The 0-1 Knapsack Problem is a problem that requires one to fill a knapsack that has a specific weight capacity with items that have two conditions, value and weight. The goal is to fill the knapsack in a way that optimizes the value of the bag, but does no exceed the weight capacity. While this seems like a very specific use case, the same techniques used in the 0-1 Knapsack Problem can be used for a variety of other resource allocation problems. To generalize the problem more consider the knapsack as the available amount of resource and the items are different activities that the resource can be allocated to. An example of this is money and marketing, you can allocate the money (resource) to things like billboards, TV commercials, social media, etc. Each platform to advertise on will yield different amounts of traffic to the company's website and cost different amounts to advertise on. The goal would be to maximize the traffic to the website by selecting the platforms that yield the most traffic while staying within the budget.

In this paper we will focus on the 0-1 Knapsack Problem itself, but it is important to keep in mind how the techniques used to solve this problem will prove useful elsewhere. There are many different variations of the 0-1 Knapsack Problem from hiking knapsacks to a thief's knapsack. In this paper we will focus on the following sample problem:

> Suppose that you are going 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 k items to choose from. Each item has two attributes, the value (which determines the importance of the 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.

Let $S = \{item_1, item_2, item_3, ..., item_k\}, w_i =$ weight of $item_i, v_i =$ value of $item_i,$ and $W =$ weight capacity of knapsack.

There are multiple ways to approach this problem. The first way is using brute force. This requires us to consider all the subsets of S which we will call N, thus $N \subseteq S$. We will need to calculate every single N while discarding those that exceed W, the weight limit of the knapsack. From the remaining subsets we will find the subset with the largest value. This method is effective and will yield the most optimized answer, however the big issue with this approach is the order. Using the brute force approach we will need to calculate 2^n subsets, so $O(2^n)$ which is of exponential order, so its pretty bad and therefore undesirable.

However, there are other ways to approach the 0-1 Knapsack problem such as various greedy approaches and dynamic programming approaches. To analyze these methods having a concrete example (see *Table 1*) will allow for better comprehension of why these methods work or do not work.

Table 1: Example from CodesDope [2]

Item Number $item_i$	**Weight** w_i	**Value** p_i
1	12	100
2	32	200
3	30	50
4	5	60
5	34	150

$$W = 50$$

Let us look at the first greedy strategy which is to pick the highest value first and then continue in decreasing order of value. In our example this would mean our subset would be $n_1 = \{item_2, item_1\}$ in that order for a total value of 300 and weight of 44. Notice how we can't go to the next highest value $item_5$ because it will exceed the weight so we have 6 weight units of wasted space. Similarly another greedy strategy is to pick the lowest weight item first and then go in ascending order of weight. In our example this would mean our subsets would be $n_2 = \{item_4, item_3, item_1\}$, which gets us a total value of 210 and a total weight of 47. Here we got closer to the maximum weight, W, but the value is a lot lower than n_2. However, observe that neither n_1 or n_2 is the optimal solution. The optimal solution would be $n = \{item_1, item_2, item_4\}$ for a total value of 360 and a total weight of 47.

The third greedy method is to first pick the items with the largest value per weight. So $item_1 = 8.333, item_2 = 3.125, item_3 = 1.667, item_4 = 12, item_5 = 4.412$. In this case our subset would be $n_3 = \{item_4, item_2, item_3\} = N$. Notice how we skipped $item_5$ because it put us over W, but $item_3$ kept us within W. In this case we were able to obtain the optimal solution, however it can still fail for the 0-1 Knapsack problem. Consider a new example:

Table 2: Example from CodesDope [2]

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

$$W = 30$$

Using the same method we get a value of 190 when the optimal solution gets one a value of 200.

This is where dynamic programming comes in. There are different dynamic programming techniques, so let us first look at the bottom-up approach. For this approach we need to create a table. Let us use an example with smaller numbers so that we can anaylze this method better.

Table 3: 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

$$W = 5$$

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. Setting up the table will look like this:

Table 4: Example from CodesDope [2]

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

Observe that for a maximum weight of 0 that we cannot take any items, so the whole column $j = 0$ would have a total value of 0. Similarly note how if we take 0 items that the total value would be 0, so the whole row $i = 0$ is also 0. Resulting in the following table:

Table 5: Example from CodesDope [2]

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

Now lets fill out the table for $i = 1$ as represented by *Table 3*.

Table 6: 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 from our example that the weight of item 1 is $w_1 = 3$ 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. More formally this means that $F(1,1) = F(1,2) = 0$ and $F(1,3) = F(1,4) = F(1,5) = 8$. We can similarly fill out the rest of the table as shown in *Table 7*. In *Table 7* we can see that $F(4,1) = 6$ 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 $v_3 + v_4 = 9 + 6$.

Table 7: 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

Therefore the maximum value we can obtain for a knapsack with a maximum weight of 5 is 15.

Notice how with this method we need to calculate $(n * W)$ values in order to find the optimal solution, so from our example we needed to make 16 calculations as the first row and column will always have the value of 0. Thus $O(n * W)$which is a much better outcome than having $O(2^n)$. However, there are cases where the order can exceed $O(2^n)$, consider a $W = n!$ where n is the number of items, this means that the order is $O(n * n!)$ which is worse than $O(2^n)$. In order to better optimize this solution we can instead simply 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.

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

To show this let us take an example with a higher W.

Table 8: 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

$$W = 30$$

First we need determine the entries we need. So for row 3 we need $F[3][30]$. Notice how we now need to know what $F[3][30]$ is which means that to determine entries needed in row 2 we compute $F[3][30]$, thus we need $F[3-1][30]$ and $F[3-1][30-w_3] = F[2][10]$. Now we need to compute $F[2][30]$, so we need $F[1][30]$ and $F[1][20]$ and then to compute $F[2][10]$ we need $F[1,10]$ and $F[1][0]$. We can stop here because we have reached $n = 1$ as well as $w \leq 0$, however only one of these conditions needs to be satisfied in order to stop.

Now that we know the entries we need we can compute the values.

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:

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

Step 4:

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

Using this method of dynamic programming is recursive with $O(minmum(2^n, n * W))$. In this case we do not need to compute the entire array instead we compute the values that are needed and store them.

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.