Homework: Sorting and Searching Algorithms

This document defines the **homework assignments** for the "Algortihms" course @ Software University. Please submit a single **`zip`** / **`rar`** / **`7z`** archive holding the solutions (source code) of all below described problems.

For this homework, we recommend that you **use the solution provided for the in-class exercise**. You can download it from here. If you're not familiar with the skeleton, read the exercise description. You are provided with a **SortableCollection** class and several sorter classes, along with **unit tests** to check their correctness.

Problem 1. Implement Insertion Sort

Insertion sort is very simple: iterate the elements and move them to the left so that they end up with a smaller (or equal) element to the left and a larger one to the right. Check out Visualgo for a visual representation.

Since all classes implementing the ISorter interface in the skeleton work with a **List<T>**, we can use the very convenient **RemoveAt** and **Insert** methods to implement this sorting algorithm in just a few lines of code.

We can start at index 1 and check to the left until we find an element which is at least equal to the current element or until we reach index 0. Keeping track of where the last larger element was found, we can remove the element and insert it there:

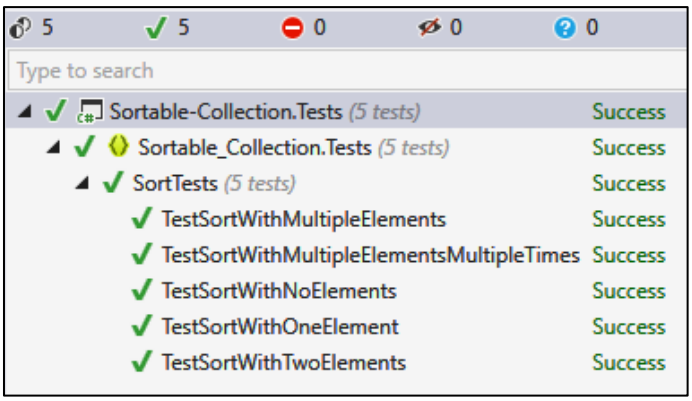
```
for (int i = 1; i < collection.Count; i++)
{
    int targetIndex = i;

    while (true) // TODO: decrement while the element at index i is smaller and targetIndex is positive
    {
        targetIndex--;
    }

    if (targetIndex < i)
    {
        T item = collection[i];
        collection.RemoveAt(i);
        collection.Insert(targetIndex, item);
    }
}
```

Change the sorter used in the unit tests and run them:

```
[TestClass]
public class SortTests
{
    private static readonly ISorter<int> TestSorter = new InsertionSorter<int>();
```



Problem 2. Implement Interpolation Search

In the SortableCollection class there is an empty method **InterpolationSearch**. Your task is to **implement** it.

Write unit tests in the **InterpolationSearchTests** class (you may use the already implemented BinarySearch unit tests as a template).

Problem 3. Implement Merge Sort

Description

Merge sort is an algorithm which takes two sorted halves of an array and merges them into one sorted array.

The idea is that an array is continuously split until the halves are of size 1 – by definition, an array with one element is sorted. Then, all the actual sorting is done by the merge procedure. We keep two pointers, one pointing to the smallest element of the left sub-array and one pointing to the smallest element of the right sub-array.

Merging requires a temporary array. In this problem, we'll initialize a single array and use it throughout the entire process. This will eliminate the constant creation and destruction of temporary arrays at each step and will improve performance.

Solution

The **MergeSort** method will take an array, a pre-existing temporary array, start index and end index of the partition to be sorted:

```
private void MergeSort(List<T> array, T[] temporaryArray, int start, int end)
{
```

It should end when start == end (partition of size 1), so we can write all logic in an if-statement. What will the sort do? It will find the midpoint and recursively sort the two halves of the array:

```
if (start < end)
{
    int middle = (end + start) / 2;

    // TODO: Sort first half (start to middle)
    // TODO: Sort second half (middle + 1 to end)

    // TODO: Merge sorted halves
}
```

Now, the **merge procedure**. It should keep two pointers as discussed – one pointing to the smallest element of the left sub-array and the other pointing to the smallest element of the right sub-array. Since the two halves are sorted, initially the left pointer points to start and the right pointer to middle + 1:

```
int leftMinIndex = start;
int rightMinIndex = middle + 1;
int tempIndex = 0;
```

Now, it's up to you to traverse both sub-arrays simultaneously and store the smallest element of both to the temporary array at **tempIndex**.

Before you copy back the elements from the temporary array to the original array (in our case list), check if there are elements left in one of the sub-arrays and move them to the temporary array as well.

Finally, copy the elements from the temporary array back to the original:

```
// Copy all elements from temp to the original array
tempIndex = 0;
leftMinIndex = start;

while (tempIndex < temporaryArray.Length && leftMinIndex <= end)
{
    array[leftMinIndex] = temporaryArray[tempIndex];
    leftMinIndex++;
    tempIndex++;
}
```

Test

Run the unit tests; they should pass (and more quickly too).

Problem 4. Implement Heap Sort

Implement the HeapSort algorithm. It works by building a min-heap from an unsorted array and then taking the smallest element repeatedly until the heap is empty. You can learn more about the Heap structure and get a complete implementation from the Advanced Tree Structures lecture (Data Structures course) or you may implement it yourself.

Test the HeapSorter using the provided unit tests.

Problem 5. Implement the Fisher-Yates Shuffle

Implement the **Fisher-Yates** shuffle algorithm. Use the Shuffle method in the **SortableCollection** class.

In the main program, write several test cases and shuffle arrays of different sizes repeatedly to show that shuffling is done properly (e.g. no clustering occurs).

Problem 6. * Implement In-place Merge Sort

Implement Merge Sort using **in-place merging** (without the use of an auxiliary array). Use the **InPlaceMergeSorter** class and then test it using the provided unit tests.

Hint: https://stackoverflow.com/questions/2571049/how-to-sort-in-place-using-the-merge-sort-algorithm

Problem 7. * Words

You can test your solution to the problem in the Judge system here.

You are given a string containing Latin letters. Write a program that finds **the number of all words with no two consecutive equal characters that can be generated by reordering the given letters**. The generated words should contain all given letters. If the given word meets the requirements it should also be considered in the count.

Input

- The input data should be read from the console.

- On the only input line there will be a single word containing all the letters that you should use for generating the words.
- The input data will always be valid and in the format described. There is no need to check it explicitly.

Output

- The output data should be printed on the console.
- On the only output line write the number of words found.

Constraints

- The number of the given letters will be between 1 and 10, inclusive.
- All given letters will be small Latin letters ('a' – 'z')
- Allowed working time for your program: 0.35 seconds. Allowed memory: 32 MB.

Examples

Input	Sample Output	Comments
xy	2	Two possible words: "xy" and "yx"
xxxy	0	It is impossible to construct a word with these letters.
aahhhaa	1	The only possible word is "ahahaha".
nopqrstuvw	3628800	There are 3628800 possible words.

Problem 8. Needles

You can test your solution to the problem in the Judge system here.

This problem is about finding the proper place of numbers in an array. From the console, you'll read a sequence of non-decreasing integers with randomly distributed "holes" among them (represented by zeros).

Then you'll be given the needles – numbers which should be inserted into the sequence, so that it remains non-decreasing (discounting the "holes"). For each needle, find the left-most index where it can be inserted.

Input

- The input should be read from the console.
- On the first line you'll be given the numbers C and N separated by a space.
- On the second line you'll be given C non-negative integers forming a non-decreasing sequence (disregarding the zeros).
- On the third line you'll be given N positive integers, the needles.
- The input data will always be valid and in the format described. There is no need to check it explicitly.

Output

- The output should be printed on the console. It should consist of a single line.
- On the only output line print N numbers separated by a space. Each number represents the left-most index at which the respective needle can be inserted.

Constraints

- All input numbers will be 32-bit signed integers.
- N will be in the range [1 … 1000].
- C will be in the range [1 … 50000].
- Allowed working time for your program: 0.1 seconds. Allowed memory: 16 MB.

Examples

Input
23 9 3 5 11 0 0 0 12 12 0 0 0 12 12 70 71 0 90 123 140 150 166 190 0 5 13 90 1 70 75 7 188 12
Output
1 13 15 0 13 15 2 21 3
Comments
5 goes to index 1 – between 3 and 5 13 goes to index 13 – 12 and 70 90 goes to index 15 – between 71 and 0 1 goes to index 0 – before 3 Etc.
Input
11 4 2 0 0 0 0 0 0 0 0 0 3 4 3 2 1
Output
11 1 0 0