**CHAPTER 1**

**1.1 INTRODUCTION**

A greedy algorithm is the most straightforward approach to solving the knapsack problem, in that it is a one-pass algorithm that constructs a single final solution. At each stage of the problem, the greedy algorithm picks the option that is locally optimal, meaning it looks like the most suitable option right now. It does not revise its previous choices as it progresses through our data set.

The steps of the algorithm we'll use to solve our knapsack problem are:

1. Sort items by worth, in descending order.
2. Start with the highest worth item. Put items into the bag until the next item on the list cannot fit.
3. Try to fill any remaining capacity with the next item on the list that can fit.

We always start by figuring out what the next most important question is. In this case, there are three main operations we need to figure out how to do:

* Sort items by worth.
* Put an item in the bag.
* Check to see if the bag is full.

**CHAPTER 2**

**2.1 LITERATURE REVIEW**

As a programmer, we tend to solve as many problems as possible using data analysis, algorithms and codes. Since we are computer science student we tend to lean in that direction to solve a common backpacking problem. We decided to use a simple greedy algorithm to solve it so we could test the theories we learnt in class. We also browsed many different sites regarding the backpacking problem and decided to go with the Greedy approach as this approach seemed more suited to the current scenario. We researched different thesis and algorithms for the backpacking problem, but according to what we learnt and what we thought was appropriate, we chose to solve the backpacking problem by using the greedy approach to solve the knapsack problem.

**CHAPTER 3**

**3.1 PROBLEM STATEMENT**

While travelling there is often a dilemma where you have to consider the cabin baggage weight limit at the airport and according to that you have to choose between different items you want. As a practical exercise, deciding what to leave behind (or get rid of altogether) entails laying out all one’s things and choosing which ones to keep. That decision is based on the item's usefulness (its worth) and its weight.

**CHAPTER 4**

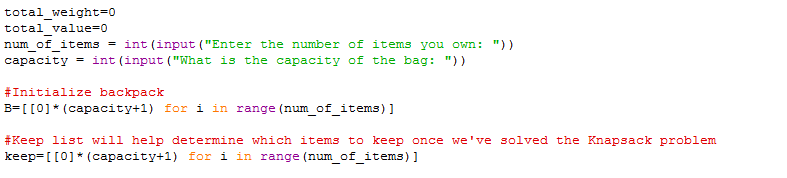
**4.1 SYSTEM DESIGN**

This code is coded in python and requires a python platform or an online compiler to run.

The main part of the entire project code is where we sort out the items based on their weight and value.

We won’t be using modules or packages, as we have tried to implement it in the simplest format possible to avoid complications.

**STEP 1:** The first step is to initialize all the lists and variable we need for the future use.

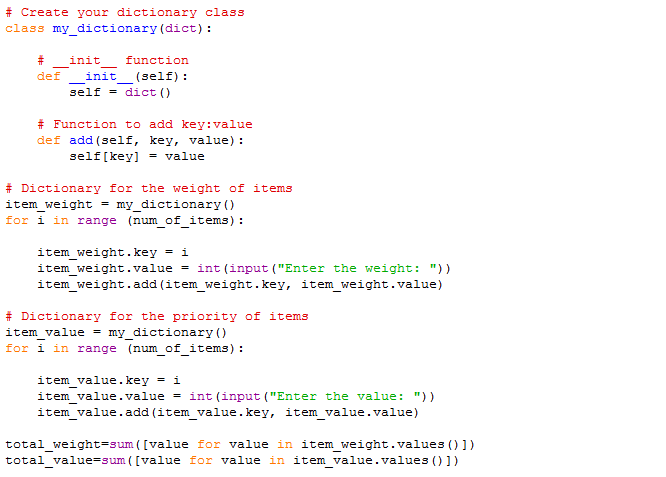
****

*Fig 4.1.1*step 1 code snippet

**STEP 2:** The next step is to create dictionary. Since we will be needing key and values for this, we shall create a class for it first. And then add the functions key and values.

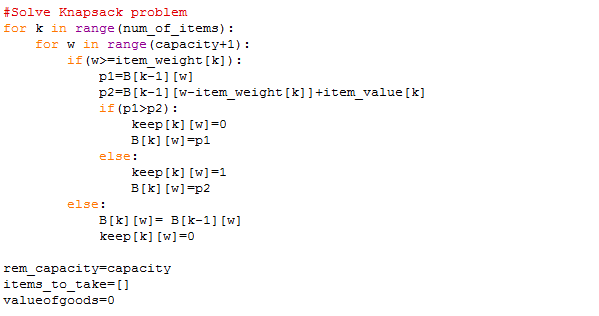
We use this dictionary class to further Create 2 dictionaries which will be used to store the weights and the priority of the items respectively.

In this we have started indexing the items from 0,1,2,….

****

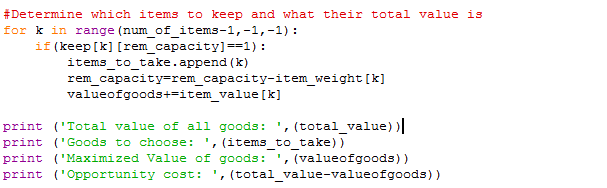
*Fig 4.1.2* Step 2 code snippet

**STEP 3:** The next step is the man step that performs the knapsack solving.

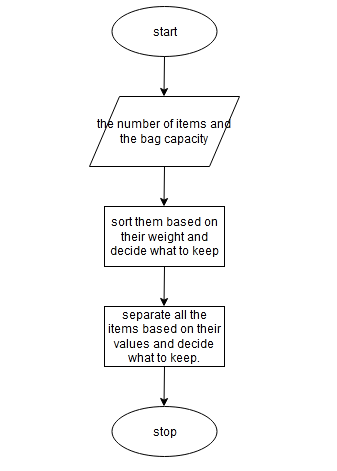
****

*Fig 4.1.3* Step 3 code snippet

**STEP 4:** The next part will be determining which items are to be kept and what their value is.

****

*Fig 4.1.4* Step 4 code snippet

****

*Fig 4.1.5* A flow chart of the basic working of the algorithm.

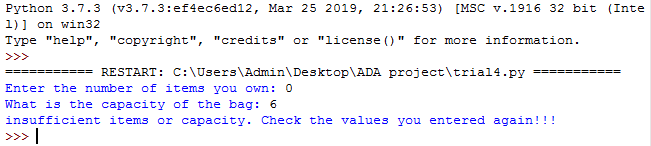
**4.2 TEST CASES**

Every algorithm has different test cases to satisfy. It depends on the problem, algorithm and the requirements to be satisfied. In this knapsack problem one of the major test cases is where if the number of items or the capacity given by the user is < 0 (less than 0).

**Test Case 1:**

If the number of items is 0, then the output would be insufficient items.

The output is a success.

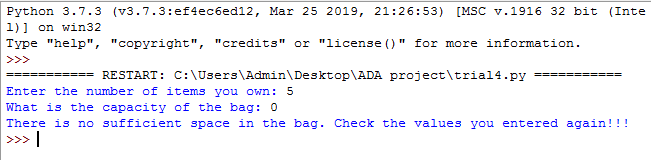


*Fig 4.2.1* Output of test case 1

**Test case 2:**

If the capacity of the bag is entered as 0 then the output should display that there isn’t enough space in the bag.

The output is a success.

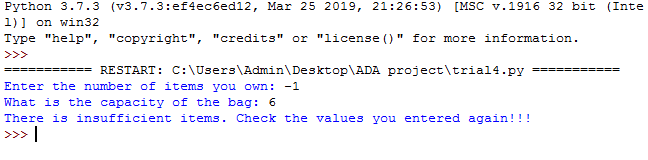


*Fig 4.2.2* Output for test case 2

**Test case 3:**

Lets say the number of items entered by the user is in negative value, then the algorithm must show a message saying insufficient items.

The output is a success.

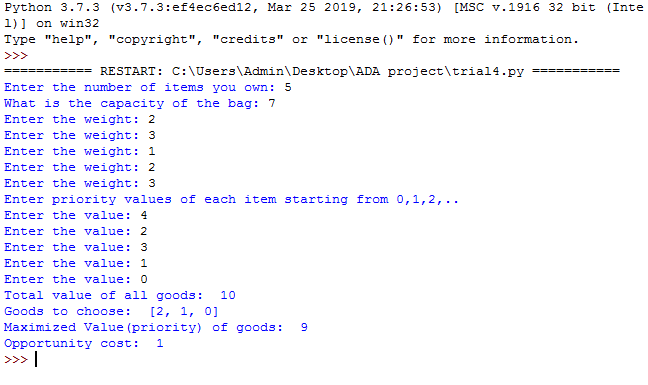


*Fig 4.2.3* Output of test case 3

**Test case 4:**

Let’s say the capacity and the number of items are >0, then the algorithm should work fine and give a suitable output.

The output is a success.



*Fig 4.2.4* Output of test case 4

**CHAPTER 5**

**5.1 EXPECTED ADVANTAGES AND LIMITATIONS**

Finding solution is quite easy with a greedy algorithm for a problem.

Analysing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer).

The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science.

The Greedy Algorithm does not solve the problem to optimality. It rather finds a local optimal solution.

It operates in linear time, which is extremely efficient. It will occasionally produce the optimal result.

**CHAPTER 6**

**6.1 CONCLUSION AND FUTURE ENHANCEMENTS**

We have made use of the greedy algorithm to solve the knapsack problem i.e. the backpacking problem. Part of choosing the right approach to solving any programming problem is taking into account the size of the input data set. In this case, it's a small one. In this scenario, a one-pass greedy algorithm will always be faster and less resource-needy simply because it has fewer steps. "Better," of course, is a subjective judgment. If speed and low resource usage is our success metric, then the greedy algorithm is clearly better. If the total worth of items in the bag is our success metric, then dynamic programming is clearly better. However, our scenario is a practical one, and only one of these algorithm designs returned an answer we would choose.

There are many scopes for future enhancements. Like I mentioned above dynamic programming is an alternative choice. Where dynamic programming on small data sets lacks in performance, it makes up in optimization. The question then becomes whether that additional optimization is worth the performance cost. The usage of genetic algorithm can improve the performance. (i.e. Generate Population, Include Greedy Solution, Selection, Crossover, Mutation ).

**REFERRENCES**

1. <https://www.wits.ac.za/media/migration/files/cs-38933-fix/migrated-pdf/pdfs-2/KnapsackShelfSpace_AllocationProblem.pdf>
2. <http://www.c4learn.com/c-programs/program-to-implement-knapsack-problem.html>
3. <https://www.sanfoundry.com/solve-fractional-knapsack-problem-using-greedy-algorithm/>