Deakin University

Data Structures and Algorithms

OnTrack Submission

Recursive Sorting

Submitted By:
Peter Stacey
pstacey
2020/08/09 20:19

Tutor:
Maksym Slavnenko

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

The implementations of MergeSort use different space complexity and the Inplace Quicksort is designed to minimise additional memory use during sorting. This involves evaluating and considering the space complexity of algorithms as part of ULO1. Additionally, this task involves implementing algorithms recursively and aligns with ULO2.

August 9, 2020

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

namespace Task_3_2
{
    /// <summary>
    /// Sorts an array of generic items using rules provided in a comparer
    /// </summary>
    class RandomizedQuickSort: ISorter
    {
        private Random Random { get; set; } = new Random();
        private int RandomPivot(int a, int b) => Random.Next(a, b);

       /// <summary>
       /// Sorts a sequence of 2 or more generic elements
       /// </summary>
       /// <param name="sequence">The array of generic elements to sort</param>
       /// <param name="comparer">IComparer to use for comparison</param>
       /// <typeparam name="K">The type of the elements</typeparam>
       /// <exception cref="System.ArgumentNullException">Thrown if the sequence
       /// of elements is null</exception>
        public void Sort<K>(K[] sequence, IComparer<K> comparer) where K :
        ↪ IComparable<K>
        {
            if (sequence == null) throw new ArgumentNullException();
            if (sequence.Length < 2) return; // Nothing to do if fewer than 2
            ↪ elements
            if (comparer == null) comparer = Comparer<K>.Default;
            Sort(sequence, comparer, 0, sequence.Length - 1);
        }

        /// <summary>
        /// Recursively quick sorts an array of elements in place, using a random
        /// pivot selection
        /// </summary>
        /// <param name="sequence">The sequence of elements</param>
        /// <param name="a">The index of one element to swap</param>
        /// <typeparam name="K">The index of the other element to swap</typeparam>
        private void Swap<K>(K[] sequence, int a, int b)
        {
            K temp = sequence[a];
            sequence[a] = sequence[b];
            sequence[b] = temp;
        }

        /// <summary>
        /// Recursively quick sorts an array of elements in place, using a random
        /// pivot selection
        /// </summary>
        /// <param name="sequence">The sequence of the elements to sort</param>
        /// <param name="comparer">The IComparer to use</param>
        /// <param name="a">The left index of the part to sort</param>
        /// <param name="b">The right index of the part to sort</param>
```

```
52          /// <typeparam name="K">The type of the elements in the array</typeparam>
53          private void Sort<K>(K[] sequence, IComparer<K> comparer,int a,int b) where
             ↪ K : IComparable<K>
54          {
55              if (a >= b) return;
56
57              int left = a;
58              int right = b - 1;
59
60              Swap(sequence, RandomPivot(a, b), b);
61              K pivot = sequence[b];
62
63              while(left <= right)
64              {
65                  while (left <= right && comparer.Compare(sequence[left], pivot) <
                     ↪ 0) left++;
66                  while (left <= right && comparer.Compare(sequence[right], pivot) >
                     ↪ 0) right--;
67                  if(left <= right)
68                  {
69                      Swap(sequence, left, right);
70                      left++;
71                      right--;
72                  }
73              }
74
75              Swap(sequence, left, b);
76
77              Sort(sequence, comparer, a, left - 1);
78              Sort(sequence, comparer, left + 1, b);
79          }
80      }
81  }
```

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

namespace Task_3_2
{
    /// <summary>
    /// Implementation of the top-down, recursive merge sort. This has a time
    /// complexity of O(nlog n)
    /// </summary>
    class MergeSortTopDown : ISorter
    {
        /// <summary>
        /// Sorts an array of generic elements according to the rules
        /// of a defined IComparer. The implementation uses the top-down
        /// MergeSort with O(nlog n).
        /// </summary>
        /// <param name="sequence">The sequence of elements to sort</param>
        /// <param name="comparer">The IComparer to use for sorting</param>
        /// <typeparam name="K">The type of the elements of the array</typeparam>
        public void Sort<K>(K[] sequence, IComparer<K> comparer) where K :
        ↪ IComparable<K>
        {
            if (sequence is null) throw new ArgumentNullException();
            if (sequence.Length < 2) return;
            if (comparer is null) comparer = Comparer<K>.Default;

            int mid = sequence.Length / 2;
            K[] left = sequence[0..mid];
            K[] right = sequence[mid..sequence.Length];

            Sort(left, comparer);
            Sort(right, comparer);
            Merge(left, right, sequence, comparer);
        }

        /// <summary>
        /// Merges two arrays in order required by the rules of the IComparer
        /// </summary>
        /// <param name="left">One array of elements to merge</param>
        /// <param name="right">Second array of elements to merge</param>
        /// <param name="sequence">The sequence to merge into</param>
        /// <param name="comparer">The IComparer to use for sorting</param>
        /// <typeparam name="K">The type of the elements in the arrays</typeparam>
        private void Merge<K>(K[] left, K[] right, K[] sequence, IComparer<K>
        ↪ comparer) where K : IComparable<K>
        {
            int i = 0;
            int j = 0;
            while (i + j < sequence.Length)
            {
                if (j == right.Length || (i < left.Length &&
                ↪ comparer.Compare(left[i], right[j]) < 0))
                    sequence[i + j] = left[i++];
```

```
51                  else
52                      sequence[i + j] = right[j++];
53              }
54          }
55      }
56  }
```

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

namespace Task_3_2
{
    /// <summary>
    /// An iterative, bottom-up implementation of MergeSort
    /// </summary>
    class MergeSortBottomUp: ISorter
    {
        /// <summary>
        /// Sorts an array of generic elements in an array according to
        /// rules defined in an IComparer.
        /// </summary>
        /// <param name="sequence">The sequence of generic elements to sort</param>
        /// <param name="comparer">The IComparer to use for sorting rules</param>
        /// <typeparam name="K">The type of the elements</typeparam>
        /// <exception cref="System.ArgumentNullException">Thrown when the sequence
        /// is null</exception>
        public void Sort<K>(K[] sequence, IComparer<K> comparer) where K :
        ↪ IComparable<K>
        {
            if (sequence is null) throw new ArgumentNullException();
            if (sequence.Length < 2) return;
            if (comparer is null) comparer = Comparer<K>.Default;

            K[] nonr = sequence;
            K[] dest = new K[sequence.Length];
            K[] temp;

            for (int i = 1; i < sequence.Length; i *= 2)
            {
                for (int j = 0; j < sequence.Length; j += 2 * i) Merge(nonr, dest,
                ↪ comparer, j, i);
                temp = nonr;
                nonr = dest;
                dest = temp;
            }
            if (sequence != nonr)
                Array.Copy(nonr, 0, sequence, 0, sequence.Length);
        }

        /// <summary>
        /// Sorts and merges values from two arrays
        /// </summary>
        /// <param name="ins">First array to merge elements of</param>
        /// <param name="outs">Second array to merge elements of</param>
        /// <param name="comparer">THe rules for sorting as an IComparer</param>
        /// <param name="start">The start index to merge</param>
        /// <param name="inc">The number of elements to merge</param>
        /// <typeparam name="K">The type of the elements in the arrays</typeparam>
        private void Merge<K>(K[] ins, K[] outs, IComparer<K> comparer, int start,
        ↪ int inc) where K : IComparable<K>
```

```
51          {
52              int end1 = Math.Min(start + inc, ins.Length);
53              int end2 = Math.Min(start + 2 * inc, ins.Length);
54
55              int x = start;
56              int y = start + inc;
57              int z = start;
58
59              while (x < end1 && y < end2)
60                  if (comparer.Compare(ins[x], ins[y]) < 0)
61                      outs[z++] = ins[x++];
62                  else
63                      outs[z++] = ins[y++];
64              if (x < end1)
65                  Array.Copy(ins, x, outs, z, end1 - x);
66              else if (y < end2)
67                  Array.Copy(ins, y, outs, z, end2 - y);
68          }
69      }
70  }
```