Durham University International Study Centre

SUMMATIVE ASSIGNMENT FRONT COVER SHEET

Complete all shaded areas:

Student ID	2499888
CIS Username	kxmz59
Teaching Group	SFS5253_CSAPP
Tutor	Chris Roberts

Assignment Title	A3: Computer Science Project Report
Assignment Deadline	Tuesday 26th May @ 13:00 BST

It is very important that any work you present as yours must in fact be your own work, and not taken from another place or done by another person. Cheating, collusion (working together with another person on an assessment which is not intended to be collaborative) and copying from unacknowledged sources (plagiarism) are all serious offences and must be avoided.

DEFINITION OF ACADEMIC IMPROPRIETY
Academic impropriety is a term that covers cheating, attempts to cheat, plagiarism, collusion and any other attempts to gain an unfair advantage in assessments. Assessments include all forms of written work, presentations and all forms of examination. Academic impropriety, in any form, is a serious offence and the penalties imposed would reflect this.

DECLARATION
By entering my Student ID below I confirm that this piece of work is a result of my own work except where it forms an assessment based on group project work. In the case of a group project, the work has been prepared in collaboration with other members of the group. Material from the work of others not involved in the project has been acknowledged and quotations and paraphrases suitably indicated. Furthermore, I confirm that I understand the definition of Academic Impropriety that is used by Durham University International Study Centre.

Student ID: 2499888	Date: 25th May 2020

DUSIC IFY Computer Science with advanced Physics with Project

A3: Computer Science Project

The Rucksack Problem

Student ID: 2499888

Date: 2020-04-28

The Rucksack Problem

1 Introduction

The aim of the projects is to run and test different algorithms to find an optimal solution for the 0-1 knapsack problem.

1.1 Specification

The minimum specification of the program was as follows:

- The solution must be written using the Python programming language
- The solution must be portable (i.e. work on Linux, Microsoft Windows, etc)
- The solution must use the import function
- The solution must have parsed user input
- The solution must use a range of data types, including lists
- The solution must use at least three user defined subroutines
- The solution must import data from a csv file in the given format
- The solution must include an analysis of each algorithm, including the computational time taken to complete
- The solution must exit the program in a controlled fashion

1.2 Rucksack Problem - Theory

The knapsack problem is a popular problem that requires one to determine from a group of items with mass m, and value v, how many of these items could be included in a collection such that the total mass does not exceed or is equal to a given limit while the value is as large as possible [1]. The name originated from a daily problem faced by a person who is limited to a fixed size rucksack and must fill it with the most precious items.

1.2.1 Applications

The knapsack problem could be used as a prototype to solve other practical problems. Some of which include

- A shipping company, without exceeding the weight capacity, trying to fit as much packages into a container
- A sports club trying to build a team by recruiting the most talented players without exceeding the salary limit
- This problem is also usually seen in resource allocation also where decisions have to be taken on the whether or not to choose a project or task from a set of them with a fixed budget or time constraint.

A very practical example framed in to our everyday life is as described below

Ben is going on a journey to the desert with Tim, and he has offered to carry their food. He can carry a maximum quantity of 8kg and has 4000 cm^3 of space available in his lunch bag. He can only pick from this collection of supplies.

Food Item	Weight	Volume	Calories
Oranges	300g	$120cm^3$	100 calories
Apples	100g	$200cm^3$	120 calories
Chicken	500g	$400cm^3$	4000 calories
Beef	600g	$500cm^3$	5000 calories
Rice	300g	$500cm^3$	3000 calories
Potatoes	400g	$300cm^3$	8000 calories

With the constraints, Ben now has to decide what is the maximum number of calories he could pack considering the fact they are going on an expedition to a desert.

The knapsack problem belongs to a class of problems called Nondeterministic Polynomial time problems ,NP problems for short. This particular problem is NP-hard. This means there is no polynomial time algorithm to solve it at the moment. This has led to attempts to use this problem as a means for public-key encryption. This however failed on several occasions since they were solvable by a pseudo polynomial time algorithm These problems push a computer to go through various stages to find a solution, while the number increases with increasing input. Some NP problems like the knapsack problem have special characteristics. In the 20th century, Stephen Cook and Richard Karp showed that NP problems could be changed to a single problem of formal logic. This means that if one problem could be solved and verified with an algorithm they all could. This characteristic is known as NP completeness.

1.2.2 P, NP, NP-Hard, NP-Complete Problems

In 2000, the Clay Institute offered 1 Million Dollars each for the solution to seven key problems in Math, The Millennium Prize Problem. The NP Vs P problem is one of them [2]. In computer science, programs are usually grouped based on how hard the problem is to solve. Some problems are quite easy. For example adding a set of integers. This is easy to solve and verify if the proposed answer is correct. These group of problems are called P problems. More of such problems are finding finding the least common multiple of a number or the product of two numbers.

There are also problems that may or may not be hard to solve but are easy to verify or check. These problems have unknown difficulty. These problems are known as NP problems. Lets consider a jigsaw puzzle for example. It is very easy to verify the solution. You only need to go through each puzzle piece and verify that it is properly connected and look at the picture portrayed by the puzzle. However finding out if an unsolved jigsaw puzzle is solvable and how hard it is, is unknown. To determine whether or not it is,all possible combinations are to be tested. If no configuration produces a solution then it can be said to be unsolvable. However this may take a lot of time. Determining whether there is a faster way is unknown.

We can conclude that P is a subset of NP. The P problems we see today were once NP problems. Since a solution can be solved in polynomial time, it can also be verified in polynomial time. The million dollar question is whether the reverse is true. Is NP = P? If NP and P were equal this may mean that problems previously classified as hard are actually easy. We would therefore quickly be able to find math proofs to problems. It would also put online security at risk as it may be vulnerable to attacks.All security algorithms fall under NP.

In simple terms NP is thought of as a hard problem while P is easy? However these are not very mathematical terms. Easy problems are solutions that run in polynomial time. In the jigsaw puzzle verifying the solution takes polynomial time. However it is not known whether there is polynomial time algorithm to solve the jigsaw. Hence it is an NP problem. Other forms of NP-Class problems include the Travelling Salesman problem,Su Do Ku and more.

A problem is NP hard if all other problems in NP can be reduced to it.

NP-C or Non-deterministic Polynomial time Complete problems are the hardest type of NP problems [3]. They are problems that are NP and NP hard or is the intersection of the NP and NP-Hard Class. These are NP problems that are reduced to NP-Hard which are also NP.

NP Complete problems are Decision problems and NP-Hard problems are optimization problems. If a problem A is a Decision problem and problem B is an optimization problem. Problem A can be reduced to Problem B. For example Knapsack decision problems can be converted to Knapsack optimization problems.

The Knapsack Problem is an NP complete problem [4]. In oreder to understand it lets consider Satisfiability. Satisfiability via the Cook Levin Theorem was proven to be NP complete. This SAT problem for short was then reduced to a 3SAT problem. Since any SAT problem can be expressed as 3SAT an algorithm that

solves 3SAT must solve SAT. Thus, any 3SAT problem can be reduced to a Subset Sum problem. Similarly, Subset Sum is NP-Complete. Then a Subset Sum problem can be expressed as a 0–1 Knapsack problem. This means 0–1 Knapsack is NP-Complete [5].

1.2.3 Types of Knapsack problem

There are two types of knapsack problem.

- Fractional Knapsack Problem
- 0 -1 Knapsack problem

The fractional knapsack problem allows us to take fractions of the respective items. This means that the items are divisible. This problem can be solved by using an algorithm known as the greedy method which was first published by George Dantzig in 1957 [6].

The 0 -1 Knapsack Problem we either take the whole item or not. Fractions of the items cannot be taken or in other words is is not divisible. There are other notable mentions such as the multiple constraint knapsack problem [7]. In this project, the 0 -1 knapsack problem is the main focus.

There are multiple methods of solving this problem. In this project, the dynamic programming approach, brute force approach and the greedy method approach is tested. Each method has a different time complexity. Time complexity is the time it takes the algorithm to find a solution to the problem. Complexity can also be described as a measure of the efficiency of an algorithm in terms of time usage because each algorithm takes a while. There are different types of time complexity. Polynomial (Constant, Linear, Quadratic, Logarithmic) and exponential are examples of running times. This is also defined as the Big-O Notation.

- Constant time complexity: This takes the same time no matter the size of the input.
- Logarithmic complexity O(log n): Time grows slower as number of operations increases. An example of this is binary search
- Linear time complexity O(n): Time grows linearly with increasing input and number of operations. An example of this algorithm is iteration.
- Linearithmic complexity O(n log n): This is a mix of both logarithmic and linear time complexity. An example of this is quicksort
- Quadratic time complexity O(n^2): The computation time grows as a square of the number of inputs. An example of such is having an iteration inside another iteration.
- Cubic time complexity O(n^3): The time grows even faster here due to the presence of multiple nested iterations. When you have operations that are powers of the input it is commonly referred to as running in polynomial time.
- Exponential complexity O(2^n): This time increases exponentially for every added input. For every new element added the time increases twice as much. Algorithms with such complexity may take forever to find solutions. An example of this can be the calculation of Fibonacci numbers.

The Dynamic Programming approach has pseudo polynomial time complexity. The greedy method has a logarithmic time complexity. The brute force method has exponential time complexity.

1.2.4 Dynamic Programming Approach

Dynamic programming is a method of solving optimization problems by breaking down a complex problem into smaller problems. It is implemented to problems that have overlapping sub problems. These sub-problems are stored and used later on to reduce the number of computations. Storing these sub-problems and their solutions is known as Memoization. These sub-problems are stored so the program does not need to calculate these problems over and over again.

The knapsack problem can be solved using this approach. In order to solve this problem you need to create a table of values or a two dimensional array. The row of the table represents the individual items and the columns represents the maximum capacities our knapsack can take. So if we have a knapsack of Capacity 5 with 4 items. This is the table we obtain

Item/Capacity	0	1	2	3	4	5
Item 1						
Item 2						
Item 3						
Item 4						

In order to solve this problem we have to consider the first item and figure out the best possible value, we could get with each capacity. Then after we consider the second item and reuse the information derived from the first item to find the combined maximum. What we do for each column and row is to figure out whether or not we should include the current item.

If we do not include the current item then we look one row above and take the item directly above the current position. We take the item above, in order to subtract it from the total capacity and get its value. This is done when the current item is unable to fit or the item directly above the current position provides a maximum value instead.

If we include the current item, we subtract that item from the total weight and take the item above, which is able to fit,and add its value also.That is when we subtract the weight of the current item we move the number of columns that the weight takes away from our available space and move to the row directly above the capacity available and add that value. This is done when the item is able to fit and provides a maximum value. Filling the table above with the following items

Item	Weight	Value
1	5	60
2	3	50
3	4	70
4	2	30

Item/Capacity	0	1	2	3	4	5
Item 1	0	0	0	0	0	60
Item 2	0	0	0	50	50	60
Item 3	0	0	0	50	70	70
Item 4	0	0	30	50	70	80

Considering we used n, number of items with W capacity ,it takes O(nW) to construct the two dimensional array. It takes O(1) to fill in each cell of the table and O(n) to trace back the items and discover the optimal solution. In total this algorithm takes O(nW) + O(1) + O(n) time. Therefore the time complexity of the Dynamic Approach is O(nW). Although it appears to have polynomial time complexity it is pseudo polynomial. This would be explained in the conclusion.

1.2.5 Brute Force Approach

The brute force approach is another popular method used to find the optimal solution for the knapsack problem. The brute force approach takes all the items and finds all the possible combinations of the list of items. If there are n items, then there are 2^n possible items. It then calculates the total weight of the respective combinations. If the weight exceeds the knapsack limit it is not considered and discarded. The total profit for the individual subsets is then calculated. The group of items with the maximum profit is the optimal solution. The disadvantage with this method however is the fact that it has exponential time complexity, $O(2^n)$. The exponent increases with increasing elements in the set. This means that sets with large number of elements could take hours and even days to discover the optimal solution. This may also end up filling the computers memory in the process.

1.2.6 Greedy method approach

The greedy method approach is another approach popularly implemented in the fractional knapasack problem. The algorithm takes all the items available and finds its profit per unit weight. The items are then sorted out in descending order of profit per unit weight and summed. The whole items are summed consecutively one by one till the next whole item cannot be included. Thus a fraction of this item is then included so that it would not exceed the knapsack limit. This method is really popular and has logarithmic time complexity, thus faster than the brute force approach. However this method cannot be implemented in the 0-1 knapsack problem. It would not provide the optimal solution since fractions of items cannot be included. This algorithm was however tested. In this algorithm, for n times, the first step has a time complexity of

O(n) . It then takes O(nlogn) time to sort, while to pick the selected items it takes O(n) time. Overall it has a logarithmic time complexity.

2 Design

2.1 System Parameters/Variables

The following are the variables included in the program.

`w_I`	list of weights of items
`Val`	list of Values of items
`Cap_k`	Capacity of items
`NoOfItems`	Total number of items
`tab`	Dynamic programming table

2.2 Equations

Given a set of items with values $\{ v1, v2, v3, ...v_n \}$ and weights $\{ w1, w2, w3, ...w_n \}$ with Capacity of knapsack W, we need to determine the subset $P \subseteq \{ 1, 2...n \}$ of items to store.

Maximizes $\sum_{i \in P} v_i$ subject to $\sum_{i \in P} w_i < W$

2.2.1 Dynamic Programming Formula

Using the method of dynamic programming the basic formula used to pick the maximum value for each row is outlined below

$$P[i][w] = Max(P[i-1][w]; v_i + P[i-1][W - w_i]) \tag{1}$$

Where i represents the item, w represents the weight of that item, v represents the value of that item and W represents the Capacity of the knapsack.

Initially all the values in the table are set to 0. The program then starts to input the values in the table.Note that the program has two decision. It either takes the item or leaves it.

If we decide to leave the item, we then have to choose the item directly above it. The first part of the equation $P[i-1][w]$ executes this command. When the program starts the first row is automatically filled with zeros. So when you leave the first item the value assigned is 0. This happens to be the case when the item cannot fit into the knapsack.

If we decide to pick the item, we add its value to the previous items maximum value and subtract the weight. This is only possible if the item is capable of entering the knapsack. This occurs in the second part of the equation. $v_i + P[i-1][W - w_i]$ The v_i represents the value of the current item. This $P[i-1][W - w_i]$ represents the computed value for the previous items that are capable of entering into the knapsack at the moment.

This computation continues till all the items are considered with the total limit of the knapsack. This value is the optimal solution. This is the maximum profit that could be computed considering all items. After the values considered are added to list and printed out.

2.2.2 Greedy Method Formula

Using the greedy method, the formulas used are outlined below.

$$P = \frac{V_i}{w_i} \tag{2}$$

Where `P` represents the **profit per unit weight** of an item. V_i is the **Value** of an item and w_i is the **weight** of an item.

$$b_i = P_i \times w_i \times \chi \tag{3}$$

Where b_i represents the **benefit** or **value** obtained from the item, P_i is the **profit per unit weight**, w_i is the **weight** of the item and x is such that$(0 < x < 1)$. X is equal to 1 if item is **taken** and 0 if it is **not taken**.

2.3 Problem Decomposition

In the main program, we are presented with a main menu and the initial variables are initialized. The main menu provides us with a list of options from selecting the CSV file, enter the values, running the program, clearing all the items and quitting. If the user picks the option run program, another 3 options are provided. He is given 3 options to choose from. These options are the various methods capable of finding the optimal solution.

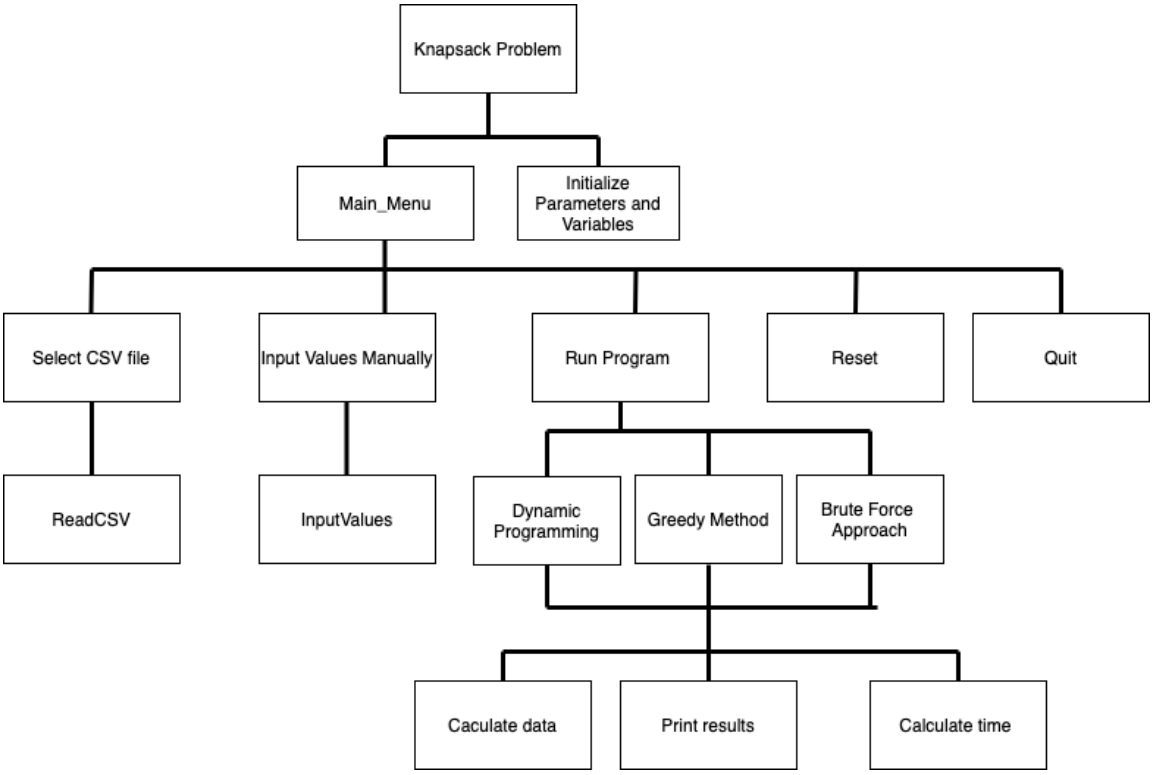

Figure 1: Diagram Structure

2.4 Test Plan

Tabulated below are test plans from the various tests. This has been divided into 3 tables for clarity.

2.4.1 Dynamic Programming test plan

Test No.	Test Description	Test Data	Expected Outcome	Actual Outcome	Improvements?
1	Test that function imports correct CSV values	example.csv	Prints the arrays for the weights of the item and their corresponding values		
2	Test non existing CSV Files	eaple.csv	print File does not exist and allow user to input another file.		
3	Test the Initial Dynamic Programming table	example.csv	Prints a 6x11 table filled with zeros		
4	Test invalid data to see if program can handle it	w	Print Invalid format with option to type a new value		
5	Test calculations of Dynamic Programming Table	example.csv	Print a 6x11 array filled with values		
6	Check Optimal Solution	example.csv	Print weight =10, Print profit = 16		
7	Print Time complexity of Dynamic Programming Method	example.csv	Print time in Nanoseconds		
8	Test Dynamic Programming Function Call	call DynamicProgramming()	Print selected item along with the optimal solution and time complexity		

2.4.2 Greedy Method test plan

Test No.	Test Description	Test Data	Expected Outcome	Actual Outcome	Improvements?
1	Test sorted values	example.csv	Print a list of sorted values in descending order of profit per weight		
2	Loop through maximums	example.csv	Loop through the list of items sorted in descending order of profit per unit weight of item		
3	Sum whole items capable of entering	example.csv	print table of items consisting of only whole items		
4	Sum whole items and fraction of item to fill knapsack	example.csv	print table of items consisting of both whole and fractional items		
5	Check Optimal Solution	knapsack_test_01.csv	print 9932		

2.4.3 Brute Force Approach test plan

Test No.	Test Description	Test Data	Expected Outcome	Actual Outcome	Improvements?
1	Test that sets are formed from csv file	example.csv	Print a list of weight and values for each corresponding item		
2	Print all possible sets from Universal set	example.csv	Print 32 sets		
3	Print value and corresponding weight of items selected	example.csv	print items 1,3,4 with corresponding weight,values		
4	Print optimal solution	example.csv	Print Best Profit = 16, print total weight = 10		

3 Implementation

Below is a flow chart of the simulation. Figure 1 is a flow chart of the main program while figure 2 is the flow chart of the option Run program.

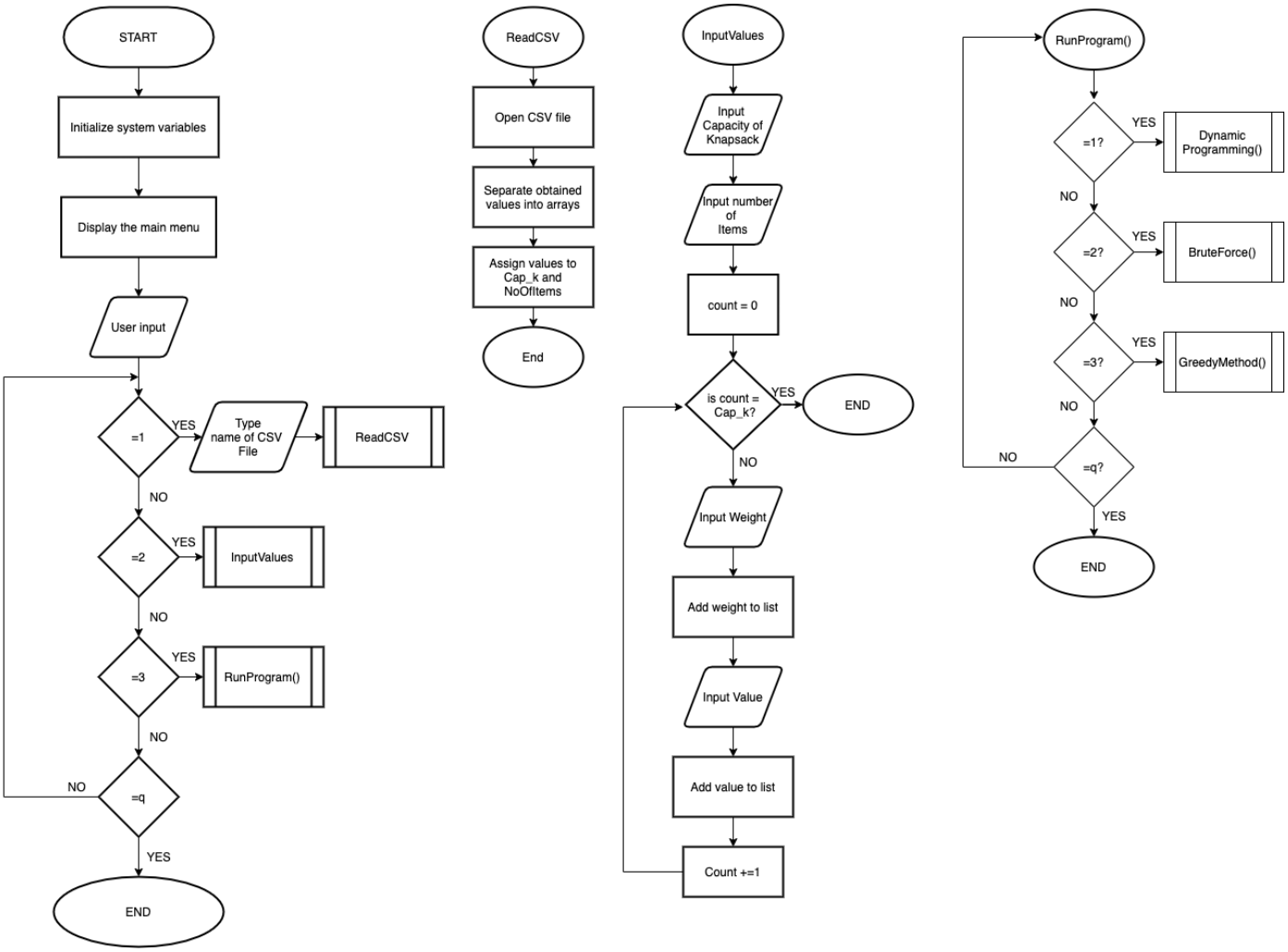

Figure 2: Flowchart: Main Menu

3.1 Data User Input

[1–3] Import csv, time and os library.

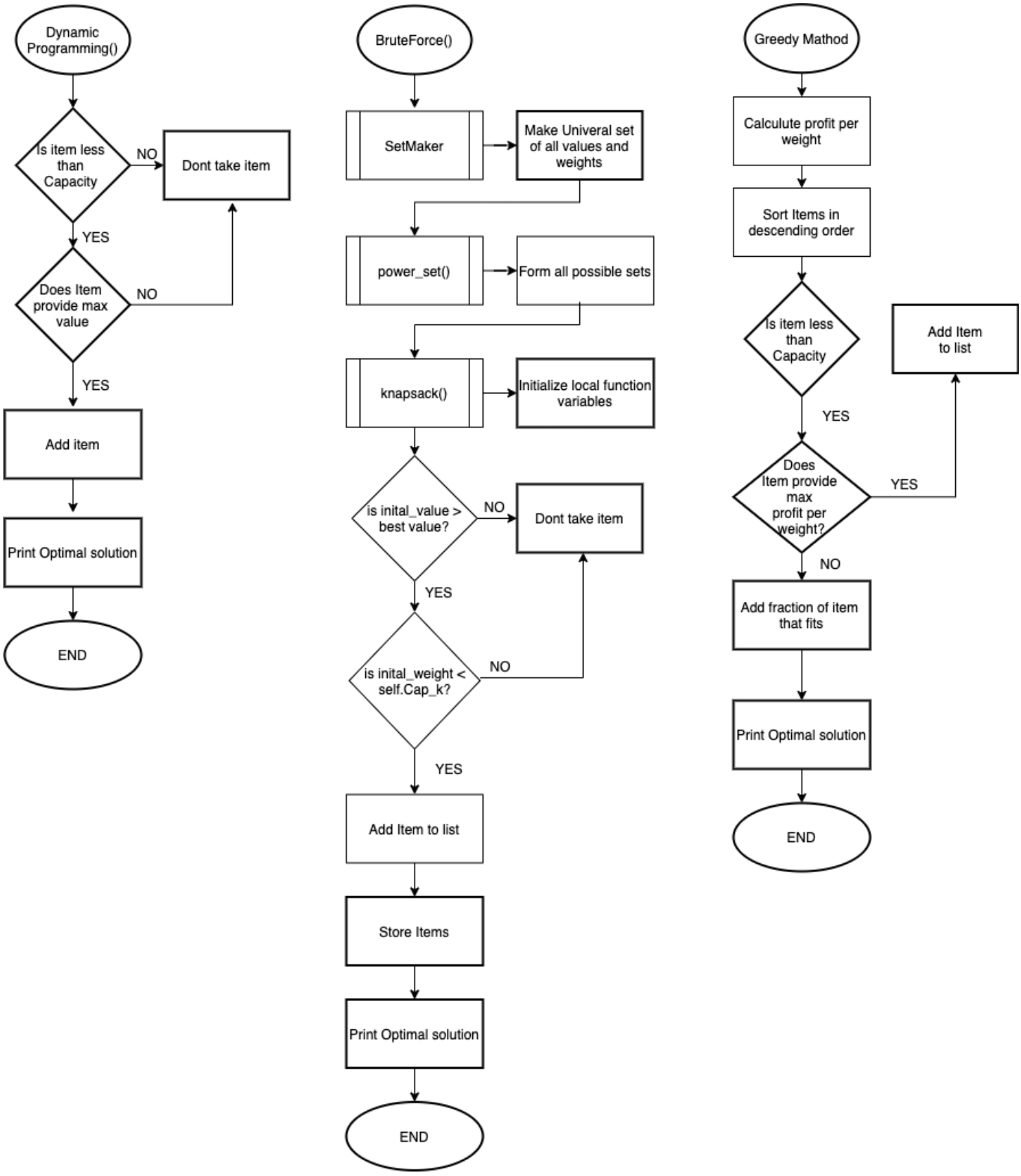

Figure 3: Flowchart: Run Program

[5–12] Initialize system variables.

[14–18] Welcomes the user to the program, Prints the name of the simulator, student ID, Date and project name

3.1.1 Input Value Function

[62–91] The `InputValues()` function enables the user to input in the values he or she decides to use manually. He or she is required to choose the number of items, the limit of the knapsack and the weights and values of the respective items. The table for the Dynamic programming table is also initialized.

[75–90] This `while loop` was created in order for all the values and weights of the respective items to be taken. If all the values are not entered the while loop shall not end. The `try except clause` was used to avoid run-time errors when invalid inputs are entered. This is done till all valid items values and weights are entered.

3.1.2 Read CSV File function

[30–53] The `ReadCSV` function reads the csv file you intend to use.It then takes the values and stores them in their respective lists. Weights are stored in the `w_I` list and the items respective values are stored in the `Val` list. The Capacity of the knapsack is then stored in the `Cap_k` variable. This is the first item in the csv file. The number of values is then calculated and stored in the `NoOfItems` variables. The values are then returned for calculation.

3.2 Clear terminal screen function

[21–28] The `clear()` function is used to clear the screen of the server. When the program is running the terminal may look quite messy after while. Clearing the terminal screen makes the program look neat and easy to run. Since there are different commands for each operating system if one command is assigned it may only work on one Server. The program checks what Server it is being run on and passes the correct function command.

3.3 Try, except function

[55–61] The `failproof()` function was created purposely for invalid input to avoid `run-time errors`. When the user enters any other input besides an integer such as a word, the program prints `'Invalid Input'` and is given the chance to enter a new input.

3.4 Main Program loop

[93–100] The `Main_Menu()` displays the main menu of the program. It provides the user with 5 options to choose from.

[102–106] The `RunProgram()` displays the menu for the run program. It enables you to choose what method you would like to use to find the optimal solution.

[252–321] This is where the main program is found. A while loop is used to run this program. The program loops till the user enters **'q'** which will end the loop exiting the program in a controlled fashion. The `Main_menu` function is called first. This displays the main menu system for the various algorithms. The user is given the option to make a choice. The user has to type a number between the range of options provided. Based on the users choice a series of events would be triggered.

[257–266] If the user types **1**, the user intends to import a CSV file. He is asked to type the name of the CSV file.If the CSV file is found it then triggers the `ReadCSV` function. If it is not found it prints **file not available** then returns to the main menu.

[268–270] If the user types **2**, the user intends to input the values. The `InputValues` function is then called.

[272–377] If the user types **3**, he intends to run the program. When this is done a while loop is triggered. This loop contains the algorithms the user intends to use to calculate the data. Firstly a menu option is displayed indicating the algorithms available to be used. The user is giving the option to choose a number between the choices available.

[275–277]If the user picks **1**, he intends to use the Dynamic programming method. The `DynamicProgramming()` functions is then called.

[280–301]If the user picks **2**, The user intends to use the brute force approach. The algorithm first checks that items are available. If no items are available **Items are not available** is returned. If items are available the `BruteForce` class is assigned and the respective functions used to implement this algorithm are also called likewise. The data obtained is then printed along with the optimal solution.

[304–306]If the user picks **3**, the user intends to use the greedy method to determine the optimal solution. The `GreedyMethod()` function is then called.

[314–321]On the main menu, If the user wants to reset or clear the list of items, he or she just needs to type in the number **4**. All the items would be cleared and all the initial variables would be set to 0 to have a new list of items.

[310–312]If the user intends to return to the main menu, the user enters **q**.

If the user intends to end the program also all he needs to enter is the letter **q** and the program will end in a controlled fashion.

3.4.1 Dynamic Programming Function

[108–143] The `DynamicProgramming()` function uses dynamic programming to find the optimal solution. The program creates a table with the number of items as rows and the limit to the knapsack as columns.This is basically a 2x2 array. This is formed using this syntax `tab = [[0 for x in range(Cap_k + 1)] for x in range(NoOfItems + 1)]`. When the values are accepted it is added to the table. This is done till all the items are considered. Note that a counter(`time.perf_counter_ns()`) is placed at the beginning and end of the algorithm to time its speed in nanoseconds.

[110-112] Algorithm checks whether items are present. If `NoOfItems == 0` this means that there are no items present. Algorithm print `'No Items are present'`.

[114–117]If items are detected it jumps the `if statement` and goes to the `else statement` . In this statement using two for loops: `for i in range(1, NoOfItems + 1)` and `for w in range(1, Cap_k + 1)` we can interact with the 2x2 array. We can either decide to add the value of an item to a cell or not based on the constraints. If the items is less than the knapsack limit, it adds the item to the previous item. If not it takes the values of the previous items, it compares the two values and takes the maximum value. When the maximum is chosen it is added to the table and the cycle continues till all items are considered.

[117-118] This is illustrated in this line of code where the `notTaking = tab[i - 1][w]` variable is assigned to the item directly above the current items. `[i-1]` represents the row above the current row and `[w]` means with the same weight constraint.

[119-126] This illustrates the constraints of the algorithm. In order for the item to be considered. The initial current weight, should be less than the knapsack limit. `w\_I[i] <= w` where `w\_I` is the weight of the item, `[i]` is the index of the particular item and `[w]` is the weight constraint of the knapsack. If these constraints are met it is passed it triggers the next command. `Val[i] + tab[i-1][w-w_I[i]]`. This adds the value of the current item to the value of the item that could fit in the space available in the knapsack. The `Val` indicates the `Value` of the item, while `[i]` represents the `item`. `tab[i-1][w-w_I[i]]` basically means the limit of the knapsack minus the current items weight and move a row up. This takes the program to a different cell in the table and adds its value to the value of the item. This means those items are capable of entering the knapsack. The solution will therefore run in O(nW) time and O(nW) space

[127–143] When the values are calculated the optimal solution is displayed with the items chosen. It is displayed in a tabular form using the help of a for loop.
[133–137] The algorithm traces back the items selected to print them out. The syntax `tab[n][w] != 0` and `tab[n][w] != tab[n - 1][w]` This basically states that if the current cell is not equal to 0 or the current cell is not equal to the cell directly above select the item.

3.4.2 Greedy Method Function

[146–203] The `GreedyMethod()` function calculates the optimal solution using the greedy method. The items are sorted and the values with the maximum profit per weight are picked till all the space is used up.

[147–149] The function is run using an if and else statement. If there is no item present the function returns the statement `'There is no item present'`. This syntax `Item_no == 0` checks whether items are present. If items are available it moves to the else statement which contains the main algorithm.

[151–160] Note that when the algorithm first starts, a counter `(time.perf_counter_ns())` is called to time the algorithm in nanoseconds. The local function variables are initialized. The first element of the list which is a '0' is removed from both the values and weights arrays. This was done to avoid run-time errors in calculation. If this was not done a zero division error would be detected and the program will crush. Using this syntax `density = sorted([[values[i] / weight[i], for i in range(Item_no)], reverse=True)` the profit per unit weight is calculated and sorted in descending. It is then added to a list along with the respective values and weights. Each subset is arranged in this format `(profit per weight, weight, value)`.

[162–188] This `while loop` only runs when Items are available and there is a limit assigned to the knapsack. The while loop contains the main algorithm of the greedy method. Firstly the variables, `max` and `index` are initialized. A for loop is then created to identify the max profit per unit weight for each loop. Since the max item would be removed from the list in each loop. The new maximum has to be found to add to the list.

[170–177]An if statement is then used to decide whether to add the whole item or not.`density[index][1]` is the second element of the subset. This represents the weight of the item. `capacity` represents the limit of the knapsack. From the if statement `if density[index][1] <= capacity` we can tell the command only executes if the weight of the item is less than the limit of the knapsack. If the items weight is less than the limit of the knapsack it is added to the knapsack. `density[index][2]` represents the value of the item being added to the knapsack. This increases the value of the knapsack and reduces the amount of space available in the knapsack. From the command `capacity -= density[index][1]` the algorithm consistently reduces the limit of the knapsack till the space is no longer enough to fit the whole item. The added items are then added to a list and stored.

[179–186]If the weight of the item is more than the limit of the knapsack, the remaining space in the knapsack is divided by the weight of the item and multiplied by the weight of the item. This is done to get the fraction of the item capable of just filling the knapsack without surpassing the limit. The quantity obtained is then multiplied by the profit per unit weight to obtain the value you would derive. The command that executes this is `value += (capacity/density[index][1]) * density[index][1]*density[index][0]`, where value is the total profit in the knapsack, `capacity` is the space available in the knapsack, `density[index][1]` and `density[index][0]` is the profit per unit weight. This item is also added to the list and stored.

[187-188]When an item is added to the list of chosen items it is removed from the original list using this `density.pop(index)`. And the number of items is reduced by 1. This is to avoid adding that same item back to the list. This is done till the knapsack limit has been reached.

[190–200]The data is then printed in tabular form displaying the optimal solution and the time complexity. The time complexity counter is stopped here and the difference between the two times is calculated to produce the total time the algorithm took to calculate the data.

3.4.3 Brute Force Class and functions

[206–247] This is a class created purposely for the Brute Force Approach. The `init` function initialzes the variables for the weight, value, the number of items and limit of knapsack. This class is made of other functions.

[214–218] The `setMaker()` function makes the universal set. It is a set composed of other sets, which includes the item, item weight and item value. It uses a for loop; `for i in range(0,self.NoOfItems)` to add all this data and returns the universal set. This basically means it is going to loop the till number of items is reached. This syntax `set.append((i,self.Weight[i],self.Value[i]))` is responsible for adding the item, item weight and item value respectively to a list called `set`.

[222–232] The `power_set()` function is used to create all the possible sets that could be formed from the universal set. This gives us all the possible combinations of items. When done this returns all the subsets. First the function checks if items are available in the set. This syntax is responsible `if (len(input) == 0):`. If there are no items present it returns an empty set. This syntax , `for small_subset in power_set(input[1:])` is responsible for looping through the universal set. This `input[1:]` enables all items in the list to be considered.

[236–247] The `knapsack()` function is used to find the optimal solution. It takes the sum of all the weights available in each subset and the profit it produces. This is evident in the command `initial_weight = sum([i[1] for i in i_set])` and `initial_value = sum([i[2] for i in i_set])`. The condition then states that the sum of weights must not exceed the limit of the knapsack. So it excludes all items that have surpassed the limit of the knapsack. The syntax `initial_value > best_value` executes this command. This syntax `initial_weight < maxValue` then looks for the maximum profit from these sets. It then takes the items present in the optimal solution and adds it to a list called `knapsack`. The maximum profit with its corresponding weight is also stored in the `;best_value;`and `total_weight` variables respectively. When all the subsets are iterated over the optimal solution is then returned.

4 Results

4.1 Test files

This is a table which contains all the CSV test files used to test the algorithms of the program

Name of test file	Knapsack Limit	Number of items	Item Weight	Item Value
example.csv	10	5	5,4,3,2,1	8,3,5,3,2
knapsack_test_01.csv	150	13	35, 9, 31, 27, 32, 5, 30, 7, 26, 151, 23, 15, 15	28, 21, 24, 32, 23, 30, 23, 5, 10, 9999, 21, 30, 24
knapsack_test_02.csv	150	10	23,31,39,44,53,38 63,85,89,82	92,57,49,68,60,43 67,84,87,72
knapsack_test_03.csv	150	5	72,80,31,68,49	24 ,13 ,23 ,15, 16
knapsack_test_04.csv	150	15	35, 9, 31, 27, 32, 5,12,9 30, 7, 26, 151, 23, 15, 15	28, 21, 24,13,33, 32, 23, 30, 23, 5, 10, 9999, 21, 30, 24

4.2 Table/Graph of Results

This is a table that displays the results of the test files

Method	Test File	Best Profit	Total Weight	Time Complexity (ns)	Optimal Solution	Reason
Dynamic Programming	example.csv	16	10	58117	DP	Less time, max profit
Brute Force Approach		16	10	94340		
Greedy Approach		16.5	10	64786		
Dynamic Programming	knapsack_test_01.csv	194	144	2282371	DP	Less time, max profit
Brute Force Approach		186	145	18213151		
Greedy Approach		9933	150	1229457		
Dynamic Programming	knapsack_test_02.csv	266	137	1695024	DP	Less time, max profit
Brute Force Approach		266	137	2504794		
Greedy Approach		280.7	150	114541		
Dynamic Programming	knapsack_test_03.csv	54	148	863389		
Brute Force Approach		47	103	143511	DP	Max profit
Greedy Approach		62.3	150	248059		
Dynamic Programming	knapsack_test_04.csv	237	150	2361729	DP	Less time, max profit
Brute Force Approach		223	147	60184023		
Greedy Approach		9932.7	150	1203196		

As shown the Dynamic programming algorithm had the most favourable algorithm compared to the others.

The Figure 4 highlights the difference in time complexity. From the graph the Brute Force approach clearly takes more time to find the optimal solution.The brute force approach has exponential time complexity. The computation time increases with increasing number of items. The more the items present the more time taken.

The Dynamic Programming Approach takes less time but the Greedy Approach takes the least time. From the graph we can tell the Greedy Method has a logarithmic time complexity. It computes the data relatively faster than the others.

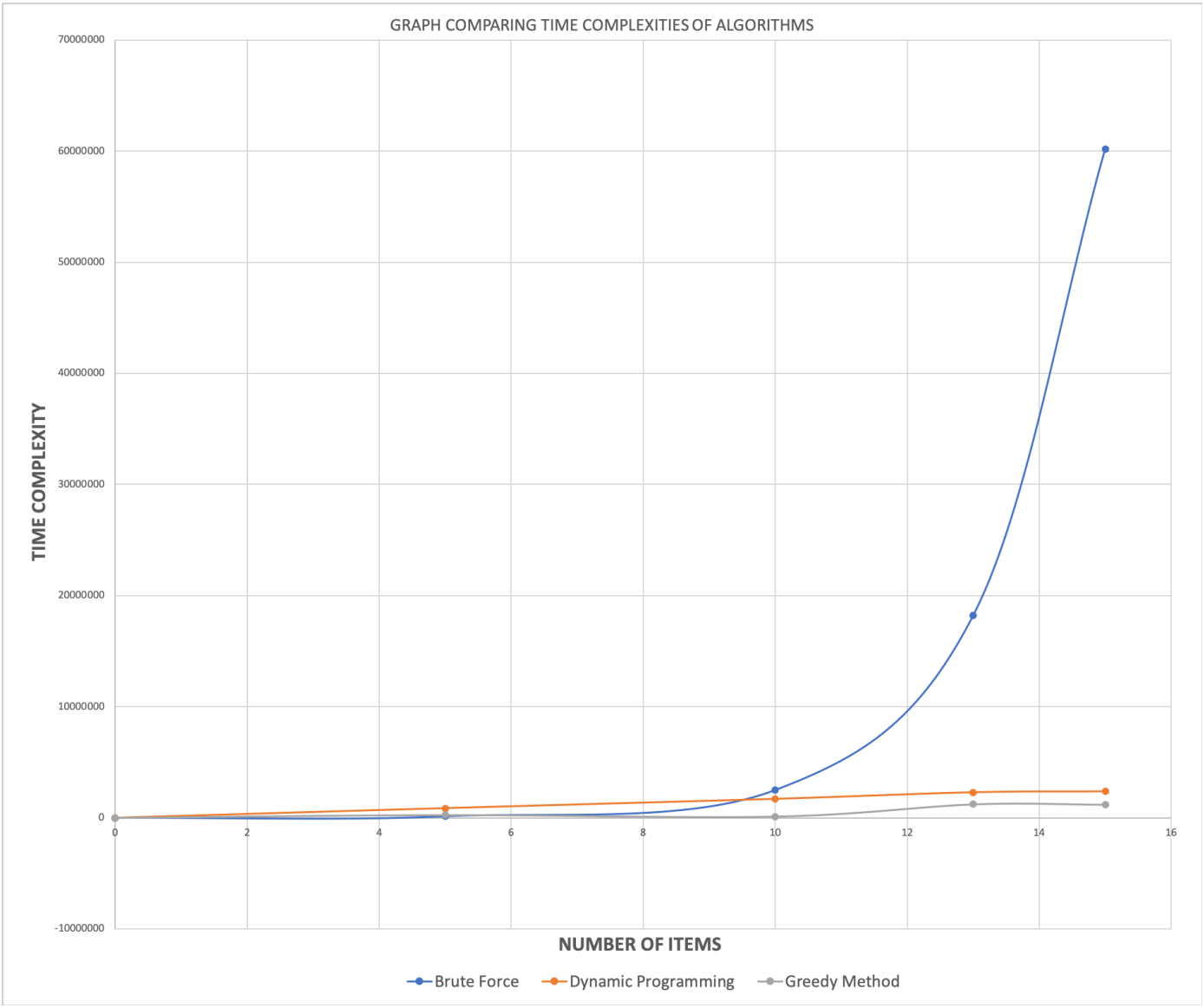

Figure 4: Comparing the time complexities of all 3 algorithms

Evident from the graph, some computations using the brute force approach are relatively quicker than the dynamic programming algorithm. This is when the number of items are small and the knapsack limit is huge.

5 Testing

5.1 Test Table

5.1.1 Dynamic Programming test table

Test No.	Test Description	Test Data	Expected Outcome	Actual Outcome	Improvements?
1	Test that function imports correct CSV values	example.csv	Print the arrays for the weights of the item and their corresponding values	Printed the arrays for the weights of the item and their corresponding values	n/a
2	Test non existing CSV Files	eaple.csv	print File does not exist and allow user to input another file.	Runtime error (FileNotFoundError)	Add a validation to check for files in path
3	Test the Initial Dynamic Programming table	example.csv	Prints a 6x11 table filled with zeros	Printed a 6x11 table filled with zeros	n/a
4	Test invalid data to see if program can handle it	w	Print Invalid format with option to type a new value	Runtime error (ValueError)	Add validation to check user inputs
5	Test calculations of Dynamic Programming Table	example.csv	Print a 6x11 array filled with values	Printed a 6x11 array filled with values	n/a
6	Check Optimal Solution	example.csv	Print weight =10, Print profit = 16	Print weight = 10, Print profit = 16	n/a
7	Print Time complexity of Dynamic Programming Method	print(t3)	Print time in Nanoseconds	Printed time in nanoseconds	n/a
8	Test Dynamic Programming Function Call	call DynamicProgramming()	Print selected item along with the optimal solution and time complexity	Printed the optimal solution with a table of selected items and time complexity	n/a

5.1.2 Greedy Method test table

Test No.	Test Description	Test Data	Expected Outcome	Actual Outcome	Improvements?
1	Test sorted values	example.csv	Print a list of sorted values in descending order of profit per unit weight	Runtime error (Zero Division Error)	Pop out zeros in arrays
2	Obtain maximum profit per unit weight for each loop	print(max)	Loop through the list of items sorted in descending order of profit per unit weight of item	Prints the maximum profit for each loop	n/a
3	Sum whole items capable of entering	example.csv	print table of items consisting of only whole items	Printed table of chosen whole items	n/a
4	Sum whole items and fraction of item to fill knapsack	example.csv	print table of items consisting of both whole and fractional items	Printed full table including fractions of items	n/a
5	Check Optimal Solution	knapsack_test_02.csv	print profit = 333 weight = 200	prints profit = 333 weight = 200	n/a

5.1.3 Brute force method test table

Test No.	Test Description	Test Data	Expected Outcome	Actual Outcome	Improvements?
1	Test that sets are formed from csv file	example.csv	Print a list of weight and values for each corresponding item	Printed a list of weight and values for each corresponding item	n/a
2	Print all possible sets from Universal set	example.csv	Print 32 sets	Printed all 32 sets	n/a
3	Print value and corresponding weight of items selected	example.csv	print items 1,3,4 with corresponding weight,values	Runtime error	Result of function was assigned to a variable
4	Print optimal solution	example.csv	Print Best Profit = 16, print total weight = 10	prints best profit = 13 total weight = 8 (logic error)	Add an equal to sign to statement 'weight < Capacity of knapsack'

5.2 Testing Evidence

5.2.1 Dynamic Programming

Test 1: Pass
Tests whether the ReadCSV() function would import the correct CSV file and add values to the appropriate list. In order to test whether it actually works the example.csv file was uploaded and the weights and values of the respective items were printed.

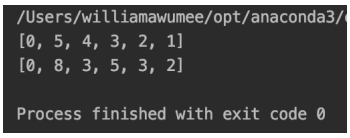

Figure 5: Dynamic Programming: Test 1

The weights and values were each added to their respective lists as seen in Figure 5 **hence the test passed**

Test 2: Pass
Tests whether a 2×2 array could be created with the number of items and the Limit of the knapsack.It was expected that a 6 x 11 matrix filled with zeros would be printed.

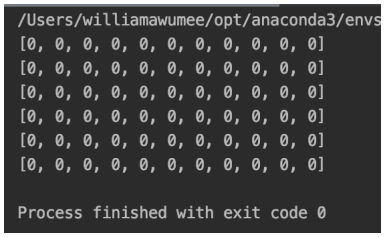

Figure 6: Dynamic Programming: Test 2

A 6 x 11 matrix filled with zeros was printed as seen in Figure 6 **hence the test passed**

Test 3: Fail
Tests for non existent CSV files. It tests what happens when a CSV File is not contained in a path.

The program crashed printing no such file or directory. A function was imported to test that the file exists and returns falls if it does not. Figure 8 is the improvement.

Test 4: Fail
Test on invalid entries. The program was to respond back notifying the user that an invalid input was entered with another chance to enter the correct input.

The program crashed as shown in Figure 9. A try except clause was added to the program. It however didn't allow the user to input a new integer but took a default one and moved on as shown in Figure 10

```
Type the name of your CSV File hereexamle.csv
Traceback (most recent call last):
  File "/Users/williamawumee/Desktop/Python/School/CS Project Notes/complete/KnapsackProblem_2499888.py", line 264, in <module>
    w_I,Val,Cap_k,NoOfItems,tab = ReadCSV()
  File "/Users/williamawumee/Desktop/Python/School/CS Project Notes/complete/KnapsackProblem_2499888.py", line 31, in ReadCSV
    with open(CSVfile) as csvfile:
FileNotFoundError: [Errno 2] No such file or directory: 'examle.csv'

Process finished with exit code 1
```

Figure 7: Dynamic Programming: Test 3

```
Type the name of your CSV File hereemple.csv
File not available
```

Figure 8: Dynamic Programming: Test 3i

```
How many Items are available?
w
Traceback (most recent call last):
  File "/Users/williamawumee/Desktop/Python/School/CS Project Notes/complete/KnapsackProblem_2499888.py", line 75, in <module>
    InputValues()
  File "/Users/williamawumee/Desktop/Python/School/CS Project Notes/complete/KnapsackProblem_2499888.py", line 50, in InputValues
    NoOfItems = int(failproof('How many Items are available?\n'))
  File "/Users/williamawumee/Desktop/Python/School/CS Project Notes/complete/KnapsackProblem_2499888.py", line 41, in failproof
    value = int(input(user))
ValueError: invalid literal for int() with base 10: 'w'
```

Figure 9: Flowchart: Dynamic Programming: Test 3

```
/Users/williamawumee/opt/anaconda3/envs/Sch
How many Items are available?w
Invalid float
What is the capacity of the knapsack?
```

Figure 10: Flowchart: Dynamic Programming: Test 3

The whole statement was therefore placed in a while loop with a counter in each clause such that if the correct input is not placed the while loop never ends. The user was then given another chance to input a valid value. This is shown in Figure 11

```
/Users/williamawumee/opt/anaconda3/
How many Items are available?
wer
Invalid float
How many Items are available?
```

Figure 11: Flowchart: Dynamic Programming: Test 3

Test 5: Pass
Tests whether the correct calculations were made and added to the table. The table was printed as shown in Figure 12 and with the help of a trace table each calculation was checked.

```
/Users/williamawumee/opt/anaconda3/envs/Scho
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 8, 8, 8, 8, 8, 8]
[0, 0, 0, 0, 3, 8, 8, 8, 8, 11, 11]
[0, 0, 0, 5, 5, 8, 8, 8, 13, 13, 13]
[0, 0, 3, 5, 5, 8, 8, 11, 13, 13, 16]
[0, 2, 3, 5, 7, 8, 10, 11, 13, 15, 16]

Process finished with exit code 0
```

Figure 12: Flowchart: Dynamic Programming: Test 5

After verifying with the trace table the values were concluded to be correct, **hence the test passed**.

Test 6: Pass
Tests whether the optimal solution could be printed and the chosen items could be traced back from the matrix.

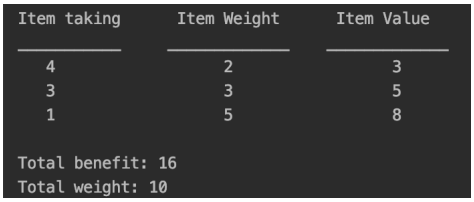

Figure 13: Flowchart: Dynamic Programming: Test 6

The optimal solution was obtained and the chosen items were traced back and printed in tabular form hence **hence the test passed**.

Test 7: Pass
Test 7 tests whether the Time complexity could be calculated

Figure 14: Flowchart: Dynamic Programming: Test 7

The time complexity was calculated and displayed in nanoseconds, **hence the test passed**.

Test 8: Pass
Tests whether the function call works and all commands are executed.

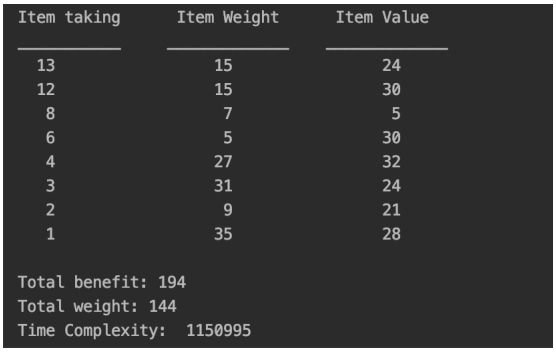

Figure 15: Flowchart: Dynamic Programming: Test 7

The optimal solution, the items chosen and the time complexity were also printed, **hence the test passed**.

5.2.2 Greedy Method

Test 1: Fail
Tests whether the profit per unit weight of an item could be calculated and sorted with its respective, unit weight and value in descending order.

It however crashed due too ZeroDivisionZero Error. From the list the integer 0 was deliberately made the first element of both lists. The 0 is mandatory for the Dynamic Programming method because in DP you have to start with 0 items and 0 capacity. However in the greedy method it is not. So it was removed from the list. The result is shown in Figure 17

Test 2: Pass
Tests whether the algorithm can access the new maximum profit per unit value for each loop after the previous maximum profit per unit value has been added to the list of chosen items. An item can not be picked twice so the next value should be accessible.

As seen a list of floats is looped through in descending order, **hence the test passed**.

```
Traceback (most recent call last):
  File "/Users/williamawumee/Desktop/Python/School/CS Project Notes/complete/KnapsackProblem_2499888.py", line 128, in <module>
    GreedyMethod(Cap_k,w_I,Val,NoOfItems)
  File "/Users/williamawumee/Desktop/Python/School/CS Project Notes/complete/KnapsackProblem_2499888.py", line 126, in GreedyMethod
    density = sorted([[values[i]/weight[i],weight[i],values[i]] for i in range(Item_no)],reverse = True) #Sorts number in decreasing order
  File "/Users/williamawumee/Desktop/Python/School/CS Project Notes/complete/KnapsackProblem_2499888.py", line 126, in <listcomp>
    density = sorted([[values[i]/weight[i],weight[i],values[i]] for i in range(Item_no)],reverse = True) #Sorts number in decreasing order
ZeroDivisionError: division by zero

Process finished with exit code 1
```

Figure 16: Flowchart: Greedy Method: Test 1

```
[[2.0, 1, 2], [1.6666666666666667, 3, 5], [1.6, 5, 8], [1.5, 2, 3], [0.75, 4, 3]]
```

Figure 17: Flowchart: Greedy Method: Test 1

```
5
2.0
1.6666666666666667
1.6
1.5
0.75

Process finished with exit code 0
```

Figure 18: Flowchart: Greedy Method: Test 2

Test 3: Pass
Tests the first phase of the Greedy Method. It checks whether all the chosen whole items can be summed till no whole item can no longer fit in the knapsack.

```
Profit per weight     Weight     Value      Profit in Knapsack     Weight Taken     Capacity remaining
_________________     ______     _____      __________________     ____________     __________________
+2.00000e+00          1          2          2                      1                9
+1.66667e+00          3          5          7                      3                6
+1.60000e+00          5          8          15                     5                1

Process finished with exit code 0
```

Figure 19: Flowchart: Greedy Method: Test 3

The program was able to tabulate the chosen items for the optimal solution. As shown there is still some space left in the knapsack. This is evident in the capacity remaining column As seen a list of floats is looped through in descending order, **hence the test passed**.

Test 4: Pass
Tests the second phase of the Greedy Method. It includes a fraction of the next item with the maximum profit per unit value and adds it to the knapsack just enough for it to fit.

```
Profit per weight     Weight     Value      Profit in Knapsack     Weight Taken     Capacity remaining
_________________     ______     _____      __________________     ____________     __________________
+2.00000e+00          1          2          2                      1                9
+1.66667e+00          3          5          7                      3                6
+1.60000e+00          5          8          15                     5                1
+1.50000e+00          2          1          16                     1                0

Total profit:  16.5

Process finished with exit code 0
```

Figure 20: Flowchart: Greedy Method: Test 4

As shown there is no space remaining currently even though the weight of the item is more than the previous space remaining. A fraction of this item was therefore taken, **hence the test passed**.

Test 5: Pass
Test 5 tests whether the optimal solution could be calculated for the knapsack_test_01.csv file. The value is then cross-checked with a dry run.

Figure 21: Flowchart: Greedy Method: Test 5

The value was verified and it corresponded with the dry run value, **hence the test passed**.

5.2.3 Brute Force

Test 1: Pass
Tests that the imported CSV files could be used to form sets containing the weights and profits of each item.

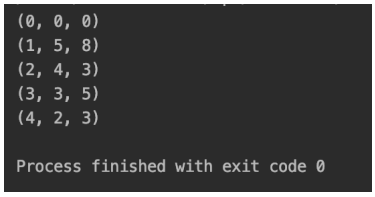

Figure 22: Flowchart: Brute Force: Test 1

The Items and their respective values and weights were printed,**hence the test passed**.

Test 2: Pass
Tests that the powerset of all these sets could be formed, to gain every possible combination

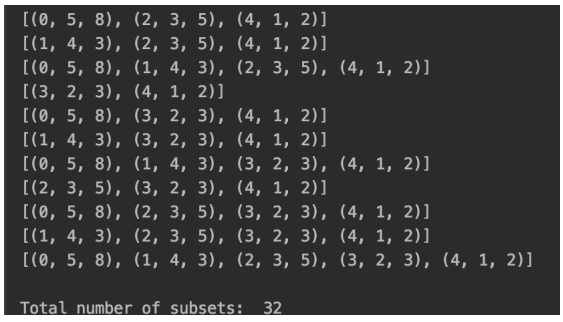

Figure 23: Flowchart: Brute Force: Test 2

All the 32 sets were printed,**hence the test passed**.

Test 3: Fail
Tests whether the items of the supposed optimal solution could be accessed.

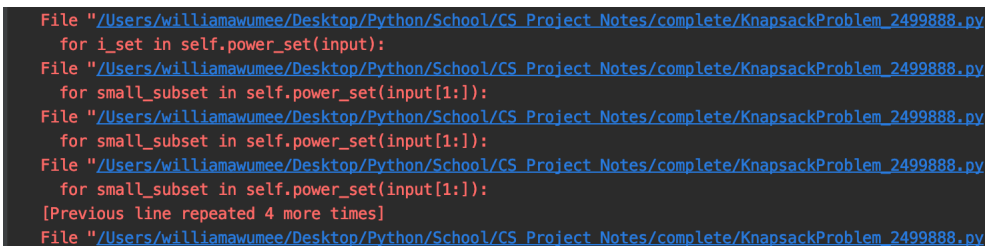

Figure 24: Flowchart: Brute Force: Test 3

The program entered an infinite loop due to the manner in which the command was interpreted. Instead of calling the function in the for loop, the result obtained from the powerset() function was assigned to a variable and passed into the for loop.

Test 4: Fail
Test 4 tests whether the correct optimal solution could be obtained

Figure 25: Flowchart: Brute Force: Test 4

This was a logic error. Instead of making the `total weight of the items` < or = to `the limit of the knapsack` it was only set to `weights` < `the limit of the knapsack` so the final solution could not be obtained. It was corrected and printed in a tabular form as shown in the Figure 26

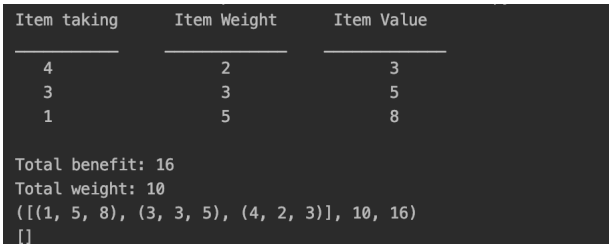

Figure 26: Flowchart: Brute Force: Test 4

5.3 Dry Runs

The Dry Run below is a test of both the dynamic programming and brute force approach for the **example.csv**

5.3.1 Dynamic Programming

This dry run was conducted to verify the values obtained from the model in Figure 12.

The result obtained from this dry run would be compared to the outcome of the 2nd array of the Figure 12. This array illustrates the result obtained after adding the first item to the knapsack.

This is obtained after adding the first item of weight = 5 and value = 8.

i	w	w_I	Val[i]	Is w_I[i]<= W	tab[i-1][w-w_I[i]]	Val[i] + tab[i-1][w-w_I[i]]	tab[i-1][w]	max	tab[i][w]
1	0	5	8	F	-	-	-	0	0
	1	5	8	F	-	-	-	0	0
	2	5	8	F	-	-	-	0	0
	3	5	8	F	-	-	-	0	0
	4	5	8	F	-	-	-	0	0
	5	5	8	T	-	8	-	8	8
	6	5	8	T	-	8	-	8	8
	7	5	8	T	-	8	-	8	8
	8	5	8	T	-	8	-	8	8
	9	5	8	T	-	8	-	8	8
	10	5	8	T	-	8	-	8	8

Comparing this result to the result obtained from the 2nd array of the model 12 we can conclude that the data calculated was correct.

The result obtained from this dry run would be compared to the outcome of the 3rd array of the Figure 12. This array illustrates the result obtained after adding the first item to the knapsack.

This is obtained after adding the first item of weight = 4 and value = 3.

i	w	w_I	Val[i]	Is w_I[i]<= W	tab[i-1][w-w_I[i]]	Val[i] + tab[i-1][w-w_I[i]]	tab[i-1][w]	max	tab[i][w]
2	0	4	3	F	-	-	0	0	0
	1	4	3	F	-	-	0	0	0
	2	4	3	F	-	-	0	0	0
	3	4	3	F	-	-	0	0	0
	4	4	3	T	0	3	0	3	3
	5	4	3	T	0	3	8	8	8
	6	4	3	T	0	3	8	8	8
	7	4	3	T	0	3	8	8	8
	8	4	3	T	0	3	8	8	8
	9	4	3	T	8	11	8	11	11
	10	4	3	T	8	11	8	11	11

Comparing this result to the result obtained from the 3rd array of the model 12 we can conclude that the data calculated was correct.

The result obtained from this dry run would be compared to the outcome of the 4th array of the Figure 12. This array illustrates the result obtained after adding the first item to the knapsack.

This is obtained after adding the first item of weight = 3 and value = 5.

i	w	w_I	Val[i]	Is w_I[i]<= W	tab[i-1][w-w_I[i]]	Val[i] + tab[i-1][w-w_I[i]]	tab[i-1][w]	max	tab[i][w]
3	0	3	5	F	-	-	0	0	0
	1	3	5	F	-	-	0	0	0
	2	3	5	F	-	-	0	0	0
	3	3	5	T	0	5	0	5	5
	4	3	5	T	0	5	3	5	5
	5	3	5	T	0	5	8	8	8
	6	3	5	T	0	5	8	8	8
	7	3	5	T	3	8	8	8	8
	8	3	5	T	8	13	8	13	13
	9	3	5	T	8	13	11	13	13
	10	3	5	T	8	13	11	13	13

Comparing this result to the result obtained from the 4th array of the model 12 we can conclude that the data calculated was correct.

5.3.2 Brute Force

This dry run was also conducted to verify the values obtained from the brute force method. Not all the iterations could be tested, however a few of them including the supposed optimal solution from the subsets were.

I_set	Initial_weight	Initial_value	Self.Cap_k	Is initial_weight <= self.Cap_k?	Is Initial_value >best_value?	Best _value	Total _weight
(1,5,8),(2,4,3) (3,3,5),(4,2,3),(5,1,2)	15	21	10	F	-	0	0
(1,5,8),(2,4,3),(3,3,5) (4,2,3)	14	18	10	F	-	0	0
(2,4,3),(3,3,5),(4,2,3) (5,1,2)	10	13	10	T	T	13	10
(1,5,8),(3,3,5),(4,2,3)	10	16	10	T	T	16	10
(1,5,8),(5,1,2),(2,4,3)	10	10	10	T	F	16	10
(1,5,8),(2,4,3),(3,3,5)	12	16	10	F	-	16	10
(5,1,2),(3,3,5),(1,5,8)	9	15	10	T	F	16	10
(1,5,8),(3,3,5)	8	13	10	T	F	10	16
(5,1,2),(3,3,5)	4	7	10	T	F	10	16
(3,3,5)	3	5	10	T	F	10	16

From the subsets chosen, the optimal solution matched our dry run. Since not all the subsets were tested in the dry run we cant conclude that this was the optimal solution but we can however verify that the calculations made were correct.

5.3.3 Greedy Method Approach

This dry run was also conducted to verify the values obtained from the greedy method.

i	density[i][1]	density[i][0]	max	index	is density[index][1]<=capacity	value	capacity
5	1	2	2	5	Yes	2	9
3	3	1.66667	1.6667	3	Yes	7	6
1	5	1.6	1.6	1	Yes	15	1
2	2	1.5	1.5	2	No	16.5	0

Comparing this result to the result obtained from the model algorithm it was concluded that the data calculated was correct. .

6 Conclusion

Three algorithms were tested to find the optimal solution. The Dynamic Programming method, Brute Force and Greedy Approach. However only the algorithms Dynamic Programming method was capable of finding the optimal solution. This is a 0-1 Knapsack problem so the program could only take whole items , the greedy approach did not produce the optimal solution. The greedy approach takes fractions of items to find the optimal solution. The greedy approach can therefore be used in the fractional knapsack problem to find the optimal solution. However in this case, since it is a 0-1 knapsack problem it would not. For n times, this algorithm took a time complexity of O(n) to find the profit per unit weight. It then took O(nlogn) time to sort, while to pick the selected items it took O(n) time. Overall it has a logarithmic time complexity.

The Brute Force approach did not produce the optimal solution either in some cases. This is quite unfortunate since the brute force approach takes into consideration all the possible outcomes and takes the optimal solution. This is probably a logic error, whoever due to limited time and resources the error could not be identified. With more time the error could be identified and the brute force would also produce the expected outcome.The Brute Force Method has exponential time complexity. Its time increases exponentially with the number of inputs. This implies that large inputs could take hours or even days to compute. It takes the alogorithm $O(2^n)$ time to form the powerset or all the possible combinations. O(n) to find the sum and O(1) to print it out. In total this produces $O(2^n)$, which is the Big-O notation for exponential time complexity. This may eventually use up the computers memory and it may begin to run slowly. Using such algorithms to more complex would require computers of high processing power.

The Dynamic Programming method did however produce optimal solutions in many cases. The Dynamic Programming method has pseudo polynomial time complexity. It took O(1) to fill in each cell of the 2 dimensional array and O(n) to trace back the items and discover the optimal solution. In total this algorithm took O(nW) + O(1) + O(n) time which is equal O(nW).

Thus Dynamic Approach had a significantly much better running time. Thus the Dynamic Approach is the best algorithm out of the lot. There was however a case where the Brute force approach in spite of its exponential time complexity had a faster computation time than the Dynamic Approach. In this case the number of items was small while the knapsack limit was huge. In such cases the brute force algorithm is quicker than the Dynamic Approach.

So does this mean we have solved the P = NP problem since the dynamic programming algorithm of the 0–1 knapsack problem runs in O(nW) time, where n is the number of items under consideration, and W is the capacity of the knapsack. Both n and W are assumed to be integers.

Using a 2D array and indexing through every possible by item and weight, a recursive approach was used to calculate the correct value for every array entry. The optimal solution was produced by building through the array. This seems to run in polynomial time, thus the P = NP problem should be solved.

Well its not exactly the case. The array is actually exponential in the size of its input. The input consists of n items and w weights. The input is stored in bits. This means that an array of dimensions size n and size w is exponentially relative to the bits used to encode it.This makes the dynamic programming algorithm a pseudo-polynomial algorithm. It looks polynomial but it isn't. It however works great for lots of practical instances, because the problem size grows very slowly.

6.1 Improvements

Some improvements were identified that could be implemented in this model, however due to limited time and resources it was not feasible.

One of such improvements is by implementing backtracking in the brute force method. The Brute Force method has a large computation time. It takes a long time for the optimal solution to be developed. To

improve the computation time backtracking could be used to optimize the Brute Force solution. Using tree representation, DFS of trees could be done. If a point is reached where the solution is no longer feasible, it is no longer explored. So for instance by creating a tree of subsets and adding the weight of each item in every part of the tree, we discard it if the weight exceeds the knapsack limit. If the weight of the item is less than the knapsack limit the tree expands adding more items the list. The values would also be totalled for each subset. This would prevent the algorithm creating sets that exceed the knapsack limit and recomputing values already included in the list.

Improvements have to be made on the logic used to create the Brute force method. The algorithm takes into consideration all possible outcomes so the optimal solution should be produced. It should have at least the some optimal solution derived from the Dynamic method approach or even better.

Another improvement is implementing the Branch and Bound algorithm. The Greedy method could be transformed into a Branch and Bound method so it could be used to find the optimal solution of the knapsack problem. From the experiment we concluded that the greedy method could not be used to find an optimal solution to the 0/1 knapsack problem however it could be transformed to the branch and bound method which could be used to find the optimal solution.

6.1.1 Branch and Bound Method

The Branch and Bound Method is a general programming technique used to improve the searching process by systematically enumerating all the solutions and disposing solutions that are not feasible [8]. Outlined below are procedures used to implement this algorithm.

First of all the items are to be arranged in decreasing order of profit per weight ratio.

Then identify the upper bound and lower bound of each node. The Upper bound is identified by filling the knapsack with the full items that can enter the knapsack from the maximum profit per weight to the least. The Upper bound is the sum of the values of this items.The lower band is obtained by adding a fraction of the item left

Next the maximization problem must be converted to negative. That is you take the negative sign for the upper bound and lower bound.

Now using a tree you decide whether to include the an item or not. This is done by choosing a path that has minimum difference of upper bound and lower bound. If the difference is equal then choose the path by comparing the upper bound and lower bound. The node with maximum difference is discarded. This is done for all nodes. When all items are considered the optimal solution is the path that leads to the last feasible node

7 Evaluation

The model meets all the specifications of the project. The first specification was for the code to be written in python code. The code was obviously written in python as witnessed in the appendix

The second specification was for the code to be portable. The line [20–27] meets this specification.

The third specification was for the code to use the import function. This specification was met by the line [1–3]

The fourth specification was for the code to have a parsed user input. This specification was met by the line [45-60] and [261–264]

The fifth specification was for the code to use a range of data types including lists. This specification was met by the line [5–12]

The sixth specification was for the code to use at least three user defined subroutines. This specification was met by the line [256–324]

The seventh specification was for the code to import data from a csv file in the format given. This specification was met by the line [29–52]

The eight specification was met by the line [127–142], [191–203] and [278–300]

The ninth specification was for the code to exit the program in a controlled fashion. This was met by the line [256]

7.1 Limitations of the model

More CSV files could not be tested due to limited time so as to get a more accurate representation of the algorithm for different Knapsack limits and number of items.

We could have tested more CSV files to get a more accurate representation of the effect of large number of items, small number of items, large knapsack limit and small knapsack limits on the different algorithms. It was discovered that when number of items are little the brute force approach is faster than the Dynamic Programming approach. More CSV files with little items should have been tested to confirm the discovery. However limited time could not permit as to do as such.

From the program it was also identified that the brute force approach did not produce the optimal solution. This is quite strange since the brute force approach is supposed to calculate for all the possible outcomes of a problem and produce the best one. However in some instances this did not happen. The Dynamic approach produced a better solution than the brute force approach, The reason this happened is unknown and is assumed to be a logic error. However due to limited time and resources the error could not be identified. With more time and appropriate resources and aid the error could be identified.

The Brute force Approach had exponential time complexity. This is because it had to calculate for all possible combinations. The algorithm has to form the powerset of all the items present. The more the items present in the set the more subsets formed. The number of subsets that can be formed is given by the formula 2^n. Thus, it increases exponentially by the number of items present. This means that this algorithm can take minutes to hours to compute or find optimal solutions to problems with large number of items. This is a problem because most problems would contain large amounts of items. This also ends up using a significant proportion of the computers memory relative to the other algorithms. When other problems are reduced to the knapsack problem, using the brute force approach would require a lot of processing power.

The computation time for the algorithms may be different on different computers based on its processing speed so the precise time using the counter may vary from others. We can therefore not conclude that this is the precise time of the algorithm. We can however assume that the relationship between the input and the operation time was correct.

The Dynamic programming method cant be used to solve problems with non-integer constraints.

The Greedy Method cannot be used to solve the 0-1 Knapsack problem. This is because in the 0-1 knapsack problem the items are non divisible. It can however only be implemented in the fractional knapsack problem.

The time calculated was represented in Nanoseconds. A function could not be identified to convert the time to nanoseconds. With more time a function could have been identified to convert it.

8 References

[1] M. Assi and R. A. Haraty. (2018) A survey of the knapsack problem, presented at: Acit. [Online]. Available: https://ieeexplore.ieee.org/document/8672677

[2] Undefined Behavior. (2017) P vs. np - an introduction. [Online]. Available: https://www.youtube.com/watch?v=OY41QYPI8cw

[3] ——. (2018) Np-complete explained (cook-levin theorem). [Online]. Available: https://www.youtube.com/watch?v=W9G_1xG77LE

[4] S. Studio. (2018) P np np-hard np-complete|,design and analysis of algorithm, english. [Online]. Available: https://www.youtube.com/watch?v=DumOqL85Ryc

[5] X. Pan and T. Zhang. (2018) Comparison and analysis of algorithms for the 0/1 knapsack problem. [Online]. Available: https://www.researchgate.net/publication/327326005_Comparison_and_Analysis_of_Algorithms_for_the_01_Knapsack_Problem

[6] S. Martello, *Knapsack problems: algorithms and computer implementations.* John Wiley & Sons Ltd., 1990.

[7] K.Klamroth and M. Wiecek. (2000) Dynamic programming approaches to the multiple criteria knapsack problem. [Online]. Available: http://www2.math.uni-wuppertal.de/~klamroth/publications/klwidp00.pdf

[8] H. Salkin and C. D. Kluyver, "The knapsack problem: A survey," *Naval Research Logistics Quarterly*, vol. 22, no. 1, pp. 127–144, 1975. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800220110

Appendices

A Code:

Listing 1: Python Code

```
import csv
import time
import os
from os import system, name

NoOfItems = 0    # Number Of Items
Cap_k = 0        # Limit of Knapsack
w_I = [0]        # List of Weight of items
Val = [0]        # List of Value of items
User_input = 'nothing'
choice = 'nothing'
tab = []         # DP table
m_input= -1
print('WELCOME TO THE KNAPSACK PROBLEM OPTIMAL SOLUTION SIMULATOR ')
print('_____________________________________________________________')
print('STUDENT ID: 2499888\n'
      'DATE: 9/04/2020\n'
      'COMPUTER SCIENCE PROJECT\n')


def clear():         #Clear terminal screen
    # for windows
    if name == 'nt':
        _ = system('cls')

        # for mac and linux(here, os.name is 'posix')
    else:
        _ = system('clear')

def ReadCSV():       #Import CSV File
    with open(CSVfile) as csvfile:
        CSV = csv.reader(csvfile, delimiter=',') #Reads name od CSV file
        w_I = [0]
        Val = [0]
        Weight = []
        Items = []

        for row in CSV:
            try:
                Weight.append(int(row[0]))  # Adds Capacity of Knapsack and weights of items
     to a list
                Val.append(int(row[1]))  # Adds the corresponding values to a list
            except:
                pass

    Cap_k = Weight[0]  # Takes the Capacity of Knapsack from the first row

    for i in Weight[1:]:  # Removes Capacity of Knapsack and adds weights to list
        w_I.append(i)

    NoOfItems = len(w_I)-1  #Calculate Number of items present
    tab = [[0 for x in range(Cap_k + 1)] for x in range(NoOfItems + 1)] #Create 2
    dimensional array to store calculated values

    return w_I, Val, Cap_k, NoOfItems, tab

def failproof(user):         #Validation of user input
    while m_input < 0:       #Runs till valid input is received
        try:
            value = int(input(user))
            return value
        except:
            print('Invalid input')

def InputValues():           #Stores User input
  clear()
  count = 0
```

```
66    global Cap_k,w_I,Val,tab,NoOfItems
67    NoOfItems = int(failproof('How many Items are available?\n'))
68    Cap_k = int(failproof('What is the capacity of the knapsack?\n'))
69    w_I = [0]
70    Val = [0]
71    tab = [[0 for x in range(Cap_k + 1)] for x in range(NoOfItems + 1)]  #Create 2 dimensional
          array to store calculated values
72    Items = []
73
74
75    while count != NoOfItems: #Runs till all values have been entered
76      clear()
77      try:
78        Weight = int(input('Enter weight \n'))
79        w_I.append(Weight)      # Add Weights entered by user to list
80      except ValueError:
81        print('Invalid Float. Enter a valid number \n')
82        continue
83
84      try:
85        Value = int(input('Enter item value \n'))
86        Val.append(Value)       # Add Values entered by user to list
87        count += 1
88      except ValueError:
89        print('Invalid Float. Enter a valid number \n')
90        continue
91    return NoOfItems,Cap_k,w_I,Val,tab
92
93  def Main_Menu():
94
95      print('\nMAIN MENU\n'
96            '1. Select CSV file\n'
97            '2. Input Values Manually\n'
98            '3. Run program\n'
99            '4. Reset\n'
100           'q. Quit \n')
101
102 def RunProgram():
103     print('\n1. Dynamic programming Approach\n'
104           '2. Brute force Approach\n'
105           '3. Greedy Method\n'
106           'q. Go back to main menu\n')
107
108 def DynamicProgramming(NoOfItems,Cap_k,tab,w_I,Val):
109     t1 = time.perf_counter_ns() #Start time Counter
110     if NoOfItems == 0:  #If items are not available
111         print('No Items are present')
112         pass
113     else:
114         for i in range(1, NoOfItems + 1):  # Row-Items available
115             for w in range(1, Cap_k + 1):  # Column- Max Capacities of knapsack
116
117                 notTaking = tab[i - 1][w]  # Takes item one row above with same Capacity of
      knapsack
118                 take = 0
119
120                 if w_I[i] <= w:  # If item weight is less than the capacity of knapsack
121                     take = Val[i] + tab[i - 1][w - w_I[i]]  # Add value of item to the max
      value of previous items able to fit
122
123                 tab[i][w] = max(notTaking, take)  # Take the max value
124
125             w = Cap_k
126             T_weight = 0
127
128         print('Item taking       Item Weight       Item Value')
129         print('-----------      -------------     -------------')
130
131         for n in range(NoOfItems, 0, -1):
132
133             if tab[n][w] != 0 and tab[n][w] != tab[n - 1][w]:  # Pick the items choosing
134                 print('%4d                %4d               %4d' % (n, w_I[n], Val[n])) #Print
       results
```

```
135                 w = w - w_I[n]      #Knapsack limit - item weight
136
137                 T_weight += w_I[n]  #Add weight of item
138
139         print('\nTotal benefit: %d' % tab[NoOfItems][Cap_k])
140         print('Total weight: %d' % T_weight)
141         t2 = time.perf_counter_ns() #End time Counter
142         t3 = (t2 - t1) #Finds the difference in time difference
143         print('Time Complexity: ', t3), ' ns'
144
145
146 def GreedyMethod(capacity, weight,values,Item_no):
147     if Item_no == 0:
148         print('There is no item present')
149         pass
150
151     else:
152         t1 = time.perf_counter_ns()     #Start time Counter
153         Items = []
154         value = 0  # Total profit
155         v = 0
156         c = 0
157         values.pop(0)  # Remove 0 from original values list to avoid Zero division
158         weight.pop(0)  # Remove 0 from original weight list to avoid Zero division
159         density = sorted([[values[i] / weight[i], weight[i], values[i]] for i in range(
    Item_no)],
160                          reverse=True)  # Sorts number in decreasing order
161
162         # Note that list is arranged in this format (Density,Weight,Value)
163         while capacity > 0 and Item_no > 0:
164             max = 0  # Maximum profit per weight
165             index = None
166             for i in range(Item_no):
167                 if density[i][1] > 0 and max < density[i][0]:  # Choose the maximum density
    form the list
168                     max = density[i][0]  # Maximum profit per weight
169                     index = i  # Stores index of the item
170
171             if density[index][1] <= capacity:  # Checks whether weight is less than Knapsack
     limit
172                 value += density[index][2]  # Adds cumulative sum of values to the knapsack
173                 v = density[index][2] # Records the value of item to print in list
174                 c = density[index][1] # Records the weight of item to print in list
175                 capacity -= density[index][1]  # Subtracts the weight of item from total
    capacity
176                 Items.append((density[index][0], density[index][1], v, value, c,capacity))
    # Creates a list of profit per weight, weight of item, value of item, Total value,
    Capacity of item taken, Capacity of knapsack remaining
177
178             else:
179                 if density[index][1] > 0:  # If item is still present
180                     value += (capacity / density[index][1]) * density[index][1] * density[
    index][0]  # Takes fraction of the item
181                     v = (capacity / density[index][1]) * density[index][1] * density[index
    ][0] #Records the fraction taken
182                     c = capacity
183                     capacity -= c
184                     Items.append((density[index][0], density[index][1], v, value, c,
    capacity)) #Adds data to a list
185                     break
186
187             density.pop(index)  # Removes item from the list
188             Item_no -= 1  # Reduces number of items
189
190         print('Profit per weight     Weight     Value      Profit in Knapsack     Weight
    Taken    Capacity remaining')
191         print('--------------------  --------   -------    ----------------------
    --------------   --------------------')
192         for i in Items:
193             print('%+6.5e          %4d       %4d            %4d                %4d
        %4d' % (
194                 i[0], i[1], i[2], i[3], i[4], i[5])
195                   )
```

```
196
197         print('\nTotal profit: ', value)
198         t2 = time.perf_counter_ns() #End time Counter
199         t3 = t2 - t1 #Find the time difference
200         print('Time Complexity: ', t3)
201         values.insert(0, 0)
202         weight.insert(0, 0)
203
204
205
206 class BruteForce: # Class for Brute Force Method
207     def __init__(self,Weight,Value,NoOfItems,Cap_k): #Initialze system variavles
208         self.Weight = Weight         # Weight of item
209         self.Value = Value           # Value of item
210         self.NoOfItems = NoOfItems   # Number of items
211         self.Cap_k = Cap_k           # Capacity of knapsack
212
213
214     def setMaker(self): #Creates a list to store item, weight and value
215         set = []
216         for i in range(0,self.NoOfItems):  #Loops till all items are considered
217             set.append((i,self.Weight[i],self.Value[i])) #Adds data to a list
218         return set
219
220
221
222     def power_set(self,input):
223         # returns a list of all subsets of the list
224         count = 0
225         if (len(input) == 0): #If list is empty returns empty sey
226             return [[]]
227         else:
228             m_subset = []  #Initialize list to store all subsets
229             for s_subset in self.power_set(input[1:]):  #Loops till all items are considered
      for every possiblity
230                 m_subset += [s_subset] # Add current item to list
231                 m_subset += [[input[0]] + s_subset] # Add remaining items to form subsets
232             return m_subset
233
234
235
236     def knapsack(self,input):
237         knapsack = [] #Initialize list to store selected items
238         total_weight = 0  # Total weight of optimal solution
239         best_value = 0    # Best profit
240         for i_set in input:     #Loops till all subsets are considered
241             initial_weight = sum([i[1] for i in i_set]) #Add all weights of items in each
      subset
242             initial_value = sum([i[2] for i in i_set])  #Add all values of items in each
      subset
243             if initial_value > best_value and initial_weight <= self.Cap_k: # Constraints of
       knapsack
244                 best_value = initial_value
245                 total_weight = initial_weight
246                 knapsack = i_set
247         return knapsack,total_weight,best_value
248
249
250 ##________MAIN PROGRAM STARTS HERE_____________________##
251
252 while User_input != 'q': # Main program runs till user decides to quit
253
254     Main_Menu()
255     User_input = input('')
256
257     if User_input == '1':
258         clear() #Clears screen terminal
259         CSVfile = input('Type the name of your CSV File here')
260         x = os.path.isfile(CSVfile) #Checks for CSV file path
261         if x: #If CSV file is found in current path
262             print('File successfuly retrieved')
263             w_I,Val,Cap_k,NoOfItems,tab = ReadCSV()
264         else: #IF CSV file cant be found in current path
```

```
265             print('File not available')
266             continue
267
268     if User_input == '2':
269         clear() #Clears terminal screen
270         InputValues()
271
272     while User_input == '3':
273         RunProgram()
274
275         choice = input('Please pick the method you would like to use.')
276         if choice == '1':
277             DynamicProgramming(NoOfItems,Cap_k,tab,w_I,Val) # Dynamic Programming algorithm
278
279
280         if choice == '2':
281             clear()     #Clears terminal screen
282             if NoOfItems == 0: #If No items are available
283                 print('Items not available')
284                 pass
285             else:
286                 t1 = time.perf_counter_ns() #Starts to time algorithm
287                 knapsack = BruteForce(w_I, Val, NoOfItems, Cap_k) #Brute Force Algorithm
288                 U_set = knapsack.setMaker() #Main set of items
289                 Subsets = knapsack.power_set(U_set) #All subsets of items
290                 Items, OptWeight, OptProfit = knapsack.knapsack(Subsets)
291
292                 print('Item taking      Item Weight      Item Value')
293                 print('-----------     -------------    -------------')
294                 for i in Items:
295                     print('%4d               %4d              %4d' % (i[0], i[1], i[2]))
296
297                 print('\nTotal weight: %4d' % OptWeight)    #Print total weight
298                 print('Best profit : %4d' % OptProfit) #Print best profit
299                 t2 =  time.perf_counter_ns() #End timer
300                 t3 = (t2 - t1) # Find the difference in time
301                 print('Time Complexity: ', t3, ' s')    #Print time
302
303
304         if choice == '3':
305             clear()     #Clears terminal screen
306             GreedyMethod(Cap_k,w_I,Val,NoOfItems)   #Greedy Method algorithm
307
308
309
310         if choice == 'q': #Goes back to the main menu
311             clear()     #Clears terminal screen
312             break
313
314     if User_input == '4': #Resets algorithms. Clears list of all items
315         clear()     #Clears terminal screen
316         NoOfItems = 0
317         Cap_k = 0
318         w_I = [0]
319         Val = [0]
320         Items = []
321         tab = [[0 for x in range(Cap_k + 1)] for x in range(NoOfItems + 1)]
```