**Batch: D2 Roll No.: 16010122323**

**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:**

|  |  |
| --- | --- |
| 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 algortihtms”,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;

}

**Code Quick Sort:**

// package Codes.Sorting;

import java.io.\*;

import java.time.\*;

import java.util.\*;

import java.lang.Math;

public class quickSort {

// A utility function to swap two elements

    static void swap(int[] arr, int i, int j)

    {

        int temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

    }

/\* This function takes last element as pivot, places

   the pivot element at its correct position in sorted

   array, and places all smaller (smaller than pivot)

   to left of pivot and all greater elements to right

   of pivot \*/

    static int partition(int[] arr, int low, int high)

    {

        // pivot

        int pivot = arr[high];

        // Index of smaller element and

        // indicates the right position

        // of pivot found so far

        int i = (low - 1);

        for(int j = low; j <= high - 1; j++)

        {

            // If current element is smaller

            // than the pivot

            if (arr[j] < pivot)

            {

                // Increment index of

                // smaller element

                i++;

                swap(arr, i, j);

            }

        }

        swap(arr, i + 1, high);

        return (i + 1);

    }

/\* The main function that implements QuickSort

          arr[] --> Array to be sorted,

          low --> Starting index,

          high --> Ending index

 \*/

    static void quickSort(int[] arr, int low, int high)

    {

        if (low < high)

        {

            // pi is partitioning index, arr[p]

            // is now at right place

            int pi = partition(arr, low, high);

            // Separately sort elements before

            // partition and after partition

            quickSort(arr, low, pi - 1);

            quickSort(arr, pi + 1, high);

        }

    }

// Function to print an array

    static void printArray(int[] arr, int size)

    {

        for(int i = 0; i < size; i++)

            System.out.print(arr[i] + " ");

        System.out.println();

    }

// Driver Code

    public static void main(String[] args)

    {

        Scanner sc = new Scanner(System.in);

        int N=sc.nextInt();

        int arr[];

        arr=new int[N];

        // System.out.println("Given Array");

        for(int i=0;i<N;i++)

        {

            double x=Math.random()\*300;

            int num= (int)x;

            arr[i]= num;

        }

        printArray(arr,N);

        long start = System.nanoTime();

        quickSort(arr, 0, N - 1);

        System.out.println("Sorted array: ");

        printArray(arr, N);

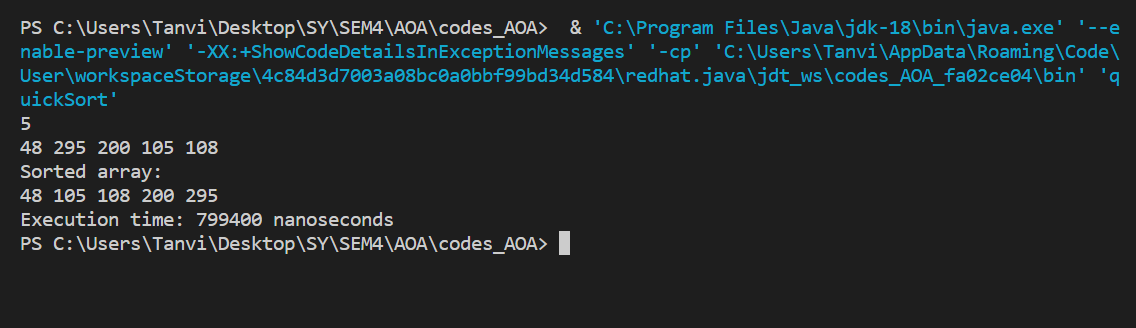
        long end = System.nanoTime();

        long execution = end - start;

        System.out.println("Execution time: " + execution + " nanoseconds");

    }

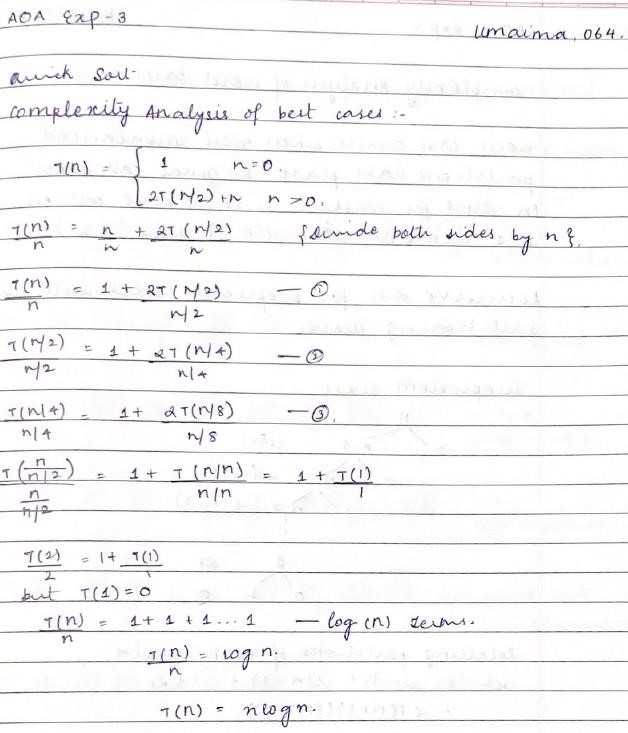
}

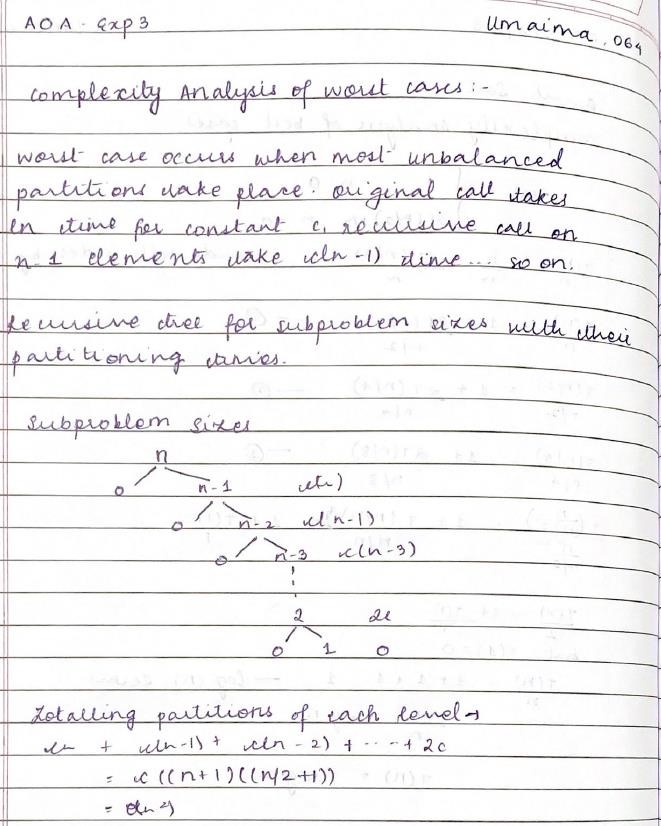
**Output Quick Sort:** ****

**The space complexity of Quick Sort:**

**O(n)**

**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* ≤ *q* ≤ 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* )

{

*n*1 ← *q* − *p* + 1  
      *n*2 ← *r* − *q*  
      Create arrays L[1 . . *n*1 + 1] and R[1 . . *n*2 + 1]  
      **FOR** *i* ← 1 **TO** *n*1  
            **DO** L[*i*] ← A[*p* + *i* − 1]  
      **FOR** *j* ← 1 **TO** *n*2  
            **DO** R[*j*] ← A[*q* + *j* ]  
      L[*n*1 + 1] ← ∞  
      R[*n*2 + 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

}

**Code Merge Sort:**

// package Codes.Sorting;

import java.util.\*;

import java.lang.Math;

public class mergeSort {

    static void merge(int arr[], int l, int m, int r)

    {

        // Find sizes of two subarrays to be merged

        int n1 = m - l + 1;

        int n2 = r - m;

        /\* Create temp arrays \*/

        int L[] = new int[n1];

        int R[] = new int[n2];

        /\*Copy data to temp arrays\*/

        for (int i = 0; i < n1; ++i)

            L[i] = arr[l + i];

        for (int j = 0; j < n2; ++j)

            R[j] = arr[m + 1 + j];

        /\* Merge the temp arrays \*/

        // Initial indexes of first and second subarrays

        int i = 0, j = 0;

        // Initial index of merged subarray array

        int k = l;

        while (i < n1 && j < n2) {

            if (L[i] <= R[j]) {

                arr[k] = L[i];

                i++;

            }

            else {

                arr[k] = R[j];

                j++;

            }

            k++;

        }

        /\* Copy remaining elements of L[] if any \*/

        while (i < n1) {

            arr[k] = L[i];

            i++;

            k++;

        }

        /\* Copy remaining elements of R[] if any \*/

        while (j < n2) {

            arr[k] = R[j];

            j++;

            k++;

        }

    }

    // Main function that sorts arr[l..r] using

    // merge()

    static void sort(int arr[], int l, int r)

    {

        if (l < r) {

            // Find the middle point

            int m =l+ (r-l)/2;

            // Sort first and second halves

            sort(arr, l, m);

            sort(arr, m + 1, r);

            // Merge the sorted halves

            merge(arr, l, m, r);

        }

    }

    /\* A utility function to print array of size n \*/

    static void printArray(int arr[])

    {

        int n = arr.length;

        for (int i = 0; i < n; ++i)

            System.out.print(arr[i] + " ");

        System.out.println();

    }

    // Driver code

    public static void main(String args[])

    {

        Scanner sc = new Scanner(System.in);

        int N=sc.nextInt();

        int arr[];

        arr=new int[N];

        // System.out.println("Given Array");

        for(int i=0;i<N;i++)

        {

            double x=Math.random()\*300;

            int num= (int)x;

            arr[i]= num;

        }

        printArray(arr);

        long start = System.nanoTime();

        // mergeSort ob = new mergeSort();

        sort(arr, 0, arr.length - 1);

        System.out.println("\nSorted array");

        printArray(arr);

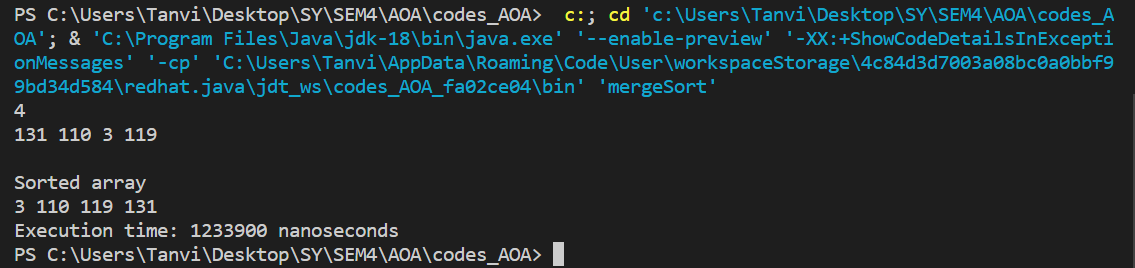
        long end = System.nanoTime();

        long execution = end - start;

        System.out.println("Execution time: " + execution + " nanoseconds");

    }

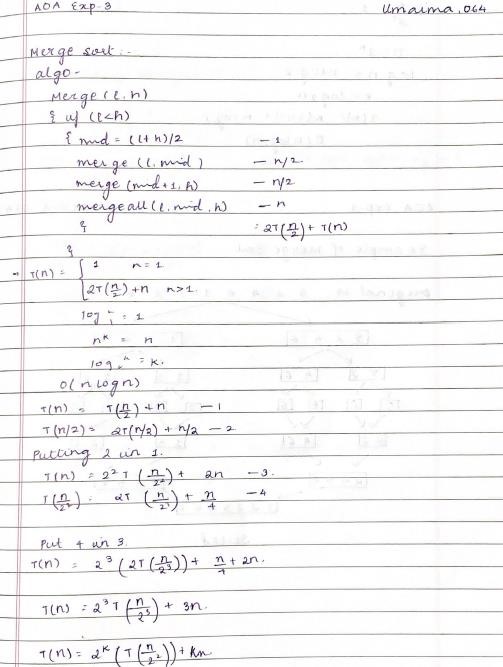
}

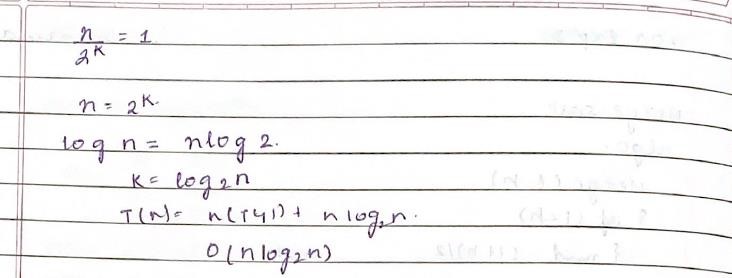
**Output Merge Sort: **

**The space complexity of Merge sort:**

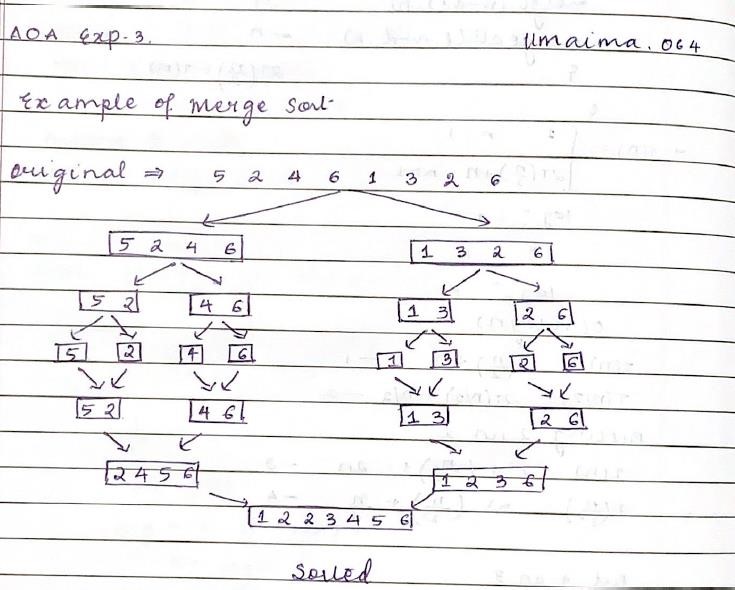
**O(N)**

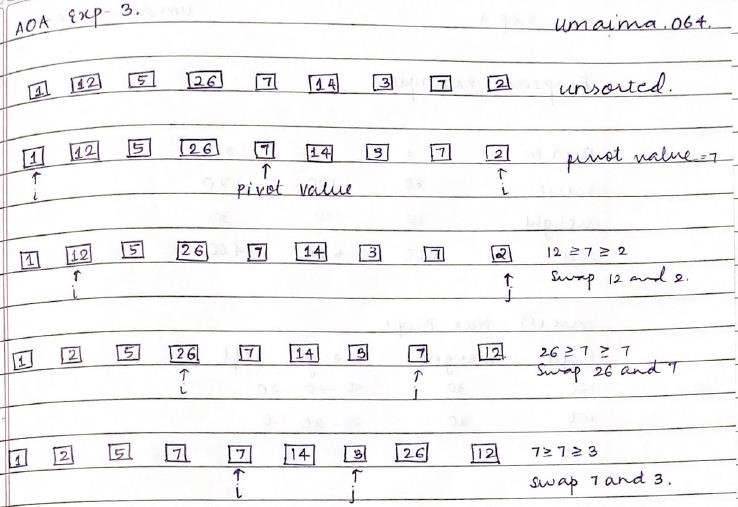
**Derivation of best case and worst-case time complexity (Merge Sort)**

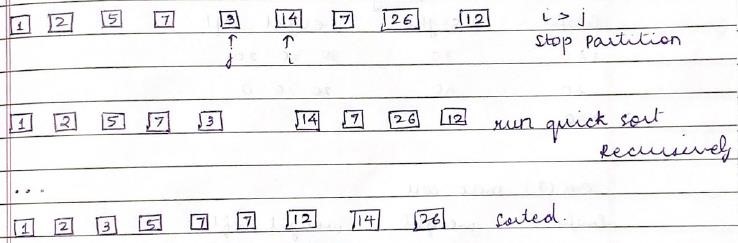




**Example for quicksort/Merge tree for merge sort:**







**Conclusion:**

Thus, through this experiment we have understood the logic behind Merge sort and Quick Sort