**Practical 2**

**Aim: Implement Sorting Algorithm(s).**

**(a) Bubble Sort (b) Selection Sort (c) Insertion Sort (d) Quick Sort.**

**Bubble Sort Algorithm:**

**bubbleSort(A[], size)**

Step 1. Take i=1 and flag=0

Step 2: if(A[i]>A[i+1] swap (A[i], A[i+1]) and set flag to 1

Step 3: i=i+1 and if(i!=size) go to Step 2

Step 4: if(i=size) size=size-1 and go to Step 1

Step 5: if(flag!=1) Exit

**selectionSort(A[], size)**

Step 1: take i=1 and first =A[i]

Step 2: if(first>A[i+1]) swap (first, A[i+1])

Step 3: i=i+1 and if(i!=size-1) go to Step 2

Step 4: if(i=size) go to Step 1 and set i=i+1

Step 5: if(first=A[size]) exit

**insertionSort(A[], size)**

Step 1: mark first element as sorted

Step 2: for each unsorted element X

'extract' the element X

Step 3: for j <- lastSortedIndex down to 0

Step 4: if current element j > X

Step 5: move sorted element to the right by 1

Step 6: break loop and insert X here

Step 7: end insertionSort

**quickSort(A[], l, u)**

. if (leftmostIndex < rightmostIndex)

pivotIndex <- partition(array,leftmostIndex, rightmostIndex)

quickSort(array, leftmostIndex, pivotIndex)

quickSort(array, pivotIndex + 1, rightmostIndex)

**partition(A[], l, u)**

set rightmostIndex as pivotIndex

storeIndex <- leftmostIndex - 1

for i <- leftmostIndex + 1 to rightmostIndex

if element[i] < pivotElement

swap element[i] and element[storeIndex]

storeIndex++

swap pivotElement and element[storeIndex+1]

return storeIndex + 1

**display(A[], size)**

Step 1: take i=0

Step 2: print A[i] and i=i+1

Step 3: if i<=size go to Step 1

Step 4: Exit

**swapping(&A, &B)**

Step 1: take temp=a

Step 2: a=b

Step 3: b=temp

Step 4: exit

**Tracing of Sorting Functions**

**bubbleSort(A[], size)**

**First Pass:**  
( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**  
( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**  
( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** )

**selectionSort(A[], size)**

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

**11** 25 12 22 64

// Find the minimum element in arr[1...4]

// and place it at beginning of arr[1...4]

11 **12** 25 22 64

// Find the minimum element in arr[2...4]

// and place it at beginning of arr[2...4]

11 12 **22** 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 **25** 64

**insertionSort(A[], size)**

12, 11, 13, 5, 6  
Let us loop for i = 1 (second element of the array) to 4 (last element of the array)  
i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12   
11, 12, 13, 5, 6  
i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13   
11, 12, 13, 5, 6  
i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.   
5, 11, 12, 13, 6  
i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.   
5, 6, 11, 12, 13

**quickSort(A[], l, u)**

quickSort(arr[], low, high)

{

if (low < high)

{

/\* pi is partitioning index, arr[pi] is now

at right place \*/

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}

**partition(A[], l, u)**

arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, **i = -1**

Traverse elements from j = low to high-1

**j = 0** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 0**

arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j

// are same

**j = 1** : Since arr[j] > pivot, do nothing

// No change in i and arr[]

**j = 2** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 1**

arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

**j = 3** : Since arr[j] > pivot, do nothing

// No change in i and arr[]

**j = 4** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 2**

arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

**j = 5** : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

**i = 3**

arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.

**Finally we place pivot at correct position by swapping**

**arr[i+1] and arr[high] (or pivot)**

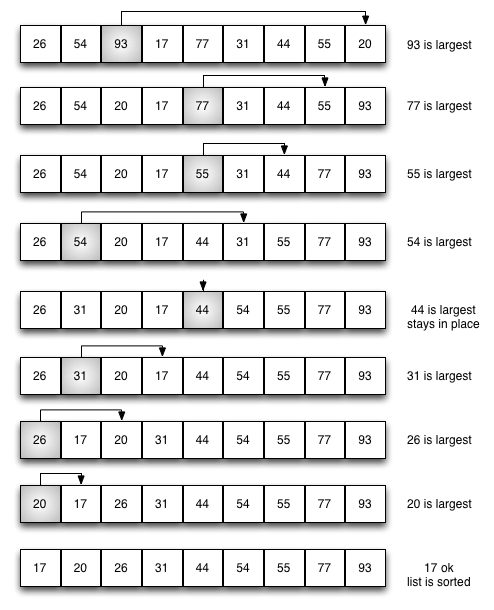
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

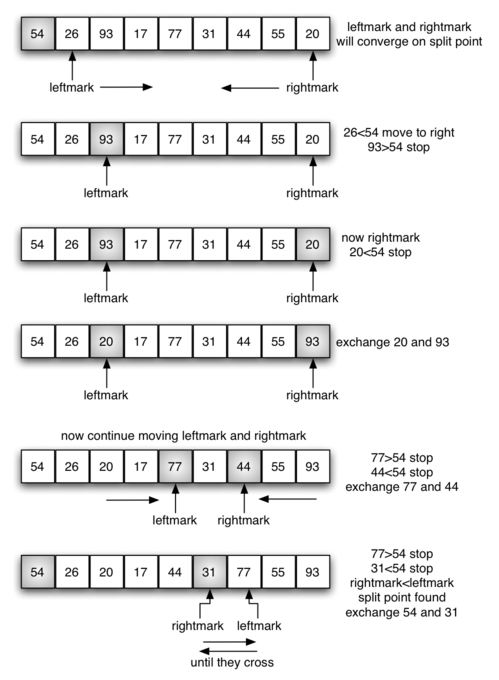
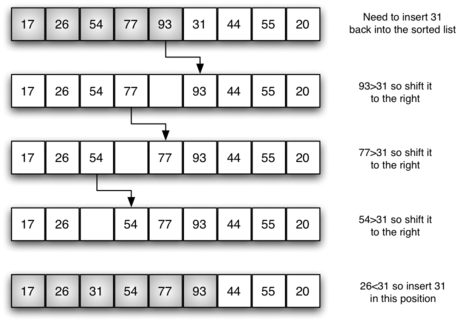
Now 70 is at its correct place. All elements smaller than

70 are before it and all elements greater than 70 are after

it.

**TRACING:**





**Code:**

#include<iostream>

#include<stdlib>

using namespace std;

void swapping(int &a, int &b) {

int temp;

temp = a;

a = b;

b = temp;

}

//-------------------------------------//

void display(int \*array, int size) {

for(int i = 0; i<size; i++)

cout << array[i] << " ";

cout << endl;

}

//----------------------------------------//

void bubbleSort(int \*array, int size) {

for(int i = 0; i<size; i++) {

int swaps = 0;

for(int j = 0; j<size-i-1; j++) {

if(array[j] > array[j+1]) {

swapping(array[j], array[j+1]);

swaps = 1; //set swap flag

}

}

if(!swaps)

break;

}

}

//--------------------------------------------------------//

void insertionSort(int \*array, int size) {

int key, j;

for(int i = 1; i<size; i++) {

key = array[i];//take value

j = i;

while(j > 0 && array[j-1]>key) {

array[j] = array[j-1];

j--;

}

array[j] = key;

}

}

//-------------------------------------------------------//

int partition(int a[],int l,int u)

{

int v,i,j,temp;

v=a[l];

i=l;

j=u+1;

do

{

do

i++;

while(a[i]<v&&i<=u);

do

j--;

while(v<a[j]);

if(i<j)

{

temp=a[i];

a[i]=a[j];

a[j]=temp;

}

}while(i<j);

a[l]=a[j];

a[j]=v;

return(j);

}

//-------------------------------------------------------//

void quickSort(int a[],int l,int u)

{

int j;

if(l<u)

{

j=partition(a,l,u);

quickSort(a,l,j-1);

quickSort(a,j+1,u);

}

}

//----------------------------------------------//

void selectionSort(int \*array, int size) {

int i, j, imin;

for(i = 0; i<size-1; i++) {

imin = i; //get index of minimum data

for(j = i+1; j<size; j++)

if(array[j] < array[imin])

imin = j;

//placing in correct position

swap(array[i], array[imin]);

}

}

//----------------------------------------------//

int main() {

int n,ch;

cout << "Enter the number of elements: ";

cin >> n;

int arr[n];

cout << "Enter elements:" << endl;

for(int i = 0; i<n; i++) {

cin >> arr[i];

}

cout << "Array before Sorting: ";

display(arr, n);

cout<<"\nMenu for sorting techniques: \n1. Bubble Sort \n2. Selection Sort \n3. Insertion Sort \n4. Quick Sort \n5. Exit";

cout<<"\nEnter your choice: (1/2/3/4/5)";

cin>>ch;

switch(ch)

{

case 1: { bubbleSort(arr, n); cout<<"\nArray after Sorting: "; display(arr, n); break; }

case 2: {selectionSort(arr, n); cout<<"\nArray after Sorting: "; display(arr, n); break; }

case 3: {insertionSort(arr, n); cout<<"\nArray after Sorting: "; display(arr, n); break; }

case 4: {quickSort(arr,0,n-1); cout<<"\nArray after sorting: "; display(arr, n); break; }

case 5: {exit(0); break; }

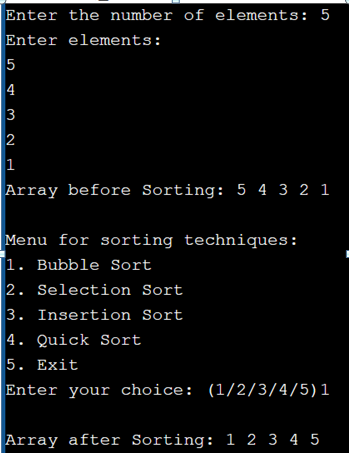
default: {cout<<"Invalid Choice"; break; }

}

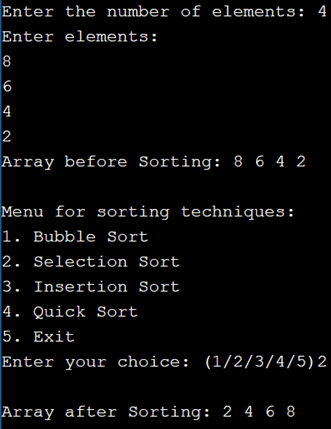
}

**Output:**

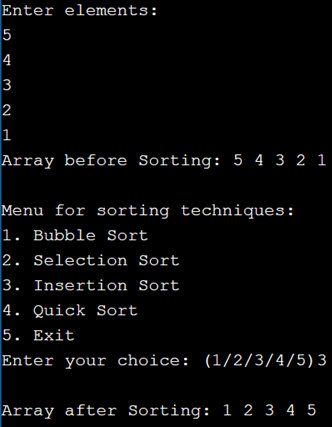
**1. BUBBLE SORT:**



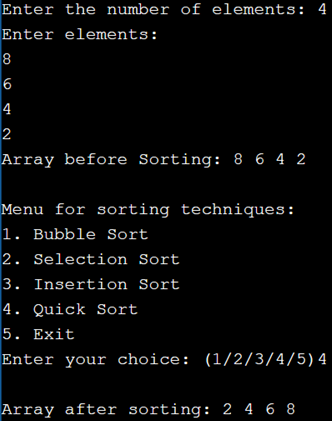
**2. SELECTION SORT:**



**3. INSERTION SORT:**



**4. QUICK SORT:**



**Conclusion:**

By performing this practical we learn about different sorting techniques like bubble sort, selection sort, quick sort and insertion sort and the different working they execute to perform sorting and their algorithm.