PRACTICAL FILE
OF
DESIGN AND ANALYSIS OF ALGORITHMS LAB
(PAPER CODE: AIML-353)

MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY SECTOR-22, ROHINI, NEW DELHI

Submitted to:	***Submitted by:***
Dr. Sakshi Pandey	**Name:- Ansh Malik**
Assistant Professor	**Roll No:- 00914811622**
AI and ML Dept.	**Semester:- 5th**
	Group:- AIML-1

Department of Artificial Intelligence & Machine Learning

Rubrics for Lab Assessment

Rubrics		0 Missing	1 Inadequate	2 Needs Improvement	3 Adequate
R1	Is able to identify the problem to be solved and define the objectives of the experiment.	No mention is made of the problem to be solved.	An attempt is made to identify the problem to be solved but it is described in a confusing manner, objectives are not relevant, objectives contain technical/ conceptual errors or objectives are not measurable.	The problem to be solved is described but there are minor omissions or vague details. Objectives are conceptually correct and measurable but may be incomplete in scope or have linguistic errors.	The problem to be solved is clearly stated. Objectives are complete, specific, concise, and measurable. They are written using correct technical terminology and are free from linguistic errors.
R2	Is able to design a reliable experiment that solves the problem.	The experiment does not solve the problem.	The experiment attempts to solve the problem but due to the nature of the design the data will not lead to a reliable solution.	The experiment attempts to solve the problem but due to the nature of the design there is a moderate chance the data will not lead to a reliable solution.	The experiment solves the problem and has a high likelihood of producing data that will lead to a reliable solution.
R3	Is able to communicate the details of an experimental procedure clearly and completely.	Diagrams are missing and/or experimental procedure is missing or extremely vague.	Diagrams are present but unclear and/or experimental procedure is present but important details are missing.	Diagrams and/or experimental procedure are present but with minor omissions or vague details.	Diagrams and/or experimental procedure are clear and complete.
R4	Is able to record and represent data in a meaningful way.	Data are either absent or incomprehensible.	Some important data are absent or incomprehensible.	All important data are present, but recorded in a way that requires some effort to comprehend.	All important data are present, organized and recorded clearly.
R5	Is able to make a judgment about the results of the experiment.	No discussion is presented about the results of the experiment .	A judgment is made about the results, but it is not reasonable or coherent.	An acceptable judgment is made about the result, but the reasoning is flawed or incomplete.	An acceptable judgment is made about the result, with clear reasoning. The effects of assumptions and experimental uncertainties are considered.

INDEX

Exp. No.	Experiment Name	Date of performing	R1	R2	R3	R4	R5	Total Marks (15)	Remarks	Signature
1.										
2.										
3.										
4.										
5.										
6.										
7.										
8.										
9.										
10.										

EXPERIMENT:- 01

Aim- To Implement

a) Merge sort in C programming Language

b) Insertion sort in C programming Language

c) Bubble sort in C programming Language

THEORY:-

a) **Merge Sort** is a comparison-based sorting algorithm that works by dividing the input array into two halves, then calling itself for these two halves, and finally it merges the two sorted halves. It is based on three principles of divide and conquer:

- ***Divide***: *Split the array into two halves.*
- ***Conquer***: *Recursively sort each half.*
- ***Merge***: *Combine the two sorted halves to produce a single sorted array.*

The merge process is used for merging two halves. It is a key process that assumes that the **left half** and **right half** are sorted and merges them into one.

b) **Insertion sort** divides the list into **sorted and unsorted part**. Initially, the **first element is already considered sorted**, while the rest of the list is considered unsorted. The algorithm then iterates through each element in the unsorted part, picking one element at a time, and inserts it into its correct position in the sorted part.

c) **Bubble sort** is one of the simplest sorting algorithms that works by comparing the adjacent elements in the list and swapping them if the elements are not in the correct order. It is an in- place and stable sorting algorithm that can sort items in data structures such as arrays and linked lists.

COMPLEXITY ANALYSIS:-

CODE:

a) **Merge Sort**

```
#include

<stdio.h> #include <stdlib.h>

void merge(int arr[], int left, int mid, int right)

   {int i, j, k;

   int n1 = mid - left + 1;

   int n2 = right - mid;

   int          lArr[n1],

   rArr[n2]; for (i = 0; i

   < n1; i++)
```

```
        lArr[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        rArr[j] = arr[mid + 1 + j];
    i = 0; j = 0; k = left; while (i
    < n1 && j < n2)
        {if (lArr[i] <= rArr[j]) {
            arr[k] = lArr[i]; i++;
        } else { arr[k] =
        rArr[j]; j++;
        }
        k++
        ;
    } while (i <
    n1) { arr[k] =
    lArr[i];i++;
    k++;
    } while (j <
    n2) { arr[k] =
    rArr[j];j++;
    k++;
    }
}
void mergeSort(int arr[], int left, int right)
    {if (left < right) { int mid = left +
        (right - left) / 2; mergeSort(arr,
        left, mid); mergeSort(arr, mid +
        1, right); merge(arr, left, mid,
        right);
    }
} int main()
{
```

```
    int arr[] = { 12, 11, 9, 7, 6, 3 }; int
    N = sizeof(arr) / sizeof(arr[0]);
    printf("Given array is:\n"); for
    (int i = 0; i < N; i++)
        printf("%d ", arr[i]);
    printf("\n"); mergeSort(arr,
    0, N - 1); printf("\nSorted
    array is:\n"); for (int i = 0;
    i < N; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

b) **Insertion sort**

```
#include
<math.h> #include <stdio.h>
void insertionSort(int arr[], int
N)
    {for (int i = 1; i < N; i++) { int
        key = arr[i]; int j = i - 1; while
        (j >= 0 && arr[j] > key) {arr[j
        + 1] = arr[j]; j = j - 1;

        }
        arr[j + 1] = key;
    }
} int main()
{
    int arr[] = { 12, 11, 19, 7, 6 }; int
    N = sizeof(arr) / sizeof(arr[0]);
    printf("Unsorted array: "); for (int
    i = 0; i < N; i++)
        {printf("%d ", arr[i]);
```

```
    }                printf("\n");
    insertionSort(arr,      N);
    printf("Sorted array: ");
    for (int i = 0; i < N; i++)
    { printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

c) **Bubble Sort** *#include*

```
<stdio.h> void swap(int* arr,
int i, int j) {int temp = arr[i];
arr[i]   =   arr[j];   arr[j]   =
temp;
} void bubbleSort(int arr[], int
n)
    {int i, j; for (i = 0; i < n -
    1; i++) { for (j = 0; j < n
    - i - 1; j++) {
        if (arr[j] > arr[j + 1])
            swap(arr, j, j + 1);
        }
    }
} int main()
{
    int arr[] = { 6, 0, 1, 7 }; int N =
    sizeof(arr)    /    sizeof(arr[0]);
    printf("Unsorted array: "); for (int
    i = 0; i < N; i++)
        {printf("%d ", arr[i]);
    }                printf("\n");
    bubbleSort(arr,        N);
```

```
    printf("Sorted array: ");
    for (int i = 0; i < N; i++)
    { printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

OUTPUT:

a)

```
Given array is:
12 11 9 7 6 3

Sorted array is:
3 6 7 9 11 12
```

b)

```
Unsorted array: 12 11 19 7 6
Sorted array: 6 7 11 12 19
```

c)

```
Unsorted array: 6 0 1 7
Sorted array: 0 1 6 7
```

VIVA QUESTIONS:-

Q1. What is the basic idea behind Merge Sort?

ANS. Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts each half, and then merges the two sorted halves to produce the sorted array.

Q2. How does Insertion Sort work?

ANS. Insertion Sort builds the sorted array one element at a time by repeatedly picking the next element and inserting it into the correct position in the already sorted portion of the array.

Q3. What is the time complexity of Bubble Sort in the best and worst cases?

ANS. The best-case time complexity of Bubble Sort is \($O(n)$\) when the array is already sorted, and the worst-case time complexity is \($O(n^2)$\) when the array is sorted in reverse order.

Q4. Why is Merge Sort more efficient than Insertion Sort for larger datasets?

ANS. Merge Sort has a time complexity of \($O(n \log n)$\) in all cases, making it more efficient than Insertion Sort's \($O(n^2)$\) complexity for large datasets, as the logarithmic factor in Merge Sort allows it to handle larger arrays more efficiently.

Q5. Can you explain the space complexity of Merge Sort?

ANS. Merge Sort has a space complexity of \($O(n)$\) because it requires additional space equal to the size of the input array for the temporary arrays used during the merging process.

EXPERIMENT :- 02

AIM: To implement

a) Quick sort in C programming Language

b) Count sort in C programming Language

c) Radix sort in C programming Language

THEORY:-

a) **Quick Sort** is a divide-and-conquer sorting algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two subarrays, according to whether they are less than or greater than the pivot. The subarrays are then sorted recursively.

Steps:

1. Choose a pivot element from the array (commonly the last element, first element, or middle element).
2. Rearrange the array so that all elements less than the pivot are on its left, and all elements greater than the pivot are on its right. This process is called partitioning.
3. Recursively apply the same steps to the left and right sub-arrays.

b) **Counting Sort** is a non-comparative sorting algorithm that works by counting the number of occurrences of each unique element in the array. The count information is then used to place the elements in the correct position in the sorted array.

Steps:

1. Find the maximum element in the array to define the range of the count array.
2. Initialize a count array with zeros, and store the count of each element.
3. Modify the count array to store the cumulative count of each element.
4. Use the cumulative count to place each element in its correct position in the output array.

c) **Radix Sort** is an integer sorting algorithm that sorts numbers by processing individual digits. It works by sorting the numbers digit by digit, starting from the least significant digit (LSD) to the most significant digit (MSD) using a stable subroutine like Counting Sort.

Steps:

1. Find the maximum number to determine the number of digits.
2. Perform Counting Sort on each digit, starting from the LSD.

COMPLEXITY ANALYSIS:

a) Quick Sort

b) Count Sort

c) Radix Sort

CODE:

a)

i) **Quick Sort**

```
#include <stdio.h> #include <stdlib.h>
int compare(const void* a, const void*
b)
{ return (*(int*)a - *(int*)b);
}         int
main()
{ int arr[] = { 7, 19 , 1, 3, 13 }; int n
   =   sizeof(arr)   /   sizeof(arr[0]);
   qsort(arr, n, sizeof(int), compare);
   printf("Sorted array: \n"); for (int
   i = 0; i < n; i++) {printf("%d ",
   arr[i]);
   }   return
0;    }
```

ii)

```
Partition
Function #include
<stdio.h> void
swap(int *a, int *b)
   {int temp = *a;
   *a = *b;
   *b = temp;
}
int partition(int arr[], int low, int high)
   {int pivot = arr[high]; int i = (low -
   1);

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

```
        {if (arr[j] < pivot) { i++;
            swap(&arr[i],
            &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
return (i + 1); }
void quickSort(int arr[], int low, int high)
    {if  (low  <  high)  {  int  pi  =
        partition(arr,     low,     high);
        quickSort(arr,  low,  pi  -  1);
        quickSort(arr, pi + 1, high);
    }
}
void printArray(int arr[], int size)
    {for (int i = 0; i < size; i++) printf("%d
        ", arr[i]);
    printf("\n");
} int
main()
{int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n); int arr[n];
    printf("Enter the elements of the array: "); for
    (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Original  array:
    ");  printArray(arr,  n);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: ");
```

```
printArray(arr,          n);
return 0;
}
```

b)

```
#include          <stdio.h>          void
countingSort(int array[], int size)
 {int output[10]; int max =
 array[0]; for (int i = 1; i <
 size; i++)
  {if (array[i] > max) max
   = array[i];
 }
 int count[10]; for (int i = 0; i
 <= max; ++i) {count[i] = 0;
 }
 for    (int    i    =    0;    i    <    size;    i++)
  {count[array[i]]++;
 }
 for (int i = 1; i <= max; i++) {count[i]
  += count[i - 1];
 }
 for (int i = size - 1; i >= 0; i--)
  { output[count[array[i]] - 1]
  = array[i];count[array[i]]--;
 }
 for (int i = 0; i < size; i++) {array[i]
  = output[i];
 }
}
void printArray(int array[], int size)
 {for (int i = 0; i < size; ++i)
```

```
  { printf("%d ", array[i]);
  }
  printf("\n");
} int main()
{
  int array[] = {7, 2, 2, 8, 3, 3, 9}; int n
  =   sizeof(array)   /   sizeof(array[0]);
  countingSort(array,                          n);
  printArray(array, n);
}
```

c)

```
#include      <stdio.h>      int
getMax(int   array[],   int   n)
{int max = array[0]; for (int
i = 1; i < n;
    i++) if (array[i] > max)
    max = array[i];
  return max;
}
void countingSort(int array[], int size, int place)
  {int output[size + 1];
  int max = (array[0] / place) % 10; for
  (int i = 1; i < size; i++) {
    if (((array[i] / place) % 10) >
      max) max = array[i];
  }
  int count[max + 1]; for
  (int i = 0; i < max;
    ++i) count[i] = 0; for (int i = 0; i < size; i++)
  count[(array[i] / place) % 10]++; for (int i =
```

```
  1; i < 10; i++) count[i] += count[i - 1]; for
  (int i = size - 1; i >= 0; i--) {
  output[count[(array[i] / place) % 10] - 1] =
  array[i];count[(array[i] / place) % 10]--;
  } for (int i = 0; i <
   size;
    i++) array[i] = output[i];
}
void radixsort(int array[], int size) {int max =
  getMax(array, size); for (int place = 1; max /
  place > 0; place *= 10)
    countingSort(array, size, place);
}
void printArray(int array[], int size)
  {for (int i = 0; i < size; ++i)
  { printf("%d ", array[i]);
  }
  printf("\n");
}
int main() {
  int array[] = {171, 432, 564, 232, 100, 45, 789};
  int n = sizeof(array) / sizeof(array[0]);
  radixsort(array, n); printArray(array, n);
}
```

OUTPUT:

a)

i)

```
Sorted array:
1 3 7 13 19
```

ii)

```
Enter the number of elements: 4
Enter the elements of the array: 1 2 3 4
Original array: 1 2 3 4
Sorted array: 1 2 3 4
```

b)

```
2 2 3 3 7 8 9
```

c)

```
45  100  171  232  432  564  789
```

VIVA QUESTIONS:-

Q1. What is the main principle behind Quick Sort, and what is its average-case time complexity?

ANS. Quick Sort works on the divide-and-conquer principle. It selects a pivot element and partitions the array into two halves: elements less than the pivot and elements greater than the pivot. It then recursively sorts the sub-arrays. The average-case time complexity of Quick Sort is O(n log n).

Q2. How does Count Sort differ from comparison-based sorting algorithms?

ANS. Count Sort is a non-comparison-based sorting algorithm that works by counting the frequency of each element in the input array and using this count to place elements in their correct positions. It is efficient for sorting integers within a known, small range. Its time complexity is O(n + k), where n is the number of elements and k is the range of the input.

Q3. Explain how Radix Sort works and when it is preferred over other sorting algorithms.

ANS. Radix Sort sorts numbers by processing each digit individually, starting from the least significant digit to the most significant one, using a stable sub-sorting algorithm like Count Sort. It is preferred when sorting integers or strings with a fixed number of digits/characters. Radix Sort's time complexity is O(d(n + k)), where d is the number of digits, n is the number of elements, and k is the base.

Q4. What is the role of the partition function in Quick Sort, and how does it affect the algorithm's efficiency?

ANS. The partition function in Quick Sort rearranges the elements in the array such that elements less than the pivot are on the left, and elements greater than the pivot are on the right. The efficiency of Quick Sort heavily depends on the partitioning method; poor pivot selection can degrade performance to $O(n^2)$ in the worst case.

Q5. Why is Count Sort not suitable for sorting arrays with a large range of elements?

ANS. Count Sort is not suitable for sorting arrays with a large range of elements because it requires additional memory proportional to the range of the input (O(k)). If the range is significantly larger than the number of elements, the space complexity and inefficiency of memory usage become impractical.