**A 0/1 Knapsack Problem to Optimize Shopping Discount under Limited Budget**

**Sudhan Bhattarai & Ebisa D. Wollega**Department of Engineering  
Colorado State University-Pueblo   
Colorado, USA   
[s.bhattarai@pack.csupueblo.edu](mailto:s.bhattarai@pack.csupueblo.edu), [ebisa.wollega@csupueblo.edu](mailto:ebisa.wollega@csupueblo.edu)

**Abstract**

Online shoppers usually look for their desired products with maximum discount possible, as larger discounts tend to influence the consumer behavior. However, the price of a product also plays significant role in buying decision because of the limited budget. In this paper, to optimize the shopping decision, a 0/1 knapsack problem is formulated, and a comparison of the traditional recursive solution method and the dynamic programming optimization approach is made. The results show that the dynamic programming method outperforms the recursive solution method in terms of computational time for large number of purchase items.

**Keywords:** 0/1 Knapsack, Recursion, Dynamic Programming, Optimization

# Introduction

Shopping is a frequent task for many people, where some may have a huge wish list but a limited capital on hand. Furthermore, the discount that is getting on each shopping is eye-catching through several coupons, promotions, and sales. In this paper, an optimization model is built to choose the items to buy from an Amazon shopping wish list to get an aggregate maximum discount yet spending only a limited budget. The problem is formulated as a 0/1 knapsack problem and solved by using recursive solution method and the dynamic programming approach.

The 0/1 knapsack (KS) problem is an optimization problem defined for a list of items with their respective values and weights. The 0/1 knapsack problem can also be defined as a single constraint integer program (Sahni, 1975). The objective of the problem is to pick a set of items from a list such that the total value picked is maximized while maintaining the total weight allowance (Black, 2019). There are different ways to solve KS problem. Branch & bound, greedy, and brute-force are a few common examples. Brute force is the simple and traditional approach in which the best combination of selected items is chosen from enumeration of all the possible combinations. 1 and 0 decisions are given to an ‘ith’ item from a list of ‘N’ items to decide whether the item is chosen or not (Hristakeva & Shrestha, 2005). Memorizing the recursive terms in brute-force enumeration and using it from storage in future is the key concept of dynamic programming. The dynamic programming is better than the branch-and-bound for tracing back optimal solution for sensitivity analysis purpose; however, it suffers from the curse of dimensionality when used for bigger problems (Bellman & Dreyfus,1959; Powell, 2007; Martello, Pisinger, & Toth, 2000). Martello et al. (2000) proposed a method to terminate the recursion by introducing tight bounds above the worst-case bounds to reduce the search space. The bound strategy for DP proposed by (Morin & Marsten, 1976) considers the items with higher weight in knapsack first which led to early lower bounds. The theory of lower bounds was further confirmed by (Toth, 1980), also adding the fact that upper bounds are low in such method as the weights of lastly assessed items are lower and also the room they leave for future items is higher. Rong et al. (2012) presented a discounted 0/1 knapsack problem with ‘n’ product groups with each group comprising of 3 items and only one item to be selected from each group to satisfy the total budget and maximizing the discount. They mainly discussed three DP algorithms: basic sequential, sparse nodes, and labeling methods. The sparse nodes method was an improvement on classical basic sequential method in which numerous unwanted non-optimal calculations are done. Unlike in the sparse nodes, the labeling method only eliminated a part of ‘dominated states’ being called an intermediate solution. Basic sequential method was faster for larger sized problems than the other two. Shahleen and Sleit (2016) compared genetic algorithm (GA), dynamic programming, and traditional greedy methods to solve KS problem. They claimed that dynamic programming is more efficient than the other two in terms of computation speed and solution quality, for problem size under 60 thousand set selection items. They recommended genetic algorithm over DP and greedy heuristic for selection items larger than 100 thousand, as GA located near optimal solution with less search space while DP took longer time for one hundred percent search to locate optimal solution. Ahmed and Younas (2011) proposed a genetic algorithm method with repair evaluation function and greedy invert mutation by transposing the initial optimal solution achieved from dynamic programming to reduce the computational time.

There have been numerous remarkable researches on modeling the optimization problems in 0/1 knapsack. As restated above, DP method, genetic algorithm and hybrid algorithms are commonly used to solve the 0/1 knapsack problem. The literature tells us about the significance of computational time and challenges on bigger problems of dynamic programming for 0/1 KS problems. This paper is an application of 0/1 knapsack problem on Amazon shopping and shows that dynamic programming is faster than traditional recursive solution method for the selected number of items.

# Problem Definition

Suppose that there are ‘n’ different items in a list with index i (i = 0, 1, 2, …, n-1). Let and be the price and discount achieved for an item ‘i’ in the item list. If there is a total “Capital” available, the goal is to try to select items in such a way that the total price of the items is constrained by the ‘Capital’ and total discount on selection (Discount) is maximum. Let, be the binary decision whether to choose an item ‘i’ or not. So, equals to ‘1’ if an item is selected, or ‘0’ otherwise.

Hence, our objective function is,

Subject to,

# Data Collection

Twenty-two wish list items data shown in Table 1 below were collected from Amazon.com in the first week of December 2019. Each item has the corresponding name, price of item after discount, original price before discount, and the discount amount offered on the original price.

Table 1: Amazon shopping wish list

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SN** | **Item** | **Price** | **Original Price** | **Discount** |
| 1 | Formal shoe | 23.10 | 36.99 | 13.89 |
| 2 | Tablet | 29.99 | 49.99 | 20.00 |
| 3 | Headphone | 59.99 | 149.99 | 90.00 |
| 4 | Printer | 19.99 | 59.99 | 40.00 |
| 5 | Sunglass | 34.99 | 89.99 | 55.00 |
| 6 | Sneaker | 55.24 | 89.95 | 34.71 |
| 7 | Running Shoe | 66.00 | 85.00 | 19.00 |
| 8 | Watch | 74.99 | 91.75 | 16.76 |
| 9 | Wireless speaker | 99.00 | 199.00 | 100.00 |
| 10 | Jogger | 40.50 | 54.00 | 13.50 |
| 11 | Arsenal kit | 83.95 | 90.00 | 6.05 |
| 12 | Gym shoe | 56.66 | 69.95 | 13.29 |
| 13 | Laptop bag | 97.99 | 113.97 | 15.98 |
| 14 | Bluetooth headphone | 78.00 | 129.99 | 51.99 |
| 15 | Skateboard | 36.99 | 59.99 | 23.00 |
| 16 | Swimsuit | 31.95 | 54.00 | 22.05 |
| 17 | PS controller | 38.99 | 64.99 | 26.00 |
| 18 | Aquarium tank | 27.89 | 32.89 | 5.00 |
| 19 | Bike helmet | 29.99 | 33.96 | 3.97 |
| 20 | Hair dryer | 38.99 | 59.99 | 21.00 |
| 21 | Bed sheet | 31.82 | 39.78 | 7.96 |
| 22 | Power bank | 39.94 | 46.99 | 7.05 |

# Solution Methods

In this section, overviews of the recursive solution method and the dynamic programming method are presented. How each method is applied to the underlying problem is also discussed.

## Recursive Solution Method

Recursive solution is a traditional algorithm used to solve the 0/1 knapsack problem. In this algorithm, a recursive function calls itself to calculate lower terms and gather the results together to locate solution of higher terms. Recursive calculations are one-time calculations in which all the recursions necessary for final result are done independent of each other. Hence, for larger problems recursive functions tend to calculate the same term for many times giving rise to computational complexity. In our solution, the last item is picked first by the algorithm and the recursive function will access the remaining items accordingly. The algorithm checks the maximum discount we can get in case of selecting the item by adding the current discount of that particular item and the future possible discounts with remaining budget once the price of selected item is subtracted from the initial budget. The rejection of the item comes if the discount from other items is more than the maximum, we can get by selecting the current item. Let, B, P, D and N represent the total budget, price list, discount list for each item and the number of items. Following is the algorithm for the recursive solution:

*define recursive (B , P, D, N):*

*if N == 0 or B == 0*

*return 0*

*if P [Nth item] > B*

*return recursive (B, P, D, N-1)*

*total discount = max (D[Nth item] + recursive (B - P[Nth item], P, D, N-1 , recurs(B, P, D, N-1)*

*return total discount*

## Dynamic Programming

In recursive solution, the value of a specific term is calculated in each iteration and hence that will increase computational time significantly making larger models impractical to compute. Dynamic programming is fast and efficient in terms of the computational time in a sense that, DP approach store the value for an item in an iteration which is used in future eliminating the need of recomputing the same value over and over again. Hence, DP will solve the smaller subproblems to find and store solution for items that are used in future for computing bigger terms and successively the optimal term.

In the following Table 2, columns represent a range of total budget available (Budget = $0, $1, $2, $3, …., $800) and rows represent the range of items that are chosen in 0/1 knapsack solution (item = 0, 1, 2, 3, …., 21). For every cell, where a row is an item and columns are budgets representing the maximum discount, we can get with decisions of whether choosing the item or not. For a column (i.e. a budget available to choose different combination of items for maximum discount value). When we decide whether to choose the next item or not, we compare two results. First, whether placing the item will increase the total discount while maintaining total budget constraint. Second, the total discount we get for the same budget with previous item insertion. Whichever is greater, we choose the value and store the value in a table.

Table 2: DP table format for 0/1 knapsack problem

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Price → Item ↓ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | …………… | 800 |
| 0 |  |  |  |  |  |  |  |  |  |  | …………… |  |
| 1 |  |  |  |  |  |  |  |  |  |  | …………… |  |
| 2 |  |  |  |  |  |  |  |  |  |  | …………… |  |
| 3 |  |  |  |  |  |  |  |  |  |  | …………… |  |
| 4 |  |  |  |  |  |  |  |  |  |  | …………… |  |
| .  .  .  21 | .  .  .  . | .  .  .  . | .  .  .  . | .  .  .  . | .  .  .  . | .  .  .  . | .  .  .  . | .  .  .  . | .  .  .  . | .  .  .  . |  |  |

The maximum discount for all cells (i\*j), i = item in row, j = budget capacity in column, is chosen and stored in that cell using the following general procedure:

1. Recalling previous best discount without inserting item (i) with budget capacity (j):

**Previous best discount = DP table [i-1, j]**

This gives the value of discount with the same budget capacity but without selecting the item (i). This is the value stored in table at (i-1, j).

1. Calculating new best discount value after we insert item (i):

**New best discount = DP table [i] + DP table [i-1, j – P[i]]**

This represent the total discount after we choose to add item (i) in our knapsack backpack. Here, if we choose to insert an item, this item’s discount money will count in our objective function but consequently total budget capacity to select other items reduces by its price. Hence, the total discount will be the item’s discount plus discount we get for budget less than the price of item which is stored in a cell in (i-1) row.

1. Max discount at cell is:

**DP table [i, j] = max (Previous best discount, New best discount)**

This decision will finally choose the maximum value of discount we get after considering up to ‘i’ items with budget limit of ‘j’. If the total discount from the previous term is bigger, there is no point of adding new item to our knapsack backpack as it will not help to increase the total discount, hence we will choose the previous best value. But if the new total discount value is bigger after we consider new item to select in our knapsack backpack, we select new value as this helped on increasing total discount value.

We keep calculating, storing, and reusing previous value of total discount for all the cells, considering up to all 22 items, and calculating for all $800 budget capacity. Finally, we will get our optimal value with maximum discount at the end of our table at cell (21, 800). Following is the algorithm for DP (Table 2 is initially an empty array):

*define Dynamic Programming (DP table, D, P, B, N):*

*loop item (0 to N-1)*

*loop budget (0 to B)*

*if item = 0 or budget = 0*

*DP table [item, budget] = 0*

*continue*

*if P (item) > budget*

*DP table [item, budget] = DP table [item – 1 , budget]*

*continue*

*previous best discount = DP table [item-1, budget]*

*new best discount = D[item] + DP table [item-1, budget - P[item]]*

*DP table [item, budget] = max (last best discount, new best discount)*

*return DP table [N-1, Budget]*

# Computation and Results

There are few assumptions made to solve the problem as follows. For the computational ease, the prices of items are rounded up to the nearest integer. More than one item may fall under the same product category, but they are available for selection. Any combination of the items from a list is accepted with maximum discount in total.

The algorithms were implemented in Python 3 programming language to solve the problem. Intel i7-8th gen, 64-bit, 1.9 GHz computer with 8 GB RAM and 4 GB GPU machine is used. The following is the results found.

Total items to select = **16** items

Total budget to spend = **$ 788**

Total discount achieved = **$ 560.19**

Table 3: Chosen items

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **SN** | **Item** | **Price** | **Discount** | **SN** | **Item** | **Price** | **Discount** |
| 1 | Formal shoe | 24 | 13.89 | 9 | Wireless speaker | 99 | 100 |
| 2 | Tablet | 30 | 20 | 10 | Jogger | 41 | 13.5 |
| 3 | Headphone | 60 | 90 | 12 | Gym shoe | 57 | 13.29 |
| 4 | Printer | 20 | 40 | 14 | Bluetooth headphone | 78 | 51.99 |
| 5 | Sunglass | 35 | 55 | 15 | Skateboard | 37 | 23 |
| 6 | Sneaker | 56 | 34.71 | 16 | Swimsuit | 32 | 22.05 |
| 7 | Running Shoe | 66 | 19 | 17 | PS controller | 39 | 26 |
| 8 | Watch | 75 | 16.76 | 20 | Hair dryer | 39 | 21 |

## Total time taken by the recursive function to get the solution = 92.1639 seconds

Total time taken by the dynamic Programming to get the solution = **0.3896** seconds

The time taken by DP to solve the problem is significantly lower as compared to the time taken by the recursive function.

The graphs in Figure 1 show the comparison of the two algorithms computation times over a range of items. The upper blue graph represents the recursive computation time, while the bottom green graph shows the DP computation time. There are not large differences in computational time for the number of items less than 16. However, the computational time for the recursive call tends to resemble exponential growth while the DP time remains fairly constant.

Table 4: Experiment results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SN | Number of items | Total budget | Time by Recursive (sec) | Time by DP (sec) |
| 1 | 10 | 300 | 0.0146031 | 0.051614 |
| 2 | 12 | 400 | 0.0604665 | 0.084242 |
| 3 | 14 | 500 | 0.2637505 | 0.167336 |
| 4 | 16 | 600 | 1.4419463 | 0.179103 |
| 5 | 18 | 700 | 5.8104126 | 0.251588 |
| 6 | 20 | 800 | 22.0314566 | 0.323022 |
| 7 | 22 | 800 | 92.1639999 | 0.389633 |

A screenshot of a cell phone

Description automatically generated

Figure 1: Graphical representation for computational time

# Conclusion

The knapsack problem is a very helpful approach to formulate daily life problems to make favorable decisions. In this paper, we took data for 22 different shopping wish list items from Amazon.com with their respective prices and discounts accrued. The objective of the research was to find the list of items to buy from the wish list such that the total money spent is within budget, and also the total discount accumulated is maximized. We formulated the problem as 0/1 knapsack and applied methods of recursive algorithm and dynamic programming to solve the problem. The results show that the performance of recursive solution method is comparable to that of the dynamic programming in terms of computational time for small size problems. However, as the size of problem increases, the computational time by recursive approach explodes compared to that of the dynamic programming. Hence, it is recommended to use the dynamic programming method to make purchase choices on Amazon.com shopping to accumulate maximum discount under limited budget. Since the dynamic programming method consumes memory, it seems interesting to apply an approximate dynamic programming (ADP) algorithm to see if the need of space for larger and expensive shopping lists could underwrite the limitations of the traditional DP.

**References**

Ahmed, Z., & Younas, I. (2011). A Dynamic Programming based GA for 0-1 Modified Knapsack Problem. *International Journal of Computer Applications, 16*(7), 1-6.

Black, P. E. (2019). knapsack problem. *Dictionary of Algorithms and Data Structures.* National Institute of Standards and Technology.

Bellman, R., & Dreyfus, S. (1959). Functional approximations and dynamic programming. Mathematical Tables and Other Aids to Computation, 247-251.

Hristakeva, M., & Shrestha, D. (2005, April). Different approaches to solve the 0/1 knapsack problem.

Martello, S., Pisinger, D., & Toth, P. (2000). New trends in exact algorithms for the 0–1 knapsack problem. *European Journal of Operational Research, 123*(2), 325-332.

Morin, T. L., & Marsten, R. E. (1976). Branch-and-bound strategies for dynamic programming. *Operations Research, 24*(4), 611-627.

Powell, W. B. (2007). Approximate Dynamic Programming: Solving the curses of dimensionality (Vol. 703). John Wiley & Sons.

Rong, A., Figueira, J. R., & Klamroth, K. (2012). Dynamic programming based algorithms for the discounted {0–1} knapsack problem. *Applied Mathematics and Computation, 218*(12), 6921-6933.

Sahni, S. (1975). Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM (JACM)*, 115 - 124.

Shaheen, A., & Sleit, A. (2016). Comparing between different approaches to solve the 0/1 Knapsack problem. *International Journal of Computer Science and Network Security (IJCSNS)., 16*(7), 1.

Toth, P. (1980). Dynamic programming algorithms for the zero-one knapsack problem. *Computing, 25*(1), 29-45.

**Mr. Sudhan Bhattarai**is a graduate student at Colorado State University-Pueblo with major in Master of Science in Industrial & Systems Engineering (MSISE). Sudhan’s research interest is primarily in the fields of data science, optimization, and artificial intelligence with the application in healthcare analytics. Sudhan’s profile can be found at <https://github.com/sudhan-bhattarai>.

**Dr. Ebisa Wollega** is an assistant professor of engineering at Colorado State University-Pueblo. His research areas include applying large-scale optimization methodologies and artificial intelligence algorithms in wide range of applications. His profile can be found at <https://www.csupueblo.edu/profile/ebisa-wollega/index.html>.