### Bubble Sort – $O(n^2)$ sort

**Idea:** Compare adjacent values and swap so that the largest is the rightmost of the two values. Repeat the process by making $n-1$ passes over the data.

**Data:** Array of $n$ elements

**Algorithm BubbleSort()**

BubbleSort sorts the array Data into ascending order. Upon completion of the sort $Data[i-1] \leq Data[i]$, for $1 \leq i \leq \text{length of Data}$.

```plaintext
for( last = \text{length of Data} – 1 down to 1 )
    for( scan = 1 to last )
        if( Data[scan-1] > Data[scan] )
            temp = Data[scan-1]
            Data[scan-1] = Data[scan]
            Data[scan] = temp
        end if
    end for
end for
```

### Selection Sort – $O(n^2)$ sort

**Idea:** Build a sorted list at the head list by selecting the smallest element in the tail of the list and placing it after the last element in the sorted list. Repeat the process by making $n-1$ passes over the data.

**Data:** Array of $n$ elements

**Algorithm SelectionSort()**

SelectionSort sorts the array Data into ascending order. Upon completion of the sort $Data[i-1] \leq Data[i]$, for $1 \leq i \leq \text{length of Data}$.

```plaintext
for( start = 0 to \text{length of Data} – 2 )
    indexOfSmallest = start
    for( scan = start + 1 to \text{length of Data} – 1 )
        if( Data[scan] < Data[indexOfSmallest] )
            indexOfSmallest = scan
        end if
    end for
    temp = Data[start]
    Data[start] = Data[indexOfSmallest]
    Data[indexOfSmallest] = temp
end for
```

Insertion Sort – $O(n^2)$ sort

Idea: Grow a sorted list at the head of the list by removing the first element following the sorted part and then shifting the hole left in the sorted portion until the proper insertion point for the removed element is reached. Reinsert the removed element in the hole.

Data: Array of $n$ elements
Algorithm InsertionSort()

    InsertionSort sorts the array Data into ascending order. Upon completion of the sort Data[$i-1$] ≤ Data[$i$], for $1 ≤ i ≤$ length of Data.

    for( holeIndex = 1 to length of Data - 1 )
        eleToInsert = Data[holeIndex ]
        scan = holeIndex-1
        while ( scan ≥ 0 and Data[scan] > eleToInsert)
            Data[scan+1] = Data[scan]
            scan = scan – 1
        end while
        Data[scan+1] = eleToInsert
    end for
end InsertionSort

Analysis of Bubble Sort

In order to analyze the time complexity (how long it takes an algorithm to execute) of an algorithm we have to decide which statements represent the work the algorithm is doing. We then need to derive a count of these statements based upon the size of the input to the algorithm.

In the case of Bubble Sort, note that the number of comparisons is proportional to the amount of work the sort does. Therefore, it is sufficient to count the number of comparisons in relation to the length of the array to measure the amount of time it takes to execute the algorithm.

Bubble Sort makes last comparisons for each pass of the inner loop. (That is start – end + 1 comparisons where start = last, end = 1, and we have to add 1 because we are counting fence posts not the space between them.) If the length of Data is $n$ then for the first pass of the outer loop we make $n-1$ comparisons. For the second pass we make $n-2$ passes. We continue this way until the last pass of the outer loop where we make 1 comparison. Hence the number of comparisons is:

$$ (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} \Rightarrow O(n^2) \text{ comparisons} $$