Write a C program to implement Selection sort, Merge sort show all the following operations in switch case, i) Enter values in array, ii) Sort using selection sort, iii) Sort using Merge sort, iv) Display array

Code:

```c
#include <stdio.h>

void selectionSort(int A[], int len) {
  for (int i = 0; i < len - 1; i++) {
    int smallest = i;
    for (int j = i + 1; j < len; j++) {
      if (A[smallest] > A[j]) {
        smallest = j;
      }
    }
    int temp = A[smallest];
    A[smallest] = A[i];
    A[i] = temp;
  }
}

void merge(int arr[], int beg, int mid, int end) {
  for (int i = beg; i <= end; i++) {
    printf("%d ", arr[i]);
  }

  int a = beg, b = mid + 1;
  int temp_arr[end - beg + 1];
  for (int i = 0; i < end - beg + 1; i++) {
    if (a > mid && b > end) {
      break;
    } else if (b > end) {
      temp_arr[i] = arr[a];
      a++;
    } else if (a > mid) {
      temp_arr[i] = arr[b];
      b++;
    } else if (arr[a] > arr[b]) {
      temp_arr[i] = arr[b];
      b++;
    } else {
      temp_arr[i] = arr[a];
      a++;
    }
  }
```

```
  printf("\nAfter sort: ");
  for (int i = 0; i < end - beg + 1; i++) {
    printf("%d ", temp_arr[i]);
    arr[beg + i] = temp_arr[i];
  }
  printf("\n\n");
}

void mergeSort(int arr[], int beg, int end) {
  if (beg < end) {
    int mid = (beg + end) / 2;
    mergeSort(arr, beg, mid);
    mergeSort(arr, mid + 1, end);
    merge(arr, beg, mid, end);
  }
}

int main() {
  int n, choice;
  printf("Enter the number of elements in the array: ");
  scanf("%d", &n);

  int arr[n];

  do {
    printf("\nMenu:\n1. Enter values in array\n2. Sort using selection
sort\n3. Sort using merge sort\n4. Display array\n5. Exit\nEnter your
choice: ");
    scanf("%d", &choice);

    switch (choice) {
    case 1:
      printf("Enter %d elements:\n", n);
      for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
      }
      break;

    case 2:
      selectionSort(arr, n);
      printf("Array sorted using Selection Sort.\n");
      break;

    case 3:
      mergeSort(arr, 0, n - 1);
      printf("Array sorted using Merge Sort.\n");
```

```
      break;

    case 4:
      printf("Array elements: ");
      for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
      }
      printf("\n");
      break;

    case 5:
      printf("Exiting the program.\n");
      break;

    default:
      printf("Invalid choice! Please enter a valid option.\n");
      break;
    }
  } while (choice != 5);
  return 0;
}
```

Output:

```
Enter the number of elements in the array: 5
Menu:
1. Enter values in array
2. Sort using selection sort
3. Sort using merge sort
4. Display array
5. Exit
Enter your choice: 1
Enter 5 elements:
54
65
23
53
12

Menu:
1. Enter values in array
2. Sort using selection sort
3. Sort using merge sort
4. Display array
5. Exit
Enter your choice: 2
Array sorted using Selection Sort.
```

```
Menu:
1. Enter values in array
2. Sort using selection sort
3. Sort using merge sort
4. Display array
5. Exit
Enter your choice: 4
Array elements: 12 23 53 54 65

Menu:
1. Enter values in array
2. Sort using selection sort
3. Sort using merge sort
4. Display array
5. Exit
Enter your choice: 1
Enter 5 elements:
12
56
97
2
4
86
15
94
23
48

Menu:
1. Enter values in array
2. Sort using selection sort
3. Sort using merge sort
4. Display array
5. Exit
Enter your choice: 3
Array sorted using Merge Sort.

Menu:
1. Enter values in array
2. Sort using selection sort
3. Sort using merge sort
4. Display array
5. Exit
Enter your choice: 4
Array elements: 2 4 12 15 23 56 86 94 97
```

2. Sort the following numbers using Quick sort.
   56,88,74,64,35,12,95,37,24

Code:

```c
#include <stdio.h>

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++;
      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);
  }
}

int main()
{
  int arr[] = {56, 88, 74, 64, 35, 12, 95, 37, 24};
  int n = sizeof(arr) / sizeof(arr[0]);
```

```c
  printf("Original Array: ");
  for (int i = 0; i < n; i++)
  {
    printf("%d ", arr[i]);
  }
  printf("\n");

  quickSort(arr, 0, n - 1);

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

  return 0;
}
```

Output:

```
PS C:\Users\Ajay kumar\Desktop\SEIT-B> cd "c:\Users\Ajay
kumar\Desktop\SEIT-B\DSA\Lab\6\" ; if ($?) { gcc quickSort.c -o
quickSort } ; if ($?) { .\quickSort }

Original Array: 56 88 74 64 35 12 95 37 24
Sorted Array: 12 24 35 37 56 64 74 88 95

PS C:\Users\Ajay kumar\Desktop\SEIT-B\DSA\Lab\6>
```

3. List and compare the running time in various sorting techniques.

Code:

Sorting algorithms have different time complexities, and their performance can vary depending on the size and characteristics of the input data. Here's a list of some common sorting techniques and a brief comparison of their running times:

Bubble Sort:
Time Complexity: O(n^2) in the worst case, where n is the number of elements.
Performance: Inefficient for large datasets.

Selection Sort:
Time Complexity: O(n^2) in the worst case.
Performance: Similar to bubble sort, it's inefficient for large datasets.

Insertion Sort:
Time Complexity: O(n^2) in the worst case.
Performance: Suitable for small datasets or nearly sorted data.

Quick Sort:
Time Complexity: O(n^2) in the worst case, but on average, it's O(n log n).
Performance: Fastest on average for general data and often used in practice.

Merge Sort:
Time Complexity: Always O(n log n), both in the worst and average cases.
Performance: Consistently efficient, especially for large datasets, but requires additional space for merging.

Heap Sort:
Time Complexity: O(n log n) in the worst and average cases.
Performance: Efficient and has a good balance between time and space complexity.

Radix Sort:
Time Complexity: O(k * n) in the worst and average cases, where k is the number of digits.
Performance: Efficient for sorting integers with limited digits.

Counting Sort:
Time Complexity: O(n + k) in the worst and average cases, where k is the range of input values.
Performance: Efficient for small integers with a limited range.

Bucket Sort:
Time Complexity: O(n^2) in the worst case but can be O(n) in the average case.
Performance: Effective for uniformly distributed data.

4.Write a program for insertion sort and show output after every pass

Code:

```c
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;

        printf("Pass %d: ", i);
        for (int k = 0; k < n; k++) {
            printf("%d ", arr[k]);
        }
        printf("\n");
    }
}

int main() {
    int n;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Original 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;
}
```

Output:

```
PS C:\Users\Ajay kumar\Desktop\SEIT-B\DSA\Lab\6> cd "c:\Users\Ajay

Enter the number of elements in the array: 4
Enter 4 elements:
4
2
3
1
Original Array: 4 2 3 1
Pass 1: 2 4 3 1
Pass 2: 2 3 4 1
Pass 3: 1 2 3 4
Sorted Array: 1 2 3 4
PS C:\Users\Ajay kumar\Desktop\SEIT-B\DSA\Lab\6>
```