Computer Algorithms ISE 1

0/1 Knapsack Problem Visualization

Name: Huma Naaz Mohammad   Batch: T3   PRN: 23510053

Link to GitHub Repository - [Link](#) | Link to Hosted Website - [Website](#)

What is the Knapsack Problem?

Given n items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible. Where does 0/1 come in? The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

How the Algorithm Works (Dynamic Programming)

To solve this, we use a method called Dynamic Programming (DP) using a table where - Rows (i) represent the items we've looked at so far and columns (w) represent the available weight capacity and each cell (DP[i][w]) holds the best value we can get using the first i items with capacity w.

The Recurrence Relation

To fill any cell DP[i][w] we look at the current item (i) and check if we take it or leave it?

1. If the item is too heavy ($w_i > w$): We must leave it. The best value is simply the best value we got without item i.

$$DP[i][w]=DP[i-1][w]$$

2. If the item fits ($w_i > w$): We check two options and pick the better one.

$$DP[i][w]=\max(DP[i-1][w],\ v_i + DP[i-1][w-wi])$$

Visualization

I built a tool using HTML, CSS, and JavaScript to show this grid-filling process step-by-step.

Fig. 1: Input Fields with Presets, Best and Worst Case

Fig. 2: Mid Execution. Showing Comparison between Options and Traversal

Fig. 3: Completed Execution with final result.

Complexity (How Fast It Is)

In order to evaluate, we need to traverse the DP table once. Thus, the time complexity is **O(n*c)**.
Where,
n: The number of items + 1
c: The capacity of the knapsack + 1

The visualization clearly shows why the time is O(n*c) because you watch the program touch every single cell in the table.