# Lab: Dynamic Programming Advanced

This document defines the lab for ["Algorithms – Advanced (C#)" course @ Software University](https://softuni.bg/trainings/3186/algorithms-advanced-with-c-sharp-january-2021#lesson-19912).

Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2589).

## Rod Cutting

Find the best way to cut up a rod with a specified length. You are also given to prices of all possible lengths starting from 0.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 0 1 5 8 9 10 17 17 20 24 30  4 | 10  2 2 |
| 0 1 5 8 9 10 17 17 20 24 30  8 | 22  2 6 |
| 0 1 5 8 9 10 17 17 20 24 30  10 | 30  10 |

## 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.

### Input

* On the first line you will receive an integer – capacity – knapsack’s capacity.
* On the next lines until "end" is not entered, you will receive an item in the following format: "{name} {weight} {value}".

### Output

* Print the used capacity of the knapsack in the following format: "Total Weight: {usedCapacity}".
* Print the total value of all items in the knapsack in the following format: "Total Value: {totalValue}".
* Print names of all items in the knapsack **sorted by name in alphabetical order**.

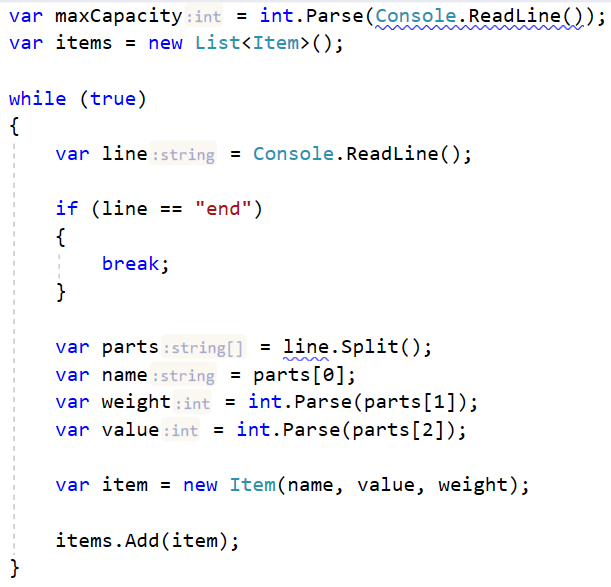
### Example

|  |  |
| --- | --- |
| **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 |

### Solution

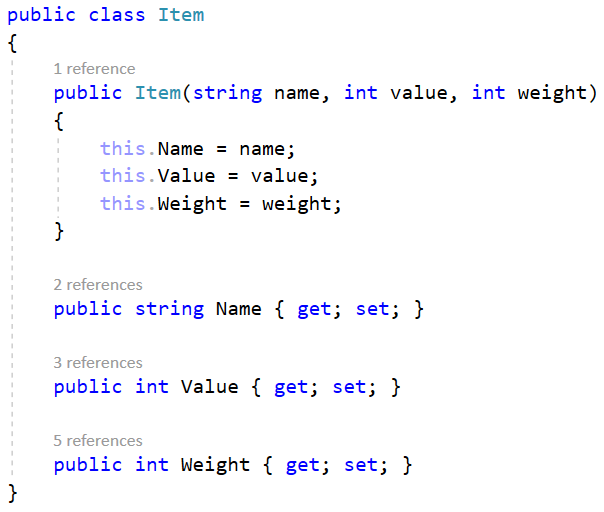
#### Read Input Data

Let’s start by reading the user input:



#### Item Class

Let’s create a simple Item class.



#### 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 the maximal price at each unit of capacity (from 0 to c). We need a matrix for this:



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

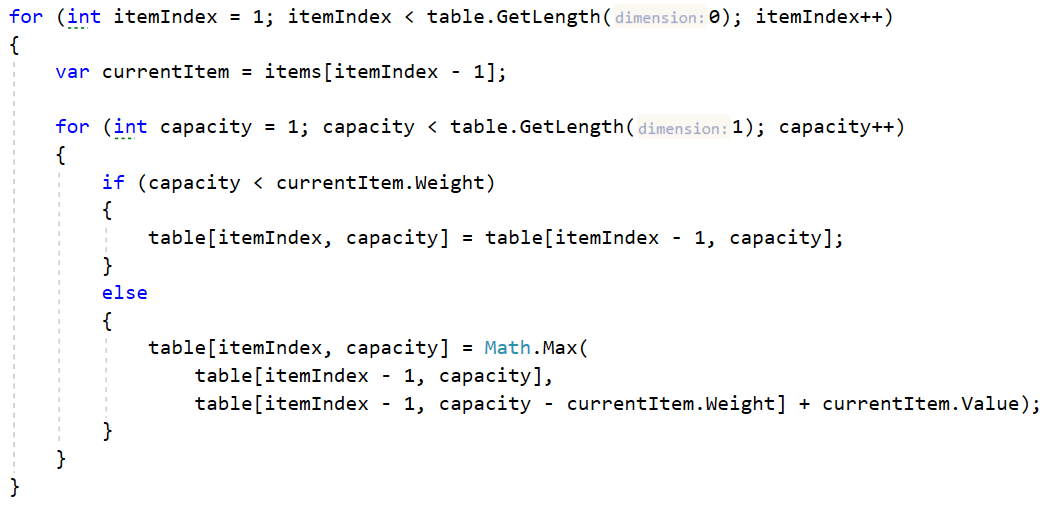
1) We have enough capacity to take it.

2) The item provides the best value 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 matrix 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 value will be the same as the best value at the given capacity c for the previous item:



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

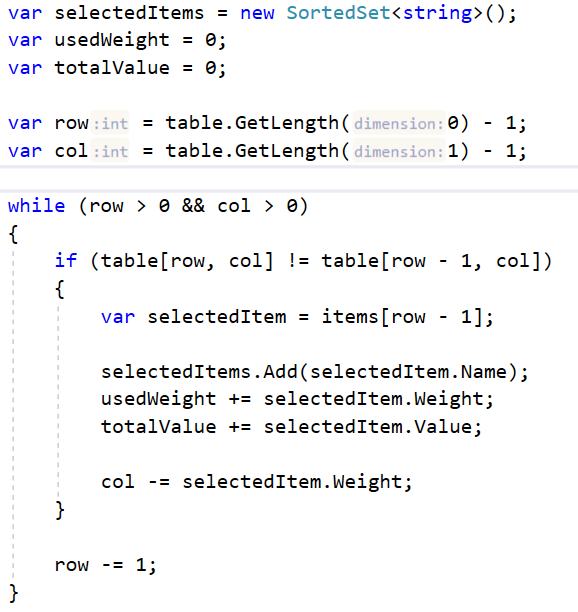
You can think about it this way – if we take the item, what is the best value we can achieve? That would be the item’s value plus the best value we have for the remaining capacity. If the result is larger than what we currently have (by default, we decided not to take the item), this means taking the 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? Backtrack your way through the matrix to find which items are in your knapsack. Start at the final index in the array and check whether the value stored at that index is equal to the value located one row above. If it isn't, then the item represented by the current row is in the knapsack.



## Longest Increasing Subsequence

Let’s have a sequence of numbers **S = {a1, a2, … an}**. 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** |
| 1 2 5 3 4 | 1 2 3 4 |
| 4 3 2 1 | 4 |
| 4 2 -1 3 5 5 | 2 3 5 |

### Solution

### 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 [**S0** … **Si**].

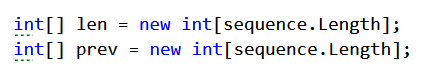
Example for S = { 3, 14, 5, 12, 15, 7, 8, 9, 11, 10, 1 }:

* 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:



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):



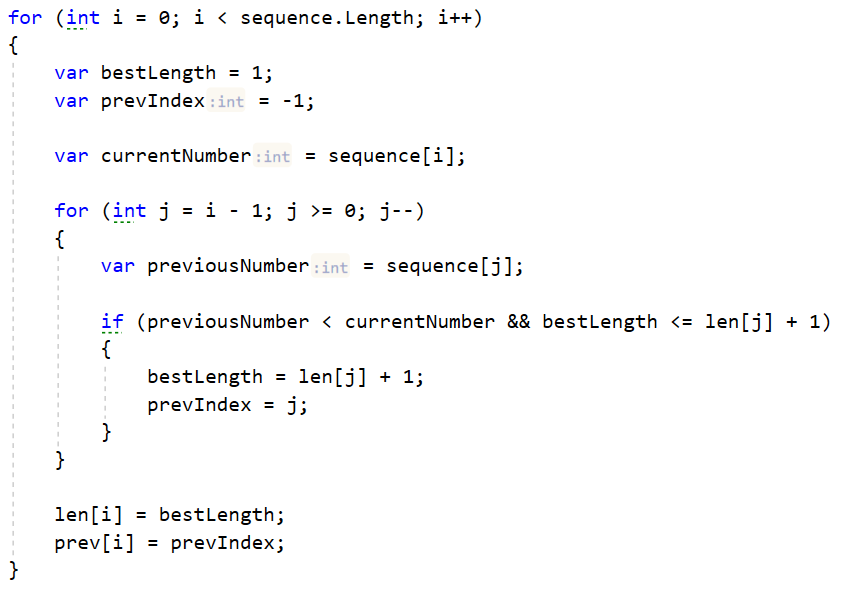
### Calculate LIS at Each Index

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.

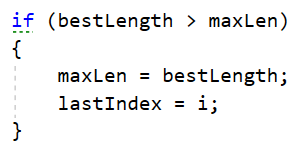
We’ll do the following:

* 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.

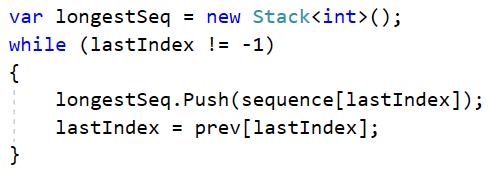


Don’t forget to keep track of the length of the longest increasing subsequence and the index of its last element:



### Recover the LIS

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:



### 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!

