Practical - 3

Aim:

Write user defined functions for the following sorting methods and compare their performance by time measurement with random data and Sorted data.

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
6. Randomized Quick Sort

Code:

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

int count_steps_bubble = 0;
int count_steps_insertion = 0;
int count_steps_selection = 0;
int count_steps_quick = 0;
int count_steps_merge = 0;
int count_steps_randomized_quick = 0;

// Utility function to print array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}



//  1) BUBBLE SORT
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
```

```
        for (int j = 0; j < n - i - 1; j++) {
            count_steps_bubble++;
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}


//  2) INSERTION SORT
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) {
            count_steps_insertion++;
            arr[j + 1] = arr[j];
            j--;
        }
    }
}


//  3) SELECTION SORT
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;

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

```
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }

        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}


// 4) QUICK SORT
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        count_steps_quick++;
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    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);
    }
}

// 5) MERGE SORT
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    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];
    }

    int i = 0, j = 0, k = l;

    while (i < n1 && j < n2) {
        count_steps_merge++;
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
```

```
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

// 6) RANDOMIZED QUICK SORT
int partition_random(int arr[], int low, int high) {
    int randomIndex = low + rand() % (high - low + 1);
    int temp = arr[randomIndex];
    arr[randomIndex] = arr[high];
    arr[high] = temp;
```

```
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        count_steps_randomized_quick++;
        if (arr[j] < pivot) {
            i++;
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1);
}

int randomized_quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition_random(arr, low, high);

        randomized_quickSort(arr, low, pi - 1);
        randomized_quickSort(arr, pi + 1, high);
    }
    return 0;
}

int main() {
    FILE *fp;
    fp = fopen("arr2.txt", "r");
    int n = 0;
```

```
    fscanf(fp, "%d", &n);

    int* array = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        fscanf(fp, "%d", &array[i]);
    }

    int* arr = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        arr[i] = array[i];
    }

    printf("Original Array: ");
    printArray(arr, n);
    printf("Size of Array: %d\n", n);

    // Bubble Sort
    int* a1 = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        a1[i] = array[i];
    }
    bubbleSort(a1, n);
    printf("\nBubble Sort: ");
    printArray(a1, n);
    printf("Number of Steps: %d\n", count_steps_bubble);

    // Insertion Sort
    int* a2 = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        a2[i] = array[i];
    }
    insertionSort(a2, n);
    printf("Insertion Sort: ");
```

```
    printArray(a2, n);
    printf("Number of Steps: %d\n", count_steps_insertion);

    // Selection Sort
    int* a3 = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        a3[i] = array[i];
    }
    selectionSort(a3, n);
    printf("Selection Sort: ");
    printArray(a3, n);
    printf("Number of Steps: %d\n", count_steps_selection);

    // Quick Sort
    int* a4 = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        a4[i] = array[i];
    }
    quickSort(a4, 0, n - 1);
    printf("Quick Sort: ");
    printArray(a4, n);
    printf("Number of Steps: %d\n", count_steps_quick);

    // Randomized Quick Sort
    int* a6 = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        a6[i] = array[i];
    }
    randomized_quickSort(a6, 0, n - 1);
    printf("Randomized Quick Sort: ");
    printArray(a6, n);
    printf("Number of Steps: %d\n", count_steps_randomized_quick);

    // Merge Sort
```

```
    int* a5 = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        a5[i] = array[i];
    }
    mergeSort(a5, 0, n - 1);
    printf("Merge Sort: ");
    printArray(a5, n);
    printf("Number of Steps: %d\n", count_steps_merge);

    return 0;
}
```

Output:

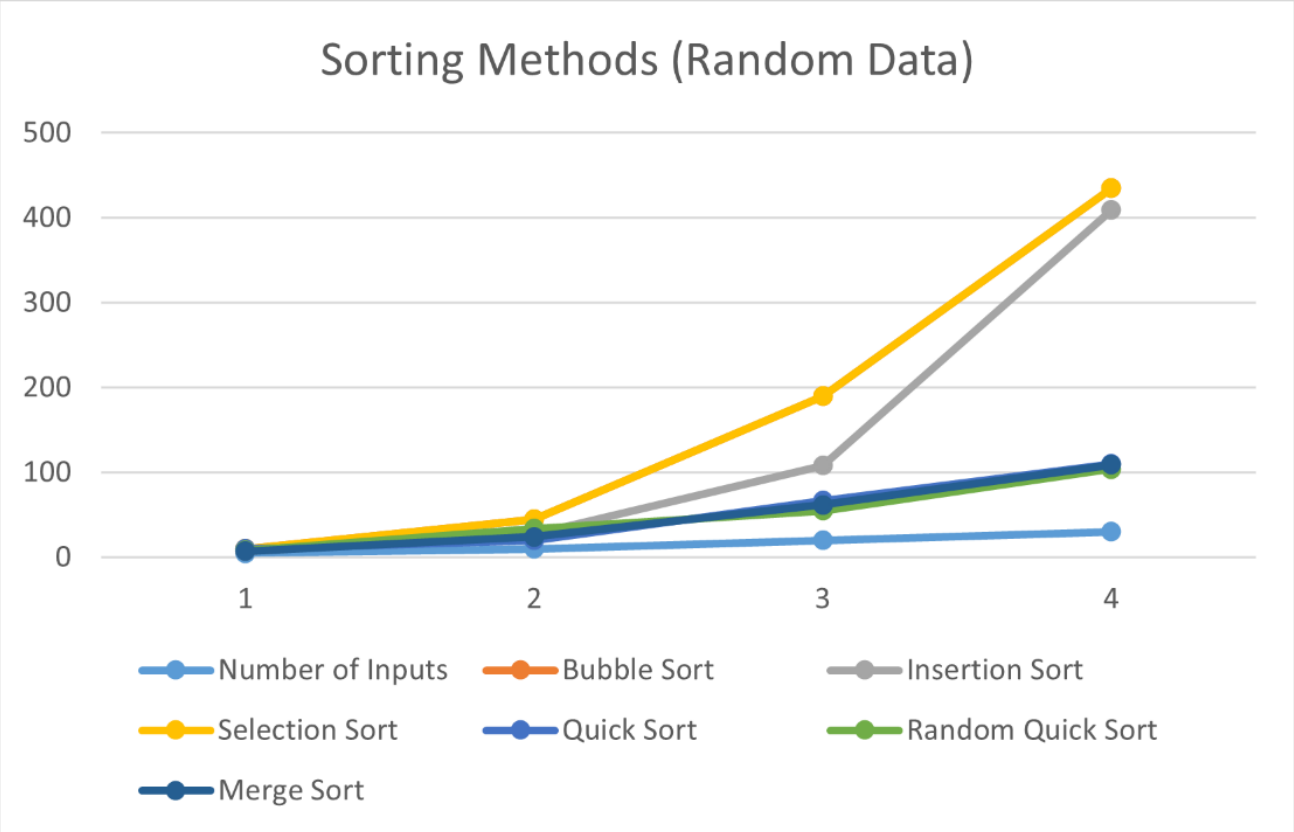
```
PS C:\Users\Lenovo\Desktop\Sem4\DAA\Prac3> .\main.exe
Original Array: 24 57 69 73 82
Size of Array: 5

Bubble Sort: 24 57 69 73 82
Number of Steps: 10
Insertion Sort: 24 57 69 73 82
Number of Steps: 0
Selection Sort: 24 57 69 73 82
Number of Steps: 10
Quick Sort: 24 57 69 73 82
Number of Steps: 10
Randomized Quick Sort: 24 57 69 73 82
Number of Steps: 7
Merge Sort: 24 57 69 73 82
Number of Steps: 7
```

Analysis:

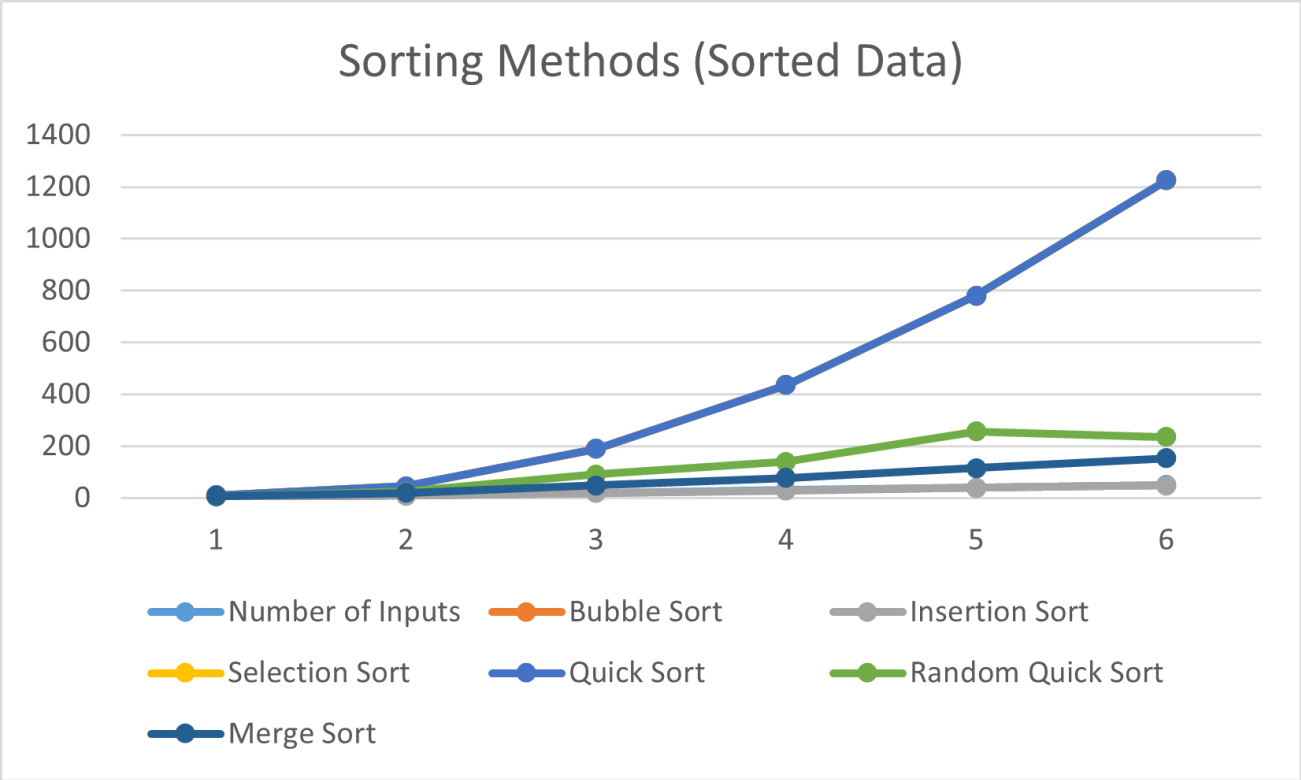
Random Data:

Number of Inputs	Bubble Sort	Insertion Sort	Selection Sort	Quick Sort	Random Quick Sort	Merge Sort
5	10	7	10	10	8	7
10	45	27	45	20	34	24
20	190	108	190	67	55	62
30	435	409	435	110	104	109
40	780	636	780	164	174	158
50	1225	1126	1225	228	238	220



Sorted Data:

Number of Inputs	Bubble Sort	Insertion Sort	Selection Sort	Quick Sort	Random Quick Sort	Merge Sort
5	10	4	10	10	7	7
10	45	9	45	45	24	19
20	190	19	190	190	91	48
30	435	29	435	435	139	77
40	780	39	780	780	257	116
50	1225	49	1225	1225	235	153



Conclusion:

it is clearly observed that different sorting algorithms exhibit significantly different performance depending on the input size and the nature of data. Simple comparison-based algorithms like **Bubble Sort** and **Selection Sort** show a rapid increase in the number of steps as the input size grows, confirming their $O(n^2)$ time complexity and making them inefficient for large datasets.

Insertion Sort performs comparatively better, especially on already sorted data, where it requires minimal steps. This demonstrates its adaptive nature and efficiency for small or nearly sorted inputs.

On the other hand, **Quick Sort** and **Merge Sort** consistently perform better for larger inputs. Merge Sort shows stable performance regardless of data order due to its O (n Log n) complexity, while Quick Sort performs very efficiently on random data but may degrade on sorted data depending on pivot selection.

Overall, this Practical validates the theoretical time complexities studied in class and highlights the importance of selecting an appropriate sorting algorithm based on input size and data characteristics rather than relying on a single method for all scenarios.