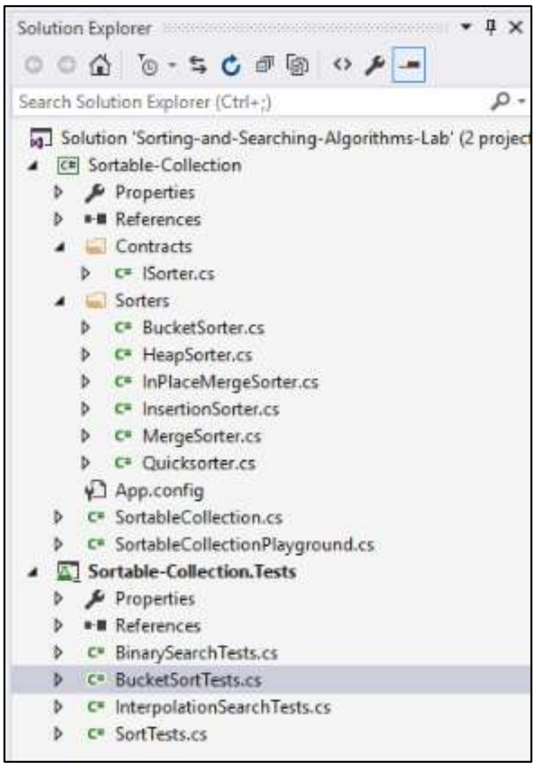
Exercises: Sorting and Searching Algorithms

This document defines the **in-class exercises** assignments for the ["Algorithms" course @ Software University](#).

For this exercise and the homework for this lecture, you are given a C# solution holding the following assets:

- **ISorter** interface – defines a single method Sort; the interface should be implemented by several classes
- **SortableCollection** class - holds comparable elements. It contains a **Sort** method which accepts a custom ISorter as parameter; it also defines three additional methods – **BinarySearch**, **InterpolationSearch** and **Shuffle**
- **Sorters** namespace – contains empty sorter classes which should be implemented
- **Unit tests** – for both the sorting and the searching algorithms



```
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Sorting-and-Searching-Algorithms-Lab' (2 projec
  Sortable-Collection
    Properties
    References
    Contracts
      ISorter.cs
    Sorters
      BucketSorter.cs
      HeapSorter.cs
      InPlaceMergeSorter.cs
      InsertionSorter.cs
      MergeSorter.cs
      Quicksorter.cs
    App.config
    SortableCollection.cs
    SortableCollectionPlayground.cs
  Sortable-Collection.Tests
    Properties
    References
    BinarySearchTests.cs
    BucketSortTests.cs
    InterpolationSearchTests.cs
    SortTests.cs
```

In this exercise, you need to implement the **Quicksorter** and **BucketSorter** classes as well as the **BinarySearch** method. The other classes and methods are intended for homework.

Part I – Study the Provided Project

Open the **Sorting-and-Searching-Algorithms-Lab** solution. The project skeleton opens correctly in **Visual Studio 2015** but can be opened in other Visual Studio versions as well and can also run in **SharpDevelop** and **Xamarin Studio**.

Below follows a description of the assets. If you wish, you could move on to Part II.

ISorter Interface

You are given an interface ISorter which defines a single **Sort** method. The method works on a generic **List<T> collection**. List<T> is convenient as it provides several useful methods that will simplify some of the tasks along with indexing.

Sorter Classes

There are several, currently empty classes, which implement ISorter. Each class will implement the Sort method using a different algorithm, e.g. Quicksorter will utilize the Quicksort algorithm to sort the elements.

```
public class Quicksorter<T> : ISorter<T> where T : IComparable<T>
{
    public void Sort(List<T> collection)
    {
        // TODO
    }
}
```

SortableCollection Class

The **SortableCollection** class has three constructors – one creates an empty SortableCollection, one receives an already created collection which it uses to create the SortableCollection, and the third one receives a variable number of arguments (the elements separated by commas).

```
public SortableCollection()
{
}

public SortableCollection(IEnumerable<T> items)
{
    this.Items = new List<T>(items);
}

public SortableCollection(params T[] items)
    : this(items.AsEnumerable())
{
}
```

There are two properties defined for the class. **Count** returns the number of elements and **Items** returns a list of the items in the collection:

```
public List<T> Items { get; } = new List<T>();

public int Count => this.Items.Count;
```

Note: We've used some new features of the C# language. In case you're having trouble compiling the code above, you can replace it with the following:

```
public List<T> Items { get; private set; }

public int Count
{
    get
    {
        return this.Items.Count;
    }
}
```

You will also need to initialize Items in the empty constructor:

```
public SortableCollection()
{
    this.Items = new List<T>();
}
```

The Sort method takes an ISorter and sorts the collection using the Sort method of the ISorter it received:

```
public void Sort(ISorter<T> sorter)
{
    sorter.Sort(this.Items);
}
```

There are two search methods and one shuffle method which are left empty for now:

```
public int BinarySearch(T item)
{
    throw new NotImplementedException();
}

public int InterpolationSearch(T item)
{
    throw new NotImplementedException();
}

public void Shuffle()
{
    throw new NotImplementedException();
}
```

Finally, there are two methods which can be useful when testing your code – **ToArray** returns the elements in the collection as an array and **ToString** prints the elements in a nice format.

```
public T[] ToArray()
{
    return this.Items.ToArray();
}

public override string ToString()
{
    return $"[{string.Join(", ", this.Items)}]";
}
```

Note: Interpolated strings are a feature in C# 6. The ToString method above is equivalent to:

```
public override string ToString()
{
    return string.Format("[{0}]", string.Join(", ", this.Items));
}
```

Unit Tests

The unit tests work with integer numbers. They test different scenarios and will help you get your implementations right.

In the beginning of the **SortTests** class we define a static ISorter, currently it will be a Quicksorter. When implementing and testing the rest of the classes for your homework, remember to change the sorter with the one you want to test, e.g. MergeSorter<int>:

```
private static readonly ISorter<int> TestSorter = new Quicksorter<int>();
```

Part II – Implement Quicksort

Problem 1. Setup

Let's declare a collection of 5 elements to work with for now. Create it in the Main method of the **SortableCollectionPlayground** class and call the Sort method using the Quicksorter. Printing the collection will be an easy way to see if the sorter works as intended:

```
var collection = new SortableCollection<int>(2, -1, 5, 0, -3);
Console.WriteLine(collection);

collection.Sort(new Quicksorter<int>());
Console.WriteLine(collection);
```

Problem 2. Learn About Quicksort

You can learn about the Quicksort algorithm from Wikipedia.

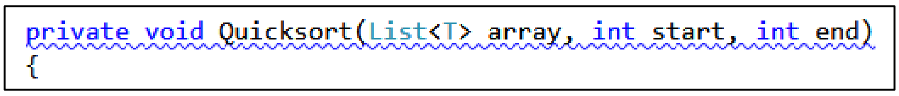
A great tool for visualizing the algorithm (along with many others) is available at Visualgo.net.

The algorithm in short:

- Quicksort takes unsorted partitions of an array and sorts them. Initially, the entire array is unsorted.
- We pick a **pivot** element, this is the first element of an unsorted partition. The goal is to move the pivot to its correct place in the array. We declare a **storeIndex** – this is the index where pivot will be moved to; initially **storeIndex = pivotIndex + 1**.
- Every element is **compared** to the pivot. If the element is smaller than pivot, it is swapped with the element currently at the storeIndex and the storeIndex is incremented. The result of this is that when we're done and we move pivot to the storeIndex, all elements to the left of it will be smaller than it, and all elements to the right will be larger. This means pivot is where it should be.
- With pivot moved to its correct place, we now have two unsorted partitions – one to the left of it and one to the right. We **call the procedure recursively** for these partitions. The bottom of the recursion is when a partition has a size of 1, which is by definition sorted.

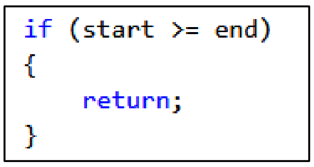
Problem 3. Write the Quicksort Method

As discussed, the algorithm works on a partition of the array, which means we need to know the start and end indices of the partition. In our case, we're using a List<T> instead of an array, so the method singature will be as follows:



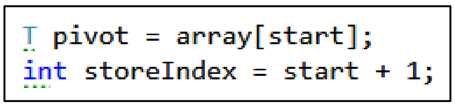
```
private void Quicksort(List<T> array, int start, int end)
{
```

First, remember to take care of the bottom of the recursion. The method should return when we have a single element; we can do this by comparing start and end:



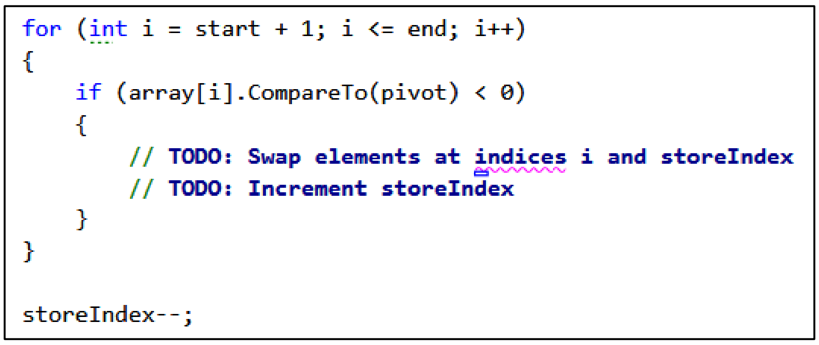
```
if (start >= end)
{
    return;
}
```

We take the first element as pivot and storeIndex will be the next index:



```
T pivot = array[start];
int storeIndex = start + 1;
```

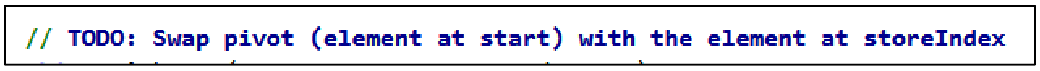
Now it's time to start comparing each element to pivot. We can use a for-loop and the **CompareTo** method (we are guaranteed to have the CompareTo method as we've placed the proper restriction on the generic type T):



```
for (int i = start + 1; i <= end; i++)
{
    if (array[i].CompareTo(pivot) < 0)
    {
        // TODO: Swap elements at indices i and storeIndex
        // TODO: Increment storeIndex
    }
}

storeIndex--;
```

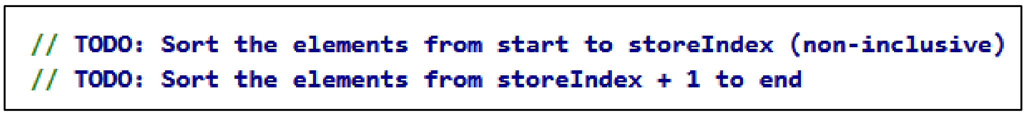
Swapping the elements is trivial, you can extract a separate Swap method for the purpose. After the loop finishes, we need to decrement storeIndex; think about why this is the case.

Swap the pivot and the element at storeIndex; this will place pivot where it should be:



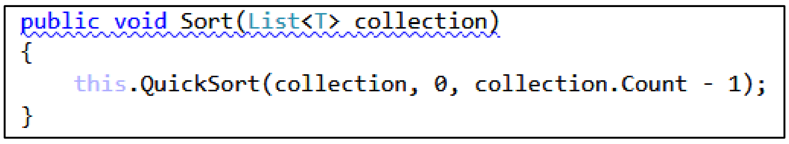
```
// TODO: Swap pivot (element at start) with the element at storeIndex
```

Finally, we need to recursively sort the two partitions to the left and to the right of pivot (now located at storeIndex):



```
// TODO: Sort the elements from start to storeIndex (non-inclusive)
// TODO: Sort the elements from storeIndex + 1 to end
```

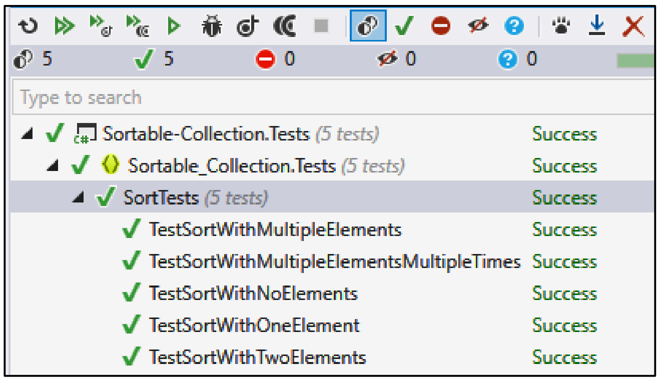
And before we test, call the Quicksort method in the Sort method of the Quicksorter class; we view the entire collection as a partition, so start = 0 and end = number of elements - 1:



```
public void Sort(List<T> collection)
{
    this.QuickSort(collection, 0, collection.Count - 1);
}
```

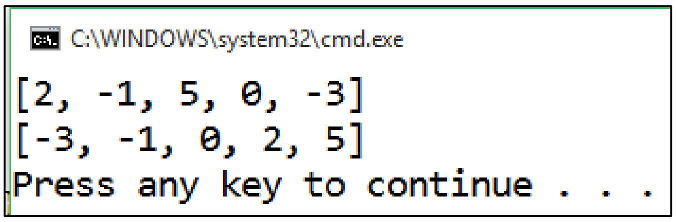
Problem 4. Test the Code

Run the unit tests in the **SortTests** class. They should all pass. If the collection has 0 or 1 elements in it, the Quicksort method will simply finish on the first step which is a correct behaviour.



```
5    ✓ 5    0    0    0
Type to search
Sortable-Collection.Tests (5 tests)                    Success
  Sortable_Collection.Tests (5 tests)                  Success
    SortTests (5 tests)                                Success
      TestSortWithMultipleElements                     Success
      TestSortWithMultipleElementsMultipleTimes        Success
      TestSortWithNoElements                           Success
      TestSortWithOneElement                           Success
      TestSortWithTwoElements                          Success
```

Run the main method. It should produce the following:



```
C:\WINDOWS\system32\cmd.exe
[2, -1, 5, 0, -3]
[-3, -1, 0, 2, 5]
Press any key to continue . . .
```

It all seems correct.

If you want, you can go through the process step by step with the debugger and see how the pivot is moved and along the way – how all elements smaller than pivot are move to the left of it.

Part III – Implement Bucket Sort

Problem 5. Learn About Bucket Sort

Bucket sort is a non-comparison algorithm, therefore it works only on numbers. You may view an example of it here.

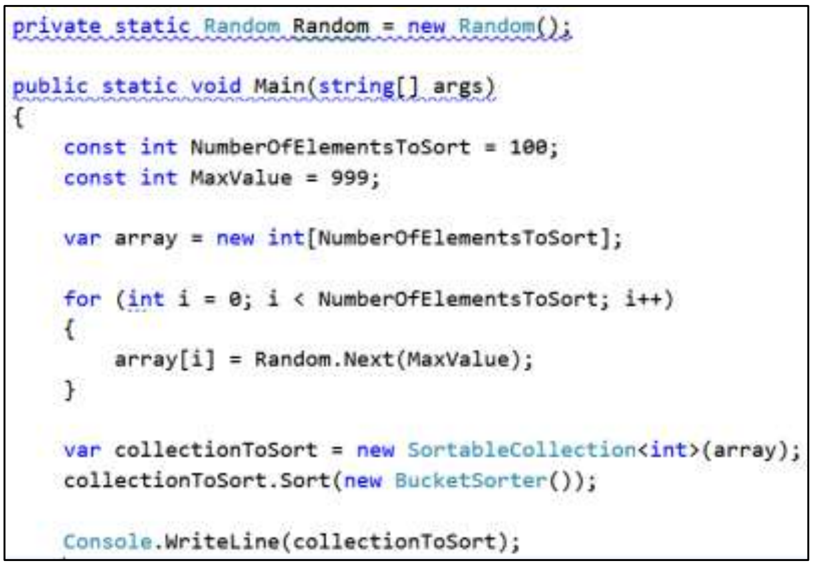
The idea is that the elements of the array are divided into buckets, then each bucket is sorted individually, usually using Insertion Sort and then the buckets are combined to produce the result. E.g. if we have numbers from 0 to 99 in the array, we can divide them into 10 buckets using their most significant digit (the tens).

A potential problem is if the elements are not distributed evenly among the buckets, which slows down the conventional sorting algorithm used in the next step.

For this problem, we'll assume we'll be sorting natural numbers (**non-negative integers**). The unit tests won't be checking for invalid data, but you can expand on the provided code and make it more flexible and hard to break.

Problem 6. Setup

Let’s assume that we’ll be sorting an array of 100 numbers in the range 1 – 999. We can generate such an array with the help of the **Random** class:

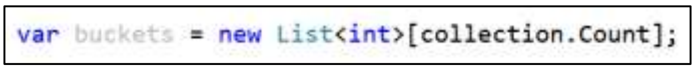


```
private static Random Random = new Random();

public static void Main(string[] args)
{
    const int NumberOfElementsToSort = 100;
    const int MaxValue = 999;

    var array = new int[NumberOfElementsToSort];

    for (int i = 0; i < NumberOfElementsToSort; i++)
    {
        array[i] = Random.Next(MaxValue);
    }

    var collectionToSort = new SortableCollection<int>(array);
    collectionToSort.Sort(new BucketSorter());

    Console.WriteLine(collectionToSort);
```

Problem 7. Write the Method

A reasonable approach is to use 100 buckets (the number of elements), decreasing the possibility of multiple numbers ending up in the same bucket.

We start by creating the buckets. We know how many buckets we’ll have, that’s the number of elements. But we don’t know how many elements will end up in any given bucket, therefore we can use an array of lists:



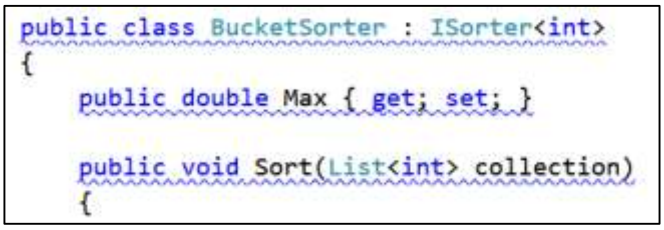
```
var buckets = new List<int>[collection.Count];
```

Next, we iterate the collection and decide where to put each number. We can distribute the numbers based on the following formula:

bucket index = element value / max value * number of buckets

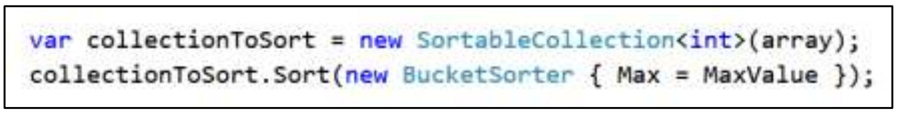
You can make sure that it is correct and will distribute each element in an existing bucket. If the numbers in the collection are uniformly distributed, than the number of elements per bucket will be small.

The BucketSorter will need to know the maximum value that’s allowed for an element; let’s store it in a property:



```
public class BucketSorter : ISorter<int>
{
    public double Max { get; set; }

    public void Sort(List<int> collection)
    {
```

In the main method, we need to pass the maximal value when creating the BucketSorter like this:



```
var collectionToSort = new SortableCollection<int>(array);
collectionToSort.Sort(new BucketSorter { Max = MaxValue });
```

We apply the formula above to find the bucket index and place it there (note that the bucket may not be initialized, we need to check that before trying to insert a number into it):

```
foreach (var element in collection)
{
    int bucketIndex = (int)(element / this.Max * collection.Count);

    if (buckets[bucketIndex] == null)
    {
        buckets[bucketIndex] = new List<int>();
    }

    buckets[bucketIndex].Add(element);
}
```

Next, iterate each bucket and sort it. Since we already have the Quicksorter class fully implemented, let's use it. Again, we have to remember that some buckets may be null:

```
var sorter = new Quicksorter<int>();

for (int i = 0; i < buckets.Length; i++)
{
    // TODO: Sort the bucket if it's initialized
}
```

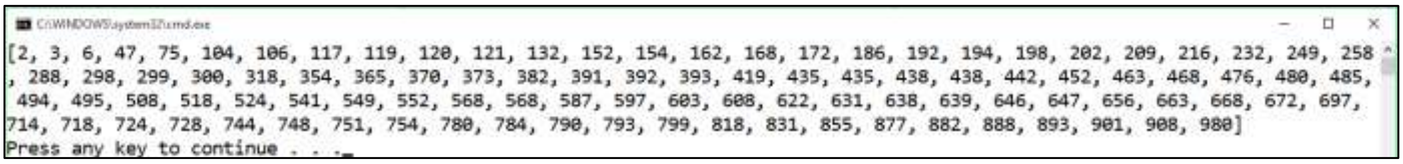
Finally, iterate the buckets and copy the now sorted numbers to the collection:

```
int index = 0;
foreach (var bucket in buckets)
{
    if (bucket == null)
    {
        continue;
    }

    foreach (var item in bucket)
    {
        collection[index] = item;
        index++;
    }
}
```

Problem 8. Test the Code

Note that this algorithm is different from the others as we need to know the maximum value allowed, therefore the unit tests that we use for the other sorters cannot test this one properly.

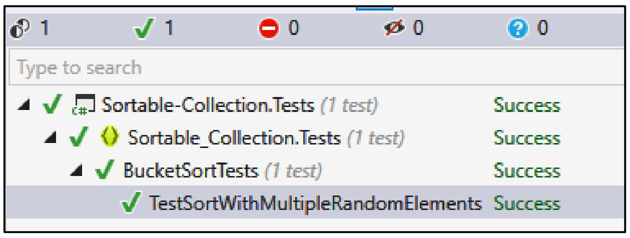
Run the main method several times and check if it's correct. You may get something like this:



```
C:\WINDOWS\system32\cmd.exe
[2, 3, 6, 47, 75, 104, 106, 117, 119, 120, 121, 132, 152, 154, 162, 168, 172, 186, 192, 194, 198, 202, 209, 216, 232, 249, 258
, 288, 298, 299, 300, 318, 354, 365, 370, 373, 382, 391, 392, 393, 419, 435, 435, 438, 438, 442, 452, 463, 468, 476, 480, 485,
 494, 495, 508, 518, 524, 541, 549, 552, 568, 568, 587, 597, 603, 608, 622, 631, 638, 639, 646, 647, 656, 663, 668, 672, 697,
714, 718, 724, 728, 744, 748, 751, 754, 780, 784, 790, 793, 799, 818, 831, 855, 877, 882, 888, 893, 901, 908, 980]
Press any key to continue . . .
```

Change the values of the MaxValue and the NumberOfElementsToSort to test whether the formula for distributing the numbers into buckets works as intended.

Finally, run the unit test in the BucketSorterTests class. It may take a while, but it should pass.



```
1   ✓ 1   0   0   0
Type to search
◢ ✓ Sortable-Collection.Tests (1 test)                 Success
  ◢ ✓ Sortable_Collection.Tests (1 test)               Success
    ◢ ✓ BucketSortTests (1 test)                       Success
        ✓ TestSortWithMultipleRandomElements            Success
```

Part IV – Implement Binary Search

Problem 9. Learn About Binary Search

First, if you're not familiar with the concept, read about binary search in Wikipedia.

Here you can find a tool which shows visually how the search is performed.

In short, if we have a **sorted collection** of comparable elements, instead of doing linear search (which takes linear time), we can eliminate half the elements at each step and finish in logarithmic time.

Binary search is a **divide-and-conquer** algorithm; we start at the middle of the collection, if we haven't found the element there, there are three possibilities:

1) The element we're looking for is smaller – then look to the left of the current element, we know all elements to the right are larger;

2) The element we're looking for is larger – look to the right of the current element;

3) The element is not present, traditionally, return -1 in that case.

Problem 10. Write the BinarySearch Method

Since we'll be checking only half of the elements after each step, we need to keep track of the start and end indices of the half we're interested in. The BinarySearch method receives only a value as a parameter, therefore we can write the logic in a separate method, e.g. **BinarySearchProcedure**, which will receive the element, called item, the start and end index.

```
private int BinarySearchProcedure(T item, int startIndex, int endIndex)
{
```

We'll be implementing a recursive algorithm, so we need to know where it ends. When we reach one of the two ends of the collection startIndex will be equal to endIndex; if we haven't found the item, at the next step startIndex will become larger, therefore we can end the recursion when startIndex > endIndex:

```
if (endIndex < startIndex)
{
    return -1;
}
```

Next step is to calculate the midpoint index. This is simple enough:

```
// calculate midpoint to cut set in half
int midpoint = startIndex + (endIndex - startIndex) / 2;
```

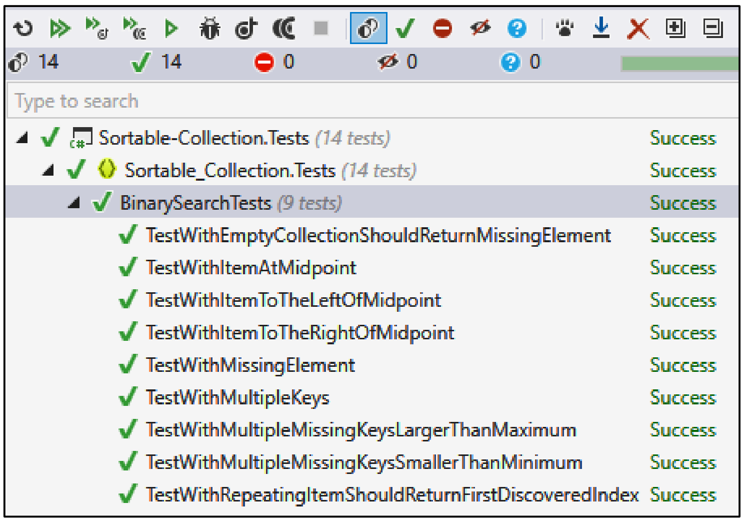
Now, it's time to compare and decide what to do.

If the element at index midpoint is smaller than item, look to the right; if it's larger, look to the left; if it's equal, just return midpoint.

```
if (this.Items[midpoint].CompareTo(item) > 0)
{
   // TODO: Search to the left - from startIndex to midpoint (exclusive)
}

if (this.Items[midpoint].CompareTo(item) < 0)
{
    // TODO: Search to the right - from midpoint + 1 to endIndex
}

return midpoint;
```

Problem 11. Test the Code

Run the unit tests. They use the **Array.BinarySearch** method in some cases to check the results. If everything is correct, all tests should pass:



```
14   ✓ 14   0   0   0
Type to search
◢ ✓ Sortable-Collection.Tests (14 tests)                          Success
  ◢ ✓ Sortable_Collection.Tests (14 tests)                        Success
    ◢ ✓ BinarySearchTests (9 tests)                               Success
        ✓ TestWithEmptyCollectionShouldReturnMissingElement       Success
        ✓ TestWithItemAtMidpoint                                  Success
        ✓ TestWithItemToTheLeftOfMidpoint                         Success
        ✓ TestWithItemToTheRightOfMidpoint                        Success
        ✓ TestWithMissingElement                                  Success
        ✓ TestWithMultipleKeys                                    Success
        ✓ TestWithMultipleMissingKeysLargerThanMaximum            Success
        ✓ TestWithMultipleMissingKeysSmallerThanMinimum           Success
        ✓ TestWithRepeatingItemShouldReturnFirstDiscoveredIndex   Success
```

That's it! Good job!