Grace Seiche

Professor Mello-Stark

CS 2223/B Term 2017

4 December 2017

Project 3

Executive Summary Report

Efficiency Report

	Dynamic Programming	Brute Force	Greedy
Time Efficiency	O(n*capacity)	O(n*2^n)	O(nlogn)
Space Efficiency	O(n*capacity)	O(2^n)	O(n)

The time efficiency of dynamic programming is O(n*capacity) because it has a set of nested for loops where the outer loop iterates from 0 through the number of objects and the inner loop iterates from 0 through the capacity of the knapsack. Thus, the time efficiency is O(n*capacity).

The space efficiency of dynamic programming is O(n*capacity) because the largest data structure created in the function is a 2D list of highest values given the number of objects and the weight. The 2D list was n by capacity, so the space efficiency is O(n*capacity).

The time efficiency of brute force is O(n*2^n) because there is a nested for loop where the outer for loop iterates 2^n times and the inner for loop iterates the number of objects time. This means that the time efficiency is O(n*2^n).

The space efficiency of brute force is O(2^n) because the largest data structure that it created in the function is a table list that holds strings represents all possible combinations of objects. Since there are two options for each object (in or out) and there are n objects, the list is 2^n items long, which is the space efficiency.

The time efficiency of greedy is O(nlogn) because in addition to the non-nested for loops, the ratios of value to weight have to be sorted from high to low. This sorting algorithm (provided by python) runs in O(nlogn) efficiency. Since this is greater than the efficiency of the for loops, that is the time efficiency.

The space efficiency of greedy is O(n) because the largest data structure created is the knapsack which can hold a maximum of n items (if the weight in the knapsack was unlimited). Therefore, the space efficiency is O(n).

Do all three solutions provide an optimal solution?

Not always. Brute force and dynamic programming provide an optimal solution, but the greedy algorithm does not always provide the optimal solution. To demonstrate this, run the program using

“input-4.txt”. The greedy method uses approximations rather than finding exact figures like dynamic programming and brute force. Since it is an approximation, the greedy method is not always guaranteed to provide the correct solution. However, the solution it gives is usually close to the actual value.

Which is the better knapsack solution and why? Is this true for all knapsack examples?

Looking only at efficiency, I would say that greedy has the best solution because it has the lowest space and time efficiencies of all three methods.

However, looking at both efficiency and accuracy, I would say that dynamic programming is the best knapsack solution because while the greedy algorithm slightly out preforms in efficiency, dynamic programming is more accurate and correct than the greedy algorithm.

There is no one perfect solution to the knapsack problem. Depending on the size of the list of items that one is trying to put in the knapsack, some algorithms are more efficient than others. Additionally, the best method also depends on the balance expected between accuracy and speed. If accuracy can be sacrificed for the sake of efficiency, then the greedy algorithm may be a better option. The better knapsack solution changes for each individual knapsack.