Deakin University

Data Structures and Algorithms

OnTrack Submission

Basic Sorting

Submitted By:
Peter Stacey
pstacey
2020/07/29 13:44

Tutor:
Maksym Slavnenko

Outcome	**Weight**
Complexity	♦◊◊◊
Implement Solutions	♦♦♦◊◊
Document solutions	♦◊◊◊

This task involves implementing three sorting algoirthms that are provided in pseudocode in the SIT221 Workbook. In that regard, it is related to using a range of algorithms in implementing programs to address specific requirements. It does not relate to either of the other two learning outcomes.

July 29, 2020

```cs
using System;
using System.Collections.Generic;
using System.Text;

namespace Vector
{
    public class Vector<T> where T : IComparable<T>
    {

        // This constant determines the default number of elements in a newly created vector.
        // It is also used to extended the capacity of the existing vector
        private const int DEFAULT_CAPACITY = 10;

        // This array represents the internal data structure wrapped by the vector class.
        // In fact, all the elements are to be stored in this private  array.
        // You will just write extra functionality (methods) to make the work with the array more convenient for the user.
        private T[] data;

        // This property represents the number of elements in the vector
        public int Count { get; private set; } = 0;

        // This property represents the maximum number of elements (capacity) in the vector
        public int Capacity
        {
            get { return data.Length; }
        }

        // This is an overloaded constructor
        public Vector(int capacity)
        {
            data = new T[capacity];
        }

        // This is the implementation of the default constructor
        public Vector() : this(DEFAULT_CAPACITY) { }

        // An Indexer is a special type of property that allows a class or structure to be accessed the same way as array for its internal collection.
        // For example, introducing the following indexer you may address an element of the vector as vector[i] or vector[0] or ...
        public T this[int index]
        {
            get
            {
                if (index >= Count || index < 0) throw new IndexOutOfRangeException();
                return data[index];
            }
```

```
46              set
47              {
48                  if (index >= Count || index < 0) throw new
                    ↪  IndexOutOfRangeException();
49                  data[index] = value;
50              }
51          }
52
53          // This private method allows extension of the existing capacity of the
            ↪  vector by another 'extraCapacity' elements.
54          // The new capacity is equal to the existing one plus 'extraCapacity'.
55          // It copies the elements of 'data' (the existing array) to 'newData' (the
            ↪  new array), and then makes data pointing to 'newData'.
56          private void ExtendData(int extraCapacity)
57          {
58              T[] newData = new T[Capacity + extraCapacity];
59              for (int i = 0; i < Count; i++) newData[i] = data[i];
60              data = newData;
61          }
62
63          // This method adds a new element to the existing array.
64          // If the internal array is out of capacity, its capacity is first extended
            ↪  to fit the new element.
65          public void Add(T element)
66          {
67              if (Count == Capacity) ExtendData(DEFAULT_CAPACITY);
68              data[Count++] = element;
69          }
70
71          // This method searches for the specified object and returns the zerobased
            ↪  index of the first occurrence within the entire data structure.
72          // This method performs a linear search; therefore, this method is an O(n)
            ↪  runtime complexity operation.
73          // If occurrence is not found, then the method returns 1.
74          // Note that Equals is the proper method to compare two objects for
            ↪  equality, you must not use operator '=' for this purpose.
75          public int IndexOf(T element)
76          {
77              for (var i = 0; i < Count; i++)
78              {
79                  if (data[i].Equals(element)) return i;
80              }
81              return -1;
82          }
83
84          public ISorter Sorter { set; get; } = new DefaultSorter();
85
86          internal class DefaultSorter : ISorter
87          {
88              public void Sort<K>(K[] sequence, IComparer<K> comparer) where K :
                ↪  IComparable<K>
89              {
90                  if (comparer == null) comparer = Comparer<K>.Default;
```

```
91                  Array.Sort(sequence, comparer);
92              }
93          }
94
95          public void Sort()
96          {
97              if (Sorter == null) Sorter = new DefaultSorter();
98              Array.Resize(ref data, Count);
99              Sorter.Sort(data, null);
100         }
101
102         public void Sort(IComparer<T> comparer)
103         {
104             if (Sorter == null) Sorter = new DefaultSorter();
105             Array.Resize(ref data, Count);
106             if (comparer == null) Sorter.Sort(data, null);
107             else Sorter.Sort(data, comparer);
108         }
109
110         public override string ToString()
111         {
112             return "[" + string.Join(", ", data[0..Count]) + "]";
113         }
114     }
115 }
```

```cs
using System;
using System.Collections.Generic;
using System.Text;

namespace Vector
{
    class BubbleSort : ISorter
    {
        public BubbleSort() { }

        public void Sort<K>(K[] sequence, IComparer<K> comparer) where K :
        ↪ IComparable<K>
        {
            if (sequence is null) throw new ArgumentNullException();
            if (sequence.Length <= 1) return;

            if (comparer == null) comparer = Comparer<K>.Default;

            // Iterative approach using Bubble-Up
            // (ie. smallest element in the array bubbled to the start)
            // This version is the same as the algorithm on Page 105
            // of the SIT221 Workbook
            for(int i = -1; i < sequence.Length - 1; i++)
            {
                for(int j = sequence.Length - 2; j > i; j--)
                {
                    if(comparer.Compare(sequence[j], sequence[j + 1]) > 0)
                    {
                        K temp = sequence[j];
                        sequence[j] = sequence[j + 1];
                        sequence[j + 1] = temp;
                    }
                }
            }

            // Iterative approach using Bubble-Down
            // (ie. largest element bubbled to the end of the array)

            //for (int i = sequence.Length; i > 0; i--)
            //{
            //    for (int j = 0; j < sequence.Length - 1; j++)
            //    {
            //        if (comparer.Compare(sequence[j], sequence[j + 1]) > 0)
            //        {
            //            K temp = sequence[j];
            //            sequence[j] = sequence[j + 1];
            //            sequence[j + 1] = temp;
            //        }
            //    }
            //}
        }
    }
}
```

```cs
using System;
using System.Collections.Generic;
using System.Text;

namespace Vector
{
    class InsertionSort : ISorter
    {
        public InsertionSort() { }

        public void Sort<K>(K[] sequence, IComparer<K> comparer) where K :
        ↪ IComparable<K>
        {
            if (sequence is null) throw new ArgumentNullException();
            if (sequence.Length <= 1) return;

            if (comparer == null) comparer = Comparer<K>.Default;

            // Iterative approach
            // Same as the algorithm on Page 108 of the SIT221 Workbook
            for (int i = 1; i < sequence.Length; i++)
            {
                int j;
                K hold = sequence[i];
                for (j = i - 1; j >= 0 && (comparer.Compare(sequence[j], hold) >
                ↪ 0); j--)
                {
                    sequence[j + 1] = sequence[j];
                }
                sequence[j + 1] = hold;
            }
        }
    }
}
```

```cs
using System;
using System.Collections.Generic;
using System.Text;

namespace Vector
{
    class SelectionSort : ISorter
    {
        public void Sort<K>(K[] sequence, IComparer<K> comparer) where K :
        ↪ IComparable<K>
        {
            if (sequence is null) throw new ArgumentNullException();
            if (sequence.Length <= 1) return;

            if (comparer == null) comparer = Comparer<K>.Default;

            // Iterative approach
            // Same as the algorithm on Page 107 of the SIT221 Workbook
            for (int i = 0; i < sequence.Length - 1; i++)
            {
                int smallest = i;
                for (int j = i + 1; j < sequence.Length; j++)
                {
                    if(comparer.Compare(sequence[smallest], sequence[j]) > 0)
                    {
                        smallest = j;
                    }
                }
                K temp = sequence[smallest];
                sequence[smallest] = sequence[i];
                sequence[i] = temp;
            }
        }
    }
}
```