

The dynamic programming approach can be constructed using a table for recording the best optimal values for each subproblem of varying number of items and weights.

Knapsack Dynamic Programming Pseudocode  
-------------------------------------------------------------  
knapSack(N, C, itemWeight[], itemValue[])  
 // Creating table for recording the stages  
 table = Array[Array[Int]](N + 1, C + 1) // Dimension of the table is (N + 1) x (C + 1)  
 for (k = 0; k <= N; k++)  
 for (w = 0; w <= C; w++)  
 if (k == 0 || w == 0)  
 table[k][w] = 0  
 else if (itemWeight[k - 1] <= w)  
 table[k][w] = max(table[k - 1][w], table[k - 1][w - itemWeight[k - 1]] + itemValue[k - 1])  
 else  
 table[k][w] = table[k - 1][w]  
return table[N][C];

Consider a problem where the knapsack capacity is 7 and there are 5 items, with   
itemValue = [2,2,4,5,3] and itemWeight = [3, 1, 3, 4, 2]  
First of all, we fill the zeroth row and column of the table with zeros.   
For zeroth row, there are no items at all so the optimal value is 0  
For zero column, the total weight is zero and no items can be added so the optimal value is 0  
Application, table

Description automatically generated  
For the first row, the first item is considered. It has weight of 3 and value of 2. For the first few weights, the item cannot be added. As soon as the weight reaches 3, the item can be added to the knapsack and the optimal value increases by 2. For the rest weights, it is the same because there is only one item to choose.   
Table

Description automatically generated  
For the second row, table[2][1] and table[2][2] = 2 because second item can be added. However, for table[2][3] = 2, it can be either first item or second item because they have similar value.   
Finally, for table[2][4] = max(table[1][4], table[1][4-2] + itemWeight[2]) = max(2, 4) = 4   
Table

Description automatically generated  
The rest of the table can be filled as follows  
Table

Description automatically generated

We can examine this code part:

for (k = 0; k <= C; k++)  
 for (w = 0; w <= W; w++)  
 if (k == 0 || w == 0)  
 table[k][w] = 0  
 else if (itemWeight[k - 1] <= w)  
 table[k][w] = max(table[k - 1][w], table[k - 1][w - itemWeight[k – 1]] + itemValue[k - 1])  
 else  
 table[k][w] = table[k - 1][w]

The operation table[k][w] = 0 takes constant time   
The operation table[k][w] = table[k - 1][w] takes constant time  for performing subtraction k - 1   
The operation table[k][w] = max(table[k - 1][w], table[k - 1][w - itemWeight[k - 1]   
+ itemValue[k - 1]]) takes constant time  for performing 6 arithmetics and 1 comparison

For each row, the code runs from to . For filling the full tables, the running time is equal to the running time for filling each row multiplied by the number of items we can choose from

The running time for the dynamic programming pseudocode is therefore at most

 (proven)

Text

Description automatically generated

We are given a knowledge that . When we fill in the the table, we only need the row that is immediate above the current filling row. We know that the upper bound of the weight for all items is U, so we need at most U columns to record the previous filled values to calculate the weights. When weight is greater than multiples of U, the algorithm starts to overwrite the previous values, discarding the information of previous weights and keep the latest relevant information of U columns. To create a loop over multiples of U weights, we can use the Modulo operator. This algorithm correctness is due to the fact that only the latest filled values are relevant towards achieving the optimal value and we are guaranteed that we will never reuse the previous values, since we only need one row above and at most U columns before.

Knapsack Dynamic Programming Pseudocode  
-------------------------------------------------------------  
knapSack(N, C, U, itemWeight[], itemValue[])  
 // Creating table for recording the stages  
 table = Array[Array[Int]](N + 1, U + 1) // Dimension of the table is (N + 1) x (U + 1)  
 for (w = 0; w <= C; w++)  
 wMod = w % U  
 for (k = 0; k <= N; k++)  
 if (k == 0)  
 table[k][wMod] = 0  
 else if (itemWeight[k - 1] <= w)  
 wModWeight = (w - itemWeight[k - 1]) % U  
 table[k][wMod] = max(table[k - 1][wMod], table[k - 1][wModWeight + itemValue[k - 1]])  
 else  
 table[k][wMod] = table[k - 1][wMod]  
return table[N][C % U];

--------------------------------------------------------------

Unlike the previous algorithm in part (c) where we fill the rows vertically along the items, this algorithm fills the columns horizontally along the weights so that the modulo operator can loop over U columns of weights.

Space complexity analysis: since we only need N rows for the items and U columns for the weights in this algorithm, we need at mostmemory complexity

In fact, as we have known that we only look up at most one previous row, the further previous rows are also redundant for calculating the new optimal values => There exists an algorithm that needs memory complexity of just 