Assignment 3

**Assignment 1**

Assignment 1 with Made Data Results

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Total Value** | **Total Weight** | **Time** |
| **Result** | 326 | 67 | 0.001 Seconds |

Assignment 1 with Data Set 10 Results

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Total Value** | **Total Weight** | **Time** |
| **Result** | 13415886 | 6323699 | 0.0009 Seconds |

This algorithm runs really fast as you can see from the results above. However, it does not give you the optimal results for Data Set 10. I saw that I could use the built-in *filter()* function to iterate through all the values that are in the tuple through the function the user has created. Therefore, I have attempted to create a simple function that subtracts the weight of an item from the total capacity. However, this script wouldn’t be accurate without some sort of a standard. In other words, it is possible that the filter would just pick the first *n* number of items that filled up the bag. Therefore, I added a *value*:*weight* ratio to define the *item\_value* of an item. Therefore, the *filter()* function takes in a sorted list of items based on the *item\_value*.

**Simple Value Based Ratio**

Simple Value Based Ratio Template with Made Data

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Total Value** | **Total Weight** | **Time** |
| **Result** | 326 | 67 | 0.014 Seconds |

Simple Value Based Ratio Template with Data Set 10

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Total Value** | **Total Weight** | **Time** |
| **Result** | 13415886 | 6323699 | 0.0229 Seconds |

This code was provided by Professor Wilck.

**Enumeration**

Enumeration with Made Data Results

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Total Value** | **Total Weight** | **Time** |
| **Result** | X | X | >Hour |

Enumeration with Data Set 10 Results

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Total Value** | **Total Weight** | **Time** |
| **Result** | 13549094 | 6402560 | 24.438 Seconds |

This code was provided by Professor Wilck.

**Recursive**

Recursive with Made Data Results

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Total Value** | **Total Weight** | **Time** |
| **Result** | 326 | 67 | 0.025 Seconds |

Recursive with Data Set 10 Results

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Total Value** | **Total Weight** | **Time** |
| **Result** | 13549094 | 6402560 | 35.78 Seconds |

As a recursive method of the knapsack problem, I have created a code that iterates through the list of items within a function called *pack(items, cap)*. This function returns an empty tuple if the input is *not* items. Then, if the input has both items and cap, but there is nothing in the *bag*, the function splits the list of items in to two lists, *initial* and *later*, where *initial* is the first list within *items* and *later* being the rest of the items. Then the function defines two other variables *put* and *dont\_put* which again, calls in the pack function. *put* is an addition of the first item in the list, *initial*, and the function *pack(items, cap)* so we are adding the item into our *bag* (explained more later), whereas *dont\_put* is a *pack(item, cap)* with no addition. Then the code moves on to another *if/else* statement. The *results* are an addition of the *initial* items and *pack(later, cap - initial[1]),* which we are running through the *pack* function again but this time without the *initial* item. This addition is running only if the *total\_value* with items within the pack is greater than the *total\_value* of the items without the item. Otherwise, we just keep *dont\_put*.

What is really interesting about this code, is the speed and the accuracy of the results. The code ran slowest for Data Set 10, compared to other codes provided by Professor Wilck and created by me. Nevertheless, the time difference is small between the two codes, and the results are the same for both codes for Data Set 10.

**Final Thoughts**

I thought that the results are rather interesting. The recursive code I have created slight time difference, actually ran slower than the *Enumeration* code provided by Professor Wilck for Data Set 10. However, it was **much** quicker than the *Enumeration code* provided by Professor Wilck when running on the dataset I have created. I think this is due to sheer simplicity of the recursive code that I have created compared to the total *enumeration* of the combination of items. However, the numbers are much larger for each items’ values and weight for Data Set 10 than the data set I have created. Nevertheless, I believe that the range between the values and weights of items are much larger for the dataset created by me, so when we do run the *Enumeration* code, it takes longer to find the optimal solution. In fact, I wasn’t able to get the results from the *enumeration* code because it ran for hours.

Another interesting thing we could point out, is the results for the newly created dataset. For all three results I could get, the output was the same, with all three codes running under a second. However, I wasn’t able to get the results for *enumeration* due to a time constraint. It would be interesting to find the “optimal” solution for this dataset, but we could be an educated guess that the result of Total Value of 326 and Total weight of 67 is the optimal solution since three of the codes gave me the same result.

It is rather difficult to explain the recursive codes in words especially because the function is called within a function. However, I tried my best trying to explain the function. I also actually had this code to submit for Assignment 1, but it was much slower than the submitted code for Assignment 1. Therefore, I actually didn’t submit this code for Assignment 1.

\*Data set was created by using the code shared by Dylan.