# Basic Algorithms

Problems for exercises and homework for the ["C# Advanced" course @ Software University](https://softuni.bg/modules/58/csharp-advanced/1189).  
You can check your solutions here: <https://judge.softuni.bg/Contests/Compete/Index/1560#0>

# Recursion

## Recursive Array Sum

Write a program that finds the sum of all elements in an integer 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 next elements.**
* The recursion should stop when there are no more elements in the array.



## Recursive Factorial

Write 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 factorial of current element - 1** (obtained by recursively calling it).
* The recursion should stop when there are no more elements in the array.



# Greedy Algorithms

## Sum of Coins\*

Write a program, which gathers a sum of money, using the least possible number of coins. This is the **range of possible coin values**:

* **{ 1, 2, 5, 10, 20, 50 }**

You will receive a **desired sum**. The goal is to **reach the sum using as few coins as possible using a greedy approach**. We’ll assume that each coin value and the desired sum are **integers**. There is a skeleton, which you can download from Judge. Use the **SumOfCoins** project.

### Examples

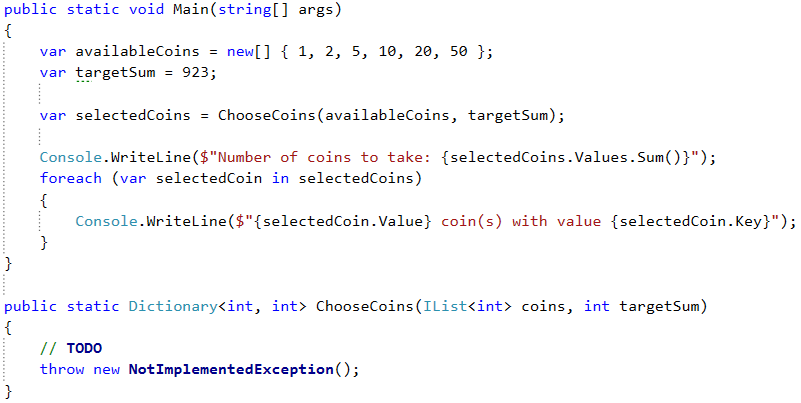
|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| Coins: 1, 2, 5, 10, 20, 50  Sum: 923 | Number of coins to take: 21  18 coin(s) with value 50  1 coin(s) with value 20  1 coin(s) with value 2  1 coin(s) with value 1 | 18\*50 + 1\*20 + 1\*2 + 1\*1 = 900 + 20 + 2 + 1 = 923 |
| Coins: 1  Sum: 42 | Number of coins to take: 42  42 coin(s) with value 1 |  |
| Coins: 3, 7  Sum: 11 | Error | Cannot reach desired sum with these coin values |
| Coins: 1, 2, 5  Sum: 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 | Solution should be fast enough to handle a combination of small coin values and a large desired sum |
| Coins: 1, 9, 10  Sum: 27 | Number of coins to take: 9  2 coin(s) with value 10  7 coin(s) with value 1 | Greedy approach produces non-optimal solution (9 coins to take instead of 3 with value 9) |

### Greedy Approach

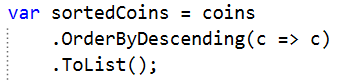
For this problem, a greedy algorithm will attempt to take the best possible coin value (which is the largest), then take the next largest coin value and so on, until the sum is reached or there are no coin values left. There may be a different amount of coins to take for each value. In one of the examples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



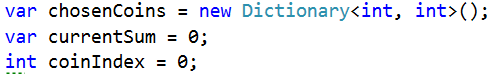
Since at each step we’ll try to take the largest value we haven’t yet tried, it would simplify our work to order the coin values in descending order. We can use LINQ:



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

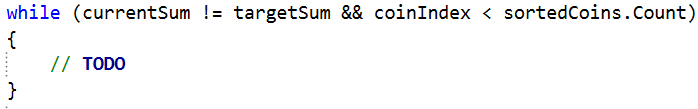
Since it’s possible to finish the algorithm without reaching the desired sum, we’ll keep track of the current amount taken in a separate variable (when we’re done, we’ll check it against the desired sum to see if we got a solution or not).



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:



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:



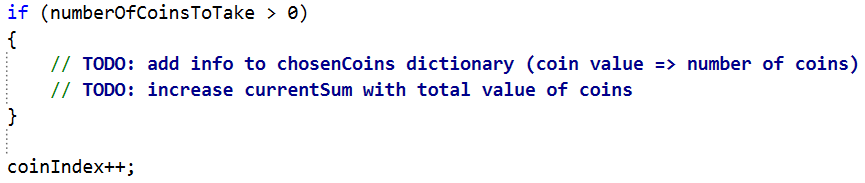
So far, we’ve accumulated some amount in the currentSum variable, the difference between targetSum and currentSum will give us the remaining sum we need to obtain:



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



All we have to 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:



Finally, return the resulting dictionary.

## Set Cover\*

Write 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 universe and no others, however, some elements are repeated. The task is to **find the smallest subset of** sets **which contains all elements in** universe**.** Use the **SetCover** project from your skeleton.

### Examples

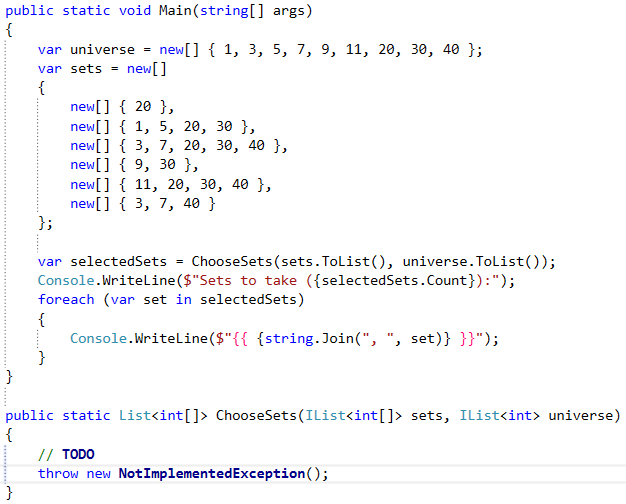
|  |  |
| --- | --- |
| **Input** | **Output** |
| Universe: 1, 2, 3, 4, 5  Number of sets: 4  1  2, 4  5  3 | Sets to take (4):  { 2, 4 }  { 1 }  { 5 }  { 3 } |
| Universe: 1, 2, 3, 4, 5  Number of sets: 4  1, 2, 3, 4, 5  2, 3, 4, 5  5  3 | Sets to take (1):  { 1, 2, 3, 4, 5 } |
| Universe: 1, 3, 5, 7, 9, 11, 20, 30, 40  Number of sets: 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 afterwards. 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

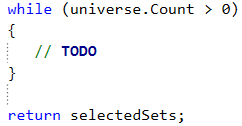
You are given sample input in the Main() method, your task is to complete the ChooseSets() method:



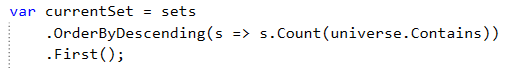
The method will return a list of arrays, so first thing’s first, initialize the resulting list:



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:

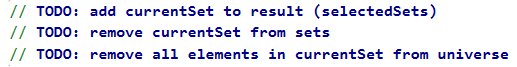


The hardest part is selecting a set. We need to get the set which has the most elements contained in the universe. We can use LINQ to sort the sets and then take the first set (the one with most elements in the universe):



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 which are in the universe.

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



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

# Simple Sorting Algorithms

## 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:



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



Implement the Merge() method:



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



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



Now, create the recursive Sort() method:



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



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):



You can now call the Sort() method:



## Qucksort\*

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](https://en.wikipedia.org/wiki/Quicksort). A great tool for visualizing the algorithm (along with many others) is available at [Visualgo.net](http://visualgo.net/sorting.html).

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**:



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



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



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



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



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:



# Searching Algorothms

## 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](https://en.wikipedia.org/wiki/Binary_search_algorithm). [Here](http://www.dave-reed.com/book/Chapter8/search.html) 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:



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



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



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:

