Exercises: Sorting and Searching Algorithms

You can check your solutions here: https://judge.softuni.bg/Contests/3188/Sorting-Searching-Algorithms.

Part I – Sorting

1. Sorting

Implement one or many of the following **sorting algorithms**:

- **`Insertion sort`**
- **`Bubble sort`**
- **`Shell sort`**

Read a sequence of numbers from the console, **sort** them and **print** them back.

Examples

Input	Output
`5 4 3 2 1`	`1 2 3 4 5`
`1 4 2 -1 0`	`-1 0 1 2 4`

2. Merge Sort

Sort an array of integers using the famous **Merge sort**.

Examples

Input	Output
`5 4 3 2 1`	`1 2 3 4 5`

Hints

Create your **`Mergesort`** class with a single Sort method

```
public class Mergesort
{
    0 references
    public static void Sort(int[] arr)
    {
    }
}
```

Create an **auxiliary array** that will help with merging subarrays

```
private static int[] auxArr;
```

Now to implement the **`Merge()`** method

```
private static void Merge(int[] arr, int leftIndex, int midIndex, int rightIndex)
```

As the two subarrays are sorted, if the **largest element in the left** is smaller than the **smallest in the right**, the two subarrays are **already merged**

```
if (IsLess(arr[midIndex], arr[midIndex + 1]))
{
    return;
}
```

If they are not, however, **transfer all elements to the auxiliary array**

```
for (int index = leftIndex; index < rightIndex + 1; index++)
{
    auxArr[index] = arr[index];
}
```

Then **merge them back** in the main array

```
int i = leftIndex;
int j = midIndex + 1;

for (int currIndex = leftIndex; currIndex <= rightIndex; currIndex++)
{
    if ([redacted])
    {
        arr[currIndex] = auxArr[j++];
    }
    else if ([redacted])
    {
        arr[currIndex] = auxArr[i++];
    }
    else if (IsLess([redacted]))
    {
        arr[currIndex] = auxArr[i++];
    }
    else
    {
        arr[currIndex] = auxArr[j++];
    }
}
```

Now to create the recursive **`Sort()`** method

```
private static void Sort(int[] arr, int leftIndex, int rightIndex)
```

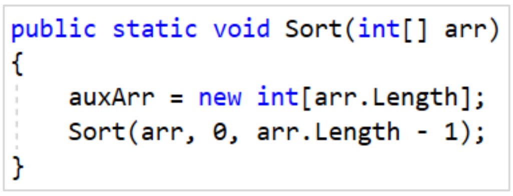
If there is **only one element** in the subarray, it is **already sorted**

```
if (leftIndex >= rightIndex)
{
    return;
}
```

If not, however, you need to **split it into two subarrays**, **sort them recursively** and then **merge them on the way up** of the recursion (as a post-action)

```
Sort(arr, leftIndex, midIndex);
Sort(arr, midIndex + 1, rightIndex);
Merge(arr, leftIndex, midIndex, rightIndex);
```

You can now call the **`Sort()`** method



```
public static void Sort(int[] arr)
{
    auxArr = new int[arr.Length];
    Sort(arr, 0, arr.Length - 1);
}
```

3. Quicksort

Sort an array of integers using the famous Quicksort algorithm.

Examples

Input	Output
5 4 3 2 1	1 2 3 4 5

Hints

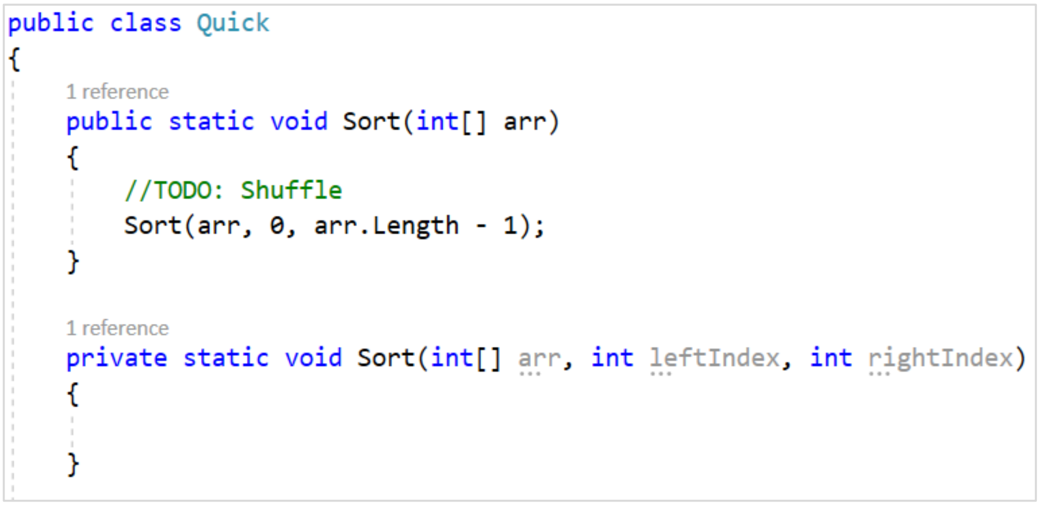
You can learn about the **`Quicksort`** algorithm from Wikipedia.

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
- We choose the **pivot**
  - We pick the first element from the unsorted partition and move it in such a way that all smaller elements are on its left and all greater, to its right
- With pivot moved to its correct place, we now have two unsorted partitions – one to the left of it and one to the right
- **Call the procedure recursively** for each partition
- The bottom of the recursion is when a partition has a size of 1, which is by definition sorted

First, define the **class** and its **sorting method**:

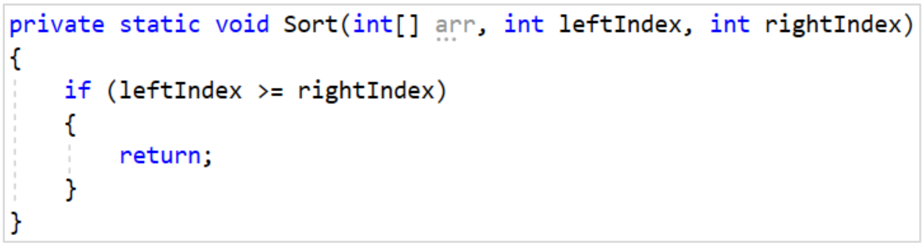


```
public class Quick
{
    1 reference
    public static void Sort(int[] arr)
    {
        //TODO: Shuffle
        Sort(arr, 0, arr.Length - 1);
    }

    1 reference
    private static void Sort(int[] arr, int leftIndex, int rightIndex)
    {

    }
```

Now to implement the private **`Sort()`** method. Don't forget to handle the **bottom of the recursion**



```
private static void Sort(int[] arr, int leftIndex, int rightIndex)
{
    if (leftIndex >= rightIndex)
    {
        return;
    }
}
```

First, find the pivot index and rearange the elements, then sort the left and right partitions recursively:

```
int pivot = Partition(arr, leftIndex, rightIndex);
Sort(arr, leftIndex, pivot - 1);
Sort(arr, pivot + 1, rightIndex);
```

Now to choose the pivot point.. we need to create a method called **`Partition()`**

```
private static int Partition(int[] arr, int leftIndex, int rightIndex)
```

If there is **only one element**, it is already partitioned and the index of the pivot is the index of its only element

```
if (leftIndex >= rightIndex)
{
    return leftIndex;
}
```

Finding the pivot point involves **rearanging all elements** in the partition so it satisfies the condition **all elements to the reft of the pivot to be smaller** from it, and **all elements to its right to be greater** than it

```
int i = leftIndex;
int j = rightIndex + 1;
while (true)
{
    while (Less(arr[++i], arr[leftIndex]))
    {
        if (i == rightIndex) break;
    }

    while (Less(arr[leftIndex], arr[--j]))
    {
        if (j == leftIndex) break;
    }

    if (i >= j) break;

    Swap(arr, i, j);
}

Swap(arr, leftIndex, j);
return j;
```

4. Inversion Count

Assume an **inversion count** is how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order then inversion count is at its maximum.

Two elements **`a[i]`** and **`a[j]`** form an inversion if **`a[i] > a[j]`** and **`i < j`**.

Find and **print the count of all inversions** in a given input array.

Examples

Input	Output	Inversions

2 4 1 3 5	3	2 1
		4 1
		4 3

`5 4 3 2 1`	`10`	`5 4`
		`5 3`
		`5 2`
		`5 1`
		`4 3`
		`4 2`
		`4 1`
		`3 2`
		`3 1`

Hints

- Use a modified version of **Merge sort**.
- Useful read: http://www.geeksforgeeks.org/counting-inversions.

5. Sort Integers by Name

You are given an array of integer numbers which you need to rearrange by their **name** in the English language. For example, the integers **0, 1, 2, 3, 4, 5, 6, 7, 8, 9** must be ordered as **8, 5, 4, 9, 1, 7, 6, 3, 2, 0** (eight, five, four, nine, one, seven, six, three, two, zero, i.e. sorted alphabetically).

Integers larger than 10 are represented in a simplified way, for example **88** is '**eight-eight**' and **1234** is '**one-two-three-four'.** That means that **88** comes before **85.** If the name of one integer starts with the name of another integer, such as in **11** (**one-one**) and **111** (**one-one-one**), the smaller integer comes first.

There are no negative integers in the input.

Input

- The input is on a single line – the integers to be rearranged, separated by a **comma** and **space**.

Output

- On the only output line, print the **rearranged integers**, in format {**n1, n2, n3 … n**}

Constraints

- The input numbers are positive signed integers
- There are no more than 50 integers in the input
- Allowed time/memory: 100 ms / 16 MB

Input	Output
`0, 1, 2, 3, 4, 5, 6, 7, 8, 9`	`8, 5, 4, 9, 1, 7, 6, 3, 2, 0`
`1111, 1, 111, 11`	`1, 11, 111, 1111`
`17, 32, 45, 88, 44`	`88, 45, 44, 17, 32`

Hints

- Create a function **`Num2Text(int)`**, which transforms number to text, as described above.
- You can use a **`Dictionary<string, string>`** to keep **numbers** and their corresponding **texts**. Sort the result with the **`.OrderBy()`** method.

6. Part II – Searching

7. Implement Binary Search

Implement an algorithm that finds the index of an element in a sorted array of integers in logarithmic time.

Examples

Input	Output	Comments

`1 2 3 4 5`
`1`
	`0`	`Index of 1 is 0`

`-1 0 1 2 4`
`1`
	`2`	`Index of 1 is 2`

`1 2 3 4 5`
`6`
	`-1`	`6 is not present in the array`

Hints

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:

- 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**;
- The element we're looking for is larger – look to the right of the current element;
- The element is **not present**, traditionally, return **-1** in that case.

Start by defining a **class** with a **method**:

```
public class BinarySearch
{
    public static int IndexOf(int[] arr, int key)
    {
    }
}
```

Inside the method, define two variables defining the **bounds** to be searched and a while loop:

```
int leftIndex = 0;
int rightIndex = arr.Length - 1;

while (leftIndex <= rightIndex)
{
    //TODO: Find index of key
}

return -1;
```

Inside the while loop, we need to find the **midpoint:**

```
int midIndex = leftIndex + (rightIndex - leftIndex) / 2;
```

If the **key** is to the left of the midpoint, move the right bound. If the key is to the right of the midpoint, move the left bound:

```
if (key < arr[midIndex])
{
    rightIndex = midIndex - 1;
}
else if (key > arr[midIndex])
{
    leftIndex = midIndex + 1;
}
else
{
    return midIndex;
}
```

That's it! Good job!

8. Searching

Implement one or many of the following sorting algorithms:

- `Linear Search`
- `Fibonacci Search`

Read a sequence of **numbers** on the first line and a **single number** on the second from the console. Find the **index** of the number in the given array. Return **`-1`** if the element is not present in the array.

Examples

Input	Output
1 2 3 4 5 1	0
1 2 3 4 5 6	-1

9. * Needles

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`

Part III – Shuffling

10. Shuffle Words

Read **words** from the console, separated by a **space**. Use the **`Fisher-Yates Shuffle`** algorithm to **shuffle** words. Print shuffled words, each on a **new line**.

Examples

Note that this is only sample output, as result after shuffling is always **different** because of the **`Random()`** class.

Input	Sample Output

`first second third fourth fifth`	`second`
	`fifth`
	`third`
	`fourth`
	`first`