**Practical - 1**

**Aim:** implementation of different sorting techniques.

**Description:**

1. **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

**Algorithm:**

1. begin BubbleSort(arr)
2. for all array elements
3. if arr[i] > arr[i+1]
4. swap(arr[i], arr[i+1])
5. end if
6. end for
7. return arr
8. end BubbleSort

Time Complexity: O(N2)

Auxiliary Space: O(1)

1. S**election sort algorithm** sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

* The subarray which is already sorted.
* The remaining subarray was unsorted.

In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

**Algorithm:**

1. Set MIN to location 0
2. Search the minimum element in the list
3. Swap with value at location MIN
4. Increment MIN to point to next element
5. Repeat until list is sorted

Time Complexity: O(N2)

Auxiliary Space: O(1)

1. **Insertion sort** is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

**Algorithm:**

1. If the element is the first element, assume that it is already sorted. Return 1.
2. Pick the next element, and store it separately in a **key.**
3. Now, compare the **key** with all elements in the sorted array.
4. If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
5. Insert the value.
6. Repeat until the array is sorted.

Time Complexity: O(N2)

Auxiliary Space: O(1)

1. **Shell sort** is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of ShellSort is to allow the exchange of far items. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h’th element are sorted.

**Algorithm:**

1. Start
2. Initialize the value of gap size. Example: h
3. Divide the list into smaller sub-part. Each must have equal intervals to h
4. Sort these sub-lists using insertion sort
5. Repeat this step 2 until the list is sorted.
6. Print a sorted list.
7. Stop.

Time Complexity:*O((n\*log n)^2)*

Auxiliary Space: O(1)

1. **Radix Sort** is the linear sorting algorithm that is used for integers. In Radix sort, digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

**Algorithm:**

radixSort(arr)

1. max = largest element in the given array
2. d = number of digits in the largest element (or, max)
3. Now, create d buckets of size 0 - 9
4. for i -> 0 to d
5. sort the array elements using counting sort (or any stable sort) according to the digits at the ith place

Time Complexity:*O(nk)*

Auxiliary Space: O(n+k)

**Codes:**

1. **Bubble Sort:**

**Code:**

#include <iostream>

using namespace std;

void printArray(int arr[], int n){

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

cout << arr[i] << " ";

}

cout << endl;

}

void bubbleSort(int arr[], int n){

cout << "Array Before Sorting: ";

printArray(arr, n);

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

bool flag = true;

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

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

int t = arr[j];

arr[j] = arr[j+1];

arr[j+1] = t;

flag = false;

}

}

if(flag)

break;

}

cout << "Array After Sorting : ";

printArray(arr, n);

}

int main()

{

int n;

cout<<"Enter how many number you are going to enter : ";

cin >> n;

int arr[n];

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

cout << "Element " << i+1 << " : ";

cin >> arr[i];

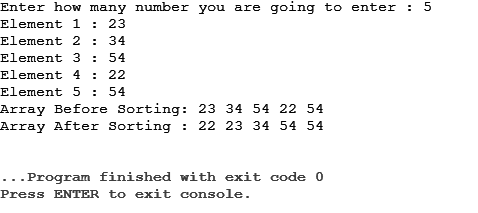
}

bubbleSort(arr, n);

return 0;

}

**Output:**



1. **Selection Sort:**

**Code:**

#include <iostream>

using namespace std;

void printArray(int arr[], int n){

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

cout << arr[i] << " ";

}

cout << endl;

}

void selectionSort(int arr[], int n){

cout << "Array Before Sorting: ";

printArray(arr, n);

int minIndex;

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

minIndex = i;

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

if(arr[minIndex] > arr[j])

minIndex = j;

}

if(minIndex != i){

int t = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = t;

}

}

cout << "Array After Sorting : ";

printArray(arr, n);

}

int main()

{

int n;

cout<<"Enter how many number you are going to enter : ";

cin >> n;

int arr[n];

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

cout << "Element " << i+1 << " : ";

cin >> arr[i];

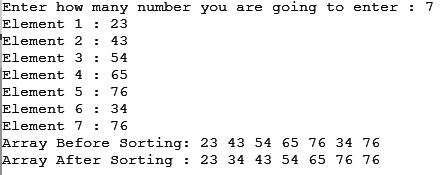
}

selectionSort(arr, n);

return 0;

}

**Output:**



1. **Insertion Sort**

**Code :**

#include <iostream>

using namespace std;

void printArray(int arr[], int n){

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

cout << arr[i] << " ";

}

cout << endl;

}

void insertionSort(int arr[], int n){

cout << "Array Before Sorting: ";

printArray(arr, n);

int j;

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

j = i;

while(j-1 >= 0 && arr[j] < arr[j-1]){

int t = arr[j];

arr[j] = arr[j-1];

arr[j-1] = t;

j--;

}

}

cout << "Array After Sorting : ";

printArray(arr, n);

}

int main()

{

int n;

cout<<"Enter how many number you are going to enter : ";

cin >> n;

int arr[n];

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

cout << "Element " << i+1 << " : ";

cin >> arr[i];

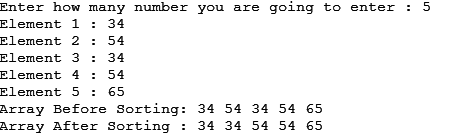
}

insertionSort(arr, n);

return 0;

}

**Output :**



1. **Shell Sort**

**Code:**

#include <iostream>

using namespace std;

void printArray(int arr[], int n){

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

cout << arr[i] << " ";

}

cout << endl;

}

void shellSort(int arr[], int n){

cout << "Array Before Sorting: ";

printArray(arr, n);

for(int gap = n/2; gap > 0; gap /= 2){

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

int temp = arr[i];

int j;

for(j = i; j >= gap && arr[j - gap] > temp; j -= gap){

arr[j] = arr[j-gap];

}

arr[j] = temp;

}

}

cout << "Array After Sorting : ";

printArray(arr, n);

}

int main()

{

int n;

cout<<"Enter how many number you are going to enter : ";

cin >> n;

int arr[n];

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

cout << "Element " << i+1 << " : ";

cin >> arr[i];

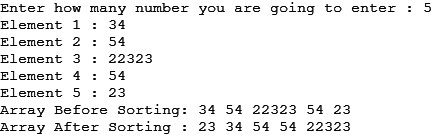
}

shellSort(arr, n);

return 0;

}

**Output:**



1. **Radix Sort**

**Code:**

#include <iostream>

using namespace std;

void printArray(int arr[], int n){

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

cout << arr[i] << " ";

}

cout << endl;

}

int maxElement(int arr[], int n){

int index = 0;

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

if(arr[i] > arr[index])

index = i;

return arr[index];

}

void countingSort(int arr[], int n, int exp){

int count[10] = {0,0,0,0,0,0,0,0,0,0}, i;

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

count[(arr[i] / exp) % 10]++;

for(i = 1; i < 10; i++)

count[i] += count[i-1];

int outputArr[n];

for(i = n-1; i >= 0; i--){

int lastDigit = (arr[i] / exp) % 10;

outputArr[count[lastDigit]-1] = arr[i];

count[lastDigit]--;

}

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

arr[i] = outputArr[i];

}

void radixSort(int arr[], int n){

cout << "Array Before Sorting: ";

printArray(arr, n);

int max = maxElement(arr, n);

for(int exp = 1; max/exp > 0; exp \*= 10){

countingSort(arr, n, exp);

}

cout << "Array After Sorting : ";

printArray(arr, n);

}

int main()

{

int n;

cout<<"Enter how many number you are going to enter : ";

cin >> n;

int arr[n];

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

cout << "Element " << i+1 << " : ";

cin >> arr[i];

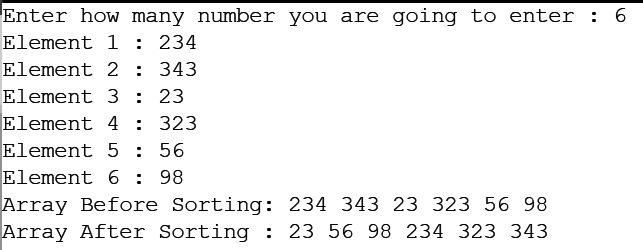
}

radixSort(arr, n);

return 0;

}

**Output:**



**Conclusion:**

All the sorting algorithms are successfully implemented.

**Practical - 2**

**Aim:** implementation of different searching algorithms

**Description:**

1. **Linear Search:**

In a linear search algorithm the values that have to be searched, one by are compared with each and every element in the array by traversing the array. If the element is searched the index of that element is returned. If an element is not found then -1 is returned.

**Algorithm:**

1. function linear(arr, n, element)
2. for i to n
3. if arr[i] is equal to element
4. return i
5. end if
6. end for
7. return -1

Time Complexity:*O(n)*

Auxiliary Space: O(1)

1. **Binary search:**

To perform binary search an array has to be sorted first. Once the array is sorted then we use the divide and conquer method to find the element in an array

**Algorithm:**

1. function find(arr, start, end, element)
2. if start == end
3. return -1
4. mid = (start+end)/2
5. if mid is greater than element
6. return find(arr, start, mid-1, element)
7. else if mid is less than element
8. return find(arr, mid+1, end, element)
9. return mid
10. end function

Time Complexity:*O(log n)*

Auxiliary Space: O(1)

**Codes:**

1. **Linear Search:**

Code:

#include <iostream>

using namespace std;

int findByLinierSearch(int arr[], int n, int value){

cout << "Finding using linear search"<<endl;

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

if(arr[i] == value)

return i;

}

return -1;

}

int main()

{

int n, value;

cout<<"Enter how many number you are going to enter : ";

cin >> n;

int arr[n];

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

cout << "Element " << i+1 << " : ";

cin >> arr[i];

}

cout<<"Enter which number you want to find : ";

cin >> value;

int index = findByLinierSearch(arr, n, value);

if(index != -1){

cout << "Your element is found at index "<<index<<endl;

}else{

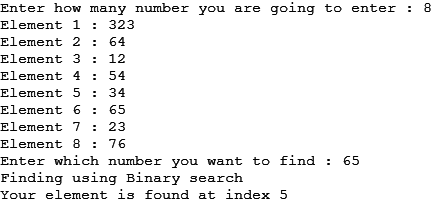
cout << "No element found That you given to search"<<endl;

}

return 0;

}

Output:



1. **Binary search**

Code:

#include <iostream>

using namespace std;

void insertionSort(int arr[], int n){

int j;

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

j = i;

while(j-1 >= 0 && arr[j] < arr[j-1]){

int t = arr[j];

arr[j] = arr[j-1];

arr[j-1] = t;

j--;

}

}

}

int findByBinarySearch(int arr[], int start, int end, int value){

int mid = (start+end)/2;

if(mid > end)

return -1;

else if(arr[mid] < value)

return findByBinarySearch(arr, mid+1, end, value);

else if(arr[mid] > value)

return findByBinarySearch(arr, start, mid-1, value);

return mid;

}

int performBinarySearch(int arr[], int n, int value){

cout << "Finding using Binary search"<<endl;

insertionSort(arr,n);

return findByBinarySearch(arr, 0, n, value);

}

int main()

{

int n, value;

cout<<"Enter how many number you are going to enter : ";

cin >> n;

int arr[n];

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

cout << "Element " << i+1 << " : ";

cin >> arr[i];

}

cout<<"Enter which number you want to find : ";

cin >> value;

int index = performBinarySearch(arr, n, value);

if(index != -1){

cout << "Your element is found"<<endl;

}else{

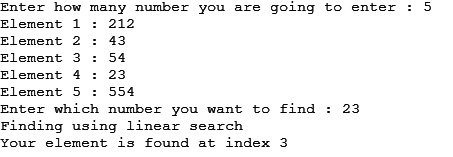
cout << "No element found That you given to search"<<endl;

}

return 0;

}

Output:



**Conclusion:**

All the searching techniques are successfully implemented.