**Lab: Dynamic Programming Advanced**

This document defines the lab for the ["Algorithms – Advanced (Java)" course @ Software University](https://softuni.bg/trainings/3611/algorithms-advanced-with-java-january-2022). Please submit your solutions (source code) of all below-described problems in [Judge](https://judge.softuni.bg/Contests/2504/Dynamic-Programming-Advanced-Lab).

# 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 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 is the so-called **Knapsack problem.**

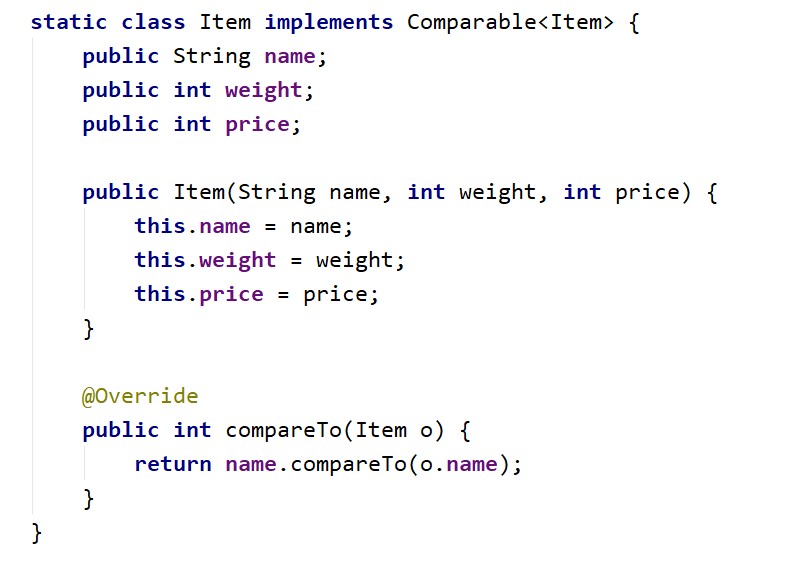
For a judge, order the items in alphabetical order.

Example:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 20  Item1 5 30  Item2 8 120  Item3 7 10  Item4 0 20  Item5 4 50  Item6 5 80  Item7 2 10  end | Total Weight: 19  Total Value: 280  Item2  Item4  Item5  Item6  Item7 |

## Setup

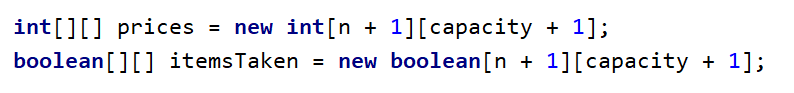
You are given a complete create a simple **Item** class.



## 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:



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

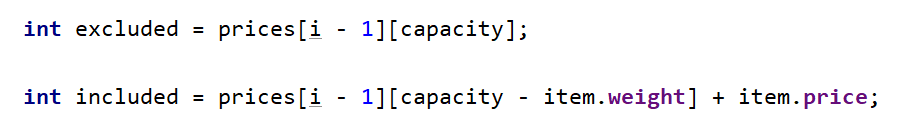
1) We have enough capacity to take it;

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

## 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:

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:



The trickiest part is to decide whether taking the item is our best option.

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.

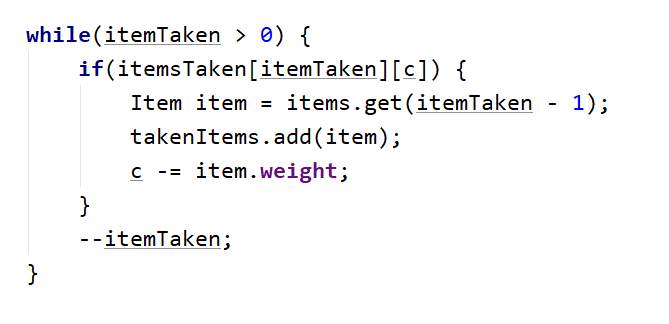
## Retrieve the Items Taken

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. In 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 an array:



Since there are multiple items and each item has weight and price, taking input from the console won’t be so easy. You either must 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.

# Longest Common Subsequence

Considering **two sequences S1 and S2**, the **longest common subsequence** is a sequence that is a subsequence of both S1 and S2. 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 | 2 |
| ink some beer  drink se ber | 10 |
| tree  team | 2 |

## 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".

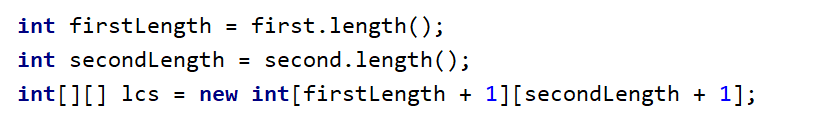
The columns will represent the substrings of the second string ("team"), again starting with an empty string and ending with the entire string.

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.

## 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:



Now, we must 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:

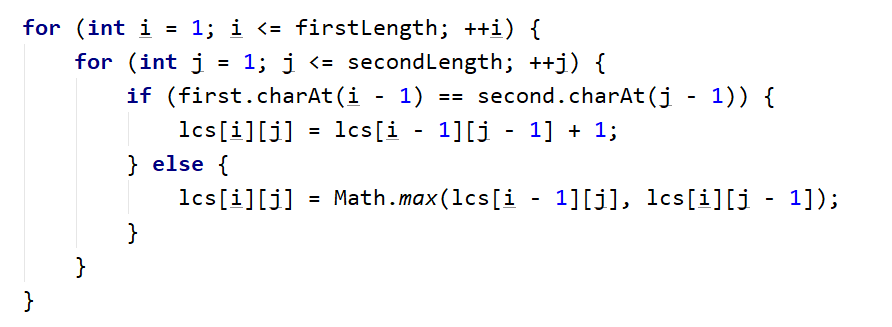
1. The last character of the first substring is equal to the last character of the second substring

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.

1. The last character of the first substring is different from the last character of the second substring

We know the LCS of all substrings is 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:



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 | |

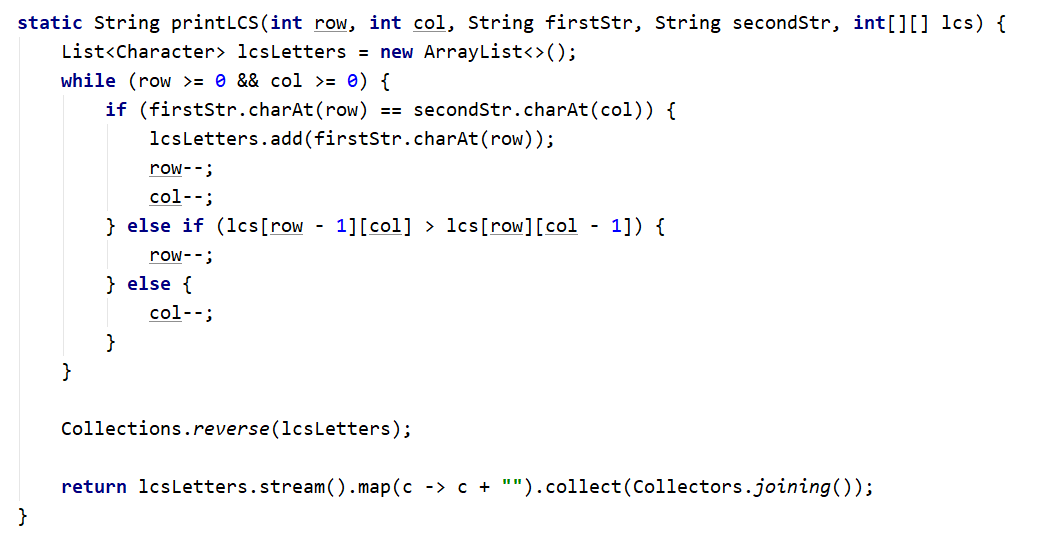
## Recover the LCS by Iterating the lcs[][] Matrix

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,

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<Charachter>**.

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 matter).



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