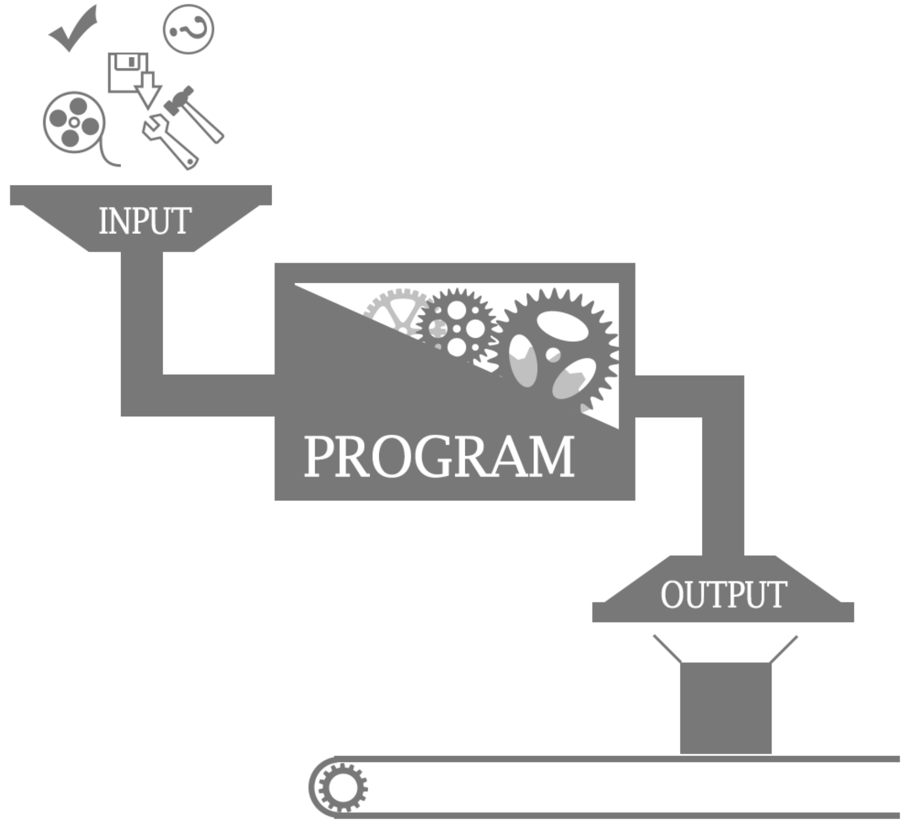
DSA\300 – Data Structures and Algorithms with C#

# Data Structures & Algorithms with C# – Lecture #3

Ahmed Mohyeldin



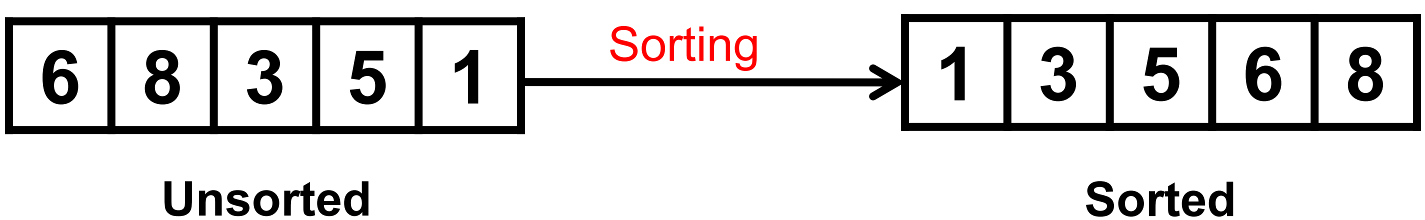

Lecture #3

# SORTING ALGORITHMS

## Making a List of Data Elements in-Order ☺

# Sorting Algorithms – The *Definition* of Sorting?

- *Sorting* is the process of arranging a set of similar information into an increasing or decreasing order. Specifically, given a sorted list *i* of *n* elements then:
  - $i_1 \leq i_2 \leq \ldots. \leq i_n$
- Example:
  - In this context, sorting means arranging an array of data elements such as integers.



# Sorting Algorithms – The *Benefits* of Sorting?

- Sorting algorithms has many practical applications in computing and science:
  - Sorting facilitates the task of searching a large data collection.
  - It helps organize business information in systematic way.
  - Sorting is frequently used by DBMS to perform vital database operations.
  - Sorted information helps analyze large amount of scientific data.
  - *…etc*.

# Sorting Algorithms – The *Types* of Sorting?

- According to the *data storage medium*, sorting methods are classified into two categories:
  - Internal Sorting:
    - The sorting of data kept in the main memory (*i.e.*, RAM), is referred to as internal sorting.
      - In internal sorting, the whole dataset to be sorted is loaded into the main memory at once.
      - The whole dataset is sorted simultaneously and then written back to the secondary device.
  - External Sorting:
    - The sorting of data stored on secondary devices (*e.g.*, disk files) is referred to as external sorting.
      - In external sorting, blocks of data are loaded into the main memory.
      - The data is sorted in the main memory and written back to the secondary device.

# Sorting Algorithms – The *Types* of Sorting? *(…cont'd)*

- Internal *vs.* External Sorting

  - Overall Execution Speed?
    - External sorting is slow compared to internal sorting, because of frequent reading from and writing to external media.

  - Memory Size Limitation?
    - Internal sorting is limited to the physical memory size.
    - External sorting works with any size of physical memory.

  - Which Type to Choose?
    - In practice, the large amount of data is often stored on disk files!
    - External sorting is usually applied in cases when data can't fit into memory entirely.

# Sorting Algorithms –The *Methods* of Sorting?

- According to the *data sorting technique*, sorting methods are classified into three general Classes:
  1. By *Exchange*. 1
     - *e.g.*, Bubble sort, Quick sort.
  2. By *Selection*. 2
     - *e.g.*, Selection sort , Shaker sort, Heap sort
  3. By *Insertion*. 3
     - *e.g.*, Insertion sort.
  - Over the past, several sorting techniques have been devised.
  - Commonly used methods include: Bubble Sort, Insertion Sort, Shell Sort, Quick Sort, Heap Sort, Merge Sort and Radix Sort.

# Sorting Algorithms – *Performance Analysis*?

- Judging the Performance of Sorting Algorithms
  - Many different algorithms exist for each of the three sorting methods.
  - Each algorithm has its merits, but the general criteria for judging a sorting algorithm are based on the following questions:

1. How fast, can it sort information in an average case?
2. How fast is its best and worst case?
3. Does it exhibit natural or unnatural behavior?
4. Does it rearrange elements with equal keys?

# Sorting Algorithms – The *Time* Analysis?

- The *major operations* that influence the running time of a sort procedure are
  - Accessing – Fetching a data item from memory.
  - Comparing – Making comparison between a pair of data items.
  - Swapping – Interchanging a pair of data items.
  - Assigning – Temporarily storing data item in a variable.

$$T(n)_{\text{Total}} = T\,(n)_{\text{Access}} + T(n)_{\text{Comparison}} + T(n)_{\text{Exchange}} + T(n)_{\text{Assignment}}$$

  - Since *comparisons* and *exchanges* are the major operations in sorting, the analysis of sorting algorithm boils down to counting these major operations.

$$T(n)_{\text{Total}} \approx T(n)_{\text{Comparisons}} + T(n)_{\text{Exchanges}}$$

# Sorting Algorithms – The *Time* Analysis? *(...cont'd)*

- Based on their complexity, the sorting algorithms are classified as *elementary* and *advanced*.
  - Elementary Methods → $O(n^2)$
    - The elementary methods use iterative procedures consisting of nested loops.
    - The efficiency is determined by the number of loops to be executed.
    - Examples: *Bubble sort*, *Insertion sort* and *Selection sort*.
  - Advanced Methods: → $O(\ n \log_2 (n)\ )$
    - The advanced sorting methods are based on recursive procedures.
    - In this case, data is partitioned into smaller blocks, which are stored and merged.
    - Therefore, this class of sorting algorithms is often referred to as *Divide-and-Conquer* methods.
    - Examples: *Quick sort*, *Merge sort*, and *Heap sort*.

# Sorting Algorithms – The *Time* Analysis?

- Elementary *vs*. Advanced Sorting Algorithms
  - *Generally*, in the *worst case* scenario, the running time of advanced methods is $O(n \log_2 (n))$. Thus, advanced methods are much faster compared to elementary methods of running time $O(n^2)$.
  - *However*, in advanced algorithms,
    - Sometimes efficiency of an algorithm also depends on the existing order of the data to be sorted:
    - Data might, for example, be presorted, reverse stored, partially sorted, or randomly sorted.
    - Therefore, the analysis of algorithm must consider the *worst case*, *best case* and *average case* scenarios.

# Sorting Algorithms – The *Time* Analysis? *(…cont'd)*

- Sort Passes?
  - In general, *n* data items can be arranged in *n*! ways.
    - For example, *10 data items* can have *over 3.5 million arrangements (10! = 3,628,800)*, of which only one arrangement would be sorted in order.
  - Since *comparisons* and *exchanges* operations are the major operations in sorting, the analysis of sorting algorithm boils down to counting these major operations during the execution of iterative order test procedure.
  - Often during sorting, the array is scanned repeatedly form one end of the array to the other.
  - The process of traversing the array is often referred to as "*Pass*"

# Sorting Algorithms – The *Bubble* Sort

Bubble Sort

# Bubble Sort – The *What*?

- The *Bubble Sort Algorithm* belongs to the *Exchange Methods* of sorting.
- The *Bubble Sort* is the *Simplest* and *Worst* of sorting routines!
- The general concept behind the *Bubble Sort* is the *repeated comparisons* and, if necessary, *exchanges of adjacent elements*.
- Its name comes from the method's similarity to *bubbles in a tank of water*, where each bubble seeks its own level.

# Bubble Sort – The *How*?

- Algorithm Procedure:

1. Compare the first two elements of the array and swap them if they are out-of-order.
2. Continue doing this up the array for each two adjacent pair of elements until you reach the last entry.
3. At this point the last entry is the largest element in the array.
4. Continue this procedure for each next largest element until the array is fully sorted.

# Bubble Sort – *Step-by-Step* Example

- *Step-by-Step Example:*
  - Sort the following list into ascending order using the Bubble Sort algorithm:

| | | | | |
|---|---|---|---|---|
| **6** | **8** | **3** | **5** | **1** |

  - ***Notes:***
    - Symbols used in the solution steps:

| | | | | | |
|---|---|---|---|---|---|
| Comparing X and Y | | **X** | **Y** | | |
| Swapping is performed on X & Y | | **Y** | **X** | | |
| X is fixed in its right position. | | | | | **X** |

# Bubble Sort – *Step-by-Step* Example

*(…cont'd)*

ascendi

| | | | | | |
|---|---|---|---|---|---|
| **Performing the 1st Pass:** | 6 | 8 | 3 | 5 | 1 |
| Comparing 1st & 2nd numbers | 6 | 8 | 3 | 5 | 1 |
| No Swapping, since 8 > 6 | 6 | 8 | 3 | 5 | 1 |
| Comparing 2nd & 3rd numbers | 6 | 8 | 3 | 5 | 1 |
| Swapping, since 3 < 8 | 6 | 3 | 8 | 5 | 1 |
| Comparing 3rd & 4th numbers | 6 | 3 | 8 | 5 | 1 |
| Swapping, since 5 < 8 | 6 | 3 | 5 | 8 | 1 |
| Comparing 4th & 5th numbers | 6 | 3 | 5 | 8 | 1 |
| Swapping, since 1 < 8 | 6 | 3 | 5 | 1 | 8 |

4

**The first pass completed and the last number ( *i.e.,* 8) is fixed.**

# Bubble Sort – *Step-by-Step* Example

*(…cont'd)*

| | | | | | |
|---|---|---|---|---|---|
| **Performing the 2nd Pass:** | 6 | 3 | 5 | 1 | 8 |
| Comparing 1st & 2nd numbers | 6 | 3 | 5 | 1 | 8 |
| Swapping, since 3 < 6 | 3 | 6 | 5 | 1 | 8 |
| Comparing 2nd & 3rd numbers | 3 | 6 | 5 | 1 | 8 |
| Swapping, since 5 < 6 | 3 | 5 | 6 | 1 | 8 |
| Comparing 3rd & 4th numbers | 3 | 5 | 6 | 1 | 8 |
| Swapping, since 1 < 6 | 3 | 5 | 1 | 6 | 8 |
| The 2nd pass is completed | 3 | 5 | 1 | 6 | 8 |

3

**The second pass is completed and number 6 is fixed.**

# Bubble Sort – *Step-by-Step* Example

*(…cont'd)*

| | | | | | |
|---|---|---|---|---|---|
| **Performing the 3rd Pass:** | 3 | 5 | 1 | 6 | 8 |
| Comparing 1st & 2nd numbers | 3 | 5 | 1 | 6 | 8 |
| 2 No, Swapping, since 5 > 3 | 3 | 5 | 1 | 6 | 8 |
| Comparing 2nd & 3rd numbers | 3 | 5 | 1 | 6 | 8 |
| Swapping, since 1 < 5 | 3 | 1 | 5 | 6 | 8 |
| The 3rd pass is completed | 3 | 1 | 5 | 6 | 8 |

**The third pass is completed and 5 is fixed.**

# Bubble Sort – *Step-by-Step* Example

*(…cont'd)*

**Performing the 4th Pass:**

| | | | | |
|---|---|---|---|---|
| 3 | 1 | 5 | 6 | 8 |

1

Comparing 1st & 2nd numbers

| | | | | |
|---|---|---|---|---|
| 3 | 1 | 5 | 6 | 8 |

Swapping, since 1 < 3

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 8 |

The 4th pass is completed

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 8 |

***Note:*** **The algorithm needs one whole pass without any swap to know it is sorted!!!**

**Now, the array is already sorted, but our algorithm does not know !**

# Bubble Sort – *Step-by-Step* Example

*(…cont'd)*

**Performing the 5th Pass:**

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 8 |

Comparing 1st & 2nd numbers

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 8 |

No, Swapping, since 3 > 1

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 8 |

The 5th pass is completed

| | | | | |
|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 8 |

***Note:*** **The algorithm performed the 5th pass without any swap, Therefore, it knows it is sorted.**

**Finally, the array is sorted, and the algorithm can terminate.**

# Bubble Sort – *Pseudocode* (1st Trial)!

```
// Bubble Sort Algorithm – Pseudocode #0:
procedure BubbleSort( A : list of sortable items )
defined as:
    n = length( A )
    for each i in 0 to n - 2 inclusive do:
        for each j in 0 to n - 2 inclusive do:
            if A[j] > A[j+1] then
              swap( A[j], A[j+1] )
            end if
        end for
    end for
end procedure

/* No Optimization: The above pseudocode run blindly
each time a complete number of cycles that it does not
account for the already sorted elements after each
pass and any possibly presorted elements!. Therefore,
a number of optimizations can be made. */

```

# Bubble Sort – *C# Implementation* (1st Trial)! *(…cont'd)*

```
// BubbleSort<T>() - Sorts the elements in the entire List<T>
//    (No. Opt.)       using the "Bubble Sort" algorithm.
static void BubbleSort<T>(T[] list)
    where T : System.IComparable<T>
{
    int n = list.Length;

    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - 1; j++)
        {
            if (list[j].CompareTo(list[j + 1]) > 0)
            {                                  // (list[j] > list[j+1])?
                T temp = list[j];              //
                list[j] = list[j + 1];         // Swap(list[j], list[j+1]);
                list[j + 1] = temp;            //
            }
        }
    }
}/* No Optimization: The above pseudocode run blindly each time a
complete number of cycles that it does not account for the already
sorted elements after each pass and any possibly presorted
elements!. Therefore, a number of optimizations can be made. */
```

or method

# Bubble Sort – *Pseudocode* (Optimization #1)

```
// Bubble Sort Algorithm – Pseudocode #1:
procedure BubbleSort( A : list of sortable items )
defined as:
    n = length( A )
    repeat
        swapped = false
        for each i in 0 to n - 2 inclusive do:
            if A[i] > A[i+1] then
              swap( A[i], A[i+1] )
              swapped = true
            end if
        end for
    until not swapped
end procedure

/* Optimization #1: A small improvement can be made if
each pass you keep track of whether or not an element
was swapped. If not, you can safely assume the list is
sorted. */
```

# Bubble Sort – *C# Implementation* #1

*(…cont'd)*

```
// BubbleSort<T>() - Sorts the elements in the entire List<T>
//    (1st Opt.)      using the "Bubble Sort" algorithm.
static void BubbleSort<T>(T[] list)
    where T : System.IComparable<T>
{
    int n = list.Length;
    bool swapped;
    do
    {
        swapped = false;
        for (int j = 0; j < n - 1; j++)
        {
            if (list[j].CompareTo(list[j + 1]) > 0)
            {                                  // (list[j] > list[j+1])?
                T temp = list[j];              //
                list[j] = list[j + 1];         // Swap (list[j], list[j+1]);
                list[j + 1] = temp;            //
                swapped = true;
            }
        }
    } while (swapped); // Break if it is already sorted.
}
```

# Bubble Sort – C# Implementation #1

(…cont'd)

```
// BubbleSort<T>() - Sorts the elements in the entire List<T>
//   (1st Opt.)       using the "Bubble Sort" algorithm.
static void BubbleSort<T>(T[] list)
    where T : System.IComparable<T>
{
    int n = list.Length;

    for (int i = 0; i < n - 1; i++)
    {
        bool swapped = false;
        for (int j = 0; j < n - 1; j++)
        {
            if (list[j].CompareTo(list[j + 1]) > 0)
            {                                    // (list[j] > list[j+1])?
                T temp = list[j];                //
                list[j] = list[j + 1];           // Swap(list[j], list[j+1]);
                list[j + 1] = temp;              //
                swapped = true;
            }
        }
        if (!swapped) break; // Break if it is already sorted.
    }
}
```

Alternative Implementation Using 2 Nested for Loops

# Bubble Sort – *Pseudocode* (Optimization #2)

```
// Bubble Sort Algorithm – Pseudocode #2:
procedure BubbleSort( A : list of sortable items )
defined as:
    n = length( A )
    repeat
        swapped = false
        for each i in 0 to n - 2 inclusive do:
            if A[i] > A[i+1] then
                swap( A[i], A[i+1] )
                swapped = true
            end if
        end for
        n = n - 1
    until not swapped
end procedure

/* Optimization #2: A second optimization can be made
if you realize that at the end of the i-th pass, the
last i numbers are already in place. */
```

# Bubble Sort – C# Implementation #2

(...cont'd)

```
// BubbleSort<T>() - Sorts the elements in the entire List<T>
//    (2nd Opt.)      using the "Bubble Sort" algorithm.
static void BubbleSort<T>(T[] list)
    where T : System.IComparable<T>
{
    int n = list.Length;
    bool swapped;
    do
    {
        swapped = false;
        for (int j = 0; j < n - 1; j++)
        {
            if (list[j].CompareTo(list[j + 1]) > 0)
            {                                  // (list[j] > list[j+1])?
                T temp = list[j];              //
                list[j] = list[j + 1];         // Swap (list[j], list[j+1]);
                list[j + 1] = temp;            //
                swapped = true;
            }
        }
        n--; // Set n to the remaining n-1 unfixed positions.
    } while (swapped); // Break if it is already sorted.
}
```

# Bubble Sort – *C# Implementation* #2

*(...cont'd)*

```
// BubbleSort<T>() - Sorts the elements in the entire List<T>
//    (2nd Opt.)      using the "Bubble Sort" algorithm.
static void BubbleSort<T>(T[] list)
    where T : System.IComparable<T>
{
    int n = list.Length;

    for (int i = 0; i < n - 1; i++)
    {
        bool swapped = false;
        for (int j = 0; j < n – i - 1; j++)
        {
            if (list[j].CompareTo(list[j + 1]) > 0)
            {                                    // (list[j] > list[j+1])?
                T temp = list[j];                //
                list[j] = list[j + 1];           // Swap(list[j], list[j+1]);
                list[j + 1] = temp;              //
                swapped = true;
            }
        }
        if (!swapped) break; // Break if it is already sorted.
    }
}

```

Alternative Implementation Using 2 Nested for Loops

# Bubble Sort – *C# Implementation* #2.A

*(…cont'd)*

Another Alternative Implementation Using 2 Nested for Loops

```
// BubbleSort<T>() - Sorts the elements in the entire List<T>
//    (2nd Opt.)      using the "Bubble Sort" algorithm.
static void BubbleSort<T>(T[] list)
    where T : System.IComparable<T>
{
    int n = list.Length;

    for (int i = 0, bool swapped = true; i < n – 1 && swapped; ++i)
    {
        swapped = false;
        for (int j = 0; j < n – i - 1; j++)
        {
            if (list[j].CompareTo(list[j + 1]) > 0)
            {                                  // (list[j] > list[j+1])?
                T temp = list[j];              //
                list[j] = list[j + 1];         // Swap(list[j], list[j+1]);
                list[j + 1] = temp;            //
                swapped = true;
            }
        }
        // Break if it is already sorted.
    }
}
```

# Bubble Sort – *Pseudocode* (Optimization #3)

```
// Bubble Sort Algorithm – Pseudocode #3:
procedure BubbleSort( A : list of sortable items )
defined as:
    n = length( A )
    repeat
        last = 0
        for each i in 0 to n - 2 inclusive do:
             if A[i] > A[i+1] then
               swap( A[i], A[i+1] )
               last = i + 1
             end if
        end for
        n = last
    until n <= 1
end procedure
/* Optimization #3: More generally, it can happen that
more than 1 element is placed in their final position
on a single pass. Therefore, you can keep track of the
last swap and decrement the range accordingly. */
```

# Bubble Sort – *C# Implementation* #3 *(…cont'd)*

```
// BubbleSort<T>() - Sorts the elements in the entire List<T>
//    (3rd Opt.)      using the "Bubble Sort" algorithm.
static void BubbleSort<T>(T[] list)
    where T : System.IComparable<T>
{
    int n = list.Length;

    do
    {
        int last = 0;
        for (int j = 0; j < n - 1; j++)
        {
            if (list[j].CompareTo(list[j + 1]) > 0)
            {                                   // (list[j] > list[j+1])?
                T temp = list[j];               //
                list[j] = list[j + 1];          // Swap(list[j], list[j+1]);
                list[j + 1] = temp;             //
                last = j + 1; // Keep record of the last one fixed.
            }
        }
        n = last; // Set n to the last fixed position.
    } while (n > 1); // Break if it is already sorted.
}
```

# Bubble Sort – *C# Implementation* #3

*(…cont'd)*

```
// BubbleSort<T>() - Sorts the elements in the entire List<T>
//    (3rd Opt.)       using the "Bubble Sort" algorithm.
static void BubbleSort<T>(T[] list)
    where T : System.IComparable<T>
{
    int n = list.Length;
    int m = n;
    for (int i = 0; i < m – 1 && n > 1; ++i)
    {
        int last = 0;
        for (int j = 0; j < n - 1; ++j)
        {
            if (list[j].CompareTo(list[j + 1]) > 0)
            {                                   // (list[j] > list[j+1])?
                T temp = list[j];               //
                list[j] = list[j + 1];          // Swap(list[j], list[j+1]
                list[j + 1] = temp;             //
                last = j + 1; // Keep record of the last one fixed.
            }
        }
        n = last; // Set n to the last fixed position.
    }
}
```

Alternative Implementation Using 2 Nested for Loops

# Bubble Sort – *C# Test Driver* (Code)

```
// Bubble Sort Algorithm - BubbleSort<T>() Test Driver.
using System;

Namespace DSA
{
    class Program
    {
        // main() – Program entry point.
        static void Main(string[] args)
        {
            Console.WriteLine("---------------------------------");
            Console.WriteLine("-  Testing DSA.BubbleSort<T>( ) -");
            Console.WriteLine("---------------------------------");

            int n;
            double[] list;
            string input;
            bool check;

            Console.Write("Enter the number of list elements: ");
            input = Console.ReadLine();
            check = int.TryParse(input, out n);
            if (!check) return;                    // continued...
```

# Bubble Sort – *C# Test Driver* (Code)

*(…cont'd)*

```
            list = new double[n]; // Create a list[n].

            for (int i = 0; i < n; ++i) // Enter the list elements:
            {
                do
                {
                    Console.Write("Enter element no.{0}: ", i + 1);
                    input = Console.ReadLine();
                    check = double.TryParse(input, out list[i]);
                } while (!check);
            }

            Console.WriteLine("Before sorting: ");
            DisplayList(list); // Display the unsorted list.
            Console.WriteLine("During sorting: ");
            BubbleSort(list);  // Perform the sorting routine.
            Console.WriteLine("After sorting: ");
            DisplayList(list); // Display the sorted list.

            Console.WriteLine("Press any key to continue...");
            Console.ReadKey(true);
        }
                                                        // continued...
```

# Bubble Sort – *C# Test Driver* (Code) *(…cont'd)*

```
// BubbleSort<T>() - Sorts the elements in the entire List<T>
//                   using the "Bubble Sort" algorithm.
static void BubbleSort<T>(T[] list) where T : System.IComparable<T>
        {
            int n = list.Length;
            int pass = 0; // Added for Tracing Purpose.
            do
            {
                int last = 0;
                for (int j = 0; j < n - 1; ++j)
                {
                    if (list[j].CompareTo(list[j + 1]) > 0)
                    {                     // (list[j] > list[j+1])?
                        T temp = list[j];      //
                        list[j] = list[j + 1]; // Swap
                        list[j + 1] = temp;    //
                        last = j + 1; // Record the last one fixed.
                }
        }
        n = last; // Set n to the last fixed position.
        Trace_SortPass (list, ++pass); // Added for Tracing Purpose.
    } while (n > 1); // Break if it is already sorted.
}                                                  // continued...
```

# Bubble Sort – C# Test Driver (Code)

(…cont'd)

```
        // DisplayList<T>() - Displays the elements in the entire
        //                    List<T> to the system console.
        static void DisplayList<T>(T[] list)
            where T : System.IComparable<T>
        {
            int n = list.Length;
            for (int i = 0; i < n; i++)
                Console.Write(list[i] + " ");
            Console.WriteLine();
        }
        // Trace_SortPass<T>() - Displays the elements in the entire
        //       List<T> to the system console after each sort pass.
        static void Trace_SortPass<T>(T[] list, int pass)
            where T : System.IComparable<T>
        {
           Console.Write("Pass #{0}: ", pass); // Write pass number.
           int n = list.Length;
           for (int i = 0; i < n; i++)  // Write the list elements.
               Console.Write(list[i] + " ");
           Console.WriteLine();
        }
    }
}
```

# Bubble Sort – *C# Test Driver* (Output)

*(...cont'd)*

- **Output:**

```
--------------------------------------------------
   Testing DSA.BubbleSort<T>( ) (Generic Version)
--------------------------------------------------
Enter the number of list elements: 5 ←
Enter element no.1: 6 ←
Enter element no.2: 8 ←
Enter element no.3: 3 ←
Enter element no.4: 5 ←
Enter element no.5: 1 ←
Before sorting:
6 8 3 5 1
During sorting:
Pass #1: 6 3 5 1 8
Pass #2: 3 5 1 6 8
Pass #3: 3 1 5 6 8
Pass #4: 1 3 5 6 8
After sorting:
1 3 5 6 8
Press any key to continue...
```

- **Note:**

← Return Key required following a user input value.

# QUESTION? – *EXEUCTION SPEED*

*Inside the **BubbleSort( )** function, which is faster, **Calling a Swap( ) function** or **placing the swapping logic inline**?*

- ☺ ANSWER ☺
  - Repetitive function calls, *e.g.*, inside loops, incurs considerable overhead over performance. Therefore, such pattern should be avoided as possible.
  - However, placing the required logic statements inline tends to generate frustrating code!
  - A practical workaround to optimize performance and write clean code is to use the *C++-Like Inline Functions Concept*, if your programming language's compiler allows such optimization ☺.


# Tip of the Day!



## Did you know…

- In C# .NET, the JIT compiler logically determines which methods to inline. But sometimes we know better than it does. With *AggressiveInlining*, we tell it that the method should be expanded inline if possible.
- Example: `// NOTE: AggressiveInlining requires.NET 4.5 or later.`

```
using System.Runtime.CompilerServices;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
static int Max(int a, int b)
{
    return (a > b)? a : b;
}
```

- **.NET Framework:** 4.5 or later.
- **Namespace:** System.Runtime.CompilerServices
- **Assembly:** mscorlib (in mscorlib.dll)

# TECHNICAL NOTE: Inlining != Lambdas



- While it's true that you can define inline lambda expressions to replace regular function declarations in C#, the compiler still ends up, mostly, creating an anonymous function!

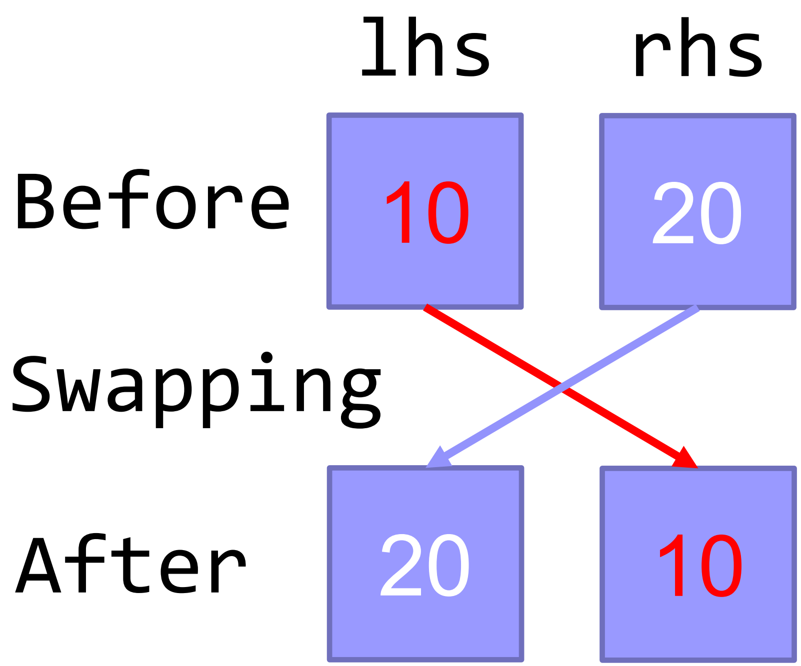
```
// Regular function declaration (C# all versions):
static int Max(int a, int b) { return (a > b)? a : b; }
// or equivalently:
// Lambda expression function declaration (C# 6.0 or later):
static int Max(int a, int b) => (a > b)? a : b;
```

- However, to tell the compiler that the method should be expanded inline if possible. Use the *AggressiveInlining* as follows:

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
static int Max(int a, int b) { return (a > b)? a : b; }
// or equivalently:
[MethodImpl(MethodImplOptions.AggressiveInlining)]
static int Max(int a, int b) => (a > b)? a : b;
```

# *Bubble Sort* Algorithm – *C# Swap<T>( )*

```
// Swap<T>( ) - Performs swapping of its arguments.
[MethodImpl(MethodImplOptions.AggressiveInlining)]
static void Swap<T>(ref T rhs, ref T lhs)
{
     T temp;
     temp = rhs;
     rhs = lhs;
     lhs = temp;
}
```




```
// How to call  Swap<T>( )?
Swap<double>(ref A[j], ref A[j+1]);//T is explicit, or
Swap(ref A[j], ref A[j+1]);        //T is inferred.
```

# Bubble Sort Algorithm – C# Swap<T>( ) (…cont'd)

```
// BubbleSort<T>() - Sorts the elements in the entire List<T>
//    (3rd Opt.)      using the "Bubble Sort" algorithm.
static void BubbleSort<T>(T[] list)
    where T : System.IComparable<T>
{
    int n = list.Length;
    int m = n;
    for (int i = 0; i < m – 1 && n > 1; ++i)
    {
        int last = 0;
        for (int j = 0; j < n - 1; ++j)
        {
            if (list[j].CompareTo(list[j + 1]) > 0)
            {                                    // (list[j] > list[j+1])?

                Swap(ref list[j], ref list[j + 1]); // Exchange.

                last = j + 1; // Keep record of the last one fixed.
            }
        }
        n = last; // Set n to the last fixed position.
    }
}
```

# Bubble Sort – The *Time* Analysis?

- Bubble Sort Utilizes Two Nested Loops.
  - The bubble sort runs at most $n - 1$ passes while executing the outer loop. In each outer loop cycle, the inner loop compares the pairs of adjacent data elements and swaps any pair that are out of order , in a range of $(n - 1) - i$, where $i \in [0$ to $n - 1]$ is the no. of done passes.
  - *What is the expected runtime in worst, average and best case scenarios*?

# Bubble Sort – The *Time* Analysis? *(…cont'd)*

- *Worst Case Analysis*:
    - In worst case, both operations are performed on all elements.
  - Bubble Sort Utilizes Two Nested Loops:
  - *Outer Loop*: executes n – 1 times (*i.e.*, n – 1 passes).
  - *Inner Loop*: executes n – 1 times during the 1st pass, n – 2 times during the 2nd pass, and so on. Each time, it makes 1 comparison & 1 call to ***Swap( )*** (*i.e.*, 3 exchanges).
  - No. of Comparisons = $1[(n-1)+(n-2)+\cdots+1]$= $n(n-1)/2$
  - No. of Exchanges =$3[(n-1)+(n-2)+\cdots+1]$ = $3n(n-1)/2$
  - *Total no. of Major Operations*:
    - No. of Comparisons + Exchanges = $2(n^2-n)$ = $2n^2-2n$
  - The Total Running Time = $O(2n^2-2n)$ or $O(n^2)$
    - Where, *n* is the number of items being sorted.

# Bubble Sort – The *Time* Analysis? *(…cont'd)*

- *Average Case Analysis*:
  - No. of Comparisons = $n(n-1)/2 = \frac{1}{2}(n^2 - n)$
  - No. of Exchanges = $3n(n-1)/4 = \frac{3}{4}(n^2 - n)$
    - *i.e.*, half no. of all possible exchanges is needed.
  - No. of Comparisons + Exchanges = $\frac{5}{4}(n^2 - n)$.
  - The Total Running Time = $O(n^2)$.
- *Best Case Analysis*:
  - No. of Comparisons = $n - 1$
    - *i.e.*, only one checking pass without swaps is needed).
  - No. of Exchanges = 0
    - *i.e.,* an already sorted list and no swaps are needed.
  - No. of Comparisons + Exchanges = $n - 1$.
  - The Total Running Time = $O(n)$.

# Bubble Sort – The *Performance* Summary?

- *Time Analysis*:
  - Worst Case $O(n^2)$ ←$O(n^2)$ comparisons, swaps
  - Average Case $O(n^2)$←$O(n^2)$ comparisons, swaps
  - Best Case $O(n)$ ←$O(n)$ comparisons, 0 swaps
    - Where, $n$ is the number of items being sorted.
  - The *Bubble Sort* is the *Simplest* and *Worst* of sorting routines!

- *Space Analysis*:
  - In-place Case = $O(1)$ auxiliary
    - *i.e.*, only requires a constant amount of additional memory space.
  - Not-In-place Case = *N/A*
    - Where, $n$ is the number of items being sorted.

# Sorting Algorithms – The *Insertion* Sort

Insertion Sort

# Insertion Sort – The *How*?

- Algorithm Procedure:

1. Insertion Sort is somewhat similar to the Bubble Sort in that we compare adjacent elements and swap them if they are out-of order.
2. Unlike the Bubble Sort however, we do not require that we find the next largest or smallest element.
3. Instead, we take the next element and insert it into the sorted list that we maintain at the beginning of the array. It runs as follows:
    1. Start with a sorted list of one element – the $1^{st}$ element.
    2. Insert the $2^{nd}$ element at the end of the sorted list then move it to its right place in the list of two elements.
    3. Insert the $3^{rd}$ element at the end of the sorted list then move it to its right place in the list of three elements.
    4. Repeat the above procedure until there is no elements to be insert into the sorted list.

# Insertion Sort – *Step-by-Step* Example?

- *Step-by-Step Example:*
  - Sort the following list into ascending order using the Insertion Sort algorithm:

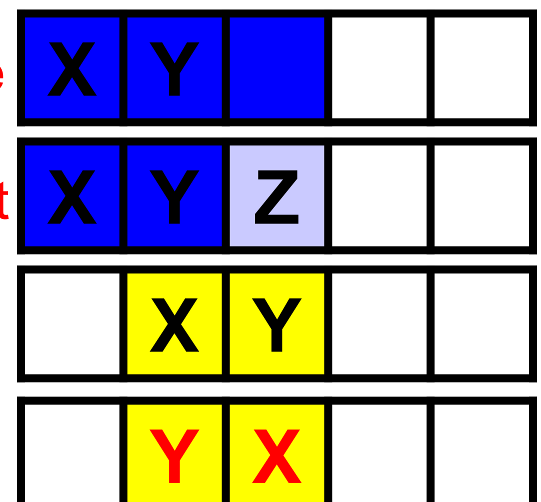
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 2 | 8 | 1 | 4 | 6 | 7 |

  - ***Notes:***
    - Symbols used in the solution steps:

Blue cells indicate elements that partly have been sorted so far in the selected sub-list.
Light Blue cells indicate an inserted element

Comparing X and Y

Swapping is performed on X & Y



# Insertion Sort – *Step-by-Step* Example? *(…cont'd)*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Initial List:** | 5 | 3 | 2 | 8 | 1 | 4 | 6 | 7 |
| Starting with a list of one element | 5 | 3 | 2 | 8 | 1 | 4 | 6 | 7 |
| Inserting the number 3 to the list | 5 | 3 | 2 | 8 | 1 | 4 | 6 | 7 |
| Comparing the numbers 5 & 3 | 5 | 3 | 2 | 8 | 1 | 4 | 6 | 7 |
| Swapping, since 3 < 5 | 3 | 5 | 2 | 8 | 1 | 4 | 6 | 7 |
| Inserting the number 2 to the list | 3 | 5 | 2 | 8 | 1 | 4 | 6 | 7 |
| Comparing the numbers 5 & 2 | 3 | 5 | 2 | 8 | 1 | 4 | 6 | 7 |
| Swapping, since 2 < 5 | 3 | 2 | 5 | 8 | 1 | 4 | 6 | 7 |
| Comparing the numbers 3 & 2 | 3 | 2 | 5 | 8 | 1 | 4 | 6 | 7 |
| Swapping, since 2 < 3 | 2 | 3 | 5 | 8 | 1 | 4 | 6 | 7 |

# Insertion Sort – *Step-by-Step* Example? *(…cont'd)*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Inserting the number 8 to the list | 2 | 3 | 5 | 8 | 1 | 4 | 6 | 7 |
| Comparing the numbers 5 & 8 | 2 | 3 | 5 | 8 | 1 | 4 | 6 | 7 |
| No swapping, since 8 > 5 | 2 | 3 | 5 | 8 | 1 | 4 | 6 | 7 |
| Inserting the number 1 to the list | 2 | 3 | 5 | 8 | 1 | 4 | 6 | 7 |
| Comparing the numbers 8 & 1 | 2 | 3 | 5 | 8 | 1 | 4 | 6 | 7 |
| Swapping, since 1 < 8 | 2 | 3 | 5 | 1 | 8 | 4 | 6 | 7 |
| Comparing the numbers 5 & 1 | 2 | 3 | 5 | 1 | 8 | 4 | 6 | 7 |
| Swapping, since 1 < 5 | 2 | 3 | 1 | 5 | 8 | 4 | 6 | 7 |
| Comparing the numbers 3 & 1 | 2 | 3 | 1 | 5 | 8 | 4 | 6 | 7 |
| Swapping, since 1 < 3 | 2 | 1 | 3 | 5 | 8 | 4 | 6 | 7 |

# Insertion Sort – *Step-by-Step* Example? *(…cont'd)*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Comparing the numbers 2 & 1 | 2 | 1 | 3 | 5 | 8 | 4 | 6 | 7 |
| Swapping, since 1 < 2 | 1 | 2 | 3 | 5 | 8 | 4 | 6 | 7 |
| Inserting the number 4 to the list | 1 | 2 | 3 | 5 | 8 | 4 | 6 | 7 |
| Comparing the numbers 8 & 4 | 1 | 2 | 3 | 5 | 8 | 4 | 6 | 7 |
| Swapping, since 4 < 8 | 1 | 2 | 3 | 5 | 4 | 8 | 6 | 7 |
| Comparing the numbers 5 & 4 | 1 | 2 | 3 | 5 | 4 | 8 | 6 | 7 |
| Swapping, since 4 < 5 | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 |
| Comparing the numbers 3 & 4 | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 |
| No swapping, since 4 > 3 | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 |
| Inserting the number 6 to the list | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 |

# Insertion Sort – *Step-by-Step* Example? *(…cont'd)*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Comparing the numbers 8 & 6 | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 |
| Swapping, since 6 < 8 | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 7 |
| Comparing the numbers 5 & 6 | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 7 |
| No swapping, since 6 > 5 | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 7 |
| Inserting the number 7 to the list | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 7 |
| Comparing the numbers 8 & 7 | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 7 |
| Swapping, since 7 < 8 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Comparing the numbers 6 & 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| No swapping, since 7 > 6 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **Finally, the list is in-order:** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# Insertion Sort – *Pseudocode* (In-Place Sort)

```
// Insertion Sort Algorithm – Pseudocode #1:
procedure InsertionSort( A : list of sortable items )
defined as:
    n = length( A )

    for i = 1 to n-1 do
           temp = A[i];
           j = i - 1;
           while j >= 0 and A[j] > temp do
                  A[j + 1] = A[j];
                  j = j - 1;
           end while
           A[j + 1] = temp;
    end for
end procedure

     /* Pseudocode #1: In-place Insertion Sort. */

```

# Insertion Sort – *C# Implementation* #1 *(...cont'd)*

```
// InsertionSort<T>() - Sorts the elements in the entire List<T>
//    (In-place #1)       using the "Insertion Sort" algorithm.
static void InsertionSort<T>(T[] list)
    where T : System.IComparable<T>
{
    int i, j; // Loop counters.
    T temp;   // Temporary variable to hold the inserted element.
    int n = list.Length; // No. of elements.

    for (i = 1; i < n; i++)
    {
        temp = list[i];
        j = i - 1;
        while (j >= 0 && list[j].CompareTo(temp) > 0)
        {                       // (list[j] > temp) ?
            list[j + 1] = list[j];
            j = j - 1;
        }
        list[j + 1] = temp;
    }
}
    /* Implementation #1: In-place implementation
            using "Nested For-While Loops".       */
```

# Insertion Sort – *C# Implementation* #2 *(…cont'd)*

```
// InsertionSort<T>() - Sorts the elements in the entire List<T>
//    (In-place #2)        using the "Insertion Sort" algorithm.
static void InsertionSort<T>(T[] list)
    where T : System.IComparable<T>
{
    int i, j; // Loop counters.
    T temp;   // Temporary variable to hold the inserted element.
    int n = list.Length; // No. of elements.

    for (i = 1; i < n; i++)
    {
        temp = list[i];

        for (j = i - 1; j >= 0 && list[j].CompareTo(temp) > 0; j--)
        {                                      // (list[j] > temp) ?
            list[j + 1] = list[j];
        }
        list[j + 1] = temp;
    }
}
    /* Implementation #2: In-place implementation
              using "Two Nested For Loops".       */
```

# Insertion Sort – *C# Test Driver* (Code)

```
// Insertion Sort Algorithm - InsertionSort<T>() Test Driver.







                    ?











// TODO: Write a C# "Test Driver" for the InsertionSort<T>( ).
//       a simple C# console program similar to the one showed
//       previously for BubbleSort<T>( ).
                                                  // continued...
```

# Insertion Sort – *C# Test Driver* (Output) *(…cont'd)*

- Output:

```
---------------------------------------------------
  Testing DSA.InsertionSort<T> (Generic Version)
---------------------------------------------------
Enter the number of list elements: 5 ←
Enter element no.1: 5 ←
Enter element no.2: 3 ←
Enter element no.3: 2 ←
Enter element no.4: 8 ←
Enter element no.5: 1 ←
Enter element no.6: 4 ←
Enter element no.7: 6 ←
Enter element no.8: 7 ←
Before sorting:
5 3 2 8 1 4 6 7
During sorting:
Pass #1: 3 5 2 8 1 4 6 7
Pass #2: 2 3 5 8 1 4 6 7
Pass #3: 2 3 5 8 1 4 6 7
Pass #4: 1 2 3 5 8 4 6 7
Pass #5: 1 2 3 4 5 8 6 7
Pass #6: 1 2 3 4 5 6 8 7
Pass #7: 1 2 3 4 5 6 7 8
After sorting:
1 2 3 4 5 6 7 8
Press any key to continue...
```

- **Note:**
  ← **Return Key required following a user input value.**

# Insertion Sort – The *Time* Analysis?

- Insertion Sort Utilizes Two Nested Loops.
  - In each cycle, the insertion sort makes comparisons between a pair of data elements and shifts the element out of order.
- *Worst Case Analysis*:
  - In worst case, both operations are performed on all elements.
  - *Outer Loop*: executes n – 1 Times → n – 1 Passes
    - No. of Comparisons = $(n - 1) + (n - 2) + \ldots + 1 = \frac{1}{2}(n^2 - n)$
    - No. of Exchanges = $(n - 1) + (n - 2) + \ldots + 1 = \frac{1}{2}(n^2 - n)$
    - No. of Comparisons + Exchanges = $n^2 - n$
  - *Inner Loop*: executes *n – 1* Times
    - No. of Data Moves = $2(n - 1)$
  - *Total no. of Major Operations*:
    - Total no. of Major Operations = $n^2 - n + 2(n - 1) = n^2 + n - 2$
  - The Total Running Time = $O(n^2 + n - 2)$ or $O(n^2)$
    - Where, *n* is the number of items being sorted.

# Insertion Sort – The *Time* Analysis? *(…cont'd)*

- *Notes*:
  - More *precise analysis* shows that insertion sort makes *$n^2/4$ comparisons* and *$n^2/8$ exchanges*.
  - The insertion sort performs *better than bubble sort and selection sort*.

# Insertion Sort – The *Performance* Summary?

- *Time Analysis*:
  - Worst Case $O(n^2)$ ← $O(n^2)$ comparisons, swaps
  - Average Case $O(n^2)$ ← $O(n^2)$ comparisons, swaps
  - Best Case $O(n)$ ← $O(n)$ comparisons, 0 swaps
    - Where, $n$ is the number of items being sorted.
  - The *insertion sort* performs *better than bubble sort and selection sort*.

- *Space Analysis*:
  - In-place Case = $O(n)$ total, $O(1)$ auxiliary.
    - *i.e.*, only requires a constant amount of additional memory.
  - Not-In-place Case = $O(n)$
    - Where, $n$ is the number of items being sorted.

# Sorting Algorithms – The *Selection* Sort

Selection Sort

# Selection Sort – The *What*?

- A *Selection Sort* selects the element with the lowest value and exchange that element with the first element. Then, from the remaining *n-1* elements, the element with the least key is found and exchange with the second element, and so forth, up to the last two elements.

# Selection Sort – The *How*?

- Algorithm Procedure:

1. Find the minimum value in the list.

2. Swap it with the value in the first position.

3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time until the remaining sub-array reduces to a single element).

# Selection Sort – *Step-by-Step* Example?

- *Step-by-Step Example:*
  - Sort the following list into ascending order using the Selection Sort algorithm:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 2 | 8 | 1 | 4 | 6 | 7 |

  - ***Notes:***
    - Symbols used in the solution steps:

Blue cell indicates the element at the starting position of the unsorted portion of the list

| | | | | |
|---|---|---|---|---|
| | X | | | |

Light Blue cells indicates the lowest min. value found to the right of the starting position.

| | | | | |
|---|---|---|---|---|
| | | X | | |

Comparing X and Y

| | | | | |
|---|---|---|---|---|
| | X | Y | | |

Swapping is performed on X & Y

| | | | | |
|---|---|---|---|---|
| | Y | X | | |

X is fixed in its right position.

| | | | | |
|---|---|---|---|---|
| | | | | X |

# Selection Sort – *Step-by-Step* Example? *(…cont'd)*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Initial List:** | 5 | 3 | 2 | 8 | 1 | 4 | 6 | 7 |
| Starting position at $1^{st}$ element | 5 | 3 | 2 | 8 | 1 | 4 | 6 | 7 |
| Min. value=1 found at $5^{th}$ element | 5 | 3 | 2 | 8 | 1 | 4 | 6 | 7 |
| Swapping $1^{st}$ and $5^{th}$ elements | 1 | 3 | 2 | 8 | 5 | 4 | 6 | 7 |
| Starting position at $2^{nd}$ element | 1 | 3 | 2 | 8 | 5 | 4 | 6 | 7 |
| Min. value=2 found at $3^{rd}$ element | 1 | 3 | 2 | 8 | 5 | 4 | 6 | 7 |
| Swapping $2^{nd}$ and $3^{rd}$ elements | 1 | 2 | 3 | 8 | 5 | 4 | 6 | 7 |
| Starting position at $3^{rd}$ element | 1 | 2 | 3 | 8 | 5 | 4 | 6 | 7 |
| No Min. value is found | 1 | 2 | 3 | 8 | 5 | 4 | 6 | 7 |
| No Swapping is needed | 1 | 2 | 3 | 8 | 5 | 4 | 6 | 7 |

# Selection Sort – *Step-by-Step* Example? *(…cont'd)*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Starting position at 4th element | 1 | 2 | 3 | 8 | 5 | 4 | 6 | 7 |
| Min. value=4 found at 6th element | 1 | 2 | 3 | 8 | 5 | 4 | 6 | 7 |
| Swapping 4th and 6th elements | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 |
| Starting position at 5th element | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 |
| No Min. value is found | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 |
| No Swapping is needed | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 |
| Starting position at 6th element | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 |
| Min. value=6 found at 7th element | 1 | 2 | 3 | 4 | 5 | 8 | 6 | 7 |
| Swapping 6th and 7th elements | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 7 |
| Starting position at 7th element | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 7 |

# Selection Sort – *Step-by-Step* Example? *(…cont'd)*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Min. value=7 found at 8th element | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 7 |
| Swapping 7th and 8th elements | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Swapping 4th and 6th elements | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Stop since the starting position at the end of list | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **Finally, the list is in-order:** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# Selection Sort – *Pseudocode*

```
// Selection Sort Algorithm – Pseudocode #1:
procedure SelectionSort( A : list of sortable items )
defined as:
    n = length( A )

    for i = 0 to n-2 do
        min = i;
        for j = i+1 to n-1 do
            if(A[j] < A[min]) then
                min = j;
        end for

        if (i != min) then
            swap(A[i], A[min]);
    end for
end procedure
```

# Selection Sort – *C# Implementation* #1

*(…cont'd)*

```
// SelectionSort<T>() - Sorts the elements in the entire List<T>
//    (In-place #1)       using the "Selection Sort" algorithm.
static void SelectionSort<T>(T[] list)
     where T : System.IComparable<T>
{
     int n = list.Length; // No. of elements.

     for (int i = 0; i < n - 1; i++)
     {
          int min = i; // Minimum value at each iteration.
          for (int j = i + 1; j < n; j++)
          {                 // (list[j] < list[min]) ?
               if (list[j].CompareTo(list[min]) < 0)
                    min = j;
          }
          if (i != min)
               Swap(ref list[i], ref list[min]); // Swap elements.
     }
}
     /* Implementation #1: In-place implementation
                  using " Two Nested For Loops".      */
```

# Selection Sort – *C# Test Driver* (Code)

```
// Selection Sort Algorithm - SelectionSort<T>() Test Driver.
```

?

```
// TODO: Write a C# "Test Driver" for the SelectionSort<T>( ).
//       a simple C# console program similar to the one showed
//       previously for BubbleSort<T>( ).
                                                    // continued...
```

# Selection Sort – *C# Test Driver* (Output) *(...cont'd)*

- **Output:**

```
----------------------------------------------------
  Testing DSA.SelectionSort<T> (Generic Version)
----------------------------------------------------
Enter the number of list elements: 5 ←
Enter element no.1: 5 ←
Enter element no.2: 3 ←
Enter element no.3: 2 ←
Enter element no.4: 8 ←
Enter element no.5: 1 ←
Enter element no.6: 4 ←
Enter element no.7: 6 ←
Enter element no.8: 7 ←
Before sorting:
5 3 2 8 1 4 6 7
During sorting:
Pass #0: 1 3 2 8 5 4 6 7
Pass #1: 1 2 3 8 5 4 6 7
Pass #2: 1 2 3 8 5 4 6 7
Pass #3: 1 2 3 4 5 8 6 7
Pass #4: 1 2 3 4 5 8 6 7
Pass #5: 1 2 3 4 5 6 8 7
Pass #6: 1 2 3 4 5 6 7 8
After sorting:
1 2 3 4 5 6 7 8
Press any key to continue...
```

- **Note:**
  ← **Return Key required following a user input value.**

# Selection Sort – The *Time* Analysis?

- Selection Sort Utilizes Two Nested Loops.
  - The insertion sort runs $n - 1$ passes while executing the outer loop. In each outer loop cycle, the inner loop searches for the minimum element value, in a sub-array of range ∈ [$i + 1$ to $n - 1$], where $i$ is the no. of done passes. Then, the outer loop swaps it with the 1st element in the sub-array before taking the next pass.
  - *What is the expected runtime in worst, average and best case scenarios*?

# Selection Sort – The *Time* Analysis? *(…cont'd)*

- *Worst Case Analysis*:
  - In worst case, both operations are performed on all elements.
  - Outer Loop: executes n – 1 times (*i.e.*, n – 1 passes). Each time, it makes 1 function call to ***Swap( )*** (*i.e.*, 3 exchanges).
    - No. of Exchanges = $3(n-1)$
  - *Inner Loop*: executes n – 1 times during the 1st pass, n – 2 times during the 2nd pass, and so on. In each time, it makes 1 comparison.
    - No. of Comparisons $=(n-1)+(n-2)+\cdots+1=\frac{1}{2}(n^2-n)$
  - *Total no. of Major Operations*: $\frac{n^2}{2}+\frac{5n}{2}-3$
    - No. of Comparisons + Exchanges $=\frac{1}{2}n^2+\frac{5}{2}n-3$
  - The Total Running Time = $O(n^2/2 + 5n/2 - 3)$ or $O(n^2)$
    - Where, *n* is the number of items being sorted.

# Selection Sort – The *Performance* Summary?

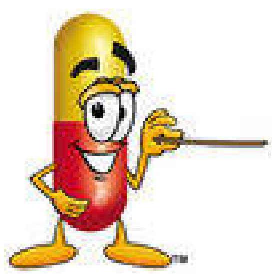
- *Time Analysis*:
  - Worst Case $O(n^2)$ ← $O(n^2)$ comparisons, $O(n)$ swaps
  - Average Case $O(n^2)$ ← $O(n^2)$ comparisons, $O(n)$ swaps
  - Best Case $O(n^2)$ ← $O(n^2)$ comparisons, 0 swaps
    - Where, *n* is the number of items being sorted.
  - *Selection sort* is *inefficient for large data set*. However, selection sort is *preferable choice, when data movement is time consuming,* for example sorting of files with short keys.
- *Space Analysis*:
  - In-place Case = $O(n)$ total , $O(1)$ *auxiliary.*
    - *i.e.*, only requires a constant amount of additional memory.
  - Not-In-place Case = *N/A*
    - Where, *n* is the number of items being sorted.

# Sorting Algorithms – The *List<T>* Class

OOP Implementation

# Sorting Algorithms – The *List<T>* Class

List <T>
Class
Diagram



T : IComparable<T>

**List**

- \- list[ ]: T
- \- count: integer
- \- capacity: integer

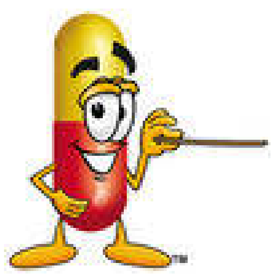
- \+ List( ) ← constructors
- \+ List(capacity: int)
- \+ List(collection: IEnumerable<T>)
- \+ ~List( )

- \+ Add(item: T) void
- \+ AddRange(collection: IEnumerable<T>) void
- \+ Clear( ) void
- \+ Display( ) void
- \+ ToString( ) string
- \+ Sort(method: SortMethod) void
- \- BubbleSort( ) void
- \- InsertionSort( ) void
- \- SelectionSort( ) void

- \+ Count: int : {READONLY }
- \+ Capacity: int {READONLY}
- \+ [ ]: T {READWRITE}

# Sorting Algorithms – The *List<T>* Class *(…cont'd)*

## Enumerated Types Diagram



**<<enumeration>>**
**SortMethod**

- **+ DefaultSort**
- **+ BubbleSort**
- **+ HeapSort**
- **+ InsertionSort**
- **+ MergeSort**
- **+ SelectionSort**
- **+ QuickSort**

# Sorting Algorithms – The *List<T>* Class *(…cont'd)*

```
// Sorting Algorithms - List<T>() Generic Class.

                              ?

// TODO: Write a C#  generic class that implements the List<T>
//       UML class diagram with a complete "Test Driver" to
//       demonstrate using the Sort( ) method.
```

# Sorting Algorithms – The *List<T>* Class *(…cont'd)*

```
// HINT: Array-Based List Data Structure - List<T> Class Skelton.
using System;
namespace Mohyeldin.DSA
{
   public class List<T> where T: System.IComparable<T>
   {
      public List () { // TODO: ... }
      public List (int capacity) { // TODO: ... }
      public List (IEnumerable<T> collection) { // TODO: ... }
      public ~List () { // TODO: ... }
      public int Count { get; protected set; } = 0;
      public int Capacity { get {// TODO: ...};init {// TODO: ...}};
      public T this[int index] { get {...} }
      public void Add(T item) {// TODO: ...}
      public void AddRange(IEnumerable<T> collection) {// TODO: ...}
      public void Clear() {// TODO: ...}
      public void Sort(SortMethod method = ...) {// TODO: ...}
      private void BubbleSort() {// TODO: ...}
      private void InsertionSort() {// TODO: ...}
      private void SelectionSort(){// TODO: ...}
      private T[] list; // The internal storage array of the List.
    }  /* (^_^) The List Class Definition - Basic Version. (^_^) */
}
```

# DEMO

Object-Oriented Implementation of Sorting Algorithms in C# – The List<T> Generic Class.

# Sorting Algorithms – *Lab Assignments*

Lab Assignments

# Sorting Algorithms – *Lab Assignments*

- Structured Programming Implementation:

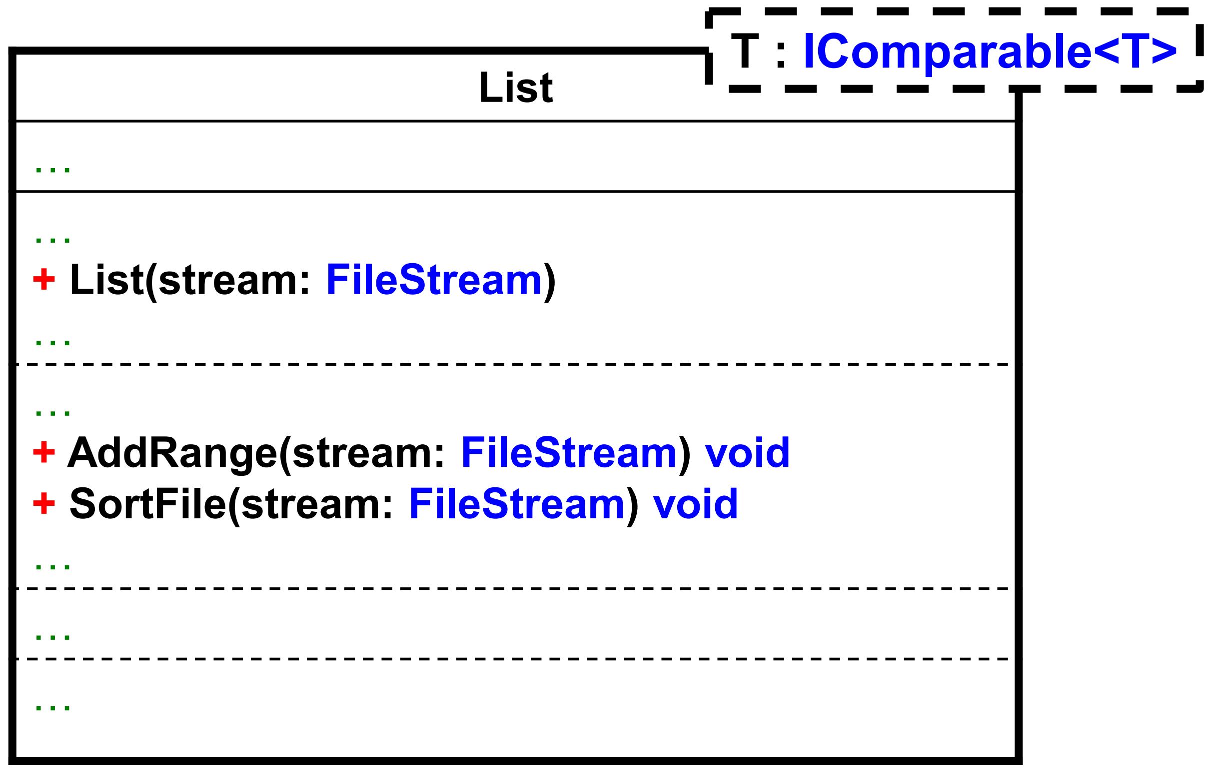
1. Create a ***Sort<T>( )*** generic static class method that implements one of the studied sort techniques on an array of any data type ***T*** that implements the **IComparable<*T*>** interface.
   - ***Bonus:*** Create a ***Point3D(x, y, z)*** structure that implements the ***IComparable*<*T*>** interface. Then, use the ***Sort<T>( )*** method to sort a list of 3D points.

   (Hint: P1 > P 2 when D1 > D2, where, $\mathrm{D} = \|P\| = \sqrt{x^2 + y^2 + z^2}$).

- Object-Oriented Programming Implementation:

2. Create a ***List<T>*** generic class that supports the studied sorting algorithms on an array member variable of any data type ***T*** that implements the ***IComparable*<*T*>** interface.
   - ***Bonus #1:*** Implement another sorting technique such as ***Shell Sort*** or ***Quick Sort***.

- Object-Oriented Programming Implementation:
  - ***Bonus #2:*** Add the indicated methods in the ***List<T>*** class diagram, shown below, to allow ***reading***, ***sorting*** and ***saving*** large data sets, which are stored on disk files in a "***comma-separated plain text format***".



# SUMMARY – Sorting Algorithms

- Sorting → is the process of arranging a set of similar information into an ascending or descending order.
- Types of Sorting → Different classifications exist:
  - Data Storage Medium → *Internal*, and *External*.
  - Data Sorting Technique → *Exchange*, *Selection*, and *Insertion*.
- Time Analysis → $T(n)_{Total} \approx T(n)_{Comparisons} + T(n)_{Exchanges}$
  - Elementary Methods → $O(n^2)$.
    - Iterative procedures with nested loops, *e.g.*, *Bubble* sort, *Selection* sort, *Insertion* sort.
  - Advanced Methods: → $O(n \log_2(n))$.
    - Divide-and-Conquer methods, *e.g.*, *Quick* sort, *Merge* sort, *Heap* sort.
- Space Analysis → Depends on whether the items are sorted In-place, or Not-in-place.
  - In-place sort → $O(1)$ and Not-in-place sort → $O(n)$.

Please consider the environment before printing this material.

Now, let's go to the DSA programming lab ☺


Lecture #3: Sorting Algorithms

Making a List of Data Elements in-Order ☺

# SUMMARY – Q & A

***Feedbacks:*** email to ameldin@gmail.com. We value your feedback!
(*Please include the following prefix in the subject field:* [DSA-C#])