Dynamic Programming

10.5281/zenodo.4037386

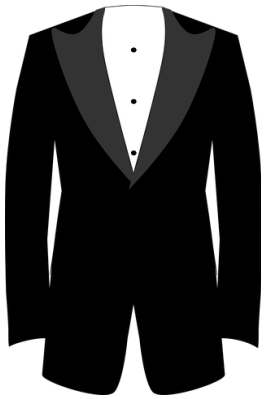
"You can never be overdressed or overeducated." — Oscar Wilde

Wedding Shop Problem

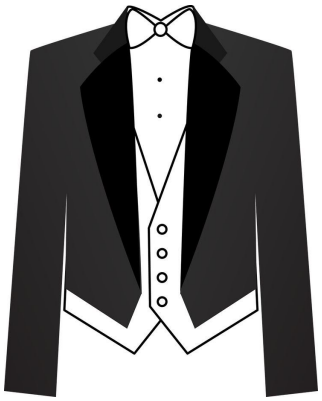
With your AI educational background, you are devoted to launching an **Wamazon** entrepreneurial e-commerce , a wedding shop! Your system will recommend the combination of the garments within customers' budget and maximize your sales price.

Tuxedo Catalogue

$60

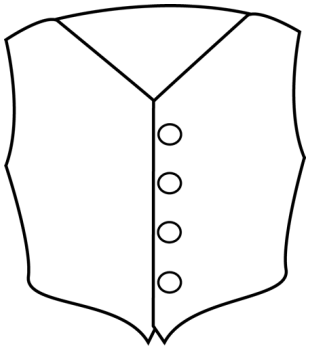


$100



$200

Vest Catalogue



$30

$50

$100

Pants Catalogue

If you dare to recommend this short to groom, well, I guess he will dare to wear it!

Greedy Algorithm to Serve Your First Customer

First customer browsing your website now!

Alright! As the store owner, you definitely want to maximize your profit by recommending customers the most expensive combination within their budget!

Tuxedo: 60, 100, 200
Vest: 30, 50, 100
Pants: 60, 80, 200

Pick the most expensive tuxedo ($200)!
Pick the most expensive vest ($100)!
With the remaining budget $60 ($360 - $200 - $100), you will just pick the pants that fit this budget!

Greedy! Greedy! Greedy! Greedy! Greedy! Greedy!

You're darn right. This is indeed an algorithm - Greedy Algorithm!

What if the prices are different now?

Tuxedo: 30, 100, 150
Vest: 30, 70, 150
Pants: 40, 50, 150

Pick the most expensive tuxedo ($150)!
Pick the most expensive vest ($150)!
With the remaining $60 ($360 - $150 - $150), you will just pick the pants ($50) that fit this budget!

Brute-Force Algorithm to Save the Falling Pants

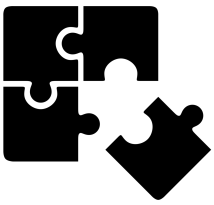
Will the Greedy Algorithm always work?

What if the customer only has $320 budget instead of $360?

Tuxedo: 30, 100, 150
Vest: 30, 70, 150
Pants: 40, 50, 150

Pick the most expensive tuxedo ($150)!
Pick the most expensive vest ($150)!
With the remaining $20 ($320 - $150 - $150), none of the pants fit this budget!

Ouch!



How to solve this problem?

(Credit to the image source: https://www.pngfuel.com/free-png/ntfvt/download)

Let us try each and every combination of the garments and see which combination has the highest price within budget $320!

Tuxedo	T1: $30	T2: $100	T3: 150
Vest	V1: $30	V2: $70	V3: $150
Pants	P1: $40	P2: $50	P3: $150

Compute the following combined prices and select the highest within $320 budget.

T1 Selected	T2 Selected	T3 Selected
V1 - P1 V1 - P2 V1 - P3 V2 - P1 V2 - P2 V2 - P3 V3 - P1 V3 - P2 V3 - P3	V1 - P1 V1 - P2 V1 - P3 V2 - P1 V2 - P2 V2 - P3 V3 - P1 V3 - P2 V3 - P3	V1 - P1 V1 - P2 V1 - P3 V2 - P1 V2 - P2 V2 - P3 V3 - P1 V3 - P2 V3 - P3

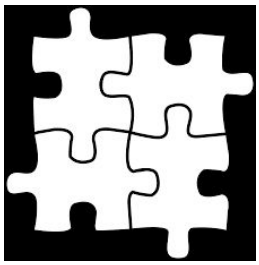
Among the total of 27 combinations, the price of T2 - V2 - P3 combination matches the budget $320.

Let us generalize this solution:

N kinds of tuxedo, N kinds of vest and N kinds of pants require N^3 combinations. If you have a total of M types of garments (including more types like shirt, socks, etc)? There will be N^M combinations.

This is called **Brute-Force Algorithm**. Brute-Force saves the falling pants!
But checking all the combinations is a lengthy process. It takes a lot of time and computing power!

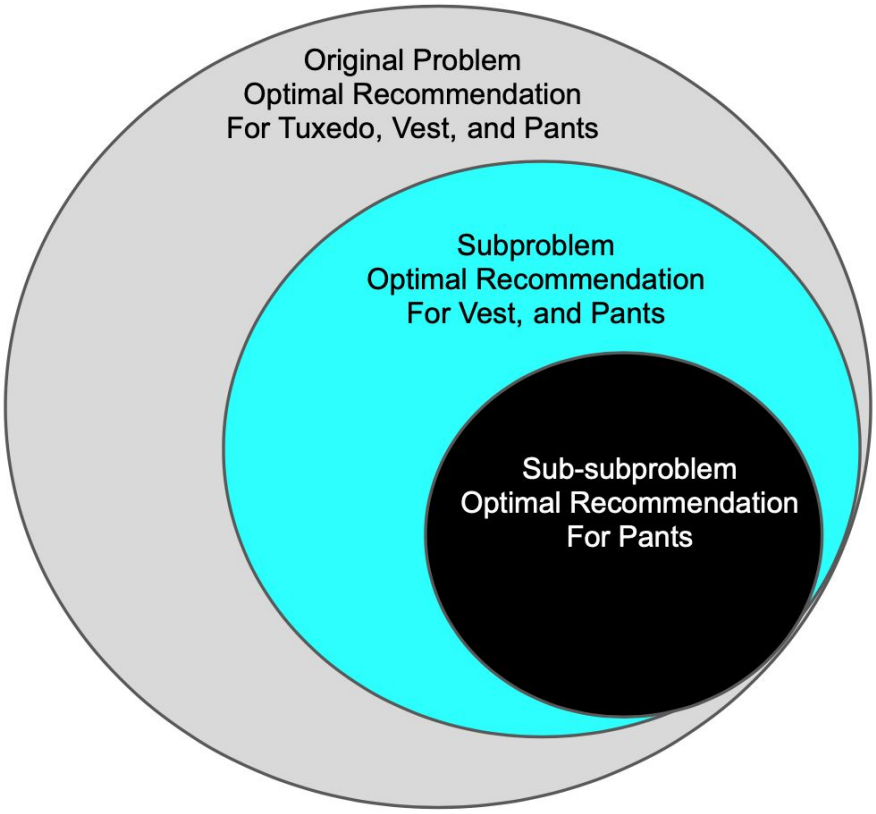
[add the story of the origin/history of brute-force algorithm]

 Better way to solve the problem

Dynamic Programming Algorithm to Remove Overlapping Steps

Solution to Subproblem Is Part of Solution to Overall Problem

 On further look at this wedding shop example, if we select Ti for Tuxedo, in order for our final selection to be optimal, our subproblem, which is the choice of Vest and Pants, must also be optimal for a reduced budget (total budget - price of Ti). And hence our sub-subproblem, which is the choice of Pants, must also be optimal for the further reduced budget (total budget - price of Ti - price of Vj).



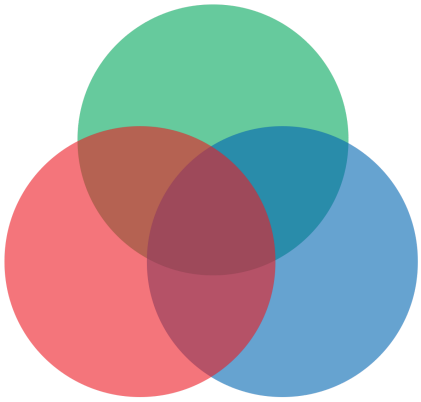
In other words, **the solution to the subproblem is part of the solution to the overall problem.**

Overlapping Subproblems

On much further look at the choices, there are some overlapping subproblems. For example, if T1-V3 selected or T3-V1 selected, they leave the same remaining budget $140 ($320 - $30 - $150) to make the optimal choice for P.

Tuxedo	T1: $30	T2: $100	T3: 150
Vest	V1: $30	V2: $70	V3: $150
Pants	P1: $40	P2: $50	P3: $150

With the increasing volume of garment variety and types, total of N^M combinations, **there will be many more overlapping subproblems**.



What if we memorize the optimal choice of V given the budget of $140? Then we do not have to re-compute it later.

In other words, **we can store the results of subproblems to avoid computing the same results again.**

Remember, one of the computer's strengths is the Big Brain, a.k.a. better memory than humans.

Genius! This is called **Dynamic Programming** algorithm!

Dynamic Programming (DP) Key Concepts

Dynamic Programming algorithm solves a complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.

Two Prerequisites

Two prerequisites for DP to be applicable

1. This problem has optimal subproblem

The solution for the subproblem is part of the solution of the original problem.

2. This problem has overlapping subproblems

This is the key characteristic of DP! The total computation of this problem is not as big as the N^M combinations in brute-force algorithm because many subproblems are overlapping

States

The distinct subproblems are called the states.

Transitions

in DP terminology

Top-Down Approach With Memo Table

Workflow

1. Initialize a DP ‘memo’ table with dummy values that are not used in the problem, e.g. ‘-1’. The DP table should have dimensions corresponding to the problem states.

2. At the start of the recursive function, check if this state has been computed before.

   (a) If it has, simply return the value from the DP memo table, O(1). (This the origin of the term ‘memoization’.)

   (b) If it has not been computed, perform the computation as per normal (only once) and then store the computed value in the DP memo table so that further calls to this sub-problem (state) return immediately.

If it has M distinct states, then it requires O(M) memory space. If computing one state (the complexity of the DP transition) requires O(k) steps, then the overall time complexity is O(kM).

Bottom-Up Approach With ...

This is actually the ‘true form’ of DP as DP was originally known as the ‘tabular method’ (computation technique involving a table). The basic steps to build bottom-up DP solution are as follows:

1. Determine the required set of parameters that uniquely describe the problem (the state). This step is similar to what we have discussed in the top-down DP earlier.
2. If there are N parameters required to represent the states, prepare an N dimensional DP table, with one entry per state. This is equivalent to the memo table in top-down DP. However, there are differences. In bottom-up DP, we only need to initialize some cells of the DP table with known initial values (the base cases). Whereas in top- down DP, we initialize the memo table completely with dummy values (usually -1) to indicate that we have not yet computed the values.
3. Now, with the base-case cells/states in the DP table already filled, determine the cells/states that can be filled next (the transitions). Repeat this process until the DP table is complete. For the bottom-up DP, this part is usually accomplished through iterations, using loops.

Reference

https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/#:~:text=Overlapping%20Subproblems%20Property%20in%20Dynamic%20Programming%20%7C%20DP%2D1,-Last%20Updated%3A%2003&text=Dynamic%20Programming%20is%20an%20algorithmic,computing%20the%20same%20results%20again.

Image source:

https://www.clipartmax.com/

DP to Optimize Fibonacci Problem