Lab: Searching, Sorting and Greedy Algorithms

This document defines the lab for "Algorithms – Fundamentals (C#)" course @ Software University.

Please submit your solutions (source code) of all below-described problems in Judge.

1. 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.

Hints

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

In short, if we have a **sorted collection** of comparable elements, instead of doing a 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 method:

```
public static int BinarySearch(int[] arr, int key)
{
    return -1;
}
```

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

```
public static int BinarySearch(int[] arr, int key)
{
    var left = 0;
    var right:int = arr.Length - 1;

    while (left <= right)
    {
        // TODO
    }

    return -1;
}
```

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

```
var mid:int = (left + right) / 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:

```
var element:int = arr[mid];

if (element == key)
{
    return mid;
}

if (key > element)
{
    left = mid + 1;
}
else
{
    right = mid - 1;
}
```

2. Selection Sort

Write an implementation of **`Selection Sort`**. You should read an array of integers and sort them.

Output

- You should print out the sorted list in the format described below.

Examples

Input	Output
5 4 3 2 1	1 2 3 4 5
13 93 37 74 61 65 5 55 17 96 52 70 17 7 89 65 16 38 42 15 86 21 93 10 31 28 36 14 65	0 1 5 7 7 9 10 12 12 13 14 15 15 16 17 17 18 19 20 20 21 21 21 22 24 25

7 68 86 97 34 27 32 86 44 51 75 29 64 0 36 33 54 20 40 60 56 51 51 25 77 75 46 47 57 18 12 27 28 29 21 22 37 74 78 34 15 71 75 20 19 76 48 98 36 76 49 83 21 44 12 85 68 24 9 80 41 66 1 54 31 55 33 88 35 32 43	27 27 28 28 29 29 31 31 32 32 33 33 34 34 35 36 36 36 37 37 38 40 41 42 43 44 44 46 47 48 49 51 51 51 52 54 54 55 55 56 57 60 61 64 65 65 65 66 68 68 70 71 74 74 75 75 75 76 76 77 78 80 83 85 86 86 86 88 89 93 93 96 97 98

3. Bubble Sort

Write an implementation of **`Bubble Sort`**. You should read an array of integers and sort them.

Output

- You should print out the sorted list in the format described below.

Examples

Input	Output
5 4 3 2 1	1 2 3 4 5
13 93 37 74 61 65 5 55 17 96 52 70 17 7 89 65 16 38 42 15 86 21 93 10 31 28 36 14 65 7 68 86 97 34 27 32 86 44 51 75 29 64 0 36 33 54 20 40 60 56 51 51 25 77 75 46 47 57 18 12 27 28 29 21 22 37 74 78 34 15 71 75 20 19 76 48 98 36 76 49 83 21 44 12 85 68 24 9 80 41 66 1 54 31 55 33 88 35 32 43	0 1 5 7 7 9 10 12 12 13 14 15 15 16 17 17 18 19 20 20 21 21 21 22 24 25 27 27 28 28 29 29 31 31 32 32 33 33 34 34 35 36 36 36 37 37 38 40 41 42 43 44 44 46 47 48 49 51 51 51 52 54 54 55 55 56 57 60 61 64 65 65 65 66 68 68 70 71 74 74 75 75 75 76 76 77 78 80 83 85 86 86 86 88 89 93 93 96 97 98

4. Insertion Sort

Write an implementation of **`Insertion Sort`**. You should read an array of integers and sort them.

Output

- You should print out the sorted list in the format described below.

Examples

Input	Output
5 4 3 2 1	1 2 3 4 5
13 93 37 74 61 65 5 55 17 96 52 70 17 7 89 65 16 38 42 15 86 21 93 10 31 28 36 14 65 7 68 86 97 34 27 32 86 44 51 75 29 64 0 36 33 54 20 40 60 56 51 51 25 77 75 46 47 57 18 12 27 28 29 21 22 37 74 78 34 15 71 75 20 19 76 48 98 36 76 49 83 21 44 12 85 68 24 9 80 41 66 1 54 31 55 33 88 35 32 43	0 1 5 7 7 9 10 12 12 13 14 15 15 16 17 17 18 19 20 20 21 21 21 22 24 25 27 27 28 28 29 29 31 31 32 32 33 33 34 34 35 36 36 36 37 37 38 40 41 42 43 44 44 46 47 48 49 51 51 51 52 54 54 55 55 56 57 60 61 64 65 65 65 66 68 68 70 71 74 74 75 75 75 76 76 77

	78 80 83 85 86 86 86 88 89 93 93 96 97 98

5. Quicksort

Sort an array of elements using the famous quicksort.

Examples

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

6. Merge Sort

Sort an array of elements using the famous merge sort.

Examples

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

7. Sum of Coins

Write a program, which receives a **set of coins** and a **target sum**. The goal is to **reach the sum using as few coins as possible**. You should use a **greedy approach**.

Constraints

- We'll assume that each coin value and the desired sum are **integers**.

Output

- If the target sum can be reached:
  - First, print the number of used coins in the following format: "**`Number of coins to take: {coins}`**".
  - For each used coin print its value and how many times has been used in the following format: **`"{counter} coin(s) with value {coinValue}"`**.
- Otherwise, print "**`Error`**".

Examples

Input	Output	Comments

1, 2, 5, 10, 20, 50 923	Number of coins to take: 21 18 coin(s) with value 50 1 coin(s) with value 20 1 coin(s) with value 2 1 coin(s) with value 1	18*50 + 1*20 + 1*2 + 1*1 = 900 + 20 + 2 + 1 = 923
3, 7 11	Error	Cannot reach desired sum with these coin values.

Greedy Approach

For this problem, a greedy algorithm will attempt to take the best possible coin value (which is the largest), then take the next largest coin value, and so on, until the sum is reached or there are no coin values left. There may be a different amount of coins to take for each value

Greedy Algorithm Implementation

Since at each step we'll try to take the largest value we haven't yet tried, it would simplify our work to order the coin values in descending order.

We can use LINQ:

```
var sortedCoins = coins
    .OrderByDescending(c => c)
    .ToList();
```

Now taking the largest coin value at each step is simply a matter of iterating the list.

We'll need several variables to keep track of:

- We'll be iterating a list, so we also need to know where we're at – an index variable.
- We will need a variable to keep track count of used coins.

```
var counter = 0;
var coinsIndex = 0;
```

Having these variables, when do we stop taking coins?

There are two possibilities:

1) We reached the desired sum.
2) We ran out of coin values.

We can put these two conditions in a while loop like this:

```
while (target > 0 && coinsIndex < sortedCoins.Count)
{
}
```

In the while loop, we need to decide how many coins to take of the current value. We take the current value from the list, we have its index, and move the index to the next one:

```
var currentCoin:int = sortedCoins[coinsIndex++];
```

So, how many coins do we take? Using integer division, we can just divide **target** over the **currentCoin** to find out:

```
var coinsCount:int = target / currentCoin;
```

If the result of the division is greater than zero then we have to increase the count of used coins and reduce the target by the result of **currentCoin * coinsCount**:

```
if (coinsCount > 0)
{
    counter += coinsCount;
    target -= currentCoin * coinsCount;
}
```

Finally, the last step is to take care of output, but this is something that you can do on your own.

8. Set Cover

Write a program that finds **the smallest subset of sets,** which **contains all elements** from a given **sequence**.

You are given two sets - a set of sets (we'll call it **sets**) and a **universe** (a sequence). The **sets** contain only elements from the **universe**, however, some elements are repeated. Your task is to find the smallest subset of **sets** that contains all elements in-**universe**.

Examples

Input	Output
1, 2, 3, 4, 5 4 1 2, 4 5 3	Sets to take (4): 2, 4 1 5 3
1, 3, 5, 7, 9, 11, 20, 30, 40 6 20 1, 5, 20, 30 3, 7, 20, 30, 40 9, 30 11, 20, 30, 40 3, 7, 40	Sets to take (4): 3, 7, 20, 30, 40 1, 5, 20, 30 9, 30 11, 20, 30, 40

Greedy Approach

Using the greedy approach, at each step, we'll take the set which contains the most elements present in the universe which we haven't yet taken. At the first step, we'll always take the set with the largest number of elements, but it gets

a bit more complicated afterward. To simplify our job (and not check against two sets at the same time), when taking a set, we can remove all elements in it from the universe. We can also remove the set from the sets we're considering.

Greedy Algorithm Implementation

First, initialize the resulting list:

```
var selectedSets = new List<int[]>();
```

As discussed in the previous section, we'll be removing elements from the universe, so we'll be repeating the next steps until the universe is empty:

```
while (universe.Count > 0)
{
}
```

The hardest part is selecting a set. We need to get the set that has the most elements contained in the universe. We can use LINQ to sort the sets and then take the first set (the one with the most elements in the universe):

```
var currentSet:int[] = sets// List<int[]>
    .OrderByDescending(
        keySelector:s:int[] => s.Count(
            predicate:e:int => universe.Contains(e)))// IOrderedEnumerable<int[]>
    .FirstOrDefault();
```

Sorting the sets at each step is probably not the most efficient approach, but it's simple enough to understand. The above LINQ query tests each element in a set to see if it is contained in the universe and sorts the sets (in descending order, from largest to smallest) based on the number of elements in each set that are in the universe.

Once we have the set we're looking for, the next steps are trivial. Complete the TODOs below:

```
// TODO: Add current set to selected sets
// TODO: Remove current set from sets
// TODO: Remove all elements in current set from the universe
```

After implementing TODOs, you should be done with this problem.