Basic Algorithms

Problems for exercises and homework for the ["C# Advanced" course @ Software University]()

You can check your solutions in [Judge]()

I. Recursion

1. Recursive Array Sum

Create a program that sums all elements in an array. Use **recursion**.

Note: In practice, recursion should not be used here (instead use an **iterative solution**), this is just an exercise.

Examples

Input	Output
1 2 3 4	10
-1 0 1	0

Hints

Write a **recursive** method. It will take as arguments the **input array** and the **current index**.

- The method should return the **current element** + the **sum of all the next elements.**
- The recursion should stop when there are no more elements in the array and 0 should be returned.

```
private static int Sum(int[] array, int index)
{
    // TODO: Set bottom of recursion
    // TODO: Return the sum of current element + sum of elements to the right
}
```

2. Recursive Factorial

Create a program that finds the factorial of a given number. Use **recursion**.

Note: In practice, recursion should not be used here. Instead, you should use an **iterative solution.** This type of solution is for exercise purposes.

Examples

Input	Output
5	120
10	3628800

Hints

Write a **recursive** method. It will take as arguments an integer number.

- The method should return the **current element** * the **result of calculating the factorial of current element - 1** (obtained by recursively calling it).
- The recursion should stop when the current element is equal to zero and 1 should be returned.

```
static long Factorial(int n)
{
    // TODO: Set bottom of recursion
    // TODO: Return the multiple of current n and factorial of n - 1
}
```

II. Greedy Algorithms

1. Sum of Coins*

Create a program, which gathers a sum of money, using the least possible number of coins. The **range of possible coin values** is **1, 2, 5, 10, 20, 50.**

The goal is to **reach the** desired sum **using as few coins as possible. You can solve the task by using a greedy approach**.

There is a skeleton, which you can download from Judge. Use the **SumOfCoins** project.

Input

As input, you will receive two lines:

- An array of integers separated by space and comma **(", ")** – coins which should be used.
- A single integer – the desired sum.

Output

- As an output on the first line print

  "Number of coins to take: {coins}"

- On the next n lines print how many coins were used, with their value

  "{numberOfCoins} coin(s) with value {valueOfCoin}"

- If you cannot reach the desire sum, throw "**InvalidOperationException()**".

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
1 42	Number of coins to take: 42 42 coin(s) with value 1	

`3, 7`
`11`
	`Error`	Cannot reach the desired sum with these coin values

`1, 2, 5`
`2031154123`
	`Number of coins to take: 406230826`
	`406230824 coin(s) with value 5`
	`1 coin(s) with value 2`
	`1 coin(s) with value 1`
		The solution should be fast enough to handle a combination of small coin values and a large desired sum

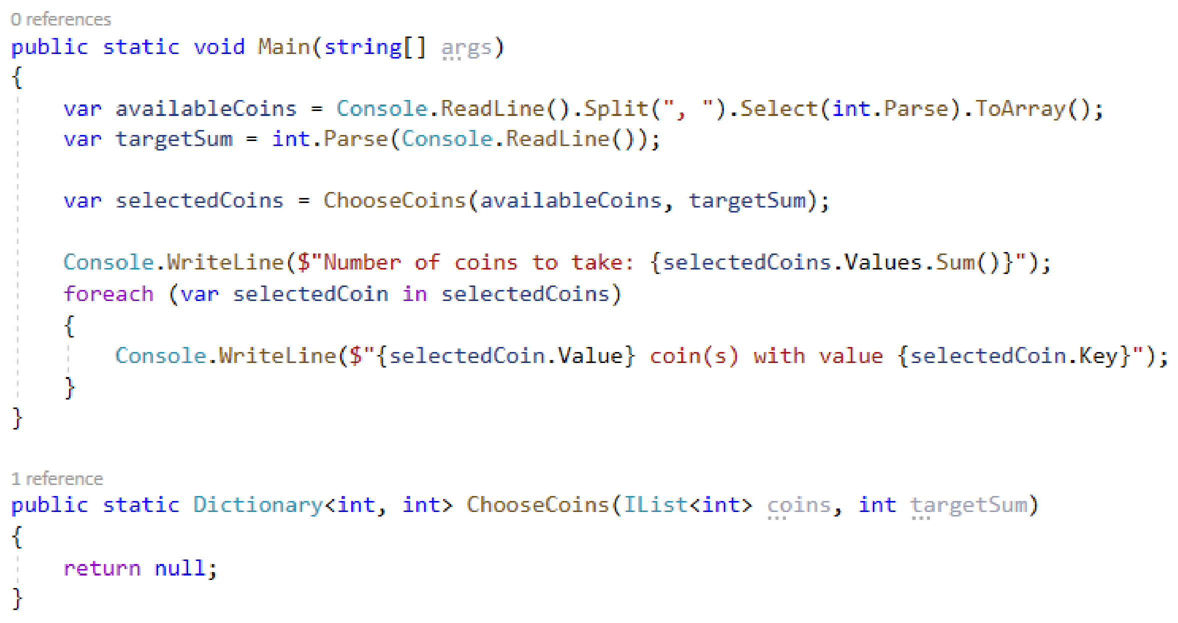
`1, 9, 10`
`27`
	`Number of coins to take: 9`
	`2 coin(s) with value 10`
	`7 coin(s) with value 1`
		The greedy approach produces a non-optimal solution (9 coins to take instead of 3 with a value of 9)

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 number of coins to take for each value. In one of the examples above, we had a very large, desired sum and relatively small coin values, which means we'll need to take a lot of coins. It would not be efficient (and may even cause an `Exception`), if we return the result as a **`List<int>`.** A more practical way to do it is to use a **`Dictionary<int, int>`**, where the keys are the coin values and the values are the number of coins to take for the specified coin value. Therefore, in the second example (coin values = { 1 }, sum = 42), instead of returning a list with 42 elements in it, we'll return a dictionary with a single key-value pair: 1 => 42.

Greedy Algorithm Implementation

You are given an implemented **`Main()`** method with sample data. Your task is to implement the **`ChooseCoins()`** method:



```
0 references
public static void Main(string[] args)
{
    var availableCoins = Console.ReadLine().Split(", ").Select(int.Parse).ToArray();
    var targetSum = int.Parse(Console.ReadLine());

    var selectedCoins = ChooseCoins(availableCoins, targetSum);

    Console.WriteLine($"Number of coins to take: {selectedCoins.Values.Sum()}");
    foreach (var selectedCoin in selectedCoins)
    {
        Console.WriteLine($"{selectedCoin.Value} coin(s) with value {selectedCoin.Key}");
    }
}

1 reference
public static Dictionary<int, int> ChooseCoins(IList<int> coins, int targetSum)
{
    return null;
}
```

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:

```
int[] sortedCoins = coins.OrderByDescending(coin => coin).ToArray();
```

Now, taking the largest coin value at each step is simply a matter of iterating the list. We'll need several variables:

- A resulting dictionary
- An index variable
- A variable for the current sum

Since it's possible to finish the algorithm without reaching the desired sum, we'll keep track of the current amount taken in a separate variable (when we're done, we'll check it against the desired sum to see if we got a solution or not).

```
Dictionary<int, int> chosenCoins = new Dictionary<int, int>();

int currentSum = 0;
int coinIndex = 0;
```

Having these variables, when do we stop taking coins? There are two possibilities:

- We have reached the desired sum
- We ran out of coin values

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

```
while (currentSum != targetSum && coinIndex < sortedCoins.Length)
{
    // TODO
}
```

Inside the body of 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:

```
int currentCoinValue = sortedCoins[coinIndex];
```

So far, we've accumulated some amount in the **`currentSum`** variable, the difference between **`targetSum`** and **`currentSum`** will give us the remaining sum we need to obtain:

```
int remainder = targetSum - currentSum;
```

So, how many coins do we take? Using integer division, we can just divide **`remainder`** over the current coin value to find out:

```
int numberOfCoins = remainder / currentCoin;
```

All we must do now is put this information in the resulting dictionary as a key-value pair (only if we can take coins with this value), then increment the current index to move on to the next coin value:

```
if (currentSum + currentCoinValue <= targetSum)
{
    // TODO: Add Information to chosenCoins dictionary (coin value => number of coins)
    // TODO: Increase currentSum with total value of coins
}

coinIndex++;
```

Outside the while loop we also should check if we can reach the desired sum with the given coins:

```
if (currentSum != targetSum)
{
    throw new InvalidOperationException();
}
```

Finally, return the resulting dictionary.

2. Set Cover*

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

In the Set Cover Problem, we are given two sets - a set of sets (we'll call it **`sets`**) and a **`universe` (a sequence)**. The **`sets`** contain all elements from the **`universe`** and no others; however, some elements are repeated. The task is to **find the smallest subset of `sets` that contains all elements in the `universe`.** Use the **`SetCover`** project from your skeleton.

Input

The input is consist of three lines:

- **Universe** - an array of integers separated by space and comma **`(", ")`** .
- **Numbers of sets** – a single integer representing the numbers of rows of the array.
- **Multidimensional (jagged)** array of integers separated by space and comma **`(", ").`**

Output

- As an output on the first line print the number of sets:

  `" Sets to take ({number of sets}):"`

- On the next n lines print actual sets in the following format:

  ```
  "{ {number1}, {number2},… }
   { {number1}, {number2},… }
   …"
  ```

Examples

Input	Output

`1, 2, 3, 4, 5`
`4`
`1`
`2, 4`
`5`
`3`
	`Sets to take (4):`
	`{ 2, 4 }`
	`{ 1 }`
	`{ 5 }`
	`{ 3 }`

`1, 2, 3, 4, 5`	`Sets to take (1):`

4 1, 2, 3, 4, 5 2, 3, 4, 5 5 3	{ 1, 2, 3, 4, 5 }

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. This is the reason for calling **`ToList()`** on both the sets and universe when calling the **`ChooseSets()`** method inside the **`Main()`** method.

Greedy Algorithm Implementation

You are given sample input in the **`Main()`** method, your task is to complete the **`ChooseSets()`** method:

```
static void Main(string[] args)
{
    int[] universe = Console.ReadLine().Split(", ").Select(int.Parse).ToArray();

    int numberOfSets = int.Parse(Console.ReadLine());
    int[][] sets = new int [numberOfSets][];

    for (int row = 0; row < sets.Length; row++)
    {
        int[] rowsValue = Console.ReadLine().Split(", ").Select(int.Parse).ToArray();
        sets[row] = new int[rowsValue.Length];

        for (int col = 0; col < sets[row].Length; col++)
        {
            sets[row][col] = rowsValue[col];
        }
    }

    List<int[]> selectedSets = ChooseSets(sets.ToList(), universe.ToList());
    Console.WriteLine($"Sets to take ({selectedSets.Count}):");

    foreach (int[] set in selectedSets)
    {
        Console.WriteLine($"{{ {string.Join(", ", set)} }}");
    }
}
```

The method will return a list of arrays, so first thing's first, initialize the resulting list:

```
List<int[]> 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)
{
    // TODO
}

return selectedSets;
```

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):

```
int[] longestSet = sets
    .OrderByDescending(s => s.Count(x => universe.Contains(x)))
    .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 currentSet to results (selectedSets)
// TODO: Remove currentSet form sets
// TODO: Remove all elements in currentSet from universe
```

This is all, we just need to run the unit tests to make sure we didn't make a mistake along the way.

III. Simple Sorting Algorithms

3. 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

Hints

Create your **Mergesort** generic class with a single **Sort** method:

```
public class Mergesort<T> where T : IComparable
{
    public static void Sort(T[] arr)
    {
    }
}
```

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

```
private static T[] aux;
```

Implement the **Merge()** method:

```
private static void Merge(T[] arr, int lo, int mid, int hi)
```

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[mid], arr[mid + 1]))
{
    return;
}
```

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

```
for (int index = lo; index < hi + 1; index++)
{
    aux[index] = arr[index];
}
```

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

```
int i = lo;
int j = mid + 1;
for (int k = lo; k <= hi; k++)
{
    if ([redacted])
    {
        arr[k] = aux[j++];
    }
    else if ([redacted])
    {
        arr[k] = aux[i++];
    }
    else if (IsLess([redacted]))
    {
        arr[k] = aux[i++];
    }
    else
    {
        arr[k] = aux[j++];
    }
}
```

Now, create the recursive **`Sort()`** method:

```
private static void Sort(T[] arr, int lo, int hi)
```

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

```
if (lo >= hi)
{
    return;
}
```

If not, 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, lo, mid);
Sort(arr, mid + 1, hi);
Merge(arr, lo, mid, hi);
```

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

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

4. Quicksort*

Sort an array of elements using the famous quicksort.

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 their left and all greater, to its right.
- With the 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
{
    public static void Sort<T>(T[] a) where T : IComparable<T>
    {
        // TODO: Shuffle
        Sort(a, 0, a.Length - 1);
    }

    private static void Sort<T>(T[] a, int lo, int hi) where T : IComparable<T>
    {
    }
}
```

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

```
private static void Sort<T>(T[] a, int lo, int hi) where T : IComparable<T>
{
    if (lo >= hi)
    {
        return;
    }
}
```

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

```
int p = Partition(a, lo, hi);
Sort(a, lo, p - 1);
Sort(a, p + 1, hi);
```

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

```
private static int Partition<T>(T[] a, int lo, int hi) where T : IComparable<T>
{
}
```

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

```
if (lo >= hi)
{
    return lo;
}
```

Finding the pivot point involves **rearranging 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 = lo;
int j = hi + 1;
while (true)
{
    while (Less(a[++i], a[lo]))
    {
        if (i == hi) break;
    }

    while (Less(a[lo], a[--j]))
    {
        if (j == lo) break;
    }

    if (i >= j) break;
    Swap(a, i, j);
}

Swap(a, lo, j);
return j;
```

IV. Searching Algorithms

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`	`0`	`Index of 1 is 0`

1		
-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. Here you can find a tool that shows visually how the search is performed.

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 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 lo = 0;
int hi = arr.Length - 1;
while (lo <= hi)
{
    // TODO: Find index of key
}

return -1;
```

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

```
int mid = [redacted];
```

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[mid])
    {
        hi = [redacted];
    }
    else if (key > arr[mid])
    {
        lo = [redacted];
    }
    else
    {
        return mid;
    }
```