Sudhan Bhattarai

**Solving a Knapsack problem with Dynamic Programming to get maximum discount in multiple Amazon items shopping with limited budget**

# Introduction

We basically do a lot of shopping in Amazon in daily basis. We often try to look for our desired products with maximum discount possible and more discount gives more satisfaction on buying. But, in other hand, price of product plays important role in our decision to buy because of our limited budget. In this project I will be making a model to decide on which products to buy within my limited budget of $1000 from a list of my desired ‘30’ different products in Amazon website with their respective price and discount. I will formulate the problem as a 0/1 knapsack problem and will use dynamic programming approach to solve the knapsack problem to get optimal solution, which is a set of items to buy under the budget of $1000 such that the discount I get in total will be maximum.

# Problem Definition:

A knapsack problem is an optimization problem defined for a list of items with respective values and weights. In knapsack decision, we usually have limited space for weight, and we try to get maximum values to fit into that weight. The objective is to pick a set of items from list such that the total value picked is maximum while maintaining the total weight allowance (Black, 2019).

Suppose, we have ‘n’ different items in a list with index i (i = 1, 2, …, n). Let and be the price and discount achieved for an item ‘i’ in the list. If we have total capital available ‘Capital’ in our knapsack backpack, we try to select items in a way such that the total weights of items are constrained by ‘Capital’ and total discount on selection (Discount) is maximum. Let, be the decision to whether choose item ‘i’ or not. So, equals to ‘0’ if we do not select item in our knapsack and equals to ‘1’ if we do.

Hence, our objective function is,

s.t,

# Data Collection:

Data for my project is my shopping wish list of 30 different items. I listed the things of my need on scratch and searched for good match in Amazon website. The list possesses item name, price of item, original price and discount offered. Item headline presented is shown in .xlsx file named ‘data’ in same directory as this .doc file. Original values for prices and discount were float values but for computational ease I rounded them up to nearest integer. Main assumptions underlined are as follows: Every item is on wish list without regarding the fact more than one items might fall in same product category, I would be happy to get any combinations of items from the list but with maximum discount in total.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SN** | **Item** | **Discounted Price** | **Original Price** | **Discount** |
| 1 | Formal shoe | $23 | $36.99 | $14 |
| 2 | Tablet | $30 | $49.99 | $20 |
| 3 | Headphone | $60 | $149.99 | $90 |
| 4 | Printer | $20 | $59.99 | $40 |
| 5 | Speaker | $45 | $59.99 | $15 |
| 6 | Sneaker | $55 | $89.95 | $35 |
| 7 | Running Shoe | $66 | $85.00 | $19 |
| 8 | Watch | $75 | $91.75 | $17 |
| 9 | Wireless speaker | $99 | $199.00 | $100 |
| 10 | Jogger | $41 | $54.00 | $14 |
| 11 | Arsenal kit | $84 | $90.00 | $6 |
| 12 | Gym shoe | $57 | $69.95 | $13 |
| 13 | Laptop bag | $98 | $113.97 | $16 |
| 14 | Bluetooth headphone | $78 | $129.99 | $52 |
| 15 | Skateboard | $37 | $59.99 | $23 |
| 16 | Sweater | $26 | $27.82 | $2 |
| 17 | PS controller | $39 | $64.99 | $26 |
| 18 | GTA V game | $85 | $99.99 | $15 |
| 19 | Bike helmet | $30 | $33.96 | $4 |
| 20 | Hair dryer | $39 | $59.99 | $21 |
| 21 | Bed sheet | $32 | $39.78 | $8 |
| 22 | Shirt | $28 | $39.99 | $12 |
| 23 | Power bank | $40 | $46.99 | $7 |
| 24 | Sunglass | $35 | $89.99 | $55 |
| 25 | Swimsuit | $32 | $54.00 | $22 |
| 26 | Aquarium tank | $28 | $32.89 | $5 |
| 27 | Blazer | $86 | $99.99 | $14 |
| 28 | Kitchen mixer | $37 | $39.67 | $2 |
| 29 | Microwave bowl | $16 | $17.99 | $2 |
| 30 | Protein powder | $24 | $28.99 | $5 |

# Solution

Solution of above stated problem is generated with collaboration of 0/1 knapsack approach and concept of dynamic programming. One popular solution method for knapsack problems is brute-force algorithm but I am not going to talk about it in this project. To give idea on my main solution method, I would like to introduce the idea of recursion first.

## Recursive Solution

Recursive solution is a popular solution to address knapsack problem. A recursive solution uses a recursive function to calculate a large term by calling the same function over and over. A recursive function call itself to calculate lower terms and gather the result 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 same term for many times giving rise to computational complexity. I haven’t shown any sole recursive methods in this project, but I have introduced an improved tool of recursion in which each calculation are stored for future use, trading space for time, called dynamic programming.

## Dynamic Programming

Dynamic programming (DP) is a problem-solving tool in which a larger problem is solved by dividing it into smaller subproblems. One classic solution method for above stated problem would be enumeration of all the possible solutions and find out the optimal. Other solution methods could be use of recursive functions. In either of those solutions, 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 computational time in a sense that, DP approach store the value for an item iteration which is used in future eliminating the need of recomputing that 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 following table, columns represent total budget of chosen items (Price = 0, 1, 2, 3, ….., 1000) and rows represent items that are chosen in knapsack backpack (item = 0, 1, 2, 3, ….., 30. For every cell, where row is an item and column is budget represent maximum discount we can get with decisions of whether choosing the items or not. For a column (i.e. a budget available to choose different combination of items for maximum discount value), when we move to decision of whether chose or not next item, we compare two results. First, whether placing the item will increase total discount while maintaining total budget constraint and second, the total discount we got for the same budget with previous item insertion. Whichever is more, we will choose the value and store the value in table.

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

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

Previous best discount = D [i-1, j]

This gives the value of discount with 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 = D [i] + D [i-1, j – P[i]]

This represent the total discount after we chose 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, 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:

D (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, total discount from previous term is bigger, there is no point of adding new item to our knapsack backpack as it won’t help to increase total discount hence, we choose previous best value. But, if 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 30 items and calculating though all $1000 budget capacity. Finally, we will get our optimal value with maximum discount at end of our table at cell (29, 1000).

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Price → Item ↓ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ……………………………….. | 1000 |
| 0 |  |  |  |  |  |  |  |  |  |  | ……………………………….. |  |
| 1 |  |  |  |  |  |  |  |  |  |  | ……………………………….. |  |
| 2 |  |  |  |  |  |  |  |  |  |  | ……………………………….. |  |
| 3 |  |  |  |  |  |  |  |  |  |  | ……………………………….. |  |
| 4 |  |  |  |  |  |  |  |  |  |  | ……………………………….. |  |
| .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . | .  .  .  .  .  . |  |  |
| 29 |  |  |  |  |  |  |  |  |  |  |  |  |

# Calculation and Result:

As the table of array used in my methodology contained 1001 columns and 29 rows, calculations on manual basis would be really hard. I wrote a program in python 3.7 to import data from excel file and then compute accordingly to form the number table designed above. On executing my program, I got following results:

Total Items to select = 20 Items

Total Budget to Spend = $977

Total Discount Achieved = $610

Time Taken (secs) on ‘10’ Executions = {0.42, 0.5, 0.48, 0.35, 0.4, 0.48, 0.44, 0.49, 0.49, 0.45},   
Average time = 0.45 secs  
(64 bit- Intel i7, 1.9 GHz, 8 GB RAM, 4 GB GPU, Python 3.7.4.)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Item** | **Discounted Price** | **Original Price** | **Discount** | **Decision** | **Price Paid** |
| Formal shoe | $23 | $36.99 | $14 | 0 | $0 |
| Tablet | $29 | $49.99 | $21 | 1 | $29 |
| Headphone | $59 | $149.99 | $91 | 1 | $59 |
| Printer | $19 | $59.99 | $41 | 1 | $19 |
| Speaker | $44 | $59.99 | $16 | 1 | $44 |
| Sneaker | $55 | $89.95 | $35 | 1 | $55 |
| Running Shoe | $66 | $85.00 | $19 | 1 | $66 |
| Watch | $74 | $91.75 | $18 | 1 | $74 |
| Wireless speaker | $99 | $199.00 | $100 | 1 | $99 |
| Jogger | $40 | $54.00 | $14 | 1 | $40 |
| Arsenal kit | $83 | $90.00 | $7 | 0 | $0 |
| Gym shoe | $56 | $69.95 | $14 | 1 | $56 |
| Laptop bag | $97 | $113.97 | $17 | 0 | $0 |
| Bluetooth headphone | $78 | $129.99 | $52 | 1 | $78 |
| Skateboard | $36 | $59.99 | $24 | 1 | $36 |
| Sweater | $25 | $27.82 | $3 | 0 | $0 |
| PS controller | $38 | $64.99 | $27 | 1 | $38 |
| GTA V game | $84 | $99.99 | $16 | 1 | $84 |
| Bike helmet | $29 | $33.96 | $5 | 0 | $0 |
| Hair dryer | $38 | $59.99 | $22 | 1 | $38 |
| Bed sheet | $31 | $39.78 | $9 | 1 | $31 |
| Shirt | $27 | $39.99 | $13 | 1 | $27 |
| Power bank | $39 | $46.99 | $8 | 1 | $39 |
| Sunglass | $34 | $89.99 | $56 | 1 | $34 |
| Swimsuit | $31 | $54.00 | $23 | 1 | $31 |
| Aquarium tank | $27 | $32.89 | $6 | 0 | $0 |
| Blazer | $85 | $99.99 | $15 | 0 | $0 |
| Kitchen mixer | $37 | $39.67 | $3 | 0 | $0 |
| Microwave bowl | $15 | $17.99 | $3 | 0 | $0 |
| Protein powder | $23 | $28.99 | $6 | 0 | $0 |
| **Total** | **$1,421** |  | **$697** | **20** | **$977** |

# Conclusion:

Knapsack problem is very helpful to formulate daily life problems to make favorable decisions. I took data of my 30 different wish list items from amazon with their respective prices and discount accrued. The objective of project was to find the list of items to buy from wish list such that the total money spent is within budget boundary and also total discount accumulated is maximum. I formulated the problem as 0/1 knapsack and applied method of dynamic programming to solve it. With my result, I found the list of items for maximum discount to be achieved by spending money within budget available with thousands of calculations in just a fraction of a second.

# References

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