

Nitin Chaudhary

9921103163

F7

APS Lab Week 1

We are given an array of n distinct numbers; where n is large numbers are randomly generated.

The task is to

a) Sort the entire array using selection sort, bubble sort, insertion sort, quick sort and merge sort. Print the total number of comparisons done in each of the sorting algorithm.

```
#include<bits/stdc++.h>
#include <sys/time.h>
int counti=0;
using namespace std;
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void insertionSort(int arr[], int n)
{
    counti=0;
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key, to one
        // position ahead of their
        // current position

        while (j >= 0 && arr[j] > key)
```

```

    {
        arr[j + 1] = arr[j];
        j = j - 1;
        counti++;
    }counti++;
    arr[j + 1] = key;
}
cout<<"\nnumber of comparisions takes place in insertion sort are"<<counti<<endl;
}

```

```

void bubbleSort(int arr[], int n)
{
    counti=0;
    int i, j;
    for (i = 0; i < n - 1; i++){

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++){
            counti++;
            if (arr[j] > arr[j + 1]){
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
    cout<<"\nnumber of comparisions takes place in bubble sort are"<<counti<<endl;
}

```

```

void selectionSort(int arr[], int n)
{
    counti=0;
    int i, j, min_idx;

    // One by one move boundary of
    // unsorted subarray
    for (i = 0; i < n-1; i++)
    {

        // Find the minimum element in
        // unsorted array
        min_idx = i;

```

```

        for (j = i+1; j < n; j++){
            counti++;
            if (arr[j] < arr[min_idx])
                min_idx = j;

            // Swap the found minimum element
            // with the first element
            if(min_idx!=i)
                swap(&arr[min_idx], &arr[i]);
        }

    }

    cout<<"\nNumber of comparisions takes place in selection Sort are "<<counti;
}

```

```

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

    // Create temp arrays
    int *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (int i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (int j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    int indexOfSubArrayOne
        = 0, // Initial index of first sub-array
        indexOfSubArrayTwo
        = 0; // Initial index of second sub-array
    int indexOfMergedArray
        = left; // Initial index of merged array

    // Merge the temp arrays back into array[left..right]

```

```

while (indexOfSubArrayOne < subArrayOne
    && indexOfSubArrayTwo < subArrayTwo) {
    counti++;
    if (leftArray[indexOfSubArrayOne]
        <= rightArray[indexOfSubArrayTwo]) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
    }
    else {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
    }
    indexOfMergedArray++;
}
// Copy the remaining elements of
// left[], if there are any
while (indexOfSubArrayOne < subArrayOne) {
    array[indexOfMergedArray]
        = leftArray[indexOfSubArrayOne];
    indexOfSubArrayOne++;
    indexOfMergedArray++;
}
// Copy the remaining elements of
// right[], if there are any
while (indexOfSubArrayTwo < subArrayTwo) {
    array[indexOfMergedArray]
        = rightArray[indexOfSubArrayTwo];
    indexOfSubArrayTwo++;
    indexOfMergedArray++;
}
delete[] leftArray;
delete[] rightArray;
}

// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSort(int array[], int const begin, int const end)

```

```

{
    if (begin >= end)
        return; // Returns recursively

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

```

```

int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i
        = (low
            - 1); // Index of smaller element and indicates
                // the right position of pivot found so far

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        counti++;
        if (arr[j] < pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

```

```

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
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);
    }
}

```

```

int main(){
int arr[5]={41,67,34,0,69};
struct timeval stop, start;
gettimeofday(&start, NULL);
insertionSort(arr,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by insertion sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr1[5]={41,67,34,0,69};
gettimeofday(&start, NULL);
bubbleSort(arr1,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by bubble sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr2[5]={41,67,34,0,69};
gettimeofday(&start, NULL);
selectionSort(arr2,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by selection sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr3[5]={41,67,34,0,69};
counti=0;
gettimeofday(&start, NULL);
mergeSort(arr3,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisons takes place in merge Sort are "<<counti;
cout<<"\nTime taken by merge sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr4[5]={41,67,34,0,69};
counti=0;

```

```

gettimeofday(&start, NULL);
quickSort(arr4,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisions takes place in quick Sort are "<<counti;
cout<<"\nTime taken by quick sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
return 0;
}

```

```

number of comparisions takes place in insertion sort are9
Time taken by insertion sort986 us
number of comparisions takes place in bubble sort are10
Time taken by bubble sort1176 us
Number of comparisions takes place in selection Sort are 10
Time taken by selection sort825 us
Number of comparisions takes place in merge Sort are 7
Time taken by merge sort0 us
Number of comparisions takes place in quick Sort are 9
Time taken by quick sort0 us
PS C:\Users\HP\Desktop\4thsem\APSlab>

```

b) Take sorted array from part a) and again run all the sorting algorithm functions. Print the total number of comparisons done in each of the sorting algorithm.

```

#include<bits/stdc++.h>
#include <sys/time.h>
int counti=0;
using namespace std;
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void insertionSort(int arr[], int n)

```

```

{
    counti=0;
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key, to one
        // position ahead of their
        // current position

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
            counti++;
        }counti++;
        arr[j + 1] = key;
    }
    cout<<"\nnumber of comparisions takes place in insertion sort are"<<counti<<endl;
}

```

```

void bubbleSort(int arr[], int n)
{
    counti=0;
    int i, j;
    for (i = 0; i < n - 1; i++){

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++){
            counti++;
            if (arr[j] > arr[j + 1]){
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
    cout<<"\nnumber of comparisions takes place in bubble sort are"<<counti<<endl;
}

```



```

}
void selectionSort(int arr[], int n)
{
    counti=0;
    int i, j, min_idx;

    // One by one move boundary of
    // unsorted subarray
    for (i = 0; i < n-1; i++)
    {

        // Find the minimum element in
        // unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++){
            counti++;
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element
        // with the first element
        if(min_idx!=i)
            swap(&arr[min_idx], &arr[i]);
        }

    }
    cout<<"\nNumber of comparisions takes place in selection Sort are "<<counti;
}

```

```

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

    // Create temp arrays
    int *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];

```

```

// Copy data to temp arrays leftArray[] and rightArray[]
for (int i = 0; i < subArrayOne; i++)
    leftArray[i] = array[left + i];
for (int j = 0; j < subArrayTwo; j++)
    rightArray[j] = array[mid + 1 + j];

int indexOfSubArrayOne
    = 0, // Initial index of first sub-array
indexOfSubArrayTwo
    = 0; // Initial index of second sub-array
int indexOfMergedArray
    = left; // Initial index of merged array

// Merge the temp arrays back into array[left..right]
while (indexOfSubArrayOne < subArrayOne
    && indexOfSubArrayTwo < subArrayTwo) {
    counti++;
    if (leftArray[indexOfSubArrayOne]
        <= rightArray[indexOfSubArrayTwo]) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
    }
    else {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
    }
    indexOfMergedArray++;
}
// Copy the remaining elements of
// left[], if there are any
while (indexOfSubArrayOne < subArrayOne) {
    array[indexOfMergedArray]
        = leftArray[indexOfSubArrayOne];
    indexOfSubArrayOne++;
    indexOfMergedArray++;
}
// Copy the remaining elements of
// right[], if there are any

```

```

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

// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Returns recursively

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

```

```

int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i
        = (low
            - 1); // Index of smaller element and indicates
                // the right position of pivot found so far

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        counti++;
        if (arr[j] < pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
}

```

```

    }
}
swap(&arr[i + 1], &arr[high]);
return (i + 1);
}

```

/* The main function that implements QuickSort

arr[] --> Array to be sorted,

low --> Starting index,

high --> Ending index */

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);
    }
}

```

```

int main(){
int arr[5]={0,34,41,67,69};
struct timeval stop, start;
gettimeofday(&start, NULL);
insertionSort(arr,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by insertion sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
gettimeofday(&start, NULL);
bubbleSort(arr,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by bubble sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
gettimeofday(&start, NULL);
selectionSort(arr,5);
gettimeofday(&stop, NULL);

```

```

cout<<"\nTime taken by selection sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
counti=0;
gettimeofday(&start, NULL);
mergeSort(arr,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisions takes place in merge Sort are "<<counti;
cout<<"\nTime taken by merge sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
counti=0;
gettimeofday(&start, NULL);
quickSort(arr,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisions takes place in quick Sort are "<<counti;
cout<<"\nTime taken by quick sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
return 0;
}

```

```

number of comparisions takes place in insertion sort are4
Time taken by insertion sort1000 us
number of comparisions takes place in bubble sort are10
Time taken by bubble sort1002 us
Number of comparisions takes place in selection Sort are 10
Time taken by selection sort1014 us
Number of comparisions takes place in merge Sort are 7
Time taken by merge sort0 us
Number of comparisions takes place in quick Sort are 10
Time taken by quick sort0 us
PS C:\Users\HP\Desktop\4thsem\APSlab>

```

c) Change the functions to take a flag “order” as argument. This order can be „d” or „a” for descending and ascending respectively. The function will sort the array in descending and ascending order depending on the flag value.

```
#include<bits/stdc++.h>
#include <sys/time.h>
int counti=0;
using namespace std;
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void insertionSort(int arr[], int n)
{
    counti=0;
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key, to one
        // position ahead of their
        // current position

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
            counti++;
        }
        counti++;
        arr[j + 1] = key;
    }
    cout<<"\nnumber of comparisions takes place in insertion sort are"<<counti<<endl;
}
```

```

void bubbleSort(int arr[], int n)
{
    counti=0;
    int i, j;
    for (i = 0; i < n - 1; i++){

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++){
            counti++;
            if (arr[j] > arr[j + 1]){
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
    cout<<"\nnumber of comparisions takes place in bubble sort are"<<counti<<endl;
}

void selectionSort(int arr[], int n)
{
    counti=0;
    int i, j, min_idx;

    // One by one move boundary of
    // unsorted subarray
    for (i = 0; i < n-1; i++)
    {

        // Find the minimum element in
        // unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++){
            counti++;
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }

        // Swap the found minimum element
        // with the first element
        if(min_idx!=i)
            swap(&arr[min_idx], &arr[i]);
    }
}

```

```

    }

}
cout<<"\nNumber of comparisions takes place in selection Sort are "<<counti;
}

```

```

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

    // Create temp arrays
    int *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (int i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (int j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    int indexOfSubArrayOne
        = 0, // Initial index of first sub-array
        indexOfSubArrayTwo
        = 0; // Initial index of second sub-array
    int indexOfMergedArray
        = left; // Initial index of merged array

    // Merge the temp arrays back into array[left..right]
    while (indexOfSubArrayOne < subArrayOne
        && indexOfSubArrayTwo < subArrayTwo) {
        counti++;
        if (leftArray[indexOfSubArrayOne]
            <= rightArray[indexOfSubArrayTwo]) {
            array[indexOfMergedArray]
                = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
    }
}

```



```

        else {
            array[indexOfMergedArray]
                = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // left[], if there are any
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // right[], if there are any
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Returns recursively

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

```

```

int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i
        = (low
            - 1); // Index of smaller element and indicates
                // the right position of pivot found so far

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        counti++;
        if (arr[j] < pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

```

```

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
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);
    }
}

```

```

void insertionSortd(int arr[], int n)
{
    counti=0;
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key, to one
        // position ahead of their
        // current position

        while (j >= 0 && arr[j] < key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
            counti++;
        }counti++;
        arr[j + 1] = key;
    }
    cout<<"\nnumber of comparisions takes place in insertion sort are"<<counti<<endl;
}

```

```

void bubbleSortd(int arr[], int n)
{
    counti=0;
    int i, j;
    for (i = 0; i < n - 1; i++){

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++){
            counti++;
            if (arr[j] < arr[j + 1]){
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
    cout<<"\nnumber of comparisions takes place in bubble sort are"<<counti<<endl;
}

void selectionSortd(int arr[], int n)
{
    counti=0;
    int i, j, min_idx;

    // One by one move boundary of
    // unsorted subarray
    for (i = 0; i < n-1; i++)
    {

        // Find the minimum element in
        // unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++){
            counti++;
            if (arr[j] < arr[min_idx])
                min_idx = j;

        }

        // Swap the found minimum element
        // with the first element
        if(min_idx!=i)
            swap(&arr[min_idx], &arr[i]);
    }
}

```

```

    }

}
cout<<"\nNumber of comparisions takes place in selection Sort are "<<counti;
}

```

```

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

    // Create temp arrays
    int *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (int i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (int j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    int indexOfSubArrayOne
        = 0, // Initial index of first sub-array
        indexOfSubArrayTwo
        = 0; // Initial index of second sub-array
    int indexOfMergedArray
        = left; // Initial index of merged array

    // Merge the temp arrays back into array[left..right]
    while (indexOfSubArrayOne < subArrayOne
        && indexOfSubArrayTwo < subArrayTwo) {
        counti++;
        if (leftArray[indexOfSubArrayOne]
            >= rightArray[indexOfSubArrayTwo]) {
            array[indexOfMergedArray]
                = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
    }
}

```

```

        else {
            array[indexOfMergedArray]
                = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // left[], if there are any
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // right[], if there are any
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSortd(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Returns recursively

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

```

```

int partitiond(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i
        = (low
            - 1); // Index of smaller element and indicates
                // the right position of pivot found so far

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        counti++;
        if (arr[j] > pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

```

```

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSortd(int arr[], int low, int high)
{
    if (low < high) {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partitiond(arr, low, high);
        // Separately sort elements before
        // partition and after partition
        quickSortd(arr, low, pi - 1);
        quickSortd(arr, pi + 1, high);
    }
}

```

```

int main(){
cout<<"Enter the choice : 0 for descending order and 1 for ascending order";
int choice;
cin>>choice;
if(choice==1){
int arr[5]={41,67,34,0,69};
struct timeval stop, start;
gettimeofday(&start, NULL);
insertionSort(arr,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by insertion sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr1[5]={41,67,34,0,69};
gettimeofday(&start, NULL);
bubbleSort(arr1,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by bubble sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr2[5]={41,67,34,0,69};
gettimeofday(&start, NULL);
selectionSort(arr2,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by selection sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr3[5]={41,67,34,0,69};
counti=0;
gettimeofday(&start, NULL);
mergeSort(arr3,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisons takes place in merge Sort are "<<counti;
cout<<"\nTime taken by merge sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr4[5]={41,67,34,0,69};
counti=0;
gettimeofday(&start, NULL);
quickSort(arr4,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisons takes place in quick Sort are "<<counti;
cout<<"\nTime taken by quick sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";

```



```

}
else{
int arr[5]={41,67,34,0,69};
struct timeval stop, start;
gettimeofday(&start, NULL);
insertionSortd(arr,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by insertion sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr1[5]={41,67,34,0,69};
gettimeofday(&start, NULL);
bubbleSortd(arr1,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by bubble sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr2[5]={41,67,34,0,69};
gettimeofday(&start, NULL);
selectionSortd(arr2,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by selection sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr3[5]={41,67,34,0,69};
counti=0;
gettimeofday(&start, NULL);
mergeSortd(arr3,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisons takes place in merge Sort are "<<counti;
cout<<"\nTime taken by merge sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
int arr4[5]={41,67,34,0,69};
counti=0;
gettimeofday(&start, NULL);
quickSortd(arr4,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisons takes place in quick Sort are "<<counti;
cout<<"\nTime taken by quick sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
}
return 0;

```

```
}
```

```
Enter the choice : 0 for descending order and 1 for ascending order0
number of comparisions takes place in insertion sort are9
Time taken by insertion sort991 us
number of comparisions takes place in bubble sort are10
Time taken by bubble sort996 us
Number of comparisions takes place in selection Sort are 10
Time taken by selection sort0 us
Number of comparisions takes place in merge Sort are 8
Time taken by merge sort0 us
Number of comparisions takes place in quick Sort are 8
Time taken by quick sort0 us
PS C:\Users\HP\Desktop\4thsem\APSlab> █
```

d) Take sorted array from part a). Again run all the sorting algorithm functions with the “order” flag changed. If array is already in ascending order pass flag value as „d” and vice versa. Print the total number of comparisons done in each of the sorting algorithm.

```
#include<bits/stdc++.h>
#include <sys/time.h>
int counti=0;
using namespace std;
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void insertionSort(int arr[], int n)
{
    counti=0;
    int i, key, j;
    for (i = 1; i < n; i++)
```

```

{
    key = arr[i];
    j = i - 1;

    // Move elements of arr[0..i-1],
    // that are greater than key, to one
    // position ahead of their
    // current position

    while (j >= 0 && arr[j] > key)
    {
        arr[j + 1] = arr[j];
        j = j - 1;
        counti++;
    }counti++;
    arr[j + 1] = key;
}
cout<<"\nnumber of comparisions takes place in insertion sort are"<<counti<<endl;
}

```

```

void bubbleSort(int arr[], int n)
{
    counti=0;
    int i, j;
    for (i = 0; i < n - 1; i++){

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++){
            counti++;
            if (arr[j] > arr[j + 1]){
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
    cout<<"\nnumber of comparisions takes place in bubble sort are"<<counti<<endl;
}

void selectionSort(int arr[], int n)
{
    counti=0;

```

```

int i, j, min_idx;

// One by one move boundary of
// unsorted subarray
for (i = 0; i < n-1; i++)
{
    // Find the minimum element in
    // unsorted array
    min_idx = i;
    for (j = i+1; j < n; j++){
        counti++;
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element
        // with the first element
        if(min_idx!=i)
            swap(&arr[min_idx], &arr[i]);
    }
}

cout<<"\nNumber of comparisions takes place in selection Sort are "<<counti;
}

```

```

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

    // Create temp arrays
    int *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (int i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (int j = 0; j < subArrayTwo; j++)

```

```

    rightArray[j] = array[mid + 1 + j];

int indexOfSubArrayOne
    = 0, // Initial index of first sub-array
    indexOfSubArrayTwo
    = 0; // Initial index of second sub-array
int indexOfMergedArray
    = left; // Initial index of merged array

// Merge the temp arrays back into array[left..right]
while (indexOfSubArrayOne < subArrayOne
    && indexOfSubArrayTwo < subArrayTwo) {
    counti++;
    if (leftArray[indexOfSubArrayOne]
        <= rightArray[indexOfSubArrayTwo]) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
    }
    else {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
    }
    indexOfMergedArray++;
}
// Copy the remaining elements of
// left[], if there are any
while (indexOfSubArrayOne < subArrayOne) {
    array[indexOfMergedArray]
        = leftArray[indexOfSubArrayOne];
    indexOfSubArrayOne++;
    indexOfMergedArray++;
}
// Copy the remaining elements of
// right[], if there are any
while (indexOfSubArrayTwo < subArrayTwo) {
    array[indexOfMergedArray]
        = rightArray[indexOfSubArrayTwo];
    indexOfSubArrayTwo++;
}

```

```

        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Returns recursively

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

```

```

int partition(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i
        = (low
            - 1); // Index of smaller element and indicates
                // the right position of pivot found so far

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        counti++;
        if (arr[j] < pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

```

```
}
```

```
/* The main function that implements QuickSort
```

```
arr[] --> Array to be sorted,
```

```
low --> Starting index,
```

```
high --> Ending index */
```

```
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);
```

```
    }
```

```
}
```

```
void insertionSortd(int arr[], int n)
```

```
{
```

```
    counti=0;
```

```
    int i, key, j;
```

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

```

{
    key = arr[i];
    j = i - 1;

    // Move elements of arr[0..i-1],
    // that are greater than key, to one
    // position ahead of their
    // current position

    while (j >= 0 && arr[j] < key)
    {
        arr[j + 1] = arr[j];
        j = j - 1;
        counti++;
    }counti++;
    arr[j + 1] = key;
}
cout<<"\nnumber of comparisions takes place in insertion sort are"<<counti<<endl;
}

```

```

void bubbleSortd(int arr[], int n)
{
    counti=0;
    int i, j;
    for (i = 0; i < n - 1; i++){

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++){
            counti++;
            if (arr[j] < arr[j + 1]){
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
    cout<<"\nnumber of comparisions takes place in bubble sort are"<<counti<<endl;
}

void selectionSortd(int arr[], int n)
{
    counti=0;

```



```

int i, j, min_idx;

// One by one move boundary of
// unsorted subarray
for (i = 0; i < n-1; i++)
{
    // Find the minimum element in
    // unsorted array
    min_idx = i;
    for (j = i+1; j < n; j++){
        counti++;
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element
        // with the first element
        if(min_idx!=i)
            swap(&arr[min_idx], &arr[i]);
    }
}

cout<<"\nNumber of comparisions takes place in selection Sort are "<<counti;
}

```

```

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

    // Create temp arrays
    int *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (int i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (int j = 0; j < subArrayTwo; j++)

```

```

    rightArray[j] = array[mid + 1 + j];

int indexOfSubArrayOne
    = 0, // Initial index of first sub-array
    indexOfSubArrayTwo
    = 0; // Initial index of second sub-array
int indexOfMergedArray
    = left; // Initial index of merged array

// Merge the temp arrays back into array[left..right]
while (indexOfSubArrayOne < subArrayOne
    && indexOfSubArrayTwo < subArrayTwo) {
    counti++;
    if (leftArray[indexOfSubArrayOne]
        >= rightArray[indexOfSubArrayTwo]) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
    }
    else {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
    }
    indexOfMergedArray++;
}
// Copy the remaining elements of
// left[], if there are any
while (indexOfSubArrayOne < subArrayOne) {
    array[indexOfMergedArray]
        = leftArray[indexOfSubArrayOne];
    indexOfSubArrayOne++;
    indexOfMergedArray++;
}
// Copy the remaining elements of
// right[], if there are any
while (indexOfSubArrayTwo < subArrayTwo) {
    array[indexOfMergedArray]
        = rightArray[indexOfSubArrayTwo];
    indexOfSubArrayTwo++;
}

```

```

        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSortd(int array[], int const begin, int const end)
{
    if (begin >= end)
        return; // Returns recursively

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

```

```

int partitiond(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i
        = (low
            - 1); // Index of smaller element and indicates
                // the right position of pivot found so far

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        counti++;
        if (arr[j] > pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

```

```
}
```

```
/* The main function that implements QuickSort
```

```
arr[] --> Array to be sorted,
```

```
low --> Starting index,
```

```
high --> Ending index */
```

```
void quickSortd(int arr[], int low, int high)
```

```
{
```

```
    if (low < high) {
```

```
        /* pi is partitioning index, arr[p] is now  
        at right place */
```

```
        int pi = partitiond(arr, low, high);
```

```
        // Separately sort elements before
```

```
        // partition and after partition
```

```
        quickSortd(arr, low, pi - 1);
```

```
        quickSortd(arr, pi + 1, high);
```

```
    }
```

```
}
```

```
int main(){
```

```
    cout<<"Enter the choice : 0 for descending order and 1 for ascending order";
```

```
    int choice;
```

```
    cin>>choice;
```

```
    if(choice==1){
```

```
        int arr[5]={0,34,41,67,69};
```

```
        struct timeval stop, start;
```

```
        gettimeofday(&start, NULL);
```

```
        insertionSort(arr,5);
```

```
        gettimeofday(&stop, NULL);
```

```
        cout<<"\nTime taken by insertion sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
```

```
        stop.tv_usec - start.tv_usec<<" us";
```

```
        gettimeofday(&start, NULL);
```

```
        bubbleSort(arr,5);
```

```
        gettimeofday(&stop, NULL);
```

```
        cout<<"\nTime taken by bubble sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
```

```
        stop.tv_usec - start.tv_usec<<" us";
```

```
        gettimeofday(&start, NULL);
```

```
        selectionSort(arr,5);
```

```
        gettimeofday(&stop, NULL);
```

```

cout<<"\nTime taken by selection sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
counti=0;
gettimeofday(&start, NULL);
mergeSort(arr,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisions takes place in merge Sort are "<<counti;
cout<<"\nTime taken by merge sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
counti=0;
gettimeofday(&start, NULL);
quickSort(arr,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisions takes place in quick Sort are "<<counti;
cout<<"\nTime taken by quick sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
}
else{
int arr[5]={0,34,41,67,69};
struct timeval stop, start;
gettimeofday(&start, NULL);
insertionSortd(arr,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by insertion sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
gettimeofday(&start, NULL);
bubbleSortd(arr,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by bubble sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
gettimeofday(&start, NULL);
selectionSortd(arr,5);
gettimeofday(&stop, NULL);
cout<<"\nTime taken by selection sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
counti=0;
gettimeofday(&start, NULL);
mergeSortd(arr,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisions takes place in merge Sort are "<<counti;

```

```

cout<<"\nTime taken by merge sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
counti=0;
gettimeofday(&start, NULL);
quickSortd(arr,0,4);
gettimeofday(&stop, NULL);
cout<<"\nNumber of comparisions takes place in quick Sort are "<<counti;
cout<<"\nTime taken by quick sort"<<(stop.tv_sec - start.tv_sec) * 1000000 +
stop.tv_usec - start.tv_usec<<" us";
}
return 0;
}

```

```

Enter the choice : 0 for descending order and 1 for ascending order1
number of comparisions takes place in insertion sort are4
Time taken by insertion sort1001 us
number of comparisions takes place in bubble sort are10
Time taken by bubble sort0 us
Number of comparisions takes place in selection Sort are 10
Time taken by selection sort1012 us
Number of comparisions takes place in merge Sort are 7
Time taken by merge sort0 us
Number of comparisions takes place in quick Sort are 10
Time taken by quick sort0 us

```

e) Implement quick sort with three overloaded functions. 1st taking pivot as first index, 2nd taking pivot as last index and 3rd taking pivot as middle index.

```
/* C++ implementation of QuickSort */
```

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
// A utility function to swap two elements
```

```
void swap(int* a, int* b)
```

```
{
```

```
    int t = *a;
```

```
    *a = *b;
```

```

        *b = t;
    }

/* 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 */
int partitionr(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i
        = (low
        - 1); // Index of smaller element and indicates
            // the right position of pivot found so far

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSortr(int arr[], int low, int high)
{
    if (low < high) {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partitionr(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSortr(arr, low, pi - 1);
        quickSortr(arr, pi + 1, high);
    }
}

```

```

int partitionl(int arr[], int low, int high)
{
    int pivot = arr[low]; // pivot
    int i
        = (low
            - 1); // Index of smaller element and indicates
                // the right position of pivot found so far

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSortl(int arr[], int low, int high)
{
    if (low < high) {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partitionl(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSortl(arr, low, pi - 1);
        quickSortl(arr, pi + 1, high);
    }
}

```

```

int partitionm(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i
        = (low
            - 1); // Index of smaller element and indicates
                // the right position of pivot found so far

```



```

        for (int j = low; j <= high - 1; j++) {
            // If current element is smaller than the pivot
            if (arr[j] < pivot) {
                i++; // increment index of smaller element
                swap(&arr[i], &arr[j]);
            }
        }
        swap(&arr[i + 1], &arr[high]);
        return (i + 1);
    }
}

void quickSortm(int arr[], int low, int high)
{
    if (low < high) {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partitionm(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSortm(arr, low, pi - 1);
        quickSortm(arr, pi + 1, high);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver Code
int main()
{
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSortl(arr, 0, n - 1);
    cout << "Sorted array: by using pivot as first element\n";
}

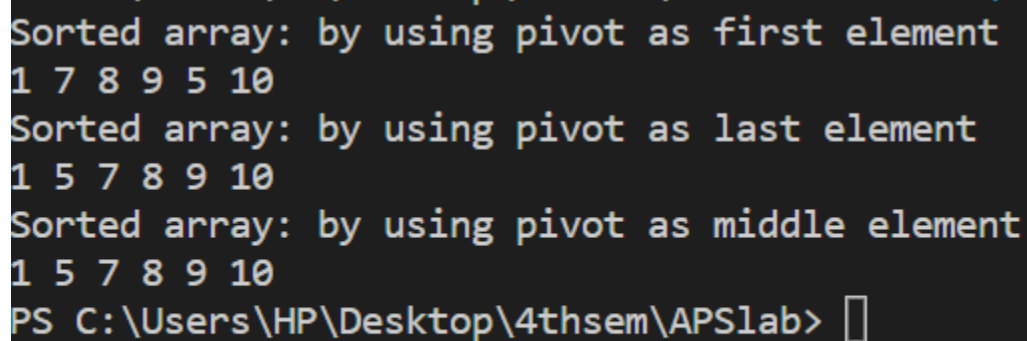
```

```

        printArray(arr, n);
int arr1[] = { 10, 7, 8, 9, 1, 5 };
quickSortr(arr1, 0, n - 1);
    cout << "Sorted array: by using pivot as last element \n";
    printArray(arr1, n);
int arr2[] = { 10, 7, 8, 9, 1, 5 };
quickSortm(arr2, 0, n - 1);
    cout << "Sorted array: by using pivot as middle element \n";
    printArray(arr2, n);
    return 0;
}

// This code is contributed by rathbhupendra

```



```

Sorted array: by using pivot as first element
1 7 8 9 5 10
Sorted array: by using pivot as last element
1 5 7 8 9 10
Sorted array: by using pivot as middle element
1 5 7 8 9 10
PS C:\Users\HP\Desktop\4thsem\APSlab>

```

f) Analyse complexity of quick sort and write down your observation of best and worst case

$T(n) = T(0) + T(n-1) + \Theta(n)$ which is equivalent to $T(n) = T(n-1) + \Theta(n)$
 So, quicksort's worst-case running time is $\Theta(n^2)$.

The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.

$T(n) = 2T(n/2) + \Theta(n)$ The solution for the above recurrence is $(n \log n)$.

Average case of quick sort is also same as it's best case.

g) Analyse complexity of merge sort and write down your observation of best and worst

Case

Time Complexity: $O(N \log(N))$, Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \theta(n)$$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is $\theta(N \log(N))$. The time complexity of Merge Sort is $\theta(N \log(N))$ in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

h) Analyse complexity of bubble sort and write down your observation of best and worst

Case

Worst and Average Case Time Complexity: $O(N^2)$. The worst case occurs when an array is reverse sorted.

Best Case Time Complexity: $O(N)$. The best case occurs when an array is already sorted.

i) Analyse complexity of selection sort and write down your observation of best and worst

Case

Time Complexity: The time complexity of Selection Sort is $O(N^2)$ as there are two nested loops:

- One loop to select an element of Array one by one = $O(N)$
- Another loop to compare that element with every other Array element = $O(N)$

Therefore overall complexity = $O(N) * O(N) = O(N*N) = O(N^2)$

j) Analyse complexity of insertion sort and write down your observation of best and worst

Case

Time Complexity: $O(N^2)$

k) Write a function to sort all even-placed numbers in increasing and odd-place numbers in

decreasing order. The modified array should contain all sorted

even-placed numbers

followed by reverse sorted odd-placed numbers. Analyse the

complexity of your implemented approach

Note that the first element is considered as even because of its

index 0.

Example for part k):

Input: arr[] = {0, 1, 2, 3, 4, 5, 6, 7}

Output: arr[] = {0, 2, 4, 6, 7, 5, 3, 1}

Even-place elements : 0, 2, 4, 6

Odd-place elements : 1, 3, 5, 7

Even-place elements in increasing order :

0, 2, 4, 6

Odd-Place elements in decreasing order :

7, 5, 3, 1

Input: arr[] = {3, 1, 2, 4, 5, 9, 13, 14, 12}

Output: 1, 2, 4, 6, 7, 5, 3, 1

Even-place elements : 3, 2, 5, 13, 12

Odd-place elements : 1, 4, 9, 14

Even-place elements in increasing order :

2, 3, 5, 12, 13

Odd-Place elements in decreasing order :

14, 9, 4, 1

Code-

```
#include <bits/stdc++.h>
using namespace std;
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

void bubbleSort(int arr[], int start,int n)
{
    int i, j;
    for (i = start; i < n - 1; i++){
        for (j = start; j < n - i - 1; j++){
            if (arr[j] > arr[j + 1])swap(arr[j], arr[j + 1]);
        }
    }
}

void bubbleSortd(int arr[],int start, int n)
{
    int i, j;
    for (i = start; i < n - 1; i++){
        for (j = start; j < n - i - 1+start; j++){
            if (arr[j] < arr[j + 1])swap(arr[j], arr[j + 1]);
        }
        // printArray(arr,n);
    }
}

int main()
{
    //int arr[] = {0, 1, 2, 3, 4, 5, 6, 7}; //{3, 1, 2, 4, 5, 9, 13, 14, 12};
    cout<<"Enter the size of the array\n";
    int N;
    cin>>N;
    int arr[N];
    cout<<"\nEnter the elements of the array:";
```

```

for(int i=0;i<N;i++){
    cin>>arr[i];
}
int i=0;
int j=N-1;
while (i<j)
{
    if((i%2!=0)&&(j%2==0)){
        swap(arr[i],arr[j]);
        i++;
        j--;
    }
    else if(i%2==0)i++;
    else j--;
}
bubbleSort(arr,0,i);
bubbleSortd(arr,i,N);
cout << "Sorted array as per requirement is:\n";
printArray(arr,N);
return 0;
}

```

Enter the size of the array

8

Enter the elements of the array:0 1 2 3 4 5 6 7

Sorted array as per requirement is:

0 2 4 6 7 5 3 1

PS C:\Users\HP\Desktop\4thsem\APSlab> □

```

Enter the size of the array
9

Enter the elements of the array:3 1 2 4 5 9 13 14 12
Sorted array as per requirement is:
2 3 5 12 13 14 9 4 1
PS C:\Users\HP\Desktop\4thsem\APSlab> 

```

I) Given an integer array of which both first half and second half are sorted. Task is to merge two sorted halves of array into single sorted array. Analyse the complexity of your implemented approach

Example:

Input : A[] = { 2 ,3 , 8 ,-1 ,7 ,10 }

Output : -1 , 2 , 3 , 7 , 8 , 10

Input : A[] = {-4 , 6, 9 , -1 , 3 }

Output : -4 , -1 , 3 , 6 , 9

```

#include <bits/stdc++.h>
using namespace std;
void mergeTwoHalf(int A[], int n)
{
    int half_i = 0;
    int temp[n];
    for (int i = 0; i < n - 1; i++) {
        if (A[i] > A[i + 1]) {
            half_i = i + 1;
            break;
        }
    }
    if (half_i == 0)
        return;
    int i = 0, j = half_i, k = 0;

```

```

        while (i < half_i && j < n) {
            if (A[i] < A[j])
                temp[k++] = A[i++];
            else
                temp[k++] = A[j++];
        }
        while (i < half_i)
            temp[k++] = A[i++];
        while (j < n)
            temp[k++] = A[j++];
        for (int i = 0; i < n; i++)
            A[i] = temp[i];
    }
int main()
{
    cout<<"Enter the size of the array\n";
    int n;
    cin>>n;
    int arr[n];
    cout<<"\nEnter the elements of the array:";
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    mergeTwoHalf(arr, n);
    cout<<"The array after sorting by using merge function of merge sort:\n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}

```



```
Enter the size of the array
6

Enter the elements of the array:2 3 8 -1 7 10
The array after sorting by using merge function of merge sort:
-1 2 3 7 8 10
PS C:\Users\HP\Desktop\4thsem\APSlab> 
```

```
Enter the size of the array
5

Enter the elements of the array:-4 6 9 -1 3
The array after sorting by using merge function of merge sort:
-4 -1 3 6 9
```

The approach which I use here is to use merge function of merge sort which takes $O(n)$ time to do this task.

For the case of space complexity - the merge function uses a temp array which will take $O(n)$ space in memory to sort to sorted array.