- **Problem:** Suppose you are on a hiking trip and you need to fill your knapsack with items necessary for the trip. Your knapsack has a capacity of W and there are N items to choose from. Each item has two attributes, the value (which determines the importance of said item) and the weight of the item. The goal of the problem is to pack your knapsack with a combination of items that maximizes the total value.
- Formalized the problem is:

  Suppose there are n items.

  $$\begin{aligned} S &= \{item_1, item_2, item_3, ...\} \\ w_i &= \text{weight of } item_i \\ p_i &= \text{profit of } item_i \\ W &= \text{maximum weight of the knapsack} \\ N &\subseteq S \text{ s.t. } N \text{ is the optimal subset of N.} \end{aligned}$$

  Need to determine:

  $$\sum_{item_i \in N} p_i \text{ is maximized subject to } \sum_{item_i \in N} w_i \leq W$$

- **Brute Force:** consider all the subsets of n items, discard those that exceed W and determine the maximum of the remaining subsets. [2^n subsets of a set containing n items]
  - Exponential-time
- **Greedy Strategy:**
  - Steal items with the largest profit first (steal in non-increasing order)
    * Does not work well if most profitable item has a large weight in comparison to the profit
  - Steal lightest item first
    * Fails when light item has low profit in comparison to its weight.
  - Steal items with largest profit per unit weight first.
    * Can still fail, so not the solution for the 0-1 Knapsack
    * Only a solution for the **Fractional Knapsack Problem** which allows for fractions of an item.
- **Dynamic Programing Approach:**
  - Bottom-Up
    * The goal is to put values into a table where the rows represent the weight limit and the columns represent the items. This means that if we pick a cell (i, j) it would contain the optimal value for the first i items for a weight limit of j.

* $O(n * W)$, but can exceed the brute force method if $W = n!$ leaving us with order $O(n * n!)$

Example

Table 1: Example from CodesDope [2]

Item Number $item_i$	**Weight** w_i	**Value** p_i
1	3	8
2	2	3
3	4	9
4	1	6

$$W = 5$$

Table 2: Example from CodesDope [2]

$w \rightarrow$ $item_i \downarrow$	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0					
3	0					
4	0					

Recall: $w_1 = 3$ with a value of $v_1 = 8$. This means that for all $w < 3$ that we cannot take an item, so these cells have a 0. For any $w \geq 3$ we can put item 1 into our knapsack, so these cell values will be 8.

Table 3: Example from CodesDope [2]

$W \rightarrow$ $item_i \downarrow$	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0	3	8	8	11
3	0	0	3	8	8	11
4	0	6	6	9	14	15

Observe: $F(4, 1) = 6$ because $w_4 = 1$ and $v_4 = 6$, so we can carry that item with a $w = 1$. Also observe that $F(4, 5) = 15$ becuase we can take $item_3$ and $item_4$, such that $w_3 + w_4 = 5$ and $v_3 + v_4 = 9 + 6$.

- Recursive (Top-Down)
  - Consider $F[i][w]$ and $F[i-1][W]$ and $F[i-1][W-w_i]$ where n is the number of items and i is the item we are talking about/the row we are in. Where we start from n and end when either $n = 1$ or $w \leq 0$. Leaving us with the equation.

$$F[i][W] = \begin{cases} maximum(P[i-1][W],\ v_i + P[i-1][w-w_i]) & \text{if } w_i \leq W \\ P[n-1][W] & \text{if } w_i > W \end{cases}$$

  - $O(minimize((n * W), (2^n)))$ [The order will never exceed $O(2^n)$]

Example

Table 4: Example from *Foundations of Algorithms* [4]

Item Number $item_i$	**Weight** w_i	**Value** p_i
1	5	50
2	10	60
3	20	140

$$W = 30$$

- **Determine the entries need:**
  - Row 3 we need $F[3][30]$
  - Row 2 we compute $F[3][30]$, thus we need $F[2][30]$ and $F[3-1][30-w_3] = F[2][10]$
  - Row 2 we compute $F[2][30]$, so we need $F[1][30]$ and $F[1][20]$
  - Row 1 we compute $F[1, 10]$ and $F[1][0]$
  - Stop because $n = 1$ as well as $w \leq 0$

Step 1:

$$\begin{aligned} F[1][w] &= \begin{cases} maximum(P[0][w],\ 50 + P[0][w-5]) & \text{if } w_1 = 5 \leq w \\ P[0][5] & \text{if } w_1 = 5 > w \end{cases} \\ &= \begin{cases} 50 & \text{if } 5 \leq w \\ 0 & \text{if } 5 > w \end{cases} \end{aligned}$$

Step 2:

$$\begin{aligned} F[2][10] &= \begin{cases} maximum(P[1][10],\ 60 + P[1][0]) & \text{if } w_2 = 10 \leq 10 \\ P[1][10] & \text{if } w_2 = 10 > 10 \end{cases} \\ &= 60 \end{aligned}$$

Step 3:

$$F[2][30] = \begin{cases} maximum(P[1][30],\ 60 + P[1][20]) & \text{if } w_2 = 10 \leq 30 \\ P[1][30] & \text{if } w_2 = 10 > 30 \end{cases}$$
$$= 60 + 50 = 110$$

Step 4:

$$F[3][30] = \begin{cases} maximum(P[2][30],\ 140 + P[2][10]) & \text{if } w_3 = 20 \leq 30 \\ P[2][30] & \text{if } w_2 = 20 > 30 \end{cases}$$
$$= 140 + 60 = 200$$

References

[1] The Knapsack Problem — Data Structures and Algorithms. https://stevenschmatz.gitbooks.io/data-structures-and-algorithms/content/281/lecture_20.html.

[2] Knapsack Programming Using Dynamic Programming and its Analysis. https://www.codesdope.com/course/algorithms-knapsack-problem/.

[3] Matthew Martin. 0/1 Knapsack Problem Fix using Dynamic Programming Example. https://www.guru99.com/knapsack-problem-dynamic-programming.html, March 2020.

[4] Richard E. Neopolitan. *Foundations of Algorithms*. Jones & Bartlett Learning, the fifth edition.