The Knapsack Problem

2020

A brief explanation of the Knapsack Problem and its composing elements

Java Group

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Group Leader: Simran Supra Other group members: Mihai – Daniel Stefanescu  Malliaris Vasileios |
|  |  |  |

Table of Contents

[To Our Customers 1](#_Toc39021209)

[The Knapsack Problem What it actually does? 2](#_Toc39021210)

[What we need to get started? 3](#_Toc39021211)

[The use of Dynamic Programming Why Dynamic and not Integer? 4](#_Toc39021212)

[Running the program 6](#_Toc39021213)

[Reference 8](#_Toc39021214)

[Company Information 8](#_Toc39021215)

# To Our Customers

*Did you know that the Knapsack Problem has been studied for more than a century, with early works dating as far back as 1897?*

*In this document you will find a detailed description of the Knapsack Problem, its operational capacities and how it works when used. Also, due to its wide use there are many variations of the Knapsack Problem which will be omitted, focusing just on the 0-1 Knapsack Problem written in C#.*

# The Knapsack Problem What it actually does?

The Knapsack Problem is a problem in combinatorial optimization which receives a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Knapsack problems appear in real-world decision-making processes in a wide variety of fields such as:

* finding the least wasteful way to cut raw materials
* selection of investments and portfolios
* selection of assets for asset-backed securitization
* other knapsack cryptosystems

For a better understanding of The Knapsack Problem, here is a simplified example:

*“If an exam contains 12 questions each worth 10 points, the test-taker need only answer 10 questions to achieve a maximum possible score of 100 points. However, on tests with a heterogeneous distribution of point values, it is more difficult to provide choices. Feuerman and Weiss proposed a system in which students are given a heterogeneous test with a total of 125 possible points. The students are asked to answer all of the questions to the best of their abilities. Of the possible subsets of problems whose total point values add up to 100, a knapsack algorithm would determine which subset gives each student the highest possible score. “*

# What we need to get started?

The problem requires from the user three simple things:

* **the total weight of the Knapsack;**
* **the total number of items;**
* **designed items**;

The items have to be designed by the user. To do so, the user needs to assign an item with the following properties:

* **name** – for identification purposes;
* **weight** – shows the weight of each item individually;
* **value** – shows the value of each item individually;

The previous inputs will be used by the Knapsack function to solve the main task of the program which is the Knapsack Problem. The function can be represented explicitly in the following pseudocode:

1 *// Input:*

2 *// Values (stored in array v)*

3 *// Weights (stored in array Gram\_Weight)*

4 *// Number of distinct items (n)*

5 *// Knapsack capacity (W)*

6

7**for** i from 0 to n **do**:

8 **for** w from 0 to W **do**:

9 **if** i = 0 or w = 0 then:

10 K[i][w] = 0

11 **else if** Gram\_Weight[i-1] <= w then:

12 K[i][w] = max(value[i-1]-K[i-1][w-Gram\_Weight[i-1], K[i-1][w]);

13 **else**

14 K[i][w] = K[i-1][w]

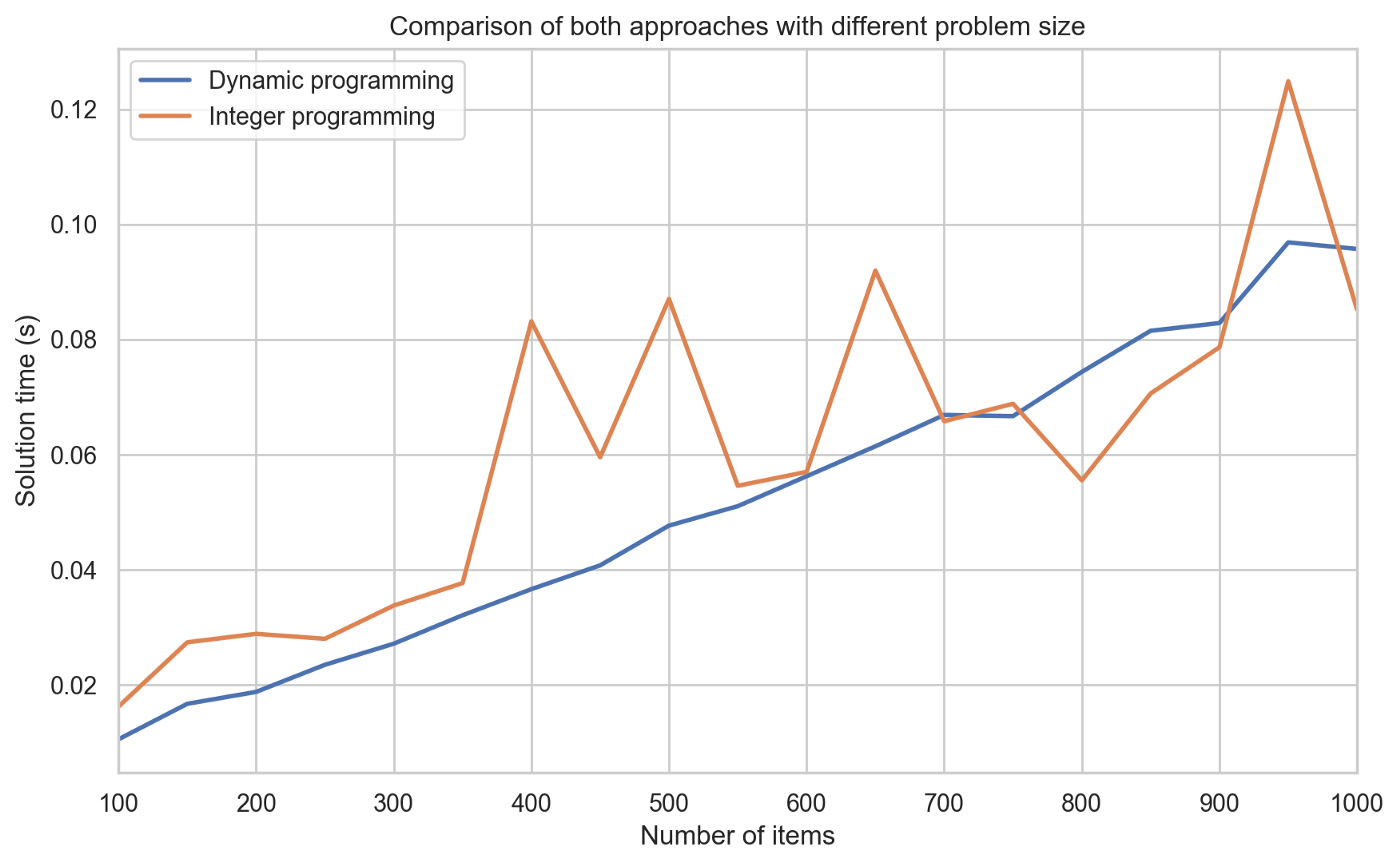
# The use of Dynamic Programming Why Dynamic and not Integer?

For an easier understanding firstly the difference of Dynamic Programming and Integer Programming must be understood.

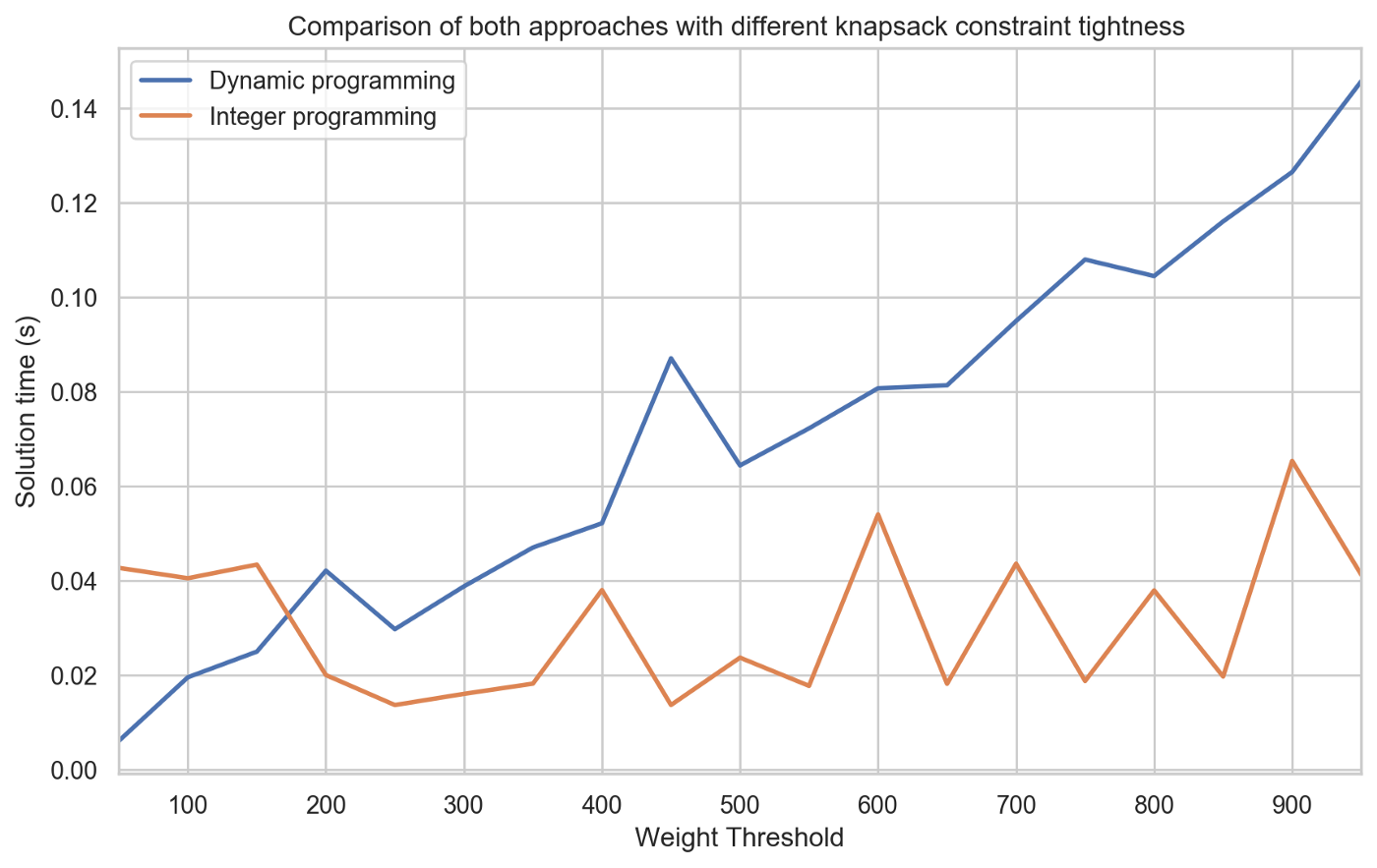
* **Dynamic Programming** is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions using a memory-based data structure;
* **Integer programming** (also referred as IP) is an operations research technique used when (typically) all the objectives and constraints are linear (in the variables) and when all the decision variables are integer or binary (i.e. 0 or 1);

Our program Knapsack function was built using Dynamic Programming, which is based on the idea that, in the optimal solution, a given item is either in the selected subset or not. This decision was made due to its efficiency while performing with larger numbers.

If the weight is a fixed number of 100 and we generate instances of Knapsack Problem with number of items ranging from 100 to 1000 the efficiency of the dynamic programming appears on the following plot:



Unfortunately, the result was not the same when it comes to weight because dynamic programming’s table size and number of iterations is directly proportional to the weight threshold, while integer programming is more directly affected by the number of integer variables and constraints. The following plot shows the difference when there are used 100 items and a random weight between 0 and 100:



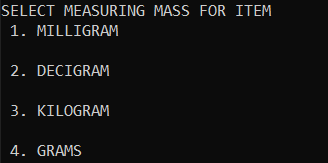
In conclusion, dynamic data was used in our C# program as it can be a solid choice regarding efficiency against integer programing but does not require a model set up and calling an external solver. Plus, dynamic programming has the bonus of the lookup table, which contains optimal solutions of the knapsack problem with different parameters.

# Running the program

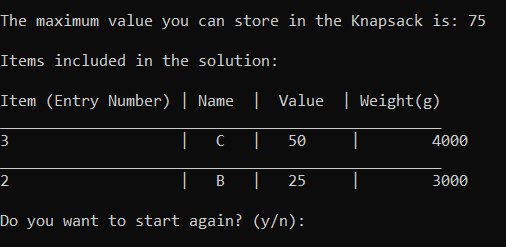
To present how our C# Knapsack Problem program works a simulation will be presented in the following section.  
Firstly, to get started the program requires the previously discussed inputs. The following values were chosen:

* **the total weight of the Knapsack: 7kg**
* **a total of 3 items:**
* **item 1:**
* **value: 10**
* **weight: 1kg**
* **name (reference letter): A**
* **item 2:**
* **value: 25**
* **weight: 3kg**
* **name (reference letter): B**
* **item 3:**
* **value: 50**
* **weight: 4kg**
* **name (reference letter): C**

During the process in which the user choses the numbers for each input, the program gives him a small selection of multiples and submultiples of weight. In this example all weights are chosen to be in kilograms. Also, the program converts all given weights in grams, result which is shown to the user.



After all the requested inputs were given the program gives the answer consisting of the maximum value that can be stored in the Knapsack and a table with the included items in the solution. The user is given the choice to repeat the process using different values.



In our case the maximum value that can be stored in the Knapsack is 75 and only item number 3 and 2 were used.

The whole process for this simulation including input writing and choosing the unit for each item the total time was 38.04 seconds.



# Reference

1. <https://en.wikipedia.org/wiki/Knapsack_problem>
2. <https://towardsdatascience.com/dynamic-program-vs-integer-program-which-one-is-better-for-the-knapsack-problem-759f41b9755d>
3. <https://en.wikipedia.org/wiki/Dynamic_programming>
4. <https://towardsdatascience.com/integer-programming-in-r-33ee6f48a3c8>

# Company Information

**Java Group**

Nixon Court, 33 Putney Road, LE2 7TG, Leicester, Leicestershire  
United Kingdom

