KnapSack-HanfengXU-ProjectCS160

Run with: python knapsack.py

Problem Definition:

There are two versions of the problem, namely integral knapsack, and fractional knapsack:

- integral knapsack: Take an item or leave it
- fractional knapsack: Can take a fraction of an item (infinitely divisible)

The problem is defined as follows: Consider a thief gets into a home to rob and he carries a knapsack. There are fixed number of items in the home — each with its own weight and value — Jewelry, with less weight and highest value vs tables, with less value but a lot heavy. To add fuel to the fire, the thief has an old knapsack which has limited capacity. In the integral knapsack version, he can't split the table into half or jewelry into 3/4ths. He either takes it or leaves it; in the fractional knapsack, he is able to take a fraction of items. The objective is to get as much value as possible while stay within the limit of the weight capacity.

Fractional Knapsack

Pre-coding Proofs and Learning:

In order to build the code, I first had to understand the problem. The book mentions:

> We can solve the fractional knapsack problem by a greedy strategy, but we cannot solve the 0-1 problem by such a strategy. To solve the fractional problem, we first compute the value per pound v_i / w_i for each item. Obeying a greedy strategy, the thief begins by taking

> as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth, until he reaches his weight limit W. Thus, by sorting the items by value per pound, the greedy algorithm runs in O(nlgn) time. We leave the proof that the fractional knapsack problem has the greedy- choice property as Exercise 16.2-1.

So I thought it would be interesting to prove the greedy choice property of fractional knapsack, and also prove why it does not work for integral knapsack.

Proof of Greedy choice property:

Let j be the item with maximum v_i/w_i. Then there exists an optimal solution in which you take as much item j as possible.

Proof:

- For the sake of contradiction, suppose that there exists an optimal solution where you did not take as much item j as possible.
- If the knapsack is not full, add some more of item j would give us a better solution with higher value. Thus we reach a contradiction and have to assume the knapsack is full.
- If the knapsack is full, there must exist some item $k \neq j$ with $v_k/w_k < v_j/w_j$ in that knapsack. Also, not all of item j is in the knapsack
- Therefore, if we take a piece of k with weight w out of the knapsack and put in a piece of j with w weight in.
- By doing so, we will increase the knapsack total value since $v_k/w_k < v_j/w_j$.
- Thus the original assumption cannot be optimal.
- We have proved by contradiction the greedy choice property of fractional knapsack.

Coding for greedy:

Since this greedy algorithm requires sorting of the items according to value per weight, I thought it would be a good idea to create items classes. Thus when items are sorted, the corresponding data values of weight and value moves with this item. In the actual algorithm, we first initialize all the items objects given the data, then the list of items are sorted in decreasing order according to their rankings of v_i/w_i. In general, I utilized "write a little, test a little", to see, for example, if the corresponding data values of weight and value are moving when it is sorted according to cost. I also tested if sorting worked as desired with multiple inputs, they are correctly rank the items. (* For ease of computation, I used flooring operator for calculating cost. As a result, items with similar cost might not be 100% precise)

The list of sorted items is looked through from left to right, if current item can fit, we decrease the capacity by its weight and increase the total value by its value, then we move to next item; if it does not fit, we fill the bag to full with fraction of that item, terminating the loop as the bag is now full .

As we test multiple inputs, it seems fractional knapsack can be solved by greedy algorithm, with runtime of O(nlogn) mainly used for sorting based on cost function.

Integral(0/1) Knapsack

Pre-Coding proofs:

First let's see why greedy does not work for the integral case.

When greedy fails:

Consider a backpack with a weight capacity of 4, and items with the following weights and values:

Item Weight: A: 3 B: 2 C: 2

Value: A: 1.8 B: 1 C: 1

Value/Weight: A: 0.6 B: 0.5 C: 0.5

If we apply *Greedy* on value you will first select item A, so the residual weight capacity will be 1. Since both items B and C weigh more than that residual value, you won't be

able to add any more items since fraction is no longer allowed. Now, we have reached a feasible solution but not optimal.

The total value for greedy is 1.8.

The optimal solution, however, is to choose items B and C, which together exactly weigh the full capacity and have a combined value of 2.

Similar to what the coin exchange problem in homework demonstrated,
Greedy algorithm can fail when it chooses a set of items that don't take up the whole available capacity. A different set of items that fills more of the available capacity will sometimes be a better choice.

Valid Solutions: Brute Force and Dynamic Programming

The book mentions that

> Both knapsack problems exhibit the optimal-substructure property. For the 0-1 problem, consider the most valuable load that weighs at most W pounds. If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_i$ that the thief can take from the n-1 original items excluding j .

First solution that came up my mind was a **brute force** one, where all possible subsets are tried and have their total weight calculated.

We can implement by using recursion: Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

To consider all subsets of items, there can be two cases for every item.

- The item is included in the optimal subset.
- The item is not included in the optimal set.

To implement this with recursion, I start with the base case: when we are out of space(weight) or when we are out of items to put in. Then, the other base case is when the weight of nth is too large for the capacity of W, we as a result do not include it in the optimal solution.

Finally we have the most important recursive cases, where we return the maximum of two case:

1. when nth item is included
2. when nth item is not included

Using recursion, we should have traversed all the possible subset, so the solution we get is optimal.

To test this solution, a few cases with easily calculated optimal solution is used. Then a more complicated test case was used, it will be later tested against the dynamic programming solution.

For the **dynamic programming** solution, I was not entirely sure how to do it. So consulted on Youtube:

The dynamic programming solution was implemented:

First, we create a 2-dimensional array of n+1 rows and w+1 columns.

A row number *i* represents the set of all the items from rows 1 to i. For instance, the values in row 3 assumes that we only have items 1, 2, and 3. A column number *j* represents the weight capacity of our knapsack.

Putting everything together, an entry in row *i*, column *j* represents the maximum value that can be obtained with items 1, 2, 3 … *i*, in a knapsack that can hold *j* weight units.

The base cases: For instance, at row 0, when we have *no items* to pick from, the maximum value that can be stored in any knapsack must be 0. Similarly, at column 0, for a knapsack which can hold 0 weight units, the maximum value that can be stored in it is 0.

The maximum value that we can obtain when we include item *i: similar to brute force solution,* we first need to compare the weight of item *i* with the knapsack's weight capacity. Obviously, if item *i* weighs more than what the knapsack can hold, we can't include it. In that case, the solution to this problem is simply the maximum value that we can obtain without item *i* (i.e. the value in the row above, at the same column).

When item *i* weighs less than the knapsack's capacity. We thus have the option to include it, if it potentially increases the maximum obtainable value. The maximum obtainable value by including item *i* is thus equals the value of item *i* itself + the

maximum value that can be obtained with the remaining capacity of the knapsack. We obviously want to make full use of the capacity of our knapsack, and not let any remaining capacity go to waste.

Therefore, at row *i* and column *j* (which represents the maximum value we can obtain there), ***we would pick either the maximum value that we can obtain without item i, or the maximum value that we can obtain with item i, whichever is larger.***

A simple knapsack problem with capacity of 8 and 4 elements will end up with the below table:

value = {1,2,5,6}

weight = {2,3,4,5}

0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1
0	0	1	2	2	3	3	3	3
0	0	1	2	5	5	6	7	7
0	0	1	2	5	6	6	7	**8**

We find the answer at the right bottom corner, and tracing back, we got item 2 and 4. Maximum value of 8.

This table and comparison to brute force algorithm demonstrated the epitome of dynamic programming, where some spaced is used to save runtime.

The brute force does not use extra space, but it runs in O(2^n) time. Whereas the DP solution uses O(N*W) for both space and runtime.

I found some complicated test cases(with 10, 20, 100 items) online and checked the results.

I used a timer to compare Brute force and DP:

num of items	3	10	20	100
Brute Force	0.000015	0.000364	0.561308	over 10 minutes, I stopped it
DP	0.000094	0.001312	0.008535	0.211874

As we observe, for smaller input, the brute force solution is faster possibly because the time it takes for DP to create the table and checks conditional statements. As brute force has exponential runtime, when the num of items is 20, DP is already much faster

and efficient. For entrees over 100 items, I waited more than 10 minutes for brute force to complete but it did not. The DP was able to finish in 0.2 sec. Thus we can see asymptotically, DP is much more efficient as it does not have exponential runtime