Practical – 7

AIM: Implementation and Time analysis of sorting algorithms – Bubble sort, Selection sort, Insertion sort, Merge sort and Quicksort.

Program

```
#include <iostream>
using namespace std;

void bubble_sort(int a[], int n)
{
  for (int i = 0; i < n; i++)
  {
    if (i < n - 1)
    {
      if (a[i] > a[i + 1])
      {
        int temp = a[i];
        a[i] = a[i + 1];
        a[i + 1] = temp;
      }
    }
  }
}

void selectionSort(int a2[], int n)
{
  for (int i = 0; i < n - 1; i++)
  {
    int min = i;
    for (int j = i + 1; j < n; j++)
    {
      if (a2[min] > a2[j])
      {
        min = j;
      }
    }

    if (min != i)
    {
      int temp = a2[i];
      a2[i] = a2[min];
      a2[min] = temp;
    }
  }
}

void insertionsort(int a[3], int n)
{

  for (int i = 1; i < n; i++)
```

```
  {

    int temp = a[i];
    int j = i - 1;

    while (j >= 0 && a[j] > a[i])
    {
      a[j + 1] = a[j];
      j = j - 1;
    }
    a[j + 1] = temp;
  }
}

void display(int a[], int n)
{
  for (size_t i = 0; i < n; i++)
    cout << a[i] << " | ";
}


int main(int argc, char const *argv[])
{
  int a1[] = {1, 4, 2, 6, 8};
  int a2[] = {1, 100, 2, 6, 8};
  int a3[] = {1, 30, 2, 6, 8};
  int n = sizeof(a1) / sizeof(a1[0]);

  cout << "Unsorted array :   ";
  display(a1, n);
  cout << endl;
  bubble_sort(a1, n);
  cout << "sorted array using Bubble sort :";
  display(a1, n);
  cout << endl;

  cout << "Unsorted array :   ";
  display(a2, n);
  cout << endl;
  selectionSort(a2, n);
  cout << "sorted array using Selection sort :";
  display(a2, n);
  cout << endl;

  cout << "Unsorted array :   ";
  display(a3, n);
  cout << endl;
  insertionsort(a3, n);
  cout << "sorted array using Insertion sort :";
  display(a3, n);

  return 0;
}
```

OUTPUT

```
Unsorted array :  1 | 4 | 2 | 6 | 8 |
sorted array using Bubble sort :1 | 2 | 4 | 6 | 8 |
Unsorted array :   1 | 100 | 2 | 6 | 8 |
sorted array using Selection sort :1 | 2 | 6 | 8 | 100 |
Unsorted array :   1 | 30 | 2 | 6 | 8 |
sorted array using Insertion sort :1 | 2 | 6 | 8 | 30 |
PS E:\BE_Milan\DS\sorting>
```

Merge sort

```
#include <iostream>
using namespace std;

void merge(int array[], int const left, int const mid,
            int const right)
{
      int const subArrayOne = mid - left + 1;
      int const subArrayTwo = right - mid;

      auto *leftArray = new int[subArrayOne],
            *rightArray = new int[subArrayTwo];

      for (auto i = 0; i < subArrayOne; i++)
            leftArray[i] = array[left + i];
      for (auto j = 0; j < subArrayTwo; j++)
            rightArray[j] = array[mid + 1 + j];

      auto indexOfSubArrayOne = 0, indexOfSubArrayTwo = 0;
      int indexOfMergedArray = left;

      while (indexOfSubArrayOne < subArrayOne
            && indexOfSubArrayTwo < subArrayTwo) {
            if (leftArray[indexOfSubArrayOne]
                  <= rightArray[indexOfSubArrayTwo]) {
                  array[indexOfMergedArray]
                        = leftArray[indexOfSubArrayOne];
                  indexOfSubArrayOne++;
            }
            else {
                  array[indexOfMergedArray]
                        = rightArray[indexOfSubArrayTwo];
                  indexOfSubArrayTwo++;
            }
            indexOfMergedArray++;
      }
```

```
	while (indexOfSubArrayOne < subArrayOne) {
		array[indexOfMergedArray]
			= leftArray[indexOfSubArrayOne];
		indexOfSubArrayOne++;
		indexOfMergedArray++;
	}

	while (indexOfSubArrayTwo < subArrayTwo) {
		array[indexOfMergedArray]
			= rightArray[indexOfSubArrayTwo];
		indexOfSubArrayTwo++;
		indexOfMergedArray++;
	}
	delete[] leftArray;
	delete[] rightArray;
}

void mergeSort(int array[], int const begin, int const end)
{
	if (begin >= end)
		return;

	int mid = begin + (end - begin) / 2;
	mergeSort(array, begin, mid);
	mergeSort(array, mid + 1, end);
	merge(array, begin, mid, end);
}

void printArray(int A[], int size)
{
	for (int i = 0; i < size; i++)
		cout << A[i] << " ";
	cout << endl;
}

int main()
{
	int arr[] = { 100, 21, 33, 5, 2, 43 };
	int arr_size = sizeof(arr) / sizeof(arr[0]);

	cout << "Input  array is \n";
	printArray(arr, arr_size);

	mergeSort(arr, 0, arr_size - 1);

	cout << "\nSorted array is \n";
	printArray(arr, arr_size);
	return 0;
}
```

Quicksort

```
#include <iostream>
using namespace std;

int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high; 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 A[], int size)
{
    for (int i = 0; i < size; i++)
        cout << A[i] << " ";
    cout << endl;
}

int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    cout << "Inputed Array\n";
    printArray(arr, 6);
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted Array\n";
    printArray(arr, 6);
}
```

Output of mergesort

```
Input  array is :100 21 33 5 2 43
Using merge sort Sorted array is :2 5 21 33 43 100
```

Output of QuickSort

```
Inputed Array
10 7 8 9 1 5
Using Quicksort Sorted Array
1 5 7 8 9 10
```

Time analysis

Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)

Applications

1. Social Media Feed Ordering:
   - Merge Sort: Used to chronologically order posts and comments in news feeds, ensuring users see the latest content first.
   - Quick Sort: Can be used to prioritize content based on engagement metrics (likes, comments) to show the most popular content first.

2. Online Shopping Product Listings:
   - Bubble Sort/Selection Sort: Used for small-scale sorting when filtering products by price or rating, suitable for simple apps.
   - Insertion Sort: Efficient for handling smaller, dynamically updated product lists within app interfaces.

- Merge Sort/Quick Sort: Ideal for large data sets of products, enabling fast sorting by various criteria (price, popularity, reviews) with good scalability.

3. Navigation Apps & Route Planning:
   - Heap Sort: Used to prioritize destinations and determine the fastest route based on real-time traffic conditions.
   - Quick Sort: Can be used to sort possible routes based on distance, estimated travel time, or user preferences.

4. Music Streaming Apps:
   - Merge Sort: Used to sort songs or playlists alphabetically, by artist, or by release date.
   - Quick Sort: Can be used to quickly sort songs based on genre, mood, or user-defined criteria.

5. To-Do List Apps:
   - Insertion Sort: Efficient for small to-do lists, keeping tasks ordered by priority or deadline.
   - Merge Sort/Quick Sort: Suitable for larger lists, enabling sorting by categories, completion status, or due dates.

6. Image/Video Editing Apps:
   - Selection Sort/Bubble Sort: Used for small sets of filters or effects, allowing simple ordering for user selection.
   - Merge Sort/Quick Sort: Can be used for large libraries of filters or effects, enabling fast sorting by category, popularity, or user ratings.

7. Real-Time Data Visualization Apps:
   - Heap Sort: Used to prioritize and display the most relevant data points in real-time charts and graphs.
   - Quick Sort: Can be used to dynamically sort incoming data streams based on various criteria, ensuring efficient visualization.

8. Educational Apps & Games:
   - Bubble Sort/Selection Sort: Introduced as simple sorting concepts in educational games and activities.
   - Merge Sort/Quick Sort: Used for more complex sorting challenges in advanced educational or competitive games