5.2 SORTING

Sorting refers to arranging data in a specific order. Sorting algorithm specifies the way to arrange data in a particular order. Most usually ascending or descending. Sorting helps in searching, data analysis, and reporting.

Examples for Sorting Algorithms

- Bubble Sort
- Radix Sort
- Insertion Sort
- Selection Sort
- Shell Sort
- Quick Sort
- Merge Sort etc

Sorting Algorithms

It is defined as an algorithm that puts the elements of a list in a certain order, which can be numerical order, alphabetical order, or any user-defined order.
There are two types of sorting:

• **Internal sorting**- deals with sorting the data stored in the computer's main memory

• **External sorting** - deals with sorting the data stored in secondary memory. External sorting is applied when there is large data that cannot be stored in the main memory.

5.2.1 SELECTION SORT

- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and

putting it at the beginning. The algorithm maintains two sub arrays in a given array.

1) The sub-array which is already sorted.

2) Remaining sub-array which is unsorted.

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

Algorithm:

1. Read the list of n elements to be sorted

2. Initially the sorted sub array is empty and the unsorted sub array is the entire list.

3. Find the smallest element in the unsorted list and Swap it with the element in the first position of sorted part

4. Repeatedly find the smallest from the remaining elements in the unsorted list and move it to the sorted sub-array

5. Continue this until all the elements in the unsorted list are sorted.

6. Print the sorted list

Example:

8 20 2 1 4 19 7 11

The smallest element 1 and swap it with the first element in the unsorted part 8.

1 20 2 **8** 4 19 7 11-pass 1

Now the smallest element 2 and swap it with the first element in the unsorted part 20.

1 **2** **20** 8 4 19 7 11-pass 2

Now the smallest element 4 and swap it with the first element in the unsorted part 20.

1 2 **4** 8 **20** 19 7 11-pass 3

Now the smallest element 7 and swap it with the first element in the unsorted part 8.

1 2 4 **7** 20 19 **8** 11-pass 4

Now the smallest element 8 and swap it with the first element in the unsorted part 20.

1 2 4 7 **8** 19 **20** 11-pass 5

Now the smallest element 11 and swap it with the first element in the unsorted part 19.

1 2 4 7 8 **11** 20 **19-pass 6**

Now the smallest element 19 and swap it with the first element in the unsorted part 20.

1 2 4 7 8 11 **19** **20**-pass 7

Finally the elements are sorted.

Program:

```
void Selectionsort(int a[], int n)
{
int i,j,min;
for(i=0; i<n-1; i++)
{
```

```
        min = i;
        for(j=i+1; j<n; j++)
        {
                if(a[j] < a[min])
                 min = j;
        }
        t = a[i];
        a[i] = a[min];
        a[min] = t;
}
cout<<"\n Sorted  List : ";
for(i=0; i<n; i++)
        cout<< a[i];
}
```

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.

Advantages:

- Very **easy to understand and implement**
- Performs **minimum number of swaps**
- **No extra memory** required (in-place sorting)
- Works well for **small lists**
- Simple logic, good for **beginners**

Disadvantages:

- **Slow for large datasets**
- Time complexity is **$O(n^2)$** in all cases
- Does not take advantage of **already sorted data**
- **Not stable** (order of equal elements may change)

Applications:

- Used to sort **small amounts of data**
- Useful when **memory is limited**
- Used in **educational purposes** to explain sorting
- Suitable when **number of swaps must be minimized**
- Used in **simple embedded systems**