SORTING

Definition given a list of elements {„„} aa an 12… , sorting is a procedure that rearranges the elements of the array such that for any two elements in the sorted list, ai and aj, ai < aj.

It may be noted that an array with a single element is deemed to be sorted. The process is carried out to facilitate searching. A sorted array is easy to search and maintain. Binary search, for instance, can be applied only to a sorted file. Searching a file, via binary search, takes On (log) time, whereas searching via linear search takes On () , which is considerably larger than the former (for larger values of n). For instance, if the value of n is 1024, the ratio of time taken by linear search is approximately 10 times of that taken in binary search.

SELECTION SORT

As discussed earlier, sorting means the arrangement of a set of numbers in an order. In selection sort, the element at the first position is compared with all other elements. The number being compared and the number, to which it is compared to, are swapped, if the number to be compared is smaller. The same procedure is repeated for the element at all the other positions.

A: Selection Sort (Min)

So the Idea of the selection sort (Min) as the following steps:

1. **Find the smallest element in the array**
2. **Exchange it with the element in the first position**
3. **Find the second smallest element and exchange it with the element in the second position**
4. **Continue until the array is sorted**

EXAMPLE:

i=0	8 3 9 7 2 6 4	min = 2
i=1	2 3 9 7 8 6 4	min = 3
i=2	2 3 9 7 8 6 4	min = 4
i=3	2 3 4 7 8 6 9	min = 6
i=4	2 3 4 6 8 7 9	min = 7
i=5	2 3 4 6 7 8 9	min = 8
end	2 3 4 6 7 8 9	

B: Selection Sort (Max)

1. **Find the largest element in the array**
2. **Exchange it with the element in the last position**
3. **Find the second largest element and exchange it with the element in the next last position**
4. **Continue until the array is sorted**

EXAMPLE :

i= 0	**8 3 9 7 2 6 4**	**max = 9**
i= 1	**8 3 4 7 2 6 9**	**max = 8**
i= 2	**6 3 4 7 2 8 9**	**max = 7**
i= 3	**6 3 4 2 7 8 9**	**max = 6**
i= 4	**2 3 4 6 7 8 9**	**max = 4**
i= 5	**2 3 4 6 7 8 9**	**max = 3**
end	**2 3 4 6 7 8 9**	

Algorithm: Selection Sort

Input: An array 'a' containing n elements.
Output: A sorted array.
Constraints: No constraints

```
SELECTION SORT (a, n) returns a
  {
  i=0;
  // (n-1) iterations
  while(i<(n-1))
    {
    j=i+1;
    while(j<n)
      {

       if(a[j]<a[i])
         {
         temp=a[j];
         a[j]=a[i];
         a[i]=temp;
         }
       j++;
       }
     i++;
     return a;
     }
  }
```

Complexity: **Since there is a loop inside another loop, the first loop runs (n − 1) times and for each (n − 1) iteration the inner loop runs (n − i − 1) times, where i is the iteration number of iterations. The complexity of the algorithm, therefore, becomes O(n2).**

Number of comparisons: $2((n-1)+(n-2)+(n-3)+\ldots 1)=\frac{n(n-1)}{2}=\frac{(n^2-n)}{2}=O(n^2)$

Problem: The algorithm has a high complexity, as discussed above.

BUBBLE SORT

It's one of the oldest sorts known. In bubble sort, the element at the first position is taken and compared with the item at the second position. The number being compared and the number to which it is compared to are swapped, if the number to be compared is smaller. The same procedure is repeated for the element at the second and the third positions.

- **The bubble sort got its name because of the way the biggest elements "bubble" to the top .**
- **It based on the property of a sorted list that any two adjacent elements are in sorted order .**
- **In a typical iteration of bubble sort each adjacent pair of elements is compared, starting with the first two elements, then the second and the third elements, and all the way to the final two elements .**
- **Each time two elements are compared, if they are already in sorted order, nothing is done to them and the next pair of elements is compared .**
- **In the case where the two elements are not in sorted order, the two elements are swapped, putting them in order.**

Algorithm: Bubble Sort

Input: an array 'a' containing n elements.
Output: a sorted array.
Constraints: no constraints

```
BUBBLE SORT (a, n) returns a
{  i=0;
// (n-1) iterations
while(i<(n-1))
  {
   j=0;
```

```
        while (j<n-1-i)
          {
           if (a[j+1]<a[j])
             {
          temp=a[j];
           a[j]=a[i];
           a[i]=temp;
           }
        j++;
        }

    i++;

    }

    return a;

}
```

Complexity: Since there is a loop inside another loop, the first loop runs (n − 1) times and for each (n − 1) iteration, the inner loop runs (n − i − 1) times, where i is the iteration number iterations. The complexity of the algorithm, therefore, becomes $O(n^2)$. Number of comparisons: $n^2/2$

Problem: The algorithm has a high complexity

Example

- Consider a set of data: 5 9 2 8 4 6 3 .
- Bubble sort first compares the first two elements, the 5 and the 9 .
- Because they are already in sorted order, nothing happens .
- The next pair of numbers, the 9 and the 2 are compared .
- Because they are not in sorted order, they are swapped and the data becomes: 5 2 9 8 4 6 3 .
- To better understand the "bubbling" nature of the sort, watch how the largest number, 9, "bubbles" to the top in the first iteration of the sort.

First iteration i=1

- **j=1 (5 9) 2 8 4 6 3 --> compare 5 and 9, no swap**
- **j=2 5 (9 2) 8 4 6 3 --> compare 9 and 2, swap**
- **j=3 5 2 (9 8) 4 6 3 --> compare 9 and 8, swap**
- **j=4 5 2 8 (9 4) 6 3 --> compare 9 and 4, swap**
- **j=5 5 2 8 4 (9 6) 3 --> compare 9 and 6, swap**
- **j=6 5 2 8 4 6 (9 3) --> compare 9 and 3, swap**
- **5 2 8 4 6 3 9 --> first iteration complete**

- **Notice that in the example above, the largest element, the 9 got swapped all the way into its correct position at the end of the list.**
- **This happens because in each comparison, the larger element is always pushed towards its place at the end of the list.**

Second iteration i=2

In the second iteration, the second-largest element will be bubbled up to its correct place in the same manner :

- **j=1 (5 2) 8 4 6 3 9 --> compare 5 and 2, swap**
- **j=2 2 (5 8) 4 6 3 9 --> compare 5 and 8, no swap**
- **j=3 2 5 (8 4) 6 3 9 --> compare 8 and 4, swap**
- **j=4 2 5 4 (8 6) 3 9 --> compare 8 and 6, swap**
- **j=5 2 5 4 6 (8 3) 9 --> compare 8 and 3, swap**
- **2 5 4 6 3 8 9 --> second iteration complete**

- **(No need to compare last two because the last element is known to be the largest).**

Third iteration i=3

- **j=1 (2 5) 4 6 3 8 9 -->compare 2 and 5 no swap ,**
- **j=2 2 (5 4) 6 3 8 9 -->compare 5 and 4 swap ,**
- **j=3 2 4 (5 6) 3 8 9 -->compare 5 and 6 no swap ,**
- **j=4 2 4 5 (6 3) 8 9 -->compare 6 and 3 swap ,**
- **2 4 5 3 6 8 9 --> Third iteration complete**

- **(No need to compare the last three elements)**

Fourth iteration i=4

- **j=1 (2 4) 5 3 6 8 9 --> compare 2 and 4, no swap**
- **j=2 2 (4 5) 3 6 8 9 --> compare 4 and 5, no swap**
- **j=3 2 4 (5 3) 6 8 9 --> compare 5 and 3, swap**
- **2 4 3 5 6 8 9 --> Fourth iteration complete**

- (No need to compare the last four elements).

Fifth iteration i=5

- **j=1 (2 4) 3 5 6 8 9 --> compare 2 and 4, no swap**
- **j=2 2 (4 3) 5 6 8 9 --> compare 4 and 3, swap**
- **2 3 4 5 6 8 9 --> Fifth iteration complete**

- (No need to compare the last five elements).

Sixth iteration i=6

- **j=1 (2 3) 4 5 6 8 9 --> compare 2 and 3, no swap**
- **2 3 4 5 6 8 9 --> Sixth iteration complete**

- **(No need to compare the last six elements).**

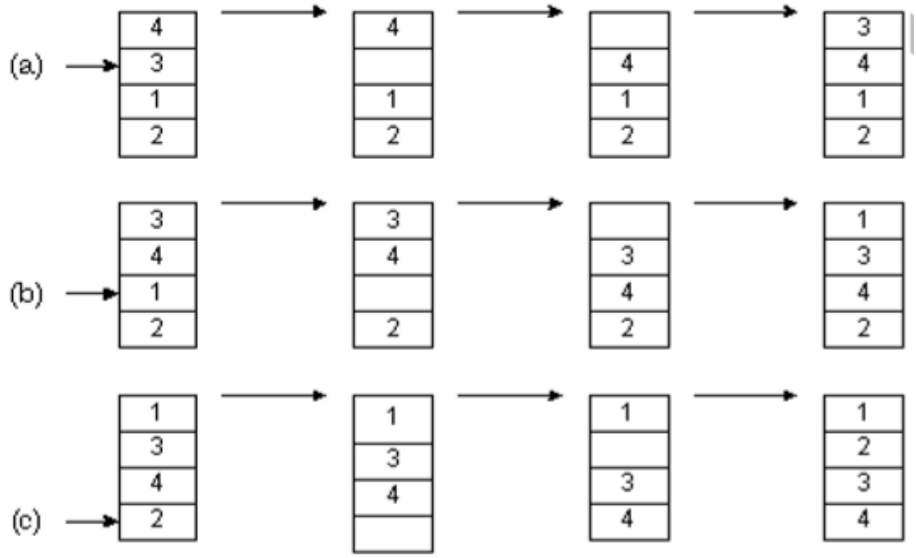
In each iteration, the largest element from amongst the yet unsorted elements is found and placed at its proper position.

INSERTION SORT

Insertion sort is a sorting technique, which inserts an item at its correct position in a partially sorted array. The technique is a well-known one as it is used in sorting playing cards. It's one of the simplest methods to sort an array is an insertion sort, If the first few objects are already sorted, and unsorted object can be inserted in the sorted set in proper place.

- An example of an insertion sort occurs in everyday life while playing cards.
- To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence.
- Both average and worst-case time is $O(n^2)$
- Best case time is O(n)

Example1: Sort the Array X[4,3,1,2]



- Starting near the top of the array in the Figure (a) we extract the 3 .
- Then the above elements are shifted down until we find the correct place to insert the 3 .
- This process repeats in Figure (b) with the next number .
- Finally, in Figure (c), we complete the sort by inserting 2 in the correct place .

Example2: Sort the Array X[8,3,9,7,2,20,4]

Inde.	X	i=1	i=2	i=3	i=4	i=5	i=6	ordered List
0	8	8	3	3	3	2	2	2
1	3	3	8	8	7	3	3	3
2	9	9	9	9	8	7	7	4
3	7	7	7	7	9	8	8	7
4	2	2	2	2	2	9	9	8
5	20	20	20	20	20	20	20	9
6	4	4	4	4	4	4	4	20

- **Number of pass : n-1 = 6**
- **Average no. of comparison is $n^2/4 = 7^2/4 = 12.25$**
- **Average no. of exchanges $n^2/4 = 7^2/4 = 12.25$**
- **Bubble sort (no. of comparison is $n^2/2 = 7^2/2 = 24.5$**
- **Bubble sort slower than Insertion sort**

Input: An array 'a' containing n elements.
Output: A sorted array.
Constraints: No constraints

```
INSERTION SORT (a, n) returns a
  {
  i=1;
  while(i<=(n-1))
    {
    temp=a[i];
    j=i;
    while((temp<a[j-1])&&(j>=0))
      {
      a[j]=a[j-1];
      j--;
      }
    a[j]=temp;
    i++;
    }
    return a;
  }
```

Complexity: Since there is a loop inside another loop, the first loop runs $(n - 1)$ times and for each $(n - 1)$ iteration, the inner loop runs for the requisite number of times. The complexity of the algorithm, therefore, becomes $O(n^2)$. As a matter of fact, the worst-case, best-case, and the average-case complexities of this algorithm is $O(n^2)$.