Exercises: Greedy Algorithms

This document defines the **in-class exercises** assignments for the "Algorithms" course @ Software University.

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`	

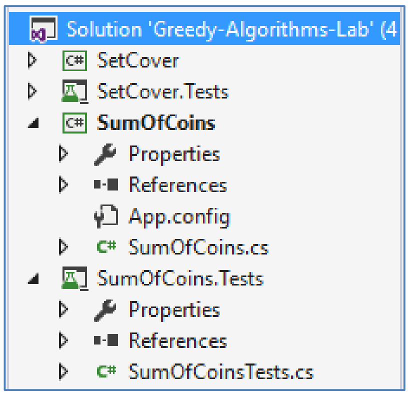
`Coins: 3, 7`
`Sum: 11`
	`Error`	`Cannot reach desired sum with these coin values`

`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)`

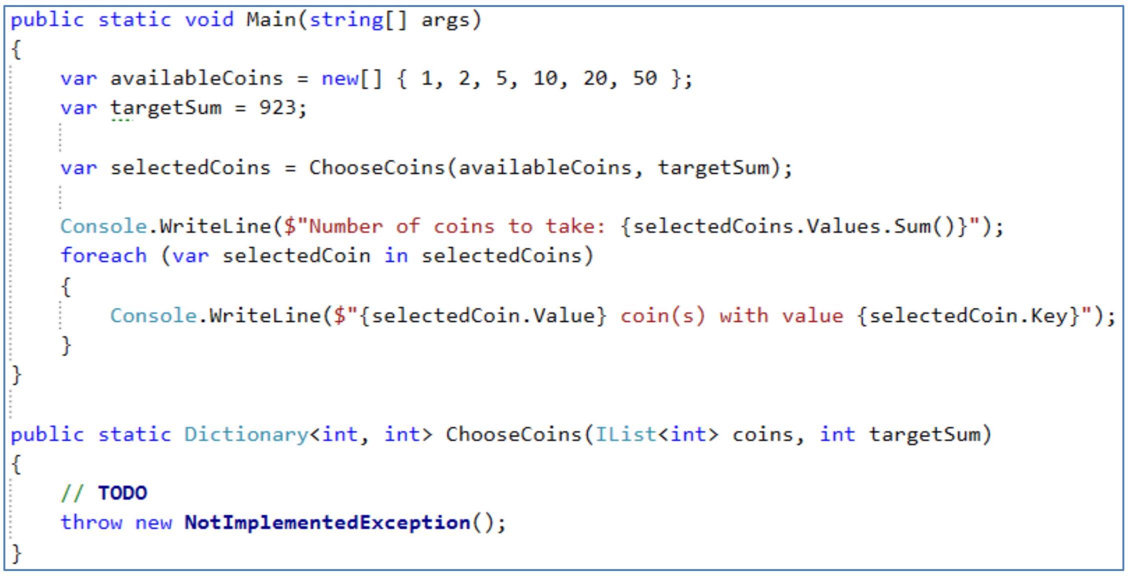
Problem 1. Provided Assets

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



```
Solution 'Greedy-Algorithms-Lab' (4
  SetCover
  SetCover.Tests
  SumOfCoins
    Properties
    References
    App.config
    SumOfCoins.cs
  SumOfCoins.Tests
    Properties
    References
    SumOfCoinsTests.cs
```

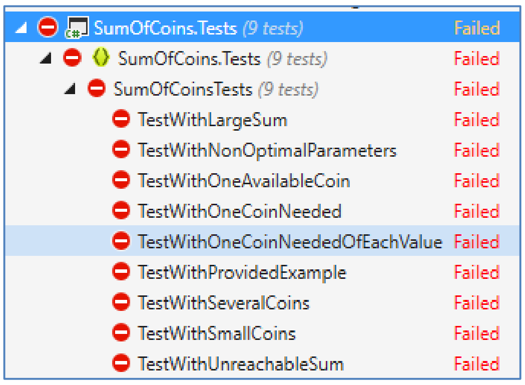
In the **SumOfCoins** project you are given an implemented **`Main()`** method with sample data. Your task is to implement the **`ChooseCoins()`** method:



```
public static void Main(string[] args)
{
    var availableCoins = new[] { 1, 2, 5, 10, 20, 50 };
    var targetSum = 923;

    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}");
    }
}

public static Dictionary<int, int> ChooseCoins(IList<int> coins, int targetSum)
{
    // TODO
    throw new NotImplementedException();
}
```

Problem 2. Run the Unit Tests to Make Sure They Fail

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



```
SumOfCoins.Tests (9 tests)                      Failed
  SumOfCoins.Tests (9 tests)                    Failed
    SumOfCoinsTests (9 tests)                   Failed
      TestWithLargeSum                          Failed
      TestWithNonOptimalParameters              Failed
      TestWithOneAvailableCoin                  Failed
      TestWithOneCoinNeeded                     Failed
      TestWithOneCoinNeededOfEachValue          Failed
      TestWithProvidedExample                   Failed
      TestWithSeveralCoins                      Failed
      TestWithSmallCoins                        Failed
      TestWithUnreachableSum                    Failed
```

Problem 3. 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.

Problem 4. 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:

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

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

Problem 5. 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).

```
var chosenCoins = new Dictionary<int, int>();
var currentSum = 0;
int coinIndex = 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 (currentSum != targetSum && coinIndex < sortedCoins.Count)
{
    // TODO
}
```

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:

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

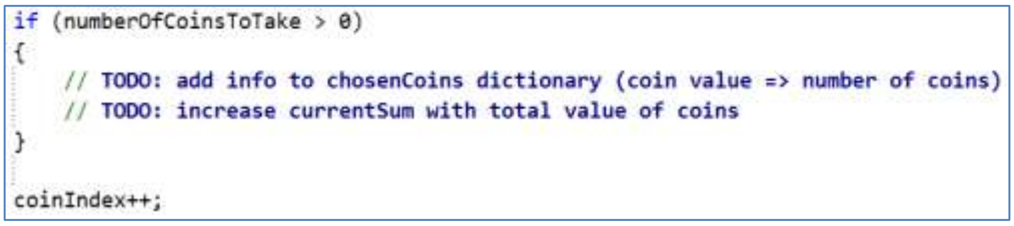
So far, we've accumulated some amount in the **`currentSum`** variable, the difference between **`targetSum`** and **`currentSum`** will give us the remaining sum we need to obtain:

```
var remainingSum = targetSum - currentSum;
```

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

```
var numberOfCoinsToTake = remainingSum / currentCoinValue;
```

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:

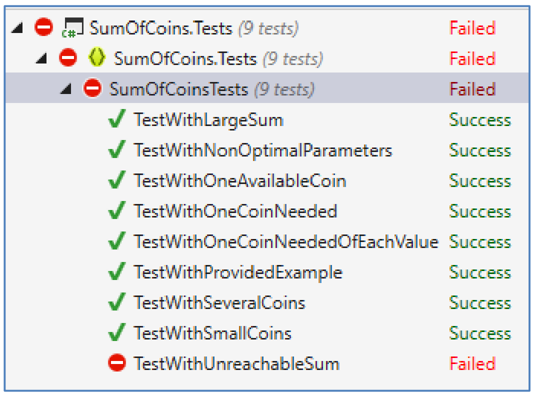


```
if (numberOfCoinsToTake > 0)
{
    // TODO: add info to chosenCoins dictionary (coin value => number of coins)
    // TODO: increase currentSum with total value of coins
}

coinIndex++;
```

Finally, return the resulting dictionary.

Problem 6. Run the Unit Tests

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

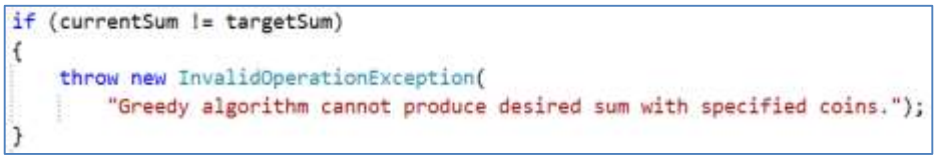


```
◢ ⛔ SumOfCoins.Tests (9 tests)                     Failed
  ◢ ⛔ SumOfCoins.Tests (9 tests)                   Failed
    ◢ ⛔ SumOfCoinsTests (9 tests)                  Failed
        ✓ TestWithLargeSum                          Success
        ✓ TestWithNonOptimalParameters              Success
        ✓ TestWithOneAvailableCoin                  Success
        ✓ TestWithOneCoinNeeded                     Success
        ✓ TestWithOneCoinNeededOfEachValue          Success
        ✓ TestWithProvidedExample                   Success
        ✓ TestWithSeveralCoins                      Success
        ✓ TestWithSmallCoins                        Success
        ⛔ TestWithUnreachableSum                   Failed
```

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.

Problem 7. Check if Sum is Reached

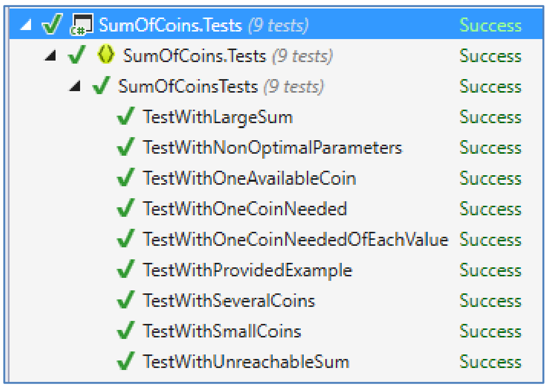
Having the **`targetSum`** and **`currentSum`**, we just need to compare them. If they're not equal, we haven't reached a solution, so an error should be thrown before returning the resulting dictionary:



```
if (currentSum != targetSum)
{
    throw new InvalidOperationException(
        "Greedy algorithm cannot produce desired sum with specified coins.");
}
```

Problem 8. 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:



```
SumOfCoins.Tests (9 tests)                          Success
  SumOfCoins.Tests (9 tests)                        Success
    SumOfCoinsTests (9 tests)                       Success
      TestWithLargeSum                              Success
      TestWithNonOptimalParameters                  Success
      TestWithOneAvailableCoin                      Success
      TestWithOneCoinNeeded                         Success
      TestWithOneCoinNeededOfEachValue              Success
      TestWithProvidedExample                       Success
      TestWithSeveralCoins                          Success
      TestWithSmallCoins                            Success
      TestWithUnreachableSum                        Success
```

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

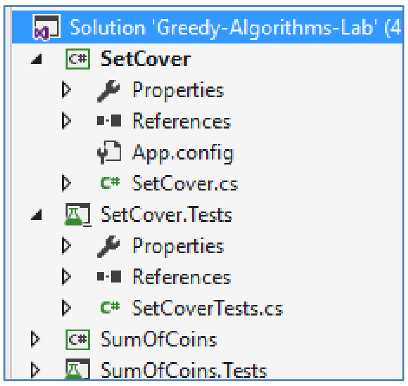
Universe: 1, 2, 3, 4, 5
Number of sets: 4
1
2, 4
5
3
	Sets to take (4):
	{ 2, 4 }
	{ 1 }
	{ 5 }
	{ 3 }

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 }

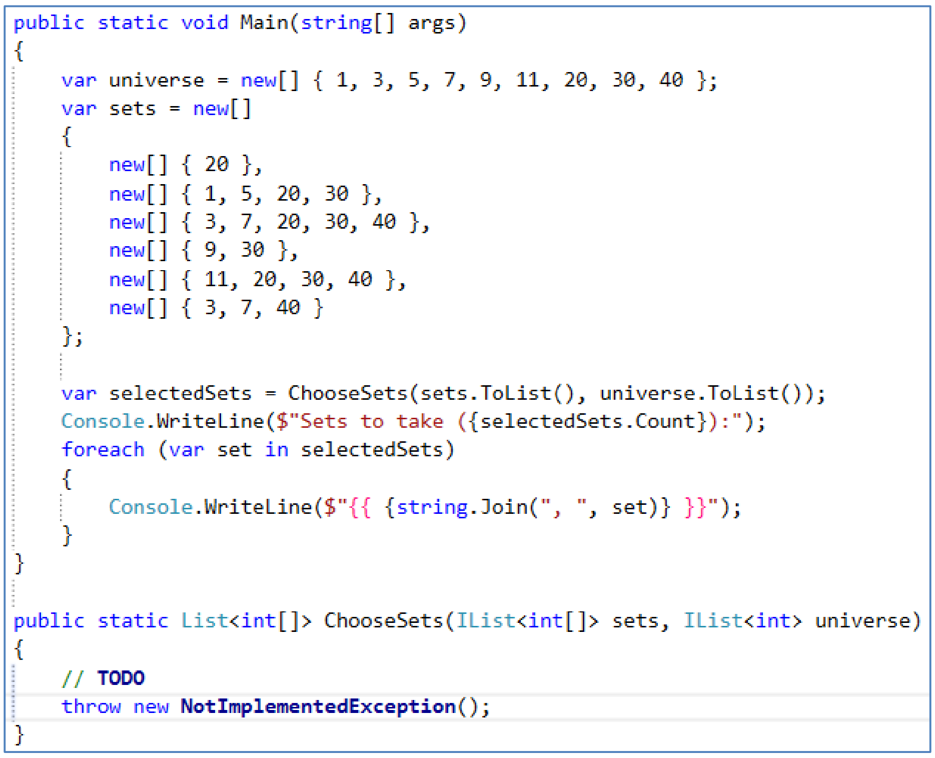
Universe: 1, 3, 5, 7, 9, 11, 20, 30, 40
Number of sets: 6
20
1, 5, 20, 30
3, 7, 20, 30, 40
9, 30
11, 20, 30, 40
3, 7, 40
	Sets to take (4):
	{ 3, 7, 20, 30, 40 }
	{ 1, 5, 20, 30 }
	{ 9, 30 }
	{ 11, 20, 30, 40 }

Problem 9. Provided Assets

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



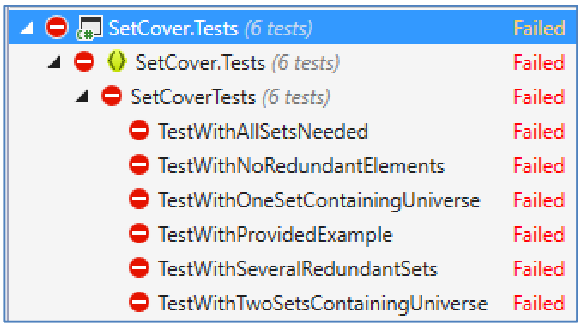
You are given sample input in the **`Main()`** method, your task is to complete the **`ChooseSets()`** method:



```
public static void Main(string[] args)
{
    var universe = new[] { 1, 3, 5, 7, 9, 11, 20, 30, 40 };
    var sets = new[]
    {
        new[] { 20 },
        new[] { 1, 5, 20, 30 },
        new[] { 3, 7, 20, 30, 40 },
        new[] { 9, 30 },
        new[] { 11, 20, 30, 40 },
        new[] { 3, 7, 40 }
    };

    var selectedSets = ChooseSets(sets.ToList(), universe.ToList());
    Console.WriteLine($"Sets to take ({selectedSets.Count}):");
    foreach (var set in selectedSets)
    {
        Console.WriteLine($"{{ {string.Join(", ", set)} }}");
    }
}

public static List<int[]> ChooseSets(IList<int[]> sets, IList<int> universe)
{
    // TODO
    throw new NotImplementedException();
}
```

Problem 10. Run the Unit Tests to Make Sure They Fail

The unit tests should all fail initially:



```
SetCover.Tests (6 tests)                          Failed
  SetCover.Tests (6 tests)                        Failed
    SetCoverTests (6 tests)                       Failed
      TestWithAllSetsNeeded                       Failed
      TestWithNoRedundantElements                 Failed
      TestWithOneSetContainingUniverse            Failed
      TestWithProvidedExample                     Failed
      TestWithSeveralRedundantSets                Failed
      TestWithTwoSetsContainingUniverse           Failed
```

Problem 11. 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.

Problem 12. Greedy Algorithm Implementation

The method will return a list of arrays, so first thing's 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)
{
    // TODO
}

return selectedSets;
```

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):

```
var currentSet = sets
    .OrderByDescending(s => s.Count(universe.Contains))
    .First();
```

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.

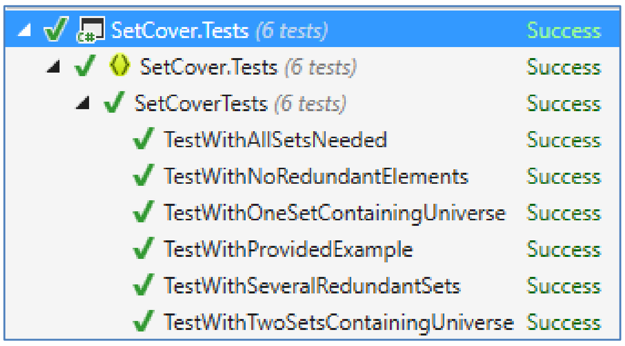
Once we have the set we're looking for, the next steps are trivial. Complete the TODOs below:

```
// TODO: add currentSet to result (selectedSets)
// TODO: remove currentSet from 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.

Problem 13. Run the Unit Tests

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



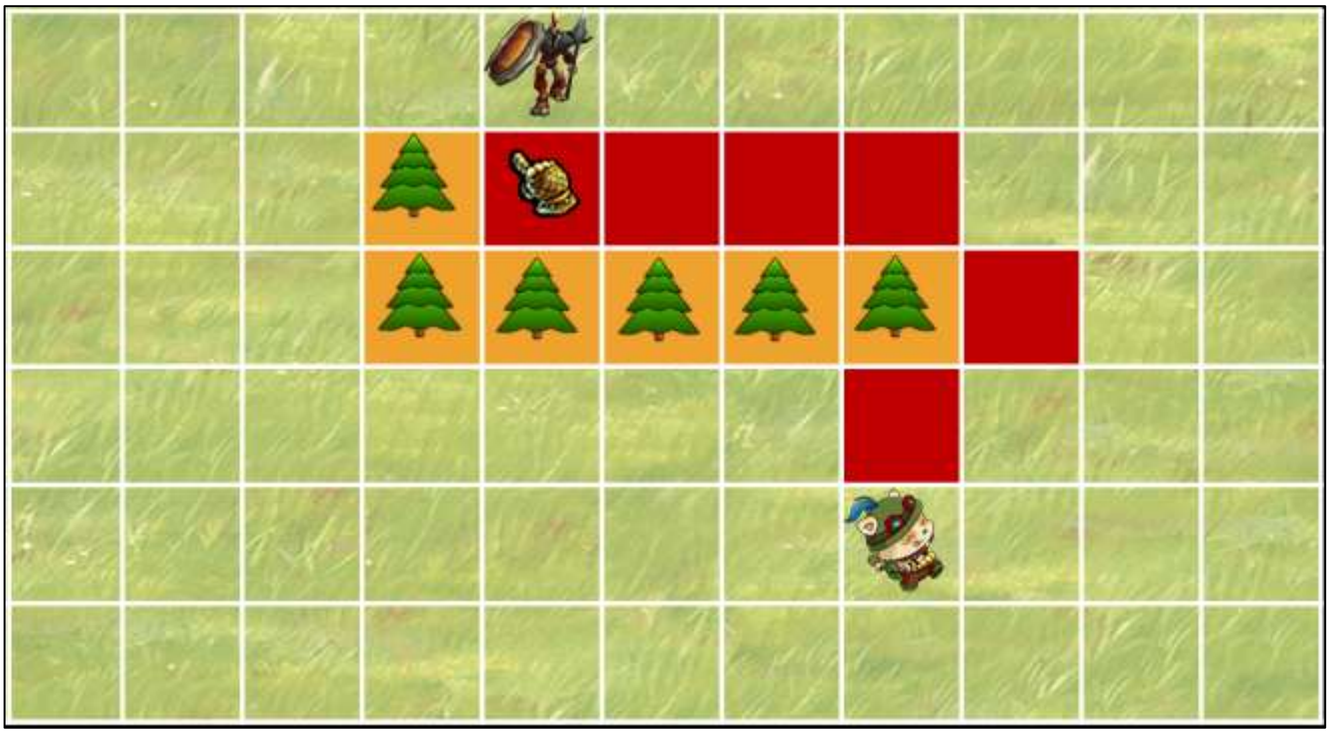
```
SetCover.Tests (6 tests)                    Success
  SetCover.Tests (6 tests)                  Success
    SetCoverTests (6 tests)                 Success
      TestWithAllSetsNeeded                 Success
      TestWithNoRedundantElements           Success
      TestWithOneSetContainingUniverse      Success
      TestWithProvidedExample               Success
      TestWithSeveralRedundantSets          Success
      TestWithTwoSetsContainingUniverse     Success
```

Congratulations, you're done!

Part III – A* Search Algorithm

In a standard task for A* Search we are given a map (usually a 2-dimensional matrix), coordinates of a starting point and coordinates of an end point and we are tasked to find the shortest path between the start and the end point, while taking in account that certain cells on the map can be blocked (non-traversable), the algorithm can also be viewed as expanding on Dijkstra's Shortest Path Algorithm

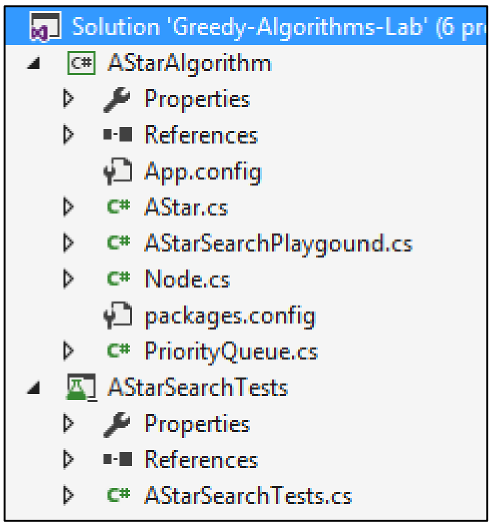
Examples: We are playing a game and we want to move our hero to the position of the pointer on the map.



The resulting shortest path can be seen marked with red on the map.

Problem 14. Provided Assets

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



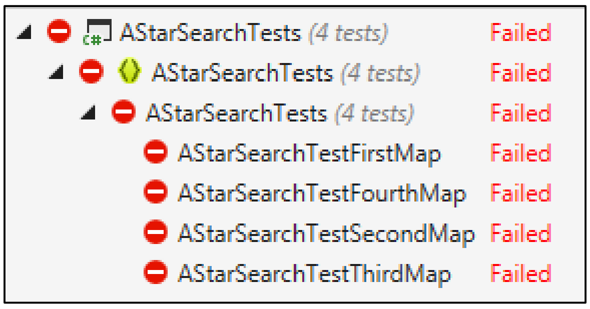
```
Solution 'Greedy-Algorithms-Lab' (6 pr
◢ AStarAlgorithm
    ▷ Properties
    ▷ References
      App.config
    ▷ AStar.cs
    ▷ AStarSearchPlaygound.cs
    ▷ Node.cs
      packages.config
    ▷ PriorityQueue.cs
◢ AStarSearchTests
    ▷ Properties
    ▷ References
    ▷ AStarSearchTests.cs
```

You are given a few sample maps and auxiliary methods in the **AStarSearchPlayground** class to help you test your algorithm, you're also given the class Node and an implementation of a Priority Queue as they will be needed for the algorithm. Your task is to implement the **FindShortestPath()** method `in the` **`AStar`** `class`:

```
public class AStar
{
    private readonly char[,] map;

    public AStar(char[,] map)
    {
        this.map = map;
    }

    public List<int[]> FindShortestPath(int[] startCoords, int[] endCoords)
    {
       throw new NotImplementedException();
    }
```

Problem 15. Run the Unit Tests to Make Sure They Fail

The unit tests should all fail initially:



```
◢ ⛔ AStarSearchTests (4 tests)        Failed
  ◢ ⛔ AStarSearchTests (4 tests)      Failed
    ◢ ⛔ AStarSearchTests (4 tests)    Failed
        ⛔ AStarSearchTestFirstMap     Failed
        ⛔ AStarSearchTestFourthMap    Failed
        ⛔ AStarSearchTestSecondMap    Failed
        ⛔ AStarSearchTestThirdMap     Failed
```

Problem 16. Initializing the required data structures

Like Dijkstra this algorithm relies on keeping track of distances and choosing the closest node so we'll need a way to keep for each cell the distance to the starting cell and also the estimated distance to the end cell (how and why will be explained later in the algorithm). In order to store the needed distances we'll be using the Node class – more specifically we'll make a matrix of Nodes (Node[,]) mirroring the Map we are given. We'll also need to extract the closest node to the start from all currently traversable nodes, so we'll need a priority queue as well. Using the quality of the Dijkstra algorithm, after we have stepped on a node we have surely taken the shortest path there, so any future paths to this node are meaningless to check – in order to save ourselves the useless operations we can keep all nodes we have visited in a HashSet<Node> and before trying to improve the distance to a node, we'll check if the node hasn't already been visited:

```
public class AStar
{
    private readonly PriorityQueue<Node> openNodesByFCost;
    private readonly HashSet<Node> closedSet;
    private readonly char[,] map;
    private readonly Node[,] graph;

    public AStar(char[,] map)
    {
        this.map = map;
        this.graph = new Node[map.GetLength(0), map.GetLength(1)];
        this.openNodesByFCost = new PriorityQueue<Node>();
        this.closedSet = new HashSet<Node>();
    }
```

Problem 17. Implementing the Main Algorithm

After we have prepared the needed structures it’s time to move on to the main algorithm. First we’ll need to get the starting Node, however since we won’t necessarily need all nodes on the map to find the shortest path we can only initialize nodes on a per need basis in order to conserve memory. To do this we’ll create a method **GetNode()**:

```
private Node GetNode(int row, int col)
{
    return this.graph[row, col] ?? (this.graph[row, col] = new Node(row, col));
}

public List<int[]> FindShortestPath(int[] startCoords, int[] endCoords)
{
    var startNode = this.GetNode(startCoords[0], startCoords[1]);
```

As discussed the method should create a Node only when it is required. The ?? operator is a ternary operator which checks if the left object is equal to null, if it is - it returns the value in the parenthesis otherwise it returns the original object. After we have the starting node, we need to set its distance from the start to 0, the property representing a node’s distance from the start is the GCost (the distance traveled on the Map/Grid). After we have set the GCost we can add the starting node to the priority queue and begin the searching process.

```
startNode.GCost = 0;
this.openNodesByFCost.Enqueue(startNode);
```

The following part will take place in a while loop, the ending conditions are either traversing all reachable nodes or finding the end Node. We start by extracting the closest node to the start from the priority queue and marking it as visited:

```
while (openNodesByFCost.Count > 0)
{
    var currentNode = this.openNodesByFCost.ExtractMin();
    this.closedSet.Add(currentNode);

    //TODO check if the current node is the end node
```

For each the current node we need to extract all neighboring nodes that are traversable (are inside the matrix and not marked with ‘W’) – since we are in a matrix the neighboring nodes are in all sides and diagonals to the current node, we can create a method **GetNeighbours()** and extract the logic of getting the neighboring nodes to it.

```
var neighbours = this.GetNeighbours(currentNode);
```

```
private List<Node> GetNeighbours(Node node)
{
    var neighbours = new List<Node>();

    var maxRow = this.graph.GetLength(0);
    var maxCol = this.graph.GetLength(1);
    for (int row = node.Row - 1; row <= node.Row + 1 && row < maxRow; row++)
    {
        if (row < 0) continue;
        for (int col = node.Col - 1; col <= node.Col + 1 && col < maxCol; col++)
        {
            //TODO Check that the col is in the matrix, is not a wall (the char 'W' on
               the map) and is not the current node
            var neighbour = this.GetNode(row, col);
            neighbours.Add(neighbour);
        }
    }

    return neighbours;
}
```

Once we have the list of neighbors we need to iterate over it and check for each node if we have found a shorter distance to it (note that in the Node constructor we initialize each node with a GCost of int.MaxValue, in this implementation we choose int.MaxValue to represent that the node is unreachable, this could easily be changed to Double.PositiveInfinity if we think that int.MaxValue will be a reachable distance). As we mentioned above if a node has already been visited (is in the HashSet<Node> closedSet) we need not check it any further as we have already found the best distance to it.

```
foreach (var neighbour in neighbours)
{
    // Skip if neighbour is already visited
    if (this.closedSet.Contains(neighbour))
    {
        continue;
    }

    var gCost = currentNode.GCost + CalculateGCost(neighbour, currentNode);
```

Here is where we start to see the specifics of A* Search, in Dijkstra we knew each node's neighbors and the distances to them thanks to the edge that connected them, here though we have a matrix and we can even move in diagonals, so we need to decide on constant distance between two cells in the matrix. A standard approach is to mark that the distance between two cells which lie on a general direction (i.e. one cell is directly on the left/right/bellow/top of the other) as 10 (this could be any number we'll use 10 here because it's easier to track) and the distance between two cells which lie next to each other on a diagonal as 14 (because 14 is the integer value of the Euclidian distance of a diagonal in a square with a side = 10), this conjecture of the distances we made is called an heuristic and is in the core of the A* Search algorithm. It is important to note that the heuristic we use for the A* Search algorithm can be different, we choose the one that would most fit the rules we wish to set for traversing the map – for instance we chose to allow movement on the diagonals and chose the cost for diagonal movement to be the Euclidian distance, but we could have decided that the cost for diagonal movement be some other arbitrary number that would fit our logic for moving on the map. Alternatively we could decide that it should be impossible to move diagonally and implement a heuristic that doesn't allow diagonal moves and in turn the neighbors of a cell will not include the cells diagonal to it. Having decided on a heuristic, we need to implement the calculations of the distances based on it – to do that we can create a method **GetDistance()** which will return the distance between two cells in the matrix based on our heuristic:

```
private static int CalculateGCost(Node node, Node prev)
{
    return GetDistance(node.Row, node.Col, prev.Row, prev.Col);
}

private static int GetDistance(int r1, int c1, int r2, int c2)
{
    var deltaX = Math.Abs(c1 - c2);
    var deltaY = Math.Abs(r1 - r2);

    if (deltaX > deltaY)
    {
        return 14 * deltaY + 10 * (deltaX - deltaY);
    }

    return 14 * deltaX + 10 * (deltaY - deltaX);
}
```

The method works in the following way it gets the absolute distance between the cells/nodes' X coordinate and the absolute distance between their Y coordinate. The distance here actually represents steps in a given direction. The smaller step count between X and Y can be taken as diagonals (because it's cheaper for us to move diagonally instead of horizontally or vertically) and the rest of the steps will be taken normally (example: we need to move 5 steps up and 3 right, then it's better for us to move 3 times to the upper right diagonal and then 2 times up).

After we have implemented our heuristic and calculated the GCost of the neighboring node we need to check if this cost is better than the previous GCost of the node – if it is we change its GCost, set its parent as the current Node (every node has a property Parent) and if the node is already in the priority queue we call the DecreaseKey operation for it, if it wasn't we add it to the queue and calculate its HCost (HCost here stands for Heuristic Cost) which represents the estimated distance from this node to the end Node – in other words we make a guess based on our heuristic of how close the end node is.

```
var gCost = currentNode.GCost + CalculateGCost(neighbour, currentNode);
if (gCost < neighbour.GCost)
{
    //TODO Improve GCost of the neighbouring node
    //TODO Set the parent of the neighbouring node to the current node

    if (!this.openNodesByFCost.Contains(neighbour))
    {
        neighbour.HCost = CalculateHCost(neighbour, endCoords);
        this.openNodesByFCost.Enqueue(neighbour);
    }
    else
    {
        this.openNodesByFCost.DecreaseKey(neighbour);
    }
}
```

As we can see we need a method for calculating the HCost of the neighboring node, luckily the HCost is the distance between the nodes based on our heuristic and we can calculate it the same way we calculated the GCost. It is important to note here that the real distance to the end node might be different as there might be obstacles along the way, but if the heuristic is well chosen we will try the most logical choices first (i.e. the directions which lead us closer to the end node).

```
private static int CalculateHCost(Node node, int[] endCoords)
{
    //TODO Calculate the distance between the current node and the end node
}
```

At this point we have calculated all the needed information for a neighboring node to add it to the priority queue, the priority queue's ExtractMin method on the other hand provides us with the element with the lowest FCost – a node's FCost is the sum of its GCost and HCost or in other words we want to first visit the nodes which are at the same time closest to the start and closest to the end as those nodes would probably lie on the shortest path from the start to the end. With this our algorithm is completed, however just finding the shortest path in itself is pretty meaningless if we don't return it, so we should reconstruct it and return it. The reconstructing is the same as other algorithms, we start from the end node and just follow the parents back until we reach the start node (whose parent is null).

```
private static List<int[]> ReconstructPath(Node currentNode)
{
    var cells = new List<int[]>();
    while (currentNode != null)
    {
        cells.Add(new[] { currentNode.Row, currentNode.Col });
        currentNode = currentNode.Parent;
    }

    return cells;
}
```

Remember when we set that condition for breaking the while loop when we find the end node, now is a good point to add some extra code to it. Since finding the end node means we have also found the shortest path we can just return the reconstructed path.

```
if (currentNode.Row == endCoords[0] && currentNode.Col == endCoords[1])
{
    return ReconstructPath(currentNode);
}
```

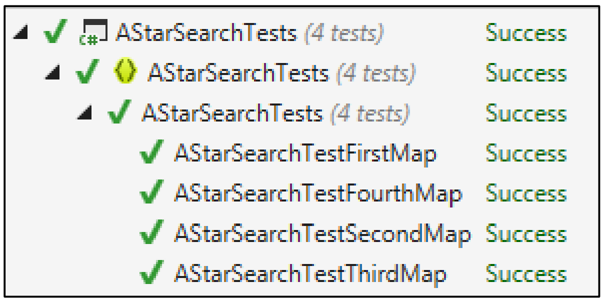
Alternatively there could be no path from the start to the end (for instance the map can be divided in two by impassable terrain), in that situation the while loop will end without finding the end node and we should return an empty list for the reconstructed path.

```
                    else
                    {
                        this.openNodesByFCost.DecreaseKey(neighbour);
                    }
                }
            }
        }

        // No path found
        return new List<int[]>(0);
}
```

With the path reconstructed we are done.

Problem 18. Run the Unit Tests

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



```
◢ ✓ AStarSearchTests (4 tests)             Success
   ◢ ✓ AStarSearchTests (4 tests)          Success
      ◢ ✓ AStarSearchTests (4 tests)       Success
            ✓ AStarSearchTestFirstMap      Success
            ✓ AStarSearchTestFourthMap     Success
            ✓ AStarSearchTestSecondMap     Success
            ✓ AStarSearchTestThirdMap      Success
```

Congratulations, you're done!