

AIM: Implementation of different sorting techniques.

Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Radix Sort, Quick Sort

A) BUBBLE SORT:

THEORY:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

Example:

First Pass:

(5 1 4 2 8) \rightarrow (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.

(1 5 4 2 8) \rightarrow (1 4 5 2 8), Swap since $5 > 4$

(1 4 5 2 8) \rightarrow (1 4 2 5 8), Swap since $5 > 2$

(1 4 2 5 8) \rightarrow (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) \rightarrow (1 4 2 5 8)

(1 4 2 5 8) \rightarrow (1 2 4 5 8), Swap since $4 > 2$

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (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) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (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)

ALGORITHM:

We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

begin BubbleSort(list)

for all elements of list

 if list[i] > list[i+1]

 swap(list[i], list[i+1])

 end if

end for

return list

end BubbleSort

PSEUDOCODE:

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable swapped which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

```
begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for

  return list

end BubbleSort
```

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

Best Case Time Complexity: $O(n)$. Best case occurs when array is already sorted.

Auxiliary Space: $O(1)$

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Sorting In Place: Yes

Stable: Yes

SOURCE CODE:

```
#include <iostream>
using namespace std;
```

```
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}
```

```
void bubble_sort( int A[ ], int n )
{
    int temp;
    for(int k = 0; k < n-1; k++)
    {
        // (n-k-1) is for ignoring comparisons of elements which have already been compared in
        earlier iterations
```

```
        for(int i = 0; i < n-k-1; i++)
        {
            if(A[ i ] > A[ i+1] )
            {
                // here swapping of positions is being done.
                temp = A[ i ];
                A[ i ] = A[ i+1 ];
                A[ i + 1] = temp;
            }

        }
        cout << "Pass" <<k <<" : ";
        printArray(A, n);
    }
}

int main()
{
    int n;
    cout << "Enter number of elements to be sorted \n";
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter element "<<i+1<<"\n";
        cin >> arr[i];
    }

    bubble_sort(arr, n);
    cout << "Using Bubble Sort, the sorted array is: \n";
    printArray(arr, n);
    return 0;

}
```

OUTPUT:

```
"C:\Users\NARENDER KESWANI\Desktop\bubblesort.exe"
Enter number of elements to be sorted
5
Enter element 1
-9
Enter element 2
9
Enter element 3
7
Enter element 4
0
Enter element 5
1
Pass0 : -9 7 0 1 9
Pass1 : -9 0 1 7 9
Pass2 : -9 0 1 7 9
Pass3 : -9 0 1 7 9
Using Bubble Sort, the sorted array is:
-9 0 1 7 9

Process returned 0 (0x0)   execution time : 14.820 s
Press any key to continue.
```

B) INSERTION SORT:

THEORY:

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array. It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead

Time Complexity: $O(n^2)$

Auxiliary Space: $O(1)$

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

Example:

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

ALGORITHM:

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 – If it is the first element, it is already sorted. return 1;

Step 2 – Pick next element

Step 3 – Compare with all elements in the sorted sub-list

Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 – Insert the value

Step 6 – Repeat until list is sorted

PSEUDOCODE:

```
procedure insertionSort( A : array of items )
  int holePosition
  int valueToInsert

  for i = 1 to length(A) inclusive do:

    /* select value to be inserted */
    valueToInsert = A[i]
    holePosition = i

    /*locate hole position for the element to be inserted */

    while holePosition > 0 and A[holePosition-1] > valueToInsert do:
      A[holePosition] = A[holePosition-1]
      holePosition = holePosition -1
    end while

    /* insert the number at hole position */
    A[holePosition] = valueToInsert

  end for
end procedure
```

SOURCE CODE:

```
#include <bits/stdc++.h>
using namespace std;

void printArray(int arr[], int n)
{
  int i;
  for (i = 0; i < n; i++)
    cout << arr[i] << " ";
}
```

```
        cout << endl;
    }

    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;
            cout << "Pass" << i << " : ";
            printArray(arr, n);
        }
    }

    int main()
    {
        int n;
        cout << "Enter number of elements to be sorted \n";
        cin >> n;
        int arr[n];
        for (int i = 0; i < n; i++)
        {
            cout << "Enter element "<<i+1<<"\n";
            cin >> arr[i];
        }

        insertionSort(arr, n);
        cout << "Using Insertion Sort, the sorted array is: \n";
        printArray(arr, n);
        return 0;
    }
```

OUTPUT:


```
"C:\Users\NARENDER KESWANI\Desktop\insertsort.exe"
Enter number of elements to be sorted
7
Enter element 1
-1
Enter element 2
6
Enter element 3
7
Enter element 4
2
Enter element 5
-5
Enter element 6
5
Enter element 7
3
Pass1 : -1 6 7 2 -5 5 3
Pass2 : -1 6 7 2 -5 5 3
Pass3 : -1 2 6 7 -5 5 3
Pass4 : -5 -1 2 6 7 5 3
Pass5 : -5 -1 2 5 6 7 3
Pass6 : -5 -1 2 3 5 6 7
Using Insertion Sort, the sorted array is:
-5 -1 2 3 5 6 7

Process returned 0 (0x0)   execution time : 18.681 s
Press any key to continue.
```

C) SELECTION SORT:

THEORY:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst-case complexities are of $O(n^2)$, where n is the number of items.

Time Complexity: $O(n^2)$ as there are two nested loops.

Auxiliary Space: $O(1)$

The good thing about selection sort is it never makes more than $O(n)$ swaps and can be useful when memory write is a costly operation.

Example:

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

Algorithm:

Step 1 – Set MIN to location 0

Step 2 – Search the minimum element in the list

Step 3 – Swap with value at location MIN

Step 4 – Increment MIN to point to next element

Step 5 – Repeat until list is sorted

Pseudocode

```
procedure selection sort
  list : array of items
  n    : size of list

  for i = 1 to n - 1
    /* set current element as minimum */
    min = i

    /* check the element to be minimum */

    for j = i+1 to n
      if list[j] < list[min] then
        min = j;
      end if
    end for
```

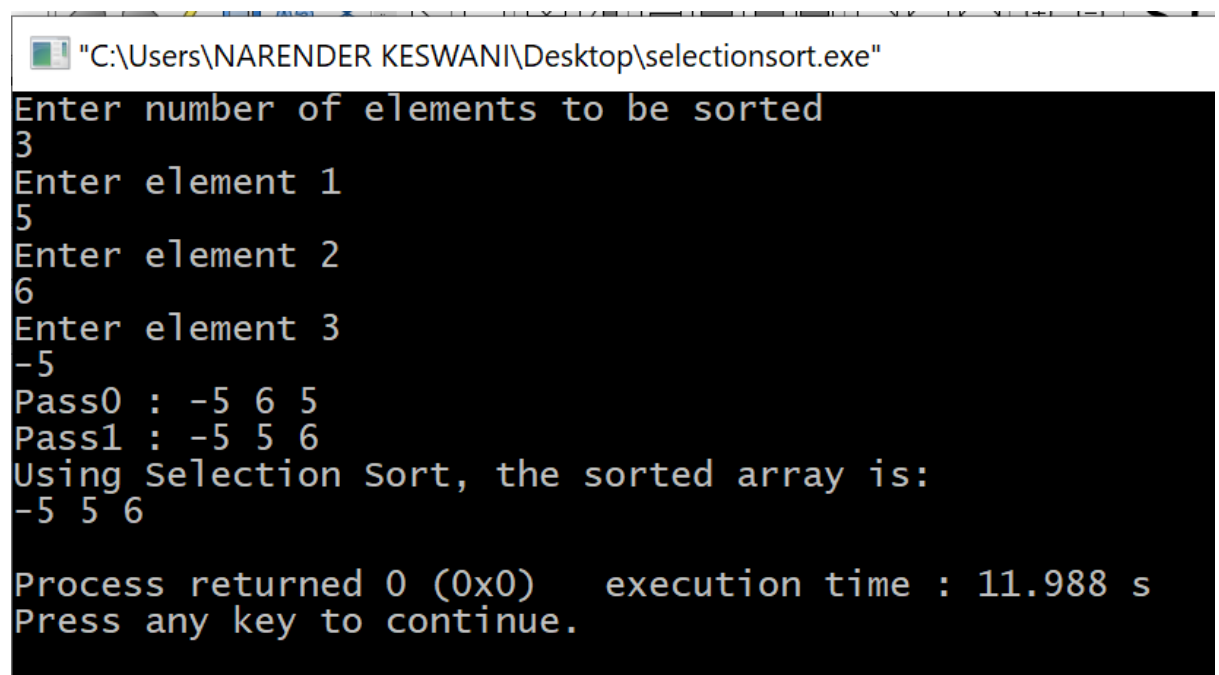
```
/* swap the minimum element with the current element*/  
if indexMin != i then  
    swap list[min] and list[i]  
end if  
end for  
  
end procedure
```

SOURCE CODE:

```
#include <bits/stdc++.h>  
using namespace std;  
  
void printArray(int arr[], int n)  
{  
    int i;  
    for (i=0; i < n; i++)  
        cout << arr[i] << " ";  
    cout << endl;  
}  
  
void swap(int *xp, int *yp)  
{  
    int temp = *xp;  
    *xp = *yp;  
    *yp = temp;  
}  
  
void selectionSort(int arr[], int n)  
{  
    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++)  
        {  
            if (arr[j] < arr[min_idx])  
                min_idx = j;  
        }  
        // Swap the found minimum element with the first element  
        swap(arr[min_idx], arr[i]);  
        cout << "Pass" << i << " : ";  
        printArray(arr, n);  
    }  
}
```

```
    }  
}  
  
int main()  
{  
    int n;  
    cout << "Enter number of elements to be sorted \n";  
    cin >> n;  
    int arr[n];  
    for (int i = 0; i < n; i++)  
    {  
        cout << "Enter element "<i+1<<"\n";  
        cin >> arr[i];  
    }  
  
    selectionSort(arr, n);  
    cout << "Using Selection Sort, the sorted array is: \n";  
    printArray(arr, n);  
    return 0;  
}
```

OUTPUT:



```
"C:\Users\NARENDER KESWANI\Desktop\selectionsort.exe"  
Enter number of elements to be sorted  
3  
Enter element 1  
5  
Enter element 2  
6  
Enter element 3  
-5  
Pass0 : -5 6 5  
Pass1 : -5 5 6  
Using Selection Sort, the sorted array is:  
-5 5 6  
  
Process returned 0 (0x0)   execution time : 11.988 s  
Press any key to continue.
```

D) SHELL SORT:

THEORY:

Shell sort is a generalized version of the insertion sort algorithm. It first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted. The interval between the elements is reduced based on the sequence used.

Worst Case Complexity: less than or equal to $O(n^2)$

Worst case complexity for shell sort is always less than or equal to $O(n^2)$.

Best Case Complexity: $O(n \log n)$

Average Case Complexity: $O(n \log n)$

Shell Sort Applications

Shell sort is used when:

calling a stack is overhead. uClibc library uses this sort.

recursion exceeds a limit. bzip2 compressor uses it.

Insertion sort does not perform well when the close elements are far apart. Shell sort helps in reducing the distance between the close elements. Thus, there will be less number of swappings to be performed.

ALGORITHM:

Following is the algorithm for shell sort.

Step 1 – Initialize the value of h

Step 2 – Divide the list into smaller sub-list of equal interval h

Step 3 – Sort these sub-lists using **insertion sort**

Step 3 – Repeat until complete list is sorted

Pseudocode

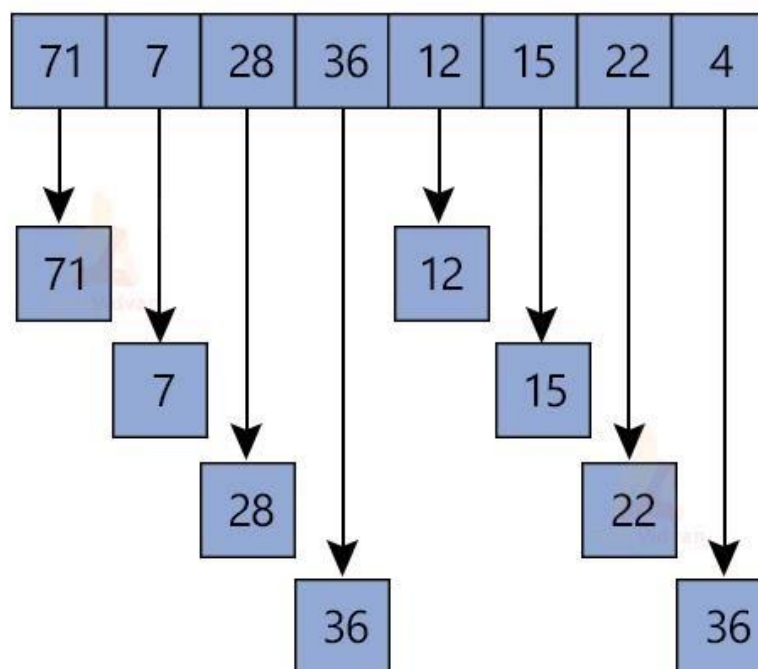
```
procedure shellSort()
  A : array of items

  /* calculate interval*/
  while interval < A.length / 3 do:
    interval = interval * 3 + 1
  end while

  while interval > 0 do:

    for outer = interval; outer < A.length; outer ++ do:
```

```
/* select value to be inserted */  
valueToInsert = A[outer]  
inner = outer;  
  
/*shift element towards right*/  
while inner > interval -1 && A[inner - interval] >= valueToInsert do:  
    A[inner] = A[inner - interval]  
    inner = inner - interval  
end while  
  
/* insert the number at hole position */  
A[inner] = valueToInsert  
  
end for  
  
/* calculate interval*/  
interval = (interval -1) /3;  
  
end while  
  
end procedure
```



SOURCE CODE:

```
#include <iostream>  
using namespace std;  
  
void printArray(int arr[], int n)
```

```
{
for (int i=0; i<n; i++)
    cout << arr[i] << " ";
}

int shellSort(int arr[], int n)
{
for (int gap = n/2; gap > 0; gap /= 2)
{
    for (int i = gap; i < n; i += 1)
    {

        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 << "Pass"<<" :";
    printArray(arr,n);
    cout << " \n";
}
return 0;
}

int main()
{
    int n;
    cout << "Enter number of elements to be sorted \n";
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter element "<<i+1<<"\n";
        cin >> arr[i];
    }

    shellSort(arr, n);
    cout << "Using Shell Sort, the sorted array is: \n";
    printArray(arr, n);
    return 0;
}
```

OUTPUT:

 "C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\shellsort.exe"

```
Enter number of elements to be sorted
5
Enter element 1
5
Enter element 2
6
Enter element 3
1
Enter element 4
3
Enter element 5
-5
Pass :-5 3 1 6 5
Pass :-5 1 3 5 6
Using Shell Sort, the sorted array is:
-5 1 3 5 6
Process returned 0 (0x0)    execution time : 10.120 s
Press any key to continue.
```

E) **QUICK SORT:**

THEORY:

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
procedure quickSort(left, right)

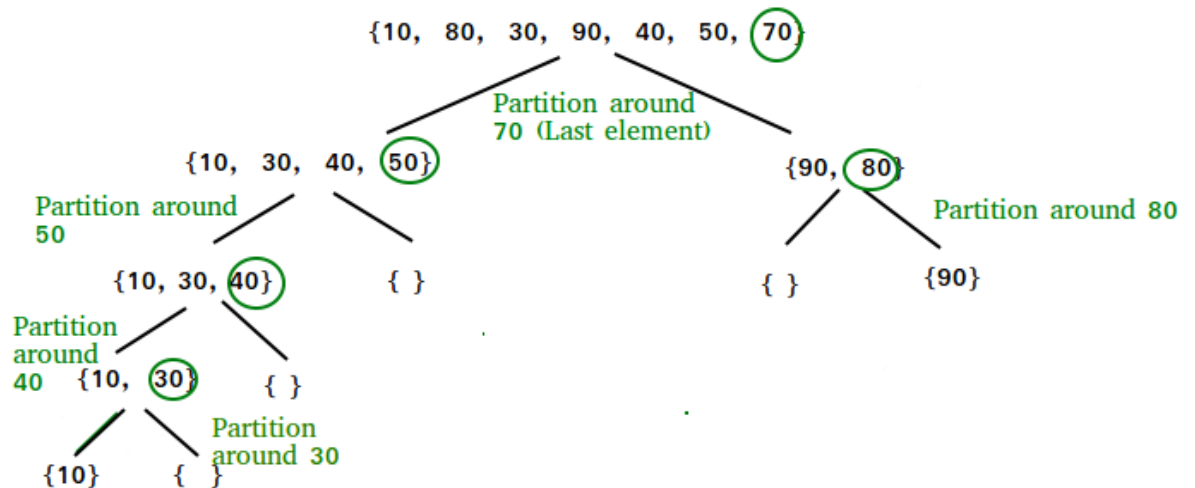
if right-left <= 0
    return
else
```



```

pivot = A[right]
partition = partitionFunc(left, right, pivot)
quickSort(left,partition-1)
quickSort(partition+1,right)
end if
end procedure

```



Worst Case Complexity [Big-O]: $O(n^2)$
Best Case Complexity [Big-omega]: $O(n \log n)$
Average Case Complexity [Big-theta]: $O(n \log n)$
Space Complexity: $O(\log n)$

SOURCE CODE:

```

#include <iostream>
using namespace std;

void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout<<arr[i]<<"\t";
}

// Swap two elements - Utility function
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

```

```
}

// partition the array using last element as pivot
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1);

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

//quicksort algorithm
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        //partition the array
        int pivot = partition(arr, low, high);
        //sort the sub arrays independently
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}

int main()
{
    int n;
    cout << "Enter number of elements to be sorted \n";
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter element "<i+1<<"\n";
        cin >> arr[i];
    }

    quickSort(arr, 0, n-1);
    cout << "Using Quick Sort, the sorted array is: \n";
    printArray(arr, n);
    return 0;
}
```

}

OUTPUT:

```
"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\quicksort.exe"
Enter number of elements to be sorted
6
Enter element 1
-5
Enter element 2
-3
Enter element 3
8
Enter element 4
9
Enter element 5
1
Enter element 6
0
Using Quick Sort, the sorted array is:
-5    -3    0    1    8    9
Process returned 0 (0x0)    execution time : 14.631 s
Press any key to continue.
```

F) RADIX SORT:

THEORY:

Radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix. For elements with more than one significant digit, this bucketing process is repeated for each digit, while preserving the ordering of the prior step, until all digits have been considered. For this reason, radix sort has also been called bucket sort and digital sort

Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place. Radix sort is implemented in places where there are numbers in large ranges.

Time Complexity

Best

$O(n+k)$

Worst	$O(n+k)$
Average	$O(n+k)$
Space Complexity	$O(\max)$

Example:

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives:

[*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives:

[*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by the most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

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

SOURCE CODE:

```
#include <iostream>
using namespace std;

// Print an array
void printArray(int arr[], int n)
{
```

```
int i;
for (i = 0; i < n; i++)
    cout << arr[i] << " ";
cout << endl;
}

// Function to get the largest element from an array
int getMax(int arr[], int n)
{
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

// Using counting sort to sort the elements in the basis of significant places
void countingSort(int arr[], int n, int place)
{
    const int max = 10;
    int output[n];
    int count[max];

    for (int i = 0; i < max; ++i)
        count[i] = 0;

    // Calculate count of elements
    for (int i = 0; i < n; i++)
        count[(arr[i] / place) % 10]++;

    // Calculate cumulative count
    for (int i = 1; i < max; i++)
        count[i] += count[i - 1];

    // Place the elements in sorted order
    for (int i = n - 1; i >= 0; i--)
    {
        output[count[(arr[i] / place) % 10] - 1] = arr[i];
        count[(arr[i] / place) % 10]--;
    }

    for (int i = 0; i < n; i++)
    {
        arr[i] = output[i];
        cout << "Pass: ";
        printArray(arr, n);
    }
}

// Main function to implement radix sort
```

```
void radixsort(int arr[], int n)
{
    // Get maximum element
    int max = getMax(arr, n);
    cout<<"The max element is "<<max<<"\n";

    // Apply counting sort to sort elements based on place value.
    for (int place = 1; max / place > 0; place *= 10)
    {
        countingSort(arr, n, place);
    }
}

int main()
{
    int n;
    cout << "Enter number of elements to be sorted \n";
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter element "<<i+1<<"\n";
        cin >> arr[i];
    }

    radixsort(arr, n);
    cout << "Using Radix Sort, the sorted array is: \n";
    printArray(arr, n);
    return 0;
}
```

OUTPUT:

 "C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\radixsort.exe"

```
Enter number of elements to be sorted
5
Enter element 1
5
Enter element 2
4
Enter element 3
3
Enter element 4
6
Enter element 5
1
The max element is 6
Pass: 1 4 3 6 1
Pass: 1 3 3 6 1
Pass: 1 3 4 6 1
Pass: 1 3 4 5 1
Pass: 1 3 4 5 6
Using Radix Sort, the sorted array is:
1 3 4 5 6

Process returned 0 (0x0)    execution time : 7.516 s
Press any key to continue.
```

CONCLUSION:

I have learned the basic sorting algorithms such as quick, shell, bubble, insertion, radix, selection, etc