**Unraveling the Knapsack Algorithm: A Guiding Force for Efficient Decision-Making and Resource Optimization**

Imagine embarking on a quest to pack your bag for a once-in-a-lifetime adventure around the world. You have a limited capacity (let’s just say, space and weight in this case), and you want to maximize the value of the items you take with you. Just like a strategic game of Jenga, where each move determines the stability of your tower, the Knapsack Algorithm emerges as a guiding force, empowering you to optimize your packing and unlock the secrets of efficient decision-making. Join me on a journey as we delve into the fascinating world of the Knapsack Algorithm and uncover the mathematical prowess behind its ingenuity.

So, what exactly is the Knapsack Algorithm or Problem? The knapsack algorithm as stated on Wikipedia is the following problem in combinatorial optimization:

*Given a set of items, each with a weight and a value, determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.*

The Knapsack Problem traces its roots back to the 1930s when Tobias Dantzig, a Polish mathematician, introduced a similar problem known as the "luggage problem." It involved maximizing the value of selected items within a weight constraint. The problem's origins lie in optimizing the cutting of steel plates to reduce waste during manufacturing. In 1951, the Knapsack Problem was formalized by mathematician George Dantzig and economist Harold Kuhn. Since then, it has been extensively studied by mathematicians, computer scientists, and operations researchers. The problem draws its name from the concept of a knapsack—a bag or backpack with a limited capacity - for example, a backpack.

Now, back to how the algorithm works – let us consider “n” items with certain weights and profits, and the maximum weight that the knapsack can hold is “W”. We need to select the items such that all of them together do not exceed the maximum weight and at the same time have the maximum profit or value. There are two ways to deal with this – 0/1 Knapsack and Fractional Knapsack.

In the 0/1 Knapsack problem, when picking an item, you either pick that complete item or you don’t i.e., 0 is when you don’t pick the object, and 1 is when you pick the object – so it’s either 1 or 0. Fractional knapsack problem is when you can pick a part of the object. That is, a fraction of the object, you don’t necessarily have to pick the complete item. For example, you have two items with weights 2kg and 3kg. In 0/1 knapsack, you either pick the complete 2kgs or no; similarly, you either pick the complete 3kg or you don’t. Whereas in the fractional knapsack, for the 2kg object, I could take either none of it, 1kg of it, or the entire object (2kgs); and similarly, for the 3kg item, I can take either none of it, 1kg, 2kg or the entire 3kgs of it into the knapsack.

Generally, the 0/1 knapsack algorithm is solved using dynamic programming and the fractional knapsack using the greedy approach. In this paper, we will plunge into the depths of the 0/1 knapsack problem, providing a comprehensive exploration of its inner workings, mathematical analysis, and algorithmic implementation using C code and some examples.

Now, to understand how the actual 0/1 Knapsack algorithm works, let us consider an example.   
Let there be 4 items of weights 3, 4, 6, 5 and values (profits) 2, 3, 1, 4 respectively. The maximum weight the knapsack here can hold is 8.

Weights = {3, 4, 6, 5}. Number of items = 4

Values = {2, 3, 1, 4} Maximum weight of the knapsack = 8

Given this, we need to choose the items so that the total weight of the items doesn’t exceed the maximum weight of the knapsack (8) and the value/profit is the maximum.

At the end, we write, for example xi = {1, 0, 0, 1} to show that we picked the items with weight 3 and 5 (i.e., values 2 and 4 respectively} and did not pick the items with weights 4 and 6 (i.e., values 3 and 1 respectively). As such, one way to figure out which items we are picking, what the total weight and the value of those items is going to be is writing down every possible combination of picking up the items and how much each possibility would value, and from this list, we then find the combination with the highest value or profit, which henceforth, would be our solution.

For example,   
 x1 = {1, 0, 0, 0}; weight = {3}; value = {2};

xk = {1, 1, 1, 0}; weight = {13}; value = {6};

.

.

.

In this way, we list every possibility, and whichever combination has the highest value and doesn’t exceed the maximum weight of the knapsack, that would be our solution.

Though this might seem easy, it would be extremely tiring and overwhelming when dealing with a larger set of items. Hence, we have a more efficient approach – using dynamic programming (aka building the “knapsack table”). This table will give us the maximum profit we can get from picking these items, without exceeding the maximum weight of the knapsack.

We start off by drawing a matrix (table),

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | pi | wi | | 2 | 3 | | 3 | 4 | | 4 | 5 | | 1 | 6 | | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 |  |  |  |  |  |  |  |  |  | | 1 |  |  |  |  |  |  |  |  |  | | 2 |  |  |  |  |  |  |  |  |  | | 3 |  |  |  |  |  |  |  |  |  | | 4 |  |  |  |  |  |  |  |  |  | |

On the top of the table, we take all the weights from 0 to 8, because the maximum value of the knapsack is 8. On the left side of the table, we take the number of items we can put in the knapsack; here 0 to 4, because the total number of items is 4. And onto the side, corresponding to the number of items, we here take the weights of the actual items in increasing order and their corresponding profits/values. Generally, you can take the items in any order. Both ascending and descending order will yield the same correct result.

As we cannot put any items in the knapsack when the maximum weight of the knapsack is 0, the first column will have zero profit/value. And when the number of items we can put in the knapsack is zero, we don’t consider the weights as well. The profits in the first row will also be zero, because we are allowed to put no items (0) in the knapsack. So we fill the first row and column with zeroes.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | pi | wi | | 2 | 3 | | 3 | 4 | | 4 | 5 | | 1 | 6 | | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 |  |  |  |  |  |  |  |  | | 2 | 0 |  |  |  |  |  |  |  |  | | 3 | 0 |  |  |  |  |  |  |  |  | | 4 | 0 |  |  |  |  |  |  |  |  | |

Now, to fill the next row, we have only 1 item (i), and the weight of that one item is 3 (wi). But for the first cell empty cell in that row, the maximum weight of the knapsack is 1. We cannot fill a bag having maximum weight 1, with an item weighing 3. So, there is no profit/value as there is no item there. It will be the same for the next cell in that row too, because the max weight of the knapsack is 2 and the weight of the item is.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | pi | wi | | 2 | 3 | | 3 | 4 | | 4 | 5 | | 1 | 6 | | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 0 |  |  |  |  |  |  | | 2 | 0 |  |  |  |  |  |  |  |  | | 3 | 0 |  |  |  |  |  |  |  |  | | 4 | 0 |  |  |  |  |  |  |  |  | |

Now, for the next cell, the weight of the item is 3 and the max weight of the knapsack is 3, so we can put the entire item in the bag, and the profit/value of this item is 2, so we fill the cell with the value 2. The next cell shows the maximum weight of the knapsack as 4, so we can take the entire item of weight 3 in the bag. And the value of the item weighing 3 is 2, so we fill that cell with 2. It will work the same for the rest of the cells in the row as well. We need to keep in mind that for this row, we are only considering 1 item.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | pi | wi | | 2 | 3 | | 3 | 4 | | 4 | 5 | | 1 | 6 | | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | | 2 | 0 |  |  |  |  |  |  |  |  | | 3 | 0 |  |  |  |  |  |  |  |  | | 4 | 0 |  |  |  |  |  |  |  |  | |

Coming to the next row, we consider 2 items i.e., the items with weights 3 & 4 with values 2 & 3 respectively. In the first empty cell here, the max weight of the knapsack is 1, so you can neither pick the item weighing 3 nor the item weighing 4. So that’ll be a 0 value; the same applies to the next cell as well. Coming to the next cell, the max weight of the knapsack is 3, so we can pick the item with weight 3 here, and its value is 2. So we fill that cell with the value 2.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | pi | wi | | 2 | 3 | | 3 | 4 | | 4 | 5 | | 1 | 6 | | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | | 2 | 0 | 0 | 0 | 2 |  |  |  |  |  | | 3 | 0 |  |  |  |  |  |  |  |  | | 4 | 0 |  |  |  |  |  |  |  |  | |

For the cell with the column that has weight 4, we can either take the item with weight 3 or weight 4, whichever has the highest profit. Now, instead of directly looking at the values and deciding which one to pick, we will start building up a mathematical formula so that it will be more efficient and easier when dealing with larger datasets. Now, to calculate the maximum value, let’s take the profit of the new item (weight 4) here, which is 3. Then, we take weight of the item we are currently considering (4) and subtract it from the maximum weight of the knapsack at that column (4 here) and to see how much weight is remaining (4-4 = 0). Now in the row above the row we are trying to fill, we go to the 0th column, take that value (0 here) and we add that number to the profit of the item we considered (3).

i.e., 4-4 = 0, go to the above row – 0th column and take that value, which is again 0, and add that to 3 i.e., 3+ 0 = 3.

The other value would be the value/profit value that is in the cell right above the cell we are trying to fill (2 here). And now, we consider the maximum of the values “3” and “2”; that is, max(3, 2) and we get 3 as the answer (maximum profit at this cell). It will be the exact same for process for the next 2 cells.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | pi | wi | | 2 | 3 | | 3 | 4 | | 4 | 5 | | 1 | 6 | | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | | 2 | 0 | 0 | 0 | 2 | 3 | 3 | 3 |  |  | | 3 | 0 |  |  |  |  |  |  |  |  | | 4 | 0 |  |  |  |  |  |  |  |  | |

Now, when the max weight of the knapsack is 7, we can take either the element with weight 3 or 4 or both, whichever has the highest value. So now we take the maximum of the 2 values. i.e., 3 (profit of the new item) + [(7-4) = 3, now go to the above row’s 3rd column, which has the value 2]. It will be (3+2 = 5). The other number will be the value just above this cell (2).

So, max (5, 2) = 5. We fill the cell with the value 5.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | pi | wi | | 2 | 3 | | 3 | 4 | | 4 | 5 | | 1 | 6 | | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | | 2 | 0 | 0 | 0 | 2 | 3 | 3 | 3 | 5 |  | | 3 | 0 |  |  |  |  |  |  |  |  | | 4 | 0 |  |  |  |  |  |  |  |  | |

We fill the rest of the values in the table in the same way.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | pi | wi | | 2 | 3 | | 3 | 4 | | 4 | 5 | | 1 | 6 | | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | | 2 | 0 | 0 | 0 | 2 | 3 | 3 | 3 | 5 | 5 | | 3 | 0 | 0 | 0 | 2 | 3 | 4 | …. |  |  | | 4 | 0 |  |  |  |  |  |  |  |  | |

Now, let us consider this cell and try writing down the actual formula here. From the above calculations, we know this is going to be max (4 + 0, 3) = 4. How we get that 0 is the column weight (6) – weight of the new item (5), we get 1; so, we go to the above row’s 1st column and take its value (which is 0). Putting this into a formula, if we consider the table to be “k”, to fill

1. k [3, 6] = max (the value above of the current cell, value of the new item + above row’s ath column’s value)

where, a = current column’s weight - weight of new item

1. k [3, 6] = max (k [3-1, 6], k [3-1, current column’s weight (6) – weight of the new item (5)] + value of current element (4))
2. k [3, 6] = max (k [ 2, 6], k [2, 1] + 4)
3. k [3, 6] = max (3, 0 + 4)
4. k [3, 6] = max (3, 4) = 4

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | pi | wi | | 2 | 3 | | 3 | 4 | | 4 | 5 | | 1 | 6 | | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | | 2 | 0 | 0 | 0 | 2 | 3 | 3 | 3 | 5 | 5 | | 3 | 0 | 0 | 0 | 2 | 3 | 4 | 4 |  |  | | 4 | 0 |  |  |  |  |  |  |  |  | |

**From this, we get the formula to be,**

**k [i, w] = max (k [i-1, w], k [i – 1, w - wi] + pi)**

Now, we fill the rest of the table,

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | pi | wi | | 2 | 3 | | 3 | 4 | | 4 | 5 | | 1 | 6 | | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | | 2 | 0 | 0 | 0 | 2 | 3 | 3 | 3 | 5 | 5 | | 3 | 0 | 0 | 0 | 2 | 3 | 4 | 4 | 5 | 6 | | 4 | 0 | 0 | 0 | 2 | 3 | 4 | 4 | 5 |  | |

Let’s just calculate the final value using the formula,

k [4, 8] = max (k [4-1, 8], k [4 – 1, 8 - 6] + 1)

k [4, 8] = max (k [3, 8], k [3, 2] + 1)

k [4, 8] = max (6, 0 + 1) = max (6, 1) = 6

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  |  | | pi | wi | | 2 | 3 | | 3 | 4 | | 4 | 5 | | 1 | 6 | | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 1 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | | 2 | 0 | 0 | 0 | 2 | 3 | 3 | 3 | 5 | 5 | | 3 | 0 | 0 | 0 | 2 | 3 | 4 | 4 | 5 | 6 | | 4 | 0 | 0 | 0 | 2 | 3 | 4 | 4 | 5 | 6 | |

Therefore, the maximum profit we can get without exceeding the maximum weight of the knapsack is 6. But, to determine which values we had to consider getting this final answer, we take a pointer at the final answer i.e., at 6. Now, we check the value above 6, if it is the same value, then we don’t take the item corresponding to where the pointer is. So, since the value above 6 is also 6, we don’t take the item (with weight 6 and value 1). Then, we move the pointer above (at 6), the value above this is 5, so we take the item (with weight 5 and value 4). Here, now, the maximum profit is 6 and the value of the item is 4 so (6 – 4 = 2). So, in the above row, we check if we have a “2”, we have it at k [2, 3], we move the pointer there. Now, the value above where the pointer is at is also 2, so we don’t take the item corresponding to the pointer (i.e., the item with weight 4 and value 3). Then, we move the pointer above (at k [1, 3] = 2). The value above this is 0, so we take the item corresponding to the pointer (i.e., the item with weight 3 and value 2).

This will be represented as,

xi = {1, 0, 0, 1}; weight = {8}; value = {6};

To summarize, the knapsack algorithm uses a dynamic programming approach to solve the knapsack problem. It builds a table (K) to store the intermediate values, where each cell K[i][w] represents the maximum value that can be achieved using the first ‘i’ items and a maximum weight of ‘w’.

**Code explanation**:

The program first starts at main.

int main(int argc, char \*argv[])c

The main function is the entry point of the program and has two parameters: argc (argument count) and argv (argument vector). These parameters allow the program to receive command-line arguments.

if (argc < 2)

{

printf("\nFile must be provided on command line...exiting.\n");

return 1;

}

This condition checks if the number of command-line arguments is less than 2 (i.e., the program name and the file name). If so, it means the file name is missing, and an error message is printed. The program exits with a return value of 1 to indicate an error.

FILE \*file = fopen(argv[1], "r");

if (file == NULL)

{

printf("Failed to open the file.\n");

return 1;

}

The fopen function is used to open the file specified in the command-line argument (argv[1]) in read mode ("r"). The file pointer file is assigned the reference to the opened file. If the file cannot be opened (due to an error or it does not exist), a failure message is printed, and the program exits with a return value of 1.

int max\_weight, num\_of\_items;

fscanf(file, "%d", &max\_weight);

fscanf(file, "%d", &num\_of\_items);

Two variables, max\_weight and num\_of\_items, are declared to store the maximum weight of the knapsack and the number of items, respectively.

The fscanf function is used to read these values from the file and store them in the respective variables.

int \*wt = malloc(num\_of\_items \* sizeof(int));

int \*val = malloc(num\_of\_items \* sizeof(int));

Two dynamically allocated arrays, wt and val, are declared using the malloc function. The size of each array is num\_of\_items \* sizeof(int), which allocates memory for storing num\_of\_items integers. These arrays will be used to store the weights and values of the items, respectively.

for (int i = 0; i < num\_of\_items; i++)

{

fscanf(file, "%d", &wt[i]);

printf("%d ", wt[i]);

}

This loop reads the weights of the items from the file using fscanf and stores them in the wt array. Additionally, it prints each weight value to the screen for display purposes.

for (int i = 0; i < num\_of\_items; i++)

{

fscanf(file, "%d", &val[i]);

printf("%d ", val[i]);

}

printf("\n");

Similar to the previous loop, this loop reads the values of the items from the file using fscanf and stores them in the val array. It also prints each value to the screen for display purposes.

fclose(file);

After reading the weights and values from the file, the file is closed using the fclose function to release the associated resources.

int maxValue = knapSack(max\_weight, wt, val, num\_of\_items);

The knapSack function is called with the provided arguments (max\_weight, wt, val, and num\_of\_items) to compute the maximum value that can be obtained from the knapsack problem. Now, the program goes to the knapsack function.

int knapSack(int max\_weight, int wt[], int val[], int num\_of\_items)

The knapSack function is defined with four parameters: max\_weight (maximum weight the knapsack can hold), wt[] (array of item weights), val[] (array of item values), and num\_of\_items (total number of items).

int i, w;

int \*\*K = malloc((num\_of\_items + 1) \* sizeof(int \*));

for (i = 0; i <= num\_of\_items; i++)

{

K[i] = malloc((max\_weight + 1) \* sizeof(int));

}

Two variables, i and w, are declared to keep track of the current item and weight being considered, respectively. A 2D array K is dynamically allocated using malloc to store the intermediate values of the knapsack problem. It has (num\_of\_items + 1) rows and (max\_weight + 1) columns to accommodate the base case and the items being considered. The nested loop allocates memory for each row of the array.

for (i = 0; i <= num\_of\_items; i++)

{

for (w = 0; w <= max\_weight; w++)

{

if (i == 0 || w == 0)

{

K[i][w] = 0;

}

else if (wt[i - 1] <= w)

{

K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);

}

else

{

K[i][w] = K[i - 1][w];

}

}

}

The nested loops iterate over each item (i) and each possible weight (w) in a bottom-up manner to fill the knapsack table K.

The first if condition checks if it is the base case (either i == 0 or w == 0). If true, it means there are no items or the weight is 0, so the value in K[i][w] is set to 0.

The else if condition checks if the weight of the current item is less than or equal to the current weight being considered (wt[i - 1] <= w). If true, it means the current item can be included in the knapsack. In this case, the maximum value is computed by considering two options:

* Including the current item: val[i - 1] + K[i - 1][w - wt[i - 1]], which adds the value of the current item to the value obtained by considering the remaining weight.
* Excluding the current item: K[i - 1][w], which considers the maximum value obtained so far without including the current item.

The maximum of these two options is stored in K[i][w].

The else condition is executed when the weight of the current item is greater than the current weight being considered, so it cannot be included. In this case, the value is simply copied from the previous row: K[i - 1][w].

For this, the max function is called (inside the nested for-loop),

int max(int a, int b)

{

return (a > b) ? a : b;

}

The max function is a simple function that takes two integers, a and b, as parameters and returns the maximum value between them. This line uses the ternary operator (?:) to perform a comparison between a and b. If a is greater than b, the value of a is returned. Otherwise, the value of b is returned.

After the nested for loop,

printTable(K, num\_of\_items, max\_weight);

The printTable function is called with the array K, number of items, and the maximum weight of the knapsack.

void printTable(int \*\*table, int num\_of\_items, int max\_weight)

The printTable function is defined with three parameters: table (the 2D knapsack table), num\_of\_items (the number of items), and max\_weight (the maximum weight of the knapsack).

printf("\nKnapsack Table:\n");

for (int i = 0; i <= num\_of\_items; i++)

{

for (int w = 0; w <= max\_weight; w++)

{

printf("%-2d ", table[i][w]);

}

printf("\n");

}

printf("\n");

This code is responsible for printing the knapsack table, which displays the intermediate values computed during the dynamic programming approach. The outer loop iterates over each row of the table, from 0 to num\_of\_items, including the base case row. The inner loop iterates over each column of the table, from 0 to max\_weight, including the base case column. The value in each cell of the table is printed using the printf function, formatted as a two-digit number with a space separator. After printing all the cells in a row, a newline character is printed to move to the next line. Now, that the printing of the table is done, the program goes back to the knapSack function, continuing from the following line.

int result = K[num\_of\_items][max\_weight];

The maximum value that can be obtained is stored in result, which corresponds to the value in the bottom-right cell of the knapsack table.

for (i = 0; i <= num\_of\_items; i++)

{

free(K[i]);

}

free(K);

After the computation is done, the memory allocated for the knapsack table K is freed using the free function to prevent memory leaks. The nested loops iterate over each row of the table, and the memory for each row is freed. Finally, the memory for the array of row pointers is freed.

return result;

The function returns the maximum value that can be obtained from the knapsack problem to the caller. This returned value is stored in maxValue variable in the main function. Now, back to the main function,

printf("The maximum value that can be obtained is: %d\n\n", maxValue);

The maxValue variable, which contains the maximum value computed by the knapSack function, is printed to the screen.

free(wt);

free(val);

The memory allocated for the wt and val arrays using malloc is freed using the free function. This step is important to release the dynamically allocated memory and prevent memory leaks.

return 0;

The program exits with a return value of 0, indicating successful execution.

**Mathematical Analysis:**

To analyze the time complexity of the Knapsack algorithm, we need to consider the number of iterations in the nested loops used to build the knapsack table.

* The outer loop iterates over i, representing the items, from 0 to num\_of\_items (inclusive). It considers each item one by one, incrementing i in each iteration.
* The inner loop iterates over w, representing the weights, from 0 to max\_weight (inclusive). It considers each weight one by one, incrementing w in each iteration.
* Therefore, the total number of iterations for building the knapsack table is equal to (num\_of\_items + 1) \* (max\_weight + 1).
* Since the time complexity of the algorithm is typically expressed in terms of the input size, we can say that the time complexity of the knapsack algorithm is O(num\_of\_items \* max\_weight) i.e., O(n\*w).

Moving on to the space complexity, it primarily depends on the space required to store the knapsack table (K), as well as the space for the input arrays of weights (wt[]) and values (val[]).

* The knapsack table K is a 2D array with (num\_of\_items + 1) rows and (max\_weight + 1) columns. Hence, the space required to store the table is proportional to the number of cells, which is (num\_of\_items + 1) \* (max\_weight + 1).
* In addition to the table, the algorithm uses arrays wt[] and val[] to store the weights and values of the items. The space required for these arrays is proportional to the number of items, which is num\_of\_items.
* Therefore, the overall space complexity of the knapsack algorithm is O(num\_of\_items \* max\_weight), considering both the knapsack table and the input arrays.

Ultimately, the Knapsack Algorithm serves as a great ally in our goal of maximising value, whether it is during our journeys, manufacturing processes, or even data and resource management. Its adaptability and proven efficiency make it a vital instrument for attaining success and optimising outcomes in a wide range of real-world circumstances. As we embark on exciting and ever-changing trips filled with memorable experiences, the lessons learnt from the Knapsack Algorithm allow us to make educated decisions, optimise our choices, and embark on rewarding journeys filled with unforgettable experiences.