# rExercises: Greedy Algorithms

This document defines the **in-class exercises** assignments for the ["Algorithms" course @ Software University](https://softuni.bg/opencourses/algorithms).

For the following exercises you are given a Visual Studio solution "**Greedy-Algorithms-Lab**" holding portions of the source code + unit tests. You can download it from the course’s page.

# Part I – Sum of Coins

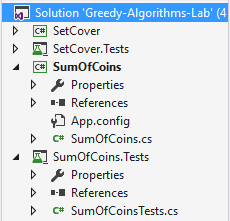
Let’s have a **range of possible coin values** (e.g. 1, 2, 5, 10, 20, 50) and a **desired sum**. In this problem, the goal is to **reach the sum using as few coins as possible using a greedy approach**. We’ll assume that each coin value and the desired sum are **integers**, but we can easily modify the problem to accommodate floating-point values using the decimal type in C#.

Examples:

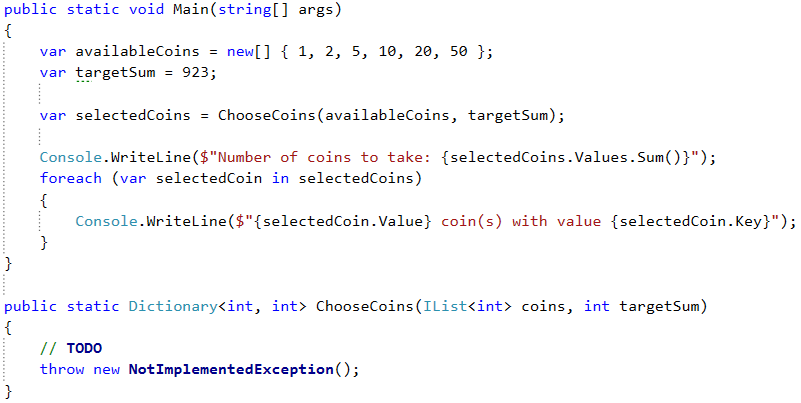
|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| Coins: 1, 2, 5, 10, 20, 50  Sum: 923 | Number of coins to take: 21  18 coin(s) with value 50  1 coin(s) with value 20  1 coin(s) with value 2  1 coin(s) with value 1 | 18\*50 + 1\*20 + 1\*2 + 1\*1 = 900 + 20 + 2 + 1 = 923 |
| 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) |

## Provided Assets

For this problem you are given one project to hold the solution and a unit test project:

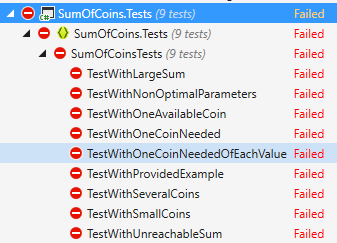


In the **SumOfCoins** project you are given an implemented Main() method with sample data. Your task is to implement the ChooseCoins() method:



## Run the Unit Tests to Make Sure They Fail

The algorithm is not implemented yet, so all unit tests should fail:



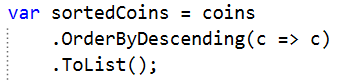
## Greedy Approach to Solution

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>; 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.

## Sort Coin Values

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:

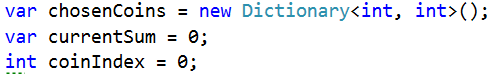


Now taking the largest coin value at each step is simply a matter of iterating the list.

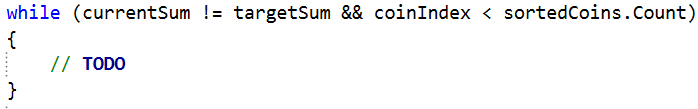
## Greedy Algorithm Implementation

We’ll need several variables to keep track of:

* We’ll need, of course, the resulting dictionary.
* We’ll be iterating a list, so we also need to know where we’re at – an index variable.
* Finally, 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: 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:



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:



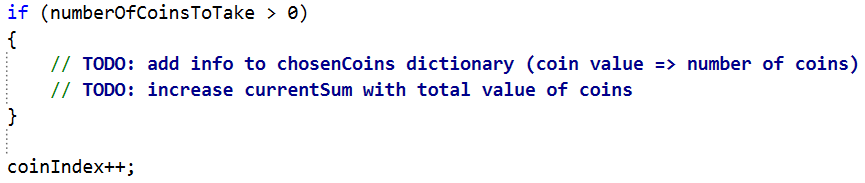
So far, we’ve accumulated some amount in the currentSum variable, the difference between targetSum and currentSum will give us the remaining sum we need to obtain:



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



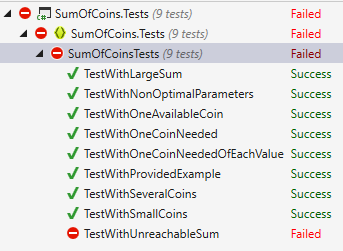
All we have to do now is put this information in the resulting dictionary as a key-value pair (only if we can take coins with this value), then increment the current index to move on to the next coin value:



Finally, return the resulting dictionary.

## Run the Unit Tests

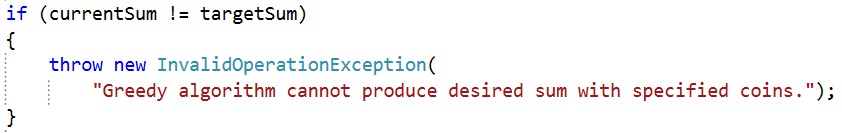
We missed something, so one of the unit tests should not pass:



As you can see, this solution doesn’t work when we have an unreachable sum. We need to throw an error if the algorithm fails to produce a result.

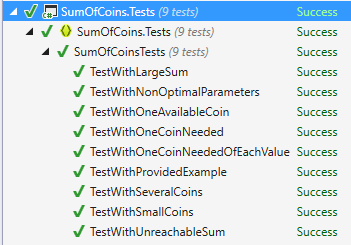
## Check if Sum is Reached

Having the targetSum and currentSum, we just need to compare them. If they’re not equal, we haven’t reached a solution, so an error should be thrown before returning the resulting dictionary:



## Run the Unit Tests One Last Time

When we’ve taken care of situations where the greedy approach fails to produce a result, the unit tests should all pass:



# Part II – Set Cover Problem

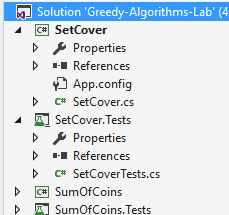
In the Set Cover Problem, we are given two sets - a set of sets (we’ll call it sets) and a universe. The sets contain all elements from universe and no others, however, some elements are repeated. The task is to **find the smallest subset of** sets **which contains all elements in** universe**.**

Examples:

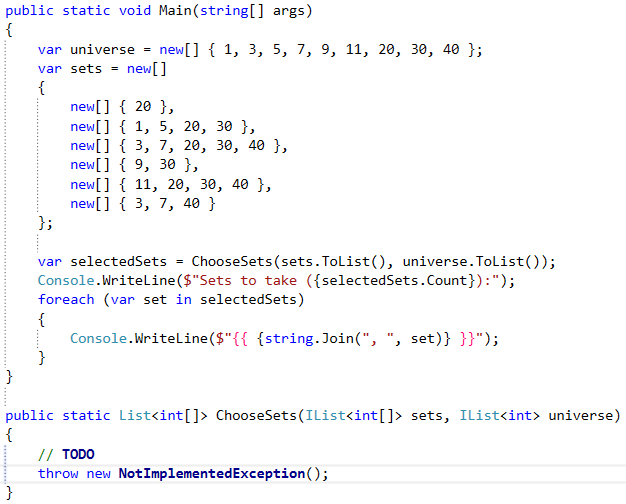
|  |  |
| --- | --- |
| **Input** | **Output** |
| Universe: 1, 2, 3, 4, 5  Number of sets: 4  1  2, 4  5  3 | Sets to take (4):  { 2, 4 }  { 1 }  { 5 }  { 3 } |
| Universe: 1, 2, 3, 4, 5  Number of sets: 4  1, 2, 3, 4, 5  2, 3, 4, 5  5  3 | Sets to take (1):  { 1, 2, 3, 4, 5 } |
| Universe: 1, 3, 5, 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 } |

## Provided Assets

For the set problem, you are given two projects – one to hold the solution and a unit test project:

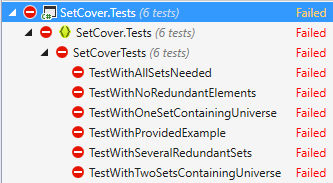


You are given sample input in the Main() method, your task is to complete the ChooseSets() method:



## Run the Unit Tests to Make Sure They Fail

The unit tests should all fail initially:



## Greedy Approach to Solution

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.

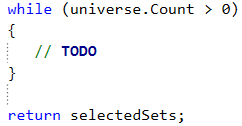
To simplify our job (and not check against two sets at the same time), when taking a set, we can remove all elements in it from the universe. We can also remove the set from the sets we’re considering. This is the reason for calling ToList() on both the sets and universe when calling the ChooseSets() method inside the Main() method.

## Greedy Algorithm Implementation

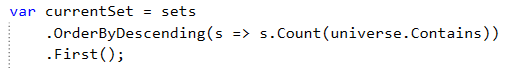
The method will return a list of arrays, so first thing’s first, initialize the resulting list:



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:

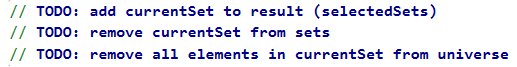


The hardest part is selecting a set. We need to get the set which has the most elements contained in the universe. We can use LINQ to sort the sets and then take the first set (the one with most elements in the universe):



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.

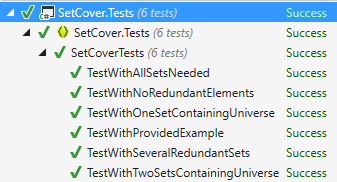
Once we have the set we’re looking for, the next steps are trivial. Complete the TODOs below:



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

## Run the Unit Tests

If you did everything correctly, the unit tests should pass:



Congratulations, you’re done!