Exercises: Dynamic Programming

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

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

Part I – Longest Increasing Subsequence

Let's have a sequence of numbers **S = {a_1, a_2, … a_n}**. An **increasing** subsequence is a sequence of numbers within S where each number is **larger** than the previous. We **do not change the relative positions** of the numbers, e.g. we do not move smaller elements to the left to obtain longer sequences. If several sequences with equal length exist, find the left-most of them. Examples:

Input	Output

`7 3 5 8 -1 6 7`	`Longest increasing subsequence (LIS)`
	`  Length: 4`
	`  Sequence: [3, 5, 6, 7]`

`11 12 13 3 14 4 15 5 6 7 8 7 16 9 8`	`Longest increasing subsequence (LIS)`
	`  Length: 7`
	`  Sequence: [3, 4, 5, 6, 7, 8, 16]`

`3 14 5 12 15 7 8 9 11 10 1`	`Longest increasing subsequence (LIS)`
	`  Length: 6`
	`  Sequence: [3, 5, 7, 8, 9, 11]`

Problem 1. Setup the Longest-Increasing-Subsequence Project

For the LIS problem you'll be using the projects **Longest-Increasing-Subsequence** and the **Longest-Increasing-Subsequence.Tests**.

The problem involves several steps:

- Read a sequence from the console.
- Find the LIS (longest increasing subsequence) using dynamic programming.
- Print the result on the console.

For now, we'll skip the first step in order to concentrate on the algorithm and use a hard-coded input sequence. Reading the sequence at a later stage is trivial. Output is easy too; if we declare a method which takes as argument the sequence and returns the LIS as an array, we simply print its length and elements:

```
static void Main()
{
    var sequence = new[] { 3, 14, 5, 12, 15, 7, 8, 9, 11, 10, 1 };
    var longestSeq = FindLongestIncreasingSubsequence(sequence);

    Console.WriteLine("Longest increasing subsequence (LIS)");
    Console.WriteLine("  Length: {0}", longestSeq.Length);
    Console.WriteLine("  Sequence: [{0}]", string.Join(", ", longestSeq));
}
```

Your task is to complete the **`FindLongestIncreasingSubsequence(int[] sequence)`** method and test it.

Problem 2. LIS: Dynamic Programming Approach

The LIS problem can be solved by dividing it into sub-problems – for each element at **index `i`** of the **sequence S**, find the LIS in the range [S_0 … S_i].

Example for **S** = { 3, 14, 5, 12, 15, 7, 8, 9, 11, 10, 1 }:

- LIS { 3 } => { 3 }
- LIS { 3, 14 } => { 3, 14 }
- LIS { 3, 14, 5 } => { 3, 5 }
- LIS { 3, 14, 5, 12 } => { 3, 5, 12 }
- etc.

For each index, we'll **keep track of the length of the LIS up to that index** and the **previous index** of the LIS. E.g., the length of the LIS at index **5** is **3**, the longest sequence ending in **`seq[5]`** is {3, 5, 7} and the index of the previous element of the subsequence (the number 5) is 2. The table below illustrates these computations:

`index`	0	1	2	3	4	5	6	7	8	9	10
`S[]`	**3**	**14**	**5**	**12**	**15**	**7**	**8**	**9**	**11**	**10**	**1**
`len[]`	1	2	2	3	4	3	4	5	6	6	1
`prev[]`	-1	0	0	2	3	2	5	6	7	7	-1
`LIS`	{3}	{3,14}	{3,5}	{3,5,12}	{3,5,12,15}	{3,5,7}	{3,5,7,8}	{3,5,7,8,9}	{3,5,7,8,9,11}	{3,5,7,8,9,10}	{1}

We need to calculate the info in the above table for every element of the original sequence **S**, so we'll need two additional arrays with length equal to the length of **S**. Translating this into code within our method, we get:

```
int[] len = new int[sequence.Length];
int[] prev = new int[sequence.Length];
```

We also need to keep track of the **length of the longest subsequence** found so far and the index at which it ends (we'll use **-1** to mark that there is no such index found currently):

```
int maxLen = 0;
int lastIndex = -1;
```

Problem 3. Calculate LIS at Each Index

To obtain the longest increasing sequence up to a given index, we just have to find the LIS up to that point to which the current element can be appended as the largest. That is why, when considering the sequence {3, 14, 5} we obtained {3, 5}; we want to know the longest sequence in which the current number (5) participates.

We'll do the following:

- loop through each number in the sequence;
- find the longest sequence up to that point which ends with a number which is smaller than the current.

Remember that we keep track of the length of each LIS in the **`len[]`** array.

```
...
for (int x = 0; x < sequence.Length; x++)
{
    len[x] = 1;
    prev[x] = -1;
    for (int i = 0; i < x; i++)
    {
        // TODO: Check if element is smaller than current and its LIS is the same length or larger
        // TODO: If condition is true:
        // -increase current length
        // -mark the index of the previous element in the new LIS
    }
```

Don’t forget to keep track of the length of the longest increasing subsequence and the index of its last element:

```
if (len[x] > maxLen)
{
    maxLen = len[x];
    lastIndex = x;
}
```

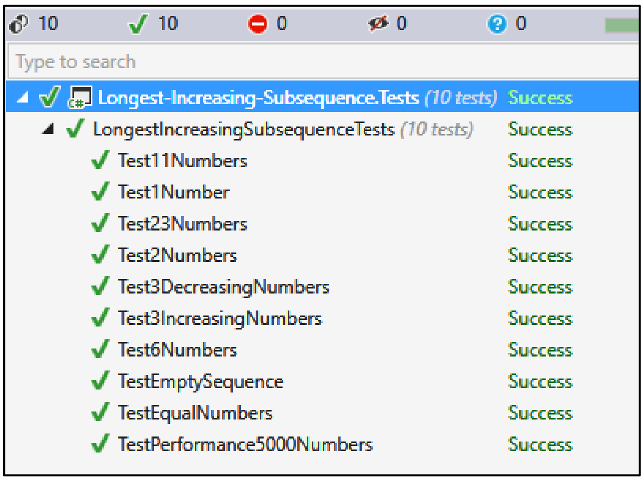
Problem 4. Recover the LIS by Walking through prev[]

Knowing the index of the last element of the LIS, we can get the whole sequence by continuously taking each previous element using the info we keep in the **prev[]** array. Store the elements in a list, reverse it and return it:

```
var longestSeq = new List<int>();
while (lastIndex != -1)
{
    longestSeq.Add(sequence[lastIndex]);
    lastIndex = prev[lastIndex];
}

longestSeq.Reverse();

return longestSeq.ToArray();
```

Problem 5. Run the Unit Tests

Run the **unit tests** to ensure your algorithm works correctly.



Problem 6. Remove the Hardcoded Values

The solution is now ready, but the program currently works on a hardcoded array. Just read the numbers from the console and you're done!

```
var sequence = Console.ReadLine()
    .Split()
    .Select(int.Parse)
    .ToArray();
```

Part II – Longest Common Subsequence

Considering **two sequences S_1 and S_2**, the **longest common subsequence** is a sequence which is a subsequence of both S_1 and S_2. For instance, if we have two strings (sequences of characters), "abc" and "adb", the LCS is "ab" – it is a subsequence of both sequences and it is the longest (there are two other subsequences – "a" and "b").

Examples:

Input	Output

abc
adb
	Longest common subsequence:
	  first = abc
	  second = adb
	  lcs = ab

ink some beer
drink se ber
	Longest common subsequence:
	  first = ink some beer
	  second = drink se ber
	  lcs = ink se ber

tree
team
	Longest common subsequence:
	  first = tree
	  second = team
	  lcs = te

Problem 7. Setup

For this problem you'll be working with the **Longest-Common-Subsequence** and the **Longest-Common-Subsequence.Tests** projects.

Let's work with hardcoded values (strings) for now and create a method to find the LCS:

```
static void Main()
{
    var firstStr = "tree";
    var secondStr = "team";

    var lcs = FindLongestCommonSubsequence(firstStr, secondStr);

    Console.WriteLine("Longest common subsequence:");
    Console.WriteLine("  first  = {0}", firstStr);
    Console.WriteLine("  second = {0}", secondStr);
    Console.WriteLine("  lcs    = {0}", lcs);
}
```

Your task will be to complete the **`FindLingestCommonSubsequence(string firstStr, string secondStr)`** method.

Problem 8. LCS: Dynamic Programming Approach

Just like the LIS problem, we can solve the LCS problem by **solving sub-problems** and keeping track of the solutions to the sub-problems (**memoization**). In the LIS problem we used an array, but here we'll be comparing two sequences, therefore, we'll need a matrix like the one below:

		t	e	a	m
	LCS("", "")	LCS("", t)	LCS("", te)	LCS("", tea)	LCS("", team)
t	LCS(t, "")	LCS(t, t)	LCS(t, te)	LCS(t, tea)	LCS(t, team)
r	LCS(tr, "")	LCS(tr, t)	LCS(tr, te)	LCS(tr, tea)	LCS(tr, team)
e	LCS(tre, "")	LCS(tre, t)	LCS(tre, te)	LCS(tre, tea)	LCS(tre, team)
e	LCS(tree, "")	LCS(tree, t)	LCS(tree, te)	LCS(tree, tea)	LCS(tree, team)

The rows will represent subsequences (substrings) of the first string ("tree"); the first row will represent a substring with length 0 (an empty string), the second row will represent a substring of length 1 ("t"), the third row will represent a substring of length 2 ("tr"), etc. The last row will represent a substring of length 4 which is the entire string "tree".

The columns will represent the substrings of the second string ("team"), again starting with an empty string and ending with the entire string.

In each cell, we'll enter the length of the LCS of the two substrings – the substring of the first string (the rows) and the second string (the columns). E.g., in the table above, cell (2, 2) will represent the LCS of "tr" and "te". Note that we assume that an empty string does not have anything in common with any other string, therefore row 0 and column 0 will be filled with zeros.

Problem 9. Find the LCS for Every Combination of Substrings

We know what to do – create a matrix of integers and calculate the LCS length for each cell. Let's begin.

The matrix should have 1 more row than the number of characters in the first string and 1 more column than the number of characters in the second string (the first row and column are the empty substrings). Therefore:

```
int firstLen = firstStr.Length + 1;
int secondLen = secondStr.Length + 1;
var lcs = new int[firstLen, secondLen];
```

Now, we have to iterate each cell of **`lcs[,]`** from top to bottom and from left to right and decide what number to put in that cell. Remember, at each step we already have the results from previous steps, so we can build on that. We have two distinct cases:

1) The last character of the first substring is equal to the last character of the second substring

This means that, compared to the cell which is to the left and up of the current one, the length of the current cell's LCS is greater by 1. Why? The cell to the left and up of the current one will hold the LCS of two substrings which are shorter by 1 than the current substrings; basically the last character (which is the same) won't be present. Adding that same character to both substrings, we'll obtain the current cell and an LCS greater by 1.

2) The last character of the first substring is different from the last character of the second substring

We know the LCS of all substrings shorter than the current ones. The longest LCS so far should be one of two – the one directly above or the one directly to the left of the current cell. Adding a character to one of the substrings used to calculate these two LCSs doesn't have any effect, therefore, the current cell's LCS is the larger of the two.

Complete the if-statement following the logic above:

```
for (int i = 1; i < firstLen; i++)
{
    for (int j = 1; j < secondLen; j++)
    {
        if (firstStr[i - 1] == secondStr[j - 1])
        {
            // TODO: LCS = LCS(cell to the left and up) + 1
        }
        else
        {
            // TODO: LCS = max(LCS(cell above), LCS(cell to the left))
        }
    }
}
```

Once done, the matrix should be filled with the length of each LCS, like so:

		t	e	a	m
	0 LCS("", "") = ""	0 LCS("", t) = ""	0 LCS("", te) = ""	0 LCS("", tea) = ""	0 LCS("", team) = ""
t	0 LCS(t, "") = ""	1 LCS(t, t) = t	1 LCS(t, te) = t	1 LCS(t, tea) = t	1 LCS(t, team) = t
r	0 LCS(tr, "") = ""	1 LCS(tr, t) = t	1 LCS(tr, te) = t	1 LCS(tr, tea) = t	1 LCS(tr, team) = t
e	0 LCS(tre, "") = ""	1 LCS(tre, t) = t	2 LCS(tre, te) = te	2 LCS(tre, tea) = te	2 LCS(tre, team) = te
e	0 LCS(tree, "") = ""	1 LCS(tree, t) = t	2 LCS(tree, te) = te	2 LCS(tree, tea) = te	2 LCS(tree, team) = te

Problem 10. Recover the LCS by Iterating the lcs[,] Matrix

Once the table is filled, all we need to do is recover what we need from it. Let's do this in a separate method, **`RetrieveLCS(string firstStr, string secondStr, int[,] lcs)`**.

We iterate the matrix starting from the bottom-right corner until we reach row 0 or column 0. We'll fill the characters in a **List<char>**.

Again, we have two distinct cases:

1) The last characters of the two substrings are the same – add the character to the list and move to the cell which is to the left and above the current one. The logic is the same as the one we used to fill the matrix.
2) The characters are different – we need to decide where to go next – up or left. We go to the cell which has the same LCS length as the current one (if both have the same length, it doesn't really matter).

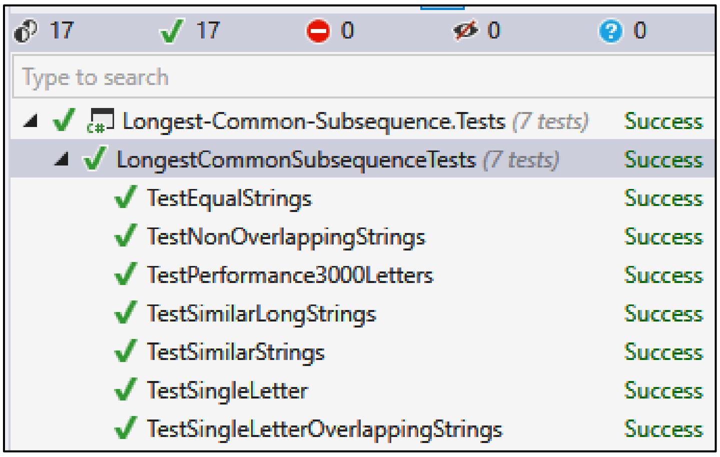
```
while (i > 0 && j > 0)
{
    if (firstStr[i - 1] == secondStr[j - 1])
    {
        // TODO: Add character to the list and move up and to the left
    }
    else if (lcs[i, j] == lcs[i - 1, j])
    {
        // TODO: move up
    }
    else
    {
        // TODO: move to the left
    }
}
```

Finally, since we obtained all the characters in reversed order, we need to reverse the list and return it as a string. We can do the following:

```
sequence.Reverse();

return new string(sequence.ToArray());
```

Problem 11. Run the Unit Tests

If you've completed the TODOs properly, the unit tests should pass:



Problem 12. Remove the Hardcoded Values

If the unit tests passed, your program is correct. You can modify the **`Main()`** method to receive the two strings from the console:

```
Console.Write("firstStr = ");
var firstStr = Console.ReadLine();

Console.Write("secondStr = ");
var secondStr = Console.ReadLine();
```

That's all!

Part III – Knapsack

Imagine you have a bag (**knapsack**) and you want to fill it with as many of your most valuable items as you can. The knapsack, of course, cannot hold an infinite number of items, it has a **weight limit (capacity)**. Based on this capacity, you need to decide which items to put in it in order to maximize the **value** of the items in the knapsack. We'll assume that the value and weight of each item and the weight limit of the knapsack are all non-negative integers. This, in essence, is the so-called **Knapsack problem.**

Example:

Input	**Output**

`Weight limit: 20`
`Item 1 – weight: 5, price: 30`
`Item 2 – weight: 8, price: 120`
`Item 3 – weight: 7, price: 10`
`Item 4 – weight: 0, price: 20`
`Item 5 – weight: 4, price: 50`
`Item 6 – weight: 5, price: 80`
`Item 7 – weight: 2, price: 10`
	`Knapsack weight capacity: 20`
	`Take the following items in the knapsack:`
	`  (weight: 8, price: 120)`
	`  (weight: 0, price: 20)`
	`  (weight: 4, price: 50)`
	`  (weight: 5, price: 80)`
	`  (weight: 2, price: 10)`
	`Total weight: 19`
	`Total price: 280`

Problem 13. Setup

You are given a complete **`Main()`** method for this problem in the **Knapsack-Problem** project as well as a simple **Item** class. The example input is hardcoded and the output is taken care of:

```
var items = new[]
{
    new Item { Weight = 5, Price = 30 },
    new Item { Weight = 8, Price = 120 },
    new Item { Weight = 7, Price = 10 },
    new Item { Weight = 0, Price = 20 },
    new Item { Weight = 4, Price = 50 },
    new Item { Weight = 5, Price = 80 },
    new Item { Weight = 2, Price = 10 }
};

var knapsackCapacity = 20;

var itemsTaken = FillKnapsack(items, knapsackCapacity);

Console.WriteLine("Knapsack weight capacity: {0}", knapsackCapacity);
Console.WriteLine("Take the following items in the knapsack:");
foreach (var item in itemsTaken)
{
    Console.WriteLine(
        "  (weight: {0}, price: {1})",
        item.Weight,
        item.Price);
}

Console.WriteLine("Total weight: {0}", itemsTaken.Sum(i => i.Weight));
Console.WriteLine("Total price: {0}", itemsTaken.Sum(i => i.Price));
```

Your task is to complete the **`FillKnapsack(Item[] items, int capacity)`** method which returns an array of Items and test it using the provided unit tests in the **Knapsack-Problem.Tests** project.

Problem 14. Knapsack: Dynamic Programming Approach

Just as with the previous problems, we'll solve the Knapsack problem by dividing it into sub-problems. If we have the sack's maximal capacity, **c**, we can find solutions for each capacity starting from 0 and incrementing by 1 until we reach **c**.

We need to keep track of two things – the maximal price at each unit of capacity (from 0 to **c**), and mark the items we want to take. Since any item can be taken or not for any given capacity, we need **matrices** to hold this info:

```
var maxPrice = new int[items.Length, capacity + 1];
var isItemTaken = new bool[items.Length, capacity + 1];
```

We will loop through each item and each capacity **c** and decide whether we'll take the item. We take an item if:

1) We have enough capacity to take it;

2) The item provides the best price compared to other items we could've taken.

Problem 15. Preparation

To find a solution for a problem using dynamic programming, we need a starting point, something to build upon. In the previous problem, in order to find the LCS we had a matrix. For both strings, we had a sub-problem in which we consider empty substrings (row 0 and column 0). This was our starting point and we could build on that to find the LCSs for each combination of substrings from there on.

In this problem, we don't have that starting point yet. We need to create it. So we'll fill the tables for the first item – fill row 0 of both matrices and for each capacity.

```
// Calculate maxPrice[0, 0...capacity]
for (int c = 0; c <= capacity; c++)
{
    if (items[0].Weight <= c)
    {
        maxPrice[0, c] = items[0].Price;
        isItemTaken[0, c] = true;
    }
}
```

Once we have that, we can start filling the rest of the matrices in order to find the solution.

Problem 16. Find Solutions for Each Item and Unit of Capacity

Having the solutions for item 0 and all possible capacities, we can complete the matrices for the rest of the items:

```
// Calculate maxPrice[1...len(items), 0...capacity]
for (int i = 1; i < items.Length; i++)
{
    for (int c = 0; c <= capacity; c++)
    {
        // TODO
    }
}
```

To find out whether an item is worth taking, let's first assume we haven't taken it. Essentially, the best price will be the same as the best price at the given capacity **c** for the previous item:

```
// Don't take item i
maxPrice[i, c] = maxPrice[i - 1, c];
```

Then, we need to check if we have enough capacity to take it:

```
// Try to take item i (if it gives better price)
var remainingCapacity = c - items[i].Weight;
if (remainingCapacity >= 0)
```

The trickiest part is to decide whether taking the item is our best option. We have a variable **`remainingCapacity`**; if the maximal price for the given item at this remaining capacity added to the current item's price gives us a better price than the current, the item is worth it.

You can think about it this way – if we take the item, what is the best price we can achieve? That would be the item's price plus the best price we have for the remaining capacity. If the result is larger than what we currently have (by default, we decided not to take the item), this means taking the item is better than not taking it.

Note that we know the value of **`maxPrice[i, remainingCapacity]`**. Since we're at capacity **`c`,** this means all smaller capacities have been filled already and **`remainingCapacity`** is indeed smaller, because from **`c`** we subtracted the current item's weight (a non-negative integer).

You need to complete the condition of the if-statement (make sure the item is worth it) and take the necessary actions if it is:

```
if (remainingCapacity >= 0
    // TODO: check if item is worth it
    )
{
    // TODO: mark the new maxPrice for this item at this capacity
    // TODO: mark the item as taken
}
```

Problem 17. Retrieve the Items Taken

If you completed the above steps correctly, you should now have a complete table like this:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Item 1 (5,30)	0	0	0	0	0	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
Item 2 (8,120)	0	0	0	0	0	30	30	30	120	120	120	120	120	150	150	150	150	150	150	150	150
Item 3 (7,10)	0	0	0	0	0	30	30	30	120	120	120	120	120	150	150	150	150	150	150	150	160
Item 4 (0,20)	20	20	20	20	20	50	50	50	140	140	140	140	140	170	170	170	170	170	170	170	180
Item 5 (4,50)	20	20	20	20	70	70	70	70	140	140	140	140	190	190	190	190	190	220	220	220	220
Item 6 (5,80)	20	20	20	20	70	100	100	100	140	150	150	150	190	220	220	220	220	270	270	270	270
Item 7 (2,10)	20	20	30	30	70	100	100	110	140	150	150	160	190	220	220	230	230	270	270	280	280

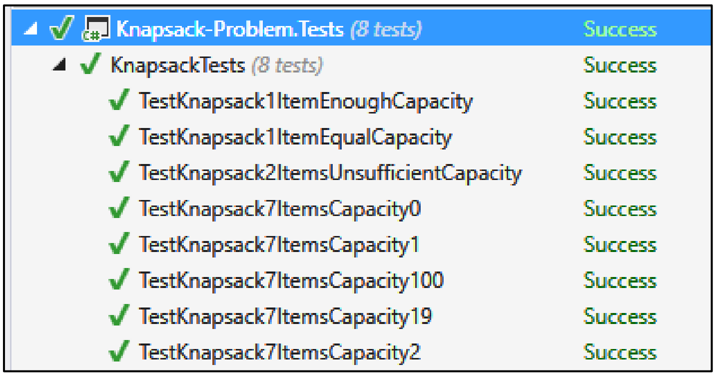
How do we extract the info we need? We start with the last item; if it's marked as taken in the **`isItemTaken[,]`** matrix for the maximal capacity, we put the item in a list and reduce the capacity by its weight. On the next step, we'll check if the previous item is marked as taken for the remaining capacity and so on.

Finally, return the items taken as array:

```
// Print the takenItems
var itemsTaken = new List<Item>();
int itemIndex = items.Length - 1;

while (itemIndex >= 0)
{
    // TODO: check if item is marked as taken at current capacity
    // TODO: if true, add item to list and reduce current capacity

    itemIndex--;
}

itemsTaken.Reverse();

return itemsTaken.ToArray();
```

Problem 18. Run the Unit Tests

If you worked correctly, all tests should pass:



Since there are multiple items and each item has weight and price, taking input from the console won't be so easy. You either have to tell the user how to enter the data or read it from a file. In any case, this will take time and is not crucial for the algorithm, so you may skip the "Remove the Hardcoded Values" step this time.