot
        int pivot = array[high];
// Index of the smaller element
        int i = low - 1;
// Traverse the array and rearrange elements
        for (int j = low; j < high; j++) {
            if (array[j] <= pivot) {
                i++;
// Swap array[i] and array[j]
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
// Swap array[i + 1] and the pivot
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;
// Return the index of the pivot element
        return i + 1;
    }
    public static void printArr(int[] arr){
        System.out.println();
        for(int i=0; i<arr.length; i++){
            System.out.print(arr[i]+" ");
        }
        System.out.println();
    }
}
```

Output

Merge Sort

Average Case

```
"C:\Program Files\Java\jdk-21\bin\java.exe"
Shuffled array:
359 419 239 119 59 179 599 479 299 539
Sorted array:
59 119 179 239 299 359 419 479 539 599
```

Worst Case

```
"C:\Program Files\Java\jdk-21\bin\java.exe"
Shuffled array:
599 539 479 419 359 299 239 179 119 59

Sorted Array:
59 119 179 239 299 359 419 479 539 599
```

Best Case:

```
"C:\Program Files\Java\jdk-21\bin\java.exe"
Shuffled array:
59 119 179 239 299 359 419 479 539 599

Sorted Array:
59 119 179 239 299 359 419 479 539 599
```

Quick Sort

Average Case

```
"C:\Program Files\Java\jdk-21\bin\java.exe"
Array before sorting:
359 419 239 119 59 179 599 479 299 539
Array after sorting (Quick):
59 119 179 239 299 359 419 479 539 599
```

Worst Case

```
"C:\Program Files\Java\jdk-21\bin\java.exe"
Array before sorting:
599 539 479 419 359 299 239 179 119 59
Array after sorting (Quick):
59 119 179 239 299 359 419 479 539 599
```

Best Case:

```
"C:\Program Files\Java\jdk-21\bin\java.exe"
Array before sorting:
59 119 179 239 299 359 419 479 539 599
Array after sorting (Quick):
59 119 179 239 299 359 419 479 539 599
```

Conclusion	Merge Sort and Quick Sort have efficient time complexities of O(n log n) for their best and average case scenarios, making them suitable for sorting large datasets. However, Quick Sort can potentially have a worst-case time complexity of O(n^2), but this can be mitigated by employing careful pivot selection strategies. In contrast, Merge Sort maintains a consistent time complexity of O(n log n) across all cases, making its performance more predictable, especially in situations where worst-case performance is crucial. The decision to choose between these two algorithms often depends on additional factors, such as the requirement for a stable sorting order or the specific characteristics of the data being sorted.