Assignment 3

# Part 1: 0-1 Knapsack Problem, Dynamic Solution

To qualify the problem to be solved using Dynamic programming it needs to exhibit both optimal substructure and have overlapping sub problems. Defining a solution as well as a sub problem will allow the above properties to be proven. A candidate solution to the problem can be defined as follows:

Where represents an item with some value, and weight, that is in the solution . is a candidate solution when . is an optimal solution if .

A sub problem of , is a solution to the problem where .

## Optimal Substructure:

Given an optimal solution and its sub problem assume that isn’t optimal. This means that there is some solution to the sub problem where . As isn’t optimal we can assume . But since and are both solutions to the sub problem with weight then and are both solutions to the problem which is supposed to be an optimal solution to. However, since and it follows that and so implies a contradiction to S being optimal. This implies that must be optimal if is optimal. This follows for the sub problems of down to the trivial problem where the sub problem must satisfy .

## Recursive Solution:

Following that the problem exhibits optimal substructure, the solution set can be built up by sub problems to the total problem, making sure that each sub solution is optimal. The trivial sub problem is when the weight is 0. In this case the optimal solution is the empty set. From then on the solution to sub problem with weight can be found by examining each potential item not a part of the current sub solution. For each item it will either be a part of the sub solution or not, with the value of the optimal solution given the remaining items will be the max of the following cases:

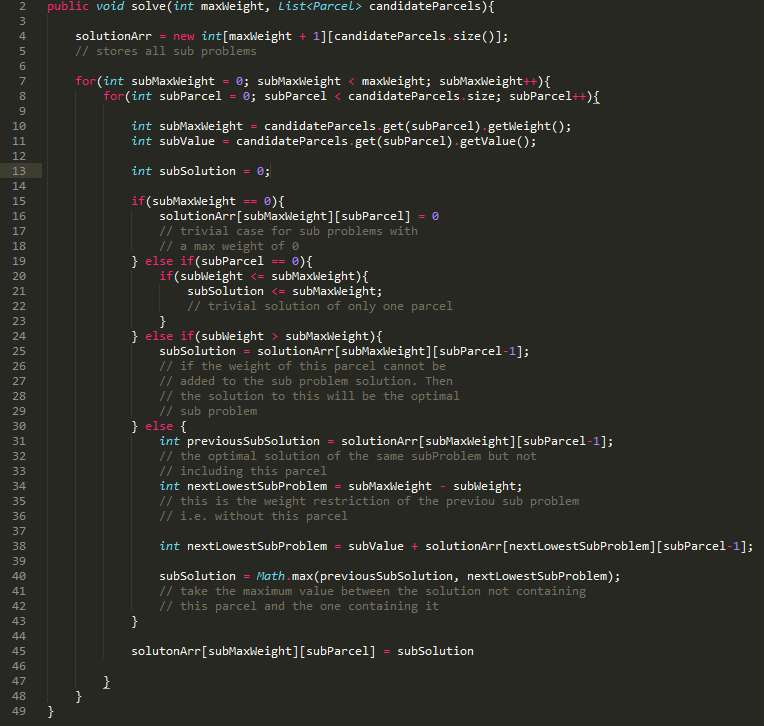
1. The value of the sub solution with the same weight as this one but excluding this item.
2. Value of this item plus the value from the sub problem with weight

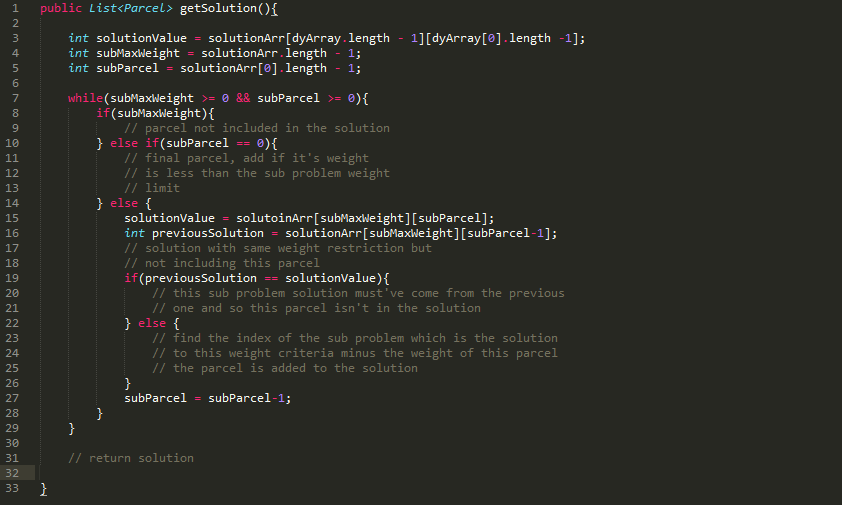
When the weight of the current item is greater then only case 1 applies.

## Overlapping Sub problems:

Based on the above recursive solution the problem will repeat sub problems when calculating greater sub solutions.

As the 0-1 knapsack problem exhibits both optimal substructure and overlapping sub problems a dynamic program will be suitable. Based on the above recursive solution a dynamic program which stores the solution to sub problems is guaranteed to give an optimal solution.





## Cost:

The dynamic algorithm will always have the same cost, as it populates an array with all the solutions to sub problems it will always be performing the same number of operations for some given input size.

The number of sub problems is dependent on the maximum weight of the knapsack and the total number of potential parcels to add to it. The exact relationship is:

Retrieving the value of the optimal packaging is as the final entry in the array of solutions is the total solution. However, to obtain the solution of which parcels are to be added the array needs to be traversed backward. This traversal decrements the number of candidate parcels to consider as it goes back through all the sub problems and so the asymptotic complexity will be .

So the total complexity of the dynamic solution is , will always dominate on it’s own and so the total complexity is just .

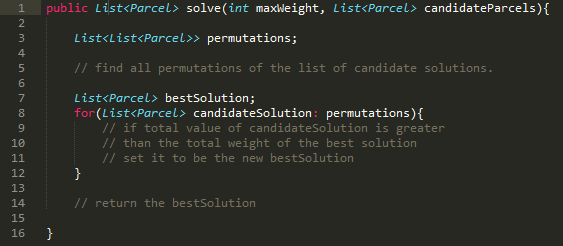
When using a barometer in the implementation this cost is evident. This barometer counts the number of iterations of the for loops in the implementation. As can be seen from the above set of trials the cost follows where is 25 here and is increasing by 1.

## Testing:

Testing was done using an online[[1]](#footnote-1) implementation for a small number of instances. I used the enumeration algorithm implemented for part 2 but with a restriction for the 0-1 Knapsack problem to generate a larger number (although the input size for each trial was small because enumeration is very inefficient) of trials. This was done as I am much more confident that the enumeration algorithm returns the correct solution since it just finds all possible combinations of parcels.

# Part 2: 0-N Knapsack Problem Solved with Brute Force

I chose to solve the 0-N Knapsack Problem using enumeration, mostly due to the suggestion rather than finding a more efficient solution. Of course the simplicity is a good motivator for this approach also.



This works simply for the 0-1 Knapsack problem and is guaranteed to return the optimal solution if the permutation is done correctly.

For the 0-N Knapsack problem it is a little more difficult to find all the permutations which are suitable. I did this by creating an intermediary list of candidate parcels, where each parcel was duplicated to appear times. This represents each parcel being able to appear a maximum number of times even though the knapsack is unbounded.

# Part 3: 0-N Dynamic Solution

This works as an extension of the 0-1 Dynamic Solution. It creates a parsed list of candidate parcels, where each parcel appears times. Then this list is solved using the 0-1 Dynamic Solution. The solution list returned by the 0-1 case using the parsed list is then parsed back into terms of the original input.

This means that the 0-N implementation is guaranteed to find the optimal solution if the 0-1 implementation is. As it just uses an extended candidate solution to find a total solution with more than one number of each parcel.

1. http://karaffeltut.com/NEWKaraffeltutCom/Knapsack/knapsack.html [↑](#footnote-ref-1)