Sorting

(Bubble Sort, Insertion Sort, Selection Sort)

Sorting

• Sorting refers to the operation of arranging data in some given order such as increasing or decreasing with numerical data or alphabetically with character data.

Comparison Sort

• A comparison sort is a sorting technique that reads the list elements and determines which of the two elements should occur first in the final sorted list through a single comparison operation.

• Some of the most well-known comparison sorts include:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Quicksort
5. Merge Sort
6. Heap Sort

• Some examples of sorts which are not comparison sorts include:

1. Counting Sort
2. Radix Sort
3. Bucket Sort

Bubble Sorting

- The bubble sort is the oldest and simplest sort in use.
- Unfortunately, it is the slowest sorting technique.
- It works by comparing each item in the list with the item next to it, and swapping them if required.
- The algorithm repeats this process until it makes a pass all the way through the list without swapping any items. This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

Algorithm: Bubble_Sort(List, N)

Here List is the collection of items and N is the total no. of items.

```
1.   Repeat steps 2 and 3 for I = 1…... N
2.       Repeat step 3 for J = 1…….N
3.           If  List[J] > List[J+1] then swap(List[J], List[J+1]).
4.   End.
```

Example

List: 10, 20, 5, 100, 25, 6

1st Pass

10, 5, 20, 25, 6, 100

2nd Pass

5, 10, 20, 6, 25, 100

3rd Pass

5, 10, 6, 20, 25, 100

4th Pass

5, 6, 10, 20, 25, 100

5th Pass

5, 6, 10, 20, 25, 100

6th Pass

5, 6, 10, 20, 25, 100

Sorted List: 5, 6, 10, 20, 25, 100

Complexity of Bubble Sort

For each pass, there are n number of comparisons in bubble sorting. For n items, there should be n passes. So,

$C(n) = n + n + \ldots\ldots\ldots\ldots\ldots\ldots.. + n$

$= n * n$

$= 0(n^2)$

Insertion Sort

- Insertion sort is well suited for sorting small data sets or for the insertion of new elements into a sorted sequence.
- Let *a*0, ..., *an*-1 be the sequence to be sorted. At the beginning and after each iteration of the algorithm, the sequence consists of two parts: the first part *a*0, ..., *ai*-1 is already sorted, the second part *ai*, ..., *an*-1 is still unsorted (*i*∊ 0, ..., *n-1*).
- In order to insert element *ai* into the sorted part, it is compared with *ai*-1, *ai*-2 etc.
- When an element *aj* with *aj* ≤ *ai* is found, *ai* is inserted behind it. If no such element is found, then *ai* is inserted at the beginning of the sequence.
- After inserting *ai* the length of the sorted part has increased by one. In the next iteration, *ai*+1 is inserted into the sorted part etc.
- While at the beginning the sorted part consists of element *a*0 only, at the end it consists of all elements *a*0, ..., *an*-1.

Algorithm: Insertion_Sort(List , N)

Here List is the list of items and N is the total number of items.

```
1.   Repeat steps 2 to 7 for I = 2…..N
2.       Set Temp := List [ I ]
3.       Set  J := I - 1
4.   Repeat steps  5 and 6 while j ≥ 1 and List[ J ]  > temp
5.           List [ J+1 ] := List [ J ]
6.            Decrement J
7.   Set List [ J+1] := Temp
8.  End.
```

Example

Given Set of Items A = {77, 33, 44, 11, 88, 22}

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
I=2, J=1	77	33	44	11	88	22
I=3, J=2	33	77	44	11	88	22
I=4, J=3	33	44	77	11	88	22
I=5, J=4	11	33	44	77	88	22
I=6, J=5	11	33	44	77	88	22
Sorted List	11	22	33	44	77	88

Complexity of Insertion Sort

Worst Case

The worst case occurs when the array A is in reverse order and each item can be compared with the maximum number (I-1) of comparisons. So,

$f(n) = 0+1+2+3\ldots+(n-1) = n(n-1)/2 = 0(n^2)$

Average Case

On the average case, there will be approximately (I-1)/2 number of comparisons. So,

$f(n) = 0+1/2+2/2+3/2\ldots+(n-1)/2 = n(n-1)/4 = 0(n^2)$

Selection Sort

- Selection is a simple sorting algorithm.

- It works by first finding the smallest element using a linear scan and swapping it into the first position in the list. Then finding the second smallest element by scanning the remaining elements, and so on.

Algorithm: Selection_Sort (List, N)

```
1.  Repeat steps 2 to 6 for I = 1 to N
2.      Set Min := I
3.      Repeat steps 4 and 5 for J = I+1 to N
4.          If List [J] < List [Min] then
5.              Set Min := J
6.      swap( List[ I ], List[Min] )
7.  End
```

Example

Given Set of Items A = {77, 33, 44, 11, 88, 22}

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
I=1, Min=4	77	33	44	11	88	22
I=2, Min=6	11	33	44	77	88	22
I=3, Min=6	11	22	44	77	88	33
I=4, Min=6	11	22	33	77	88	44
I=5, Min=6	11	22	33	44	88	77
Sorted Items	11	22	33	44	77	88

Complexity of Selection Sort

For finding the 1st smallest elements it requires n-1 comparisons, for second smallest element, n-2 comparisons and so on. So,

$f(n) = (n-1) + (n-2) + \ldots\ldots + 2 + 1 = n(n-1)/2 = O(n^2)$

END