https://www.quora.com/I-am-not-able-to-visualize-the-2D-table-we-create-for-the-0-1-Knapsack-problem-Can-anybody-help

**I am not able to visualize the 2D table we create for the 0/1 Knapsack problem. Can anybody help?**

I am doing a problem bases on 0/1 Knapsack. But I am not able to see how the elements are being filled in the 2D table. Also how do I calculate the total weight which gives the optimal solution.

Problem Link: [SPOJ.com - Problem PARTY](http://www.spoj.com/problems/PARTY/)

4 Answers

[Rudra Nil Basu](https://www.quora.com/profile/Rudra-Nil-Basu)

[Rudra Nil Basu](https://www.quora.com/profile/Rudra-Nil-Basu), C,C++, Java. Nothing more, nothing less. Next stop - Python.

[Answered Jan 30, 2016](https://www.quora.com/I-am-not-able-to-visualize-the-2D-table-we-create-for-the-0-1-Knapsack-problem-Can-anybody-help/answer/Rudra-Nil-Basu)

Let us first look at the recursive solution to the problem.

Here, we consider that wt[i] denotes the cost of ith party, and val[i] denotes the fun value for ith party. W denotes the maximum cost provided. N is the total number of parties.

Let us define a function **int** knapSack(**int** W, **int** wt[], **int** val[], **int** n) , which will return the maximum "fun" which can be obtained from a maximum of "W" cost and nnumber of parties.

What should be the base case for the above recursive function ? Obviously if maximum cost given to us is zero (W==0) , then the we won't be able to enter into any party, which is obviously not funny, thus maximum fun in this case is 0. Also, if the number of parties is zero, we won't get the chance of having the fun ( n==0 ). Thus Base case of the function is

**if** (n == 0 || W == 0)

**return** 0;

Now for the next part. We have enough money, and enough parties. Now what if the cost of the nth party (wt[n-1 ) itself is greater than cost of the whole budget (W) provided to us ? It is pointless to desperately cry for something you can't get. You know it's time to forget and move on with what God has given you rather than blaming Him for the stuff he didn't give you. So, the amount of "fun" you get is irrespective of whether you consider the nth party or not. So, we reduce the number of computations by eliminating the impossibles.

**if** (wt[n-1] > W)

**return** knapSack(W, wt, val, n-1);

Now that you have enough money to go to nth party, you are still not satisfied. You want to get as much fun as you want with the limited money you have. So you sit down near the front door with a pen and paper and start calculating whether entering this nth party is worth it or not. You have 2 choices -

i) Enter this nth party, spend the money (wt[n-1]), have the fun that this nth party can give you, and then get out of the party with your pocket being less heavier (money remaining = W-wt[n-1] ). You had some fun though ( val[n-1] ), and you have n-1 more parties to satisfy your unending quench of having fun. So, considering you took this choice, what will the current function return ? Ans - val[n-1] + knapSack(W-wt[n-1], wt, val, n-1)

ii) Or you can ignore this party and look for the other n-1 parties, but this time, your pocket is filled with the original money W , but you are yet to have some fun. So this should return knapSack(W, wt, val, n-1)

So, considering the above two choices, which one should you take ? Obviously the one which provides you more fun! So, the final statement will look like this -

**else** **return** max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),

knapSack(W, wt, val, n-1)

);

The full Greedy approach looks something like this

1. **int** knapSack(**int** W, **int** wt[], **int** val[], **int** n)
2. {
3. **if** (n == 0 || W == 0)
4. **return** 0;
6. **if** (wt[n-1] > W)
7. **return** knapSack(W, wt, val, n-1);
9. **else** **return** max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
10. knapSack(W, wt, val, n-1)
11. );
12. }

But you don't want to waste so much time to calculate the same stuff over and over again where you should be having so much fun. So, you need the help of DP. It is clear from the greedy approach that for calculating the fun for the nth party, we should calculate the fun for the n-1 th party first. So to reduce the calculations, we store the values of the lower values in a 2D array, so that we do not have to calculate the previous values again and again. So we take and a 2D matrix K[n+1][W+1].  Now , **what does K[i][j] denote ?** K[i][j] denotes the amount of fun we will have, considering that there are i (the row number) number of parties, and j (the column number) is the total Weight which is given to us (thus breaking down the problem). Consider the example, val[]={1,4,5,7} and wt[]={1,3,4,5} . K[0][0]will be equal to zero, (base case). Rather if any of i or j is 0, K[i][j] is equal to 0. What will K[1][1] denote ? The maximum amount of fun you can get when there is only 1 party of fun value val[1-1] and entry cost is wt[1-1], and your total budget is 1. What is K[1][2] ? The maximum amount of fun you can get when there  is only 1 party of fun value val[1-1] and entry cost is wt[1-1], and your total budget is 2. In this way, you fill up the array as per the golden formula - K[i][w]=max(val[i-1]+K[i-1][j-wt[i-1]],K[i-1][j]);

What is the final answer ? The maximum fun you had considering n number of parties and W as the total money you were given. Where should the answer be in the matrix ? K[n][W] .

Next comes how to calculate the optimum weight. Create another matrix **int** wg[n+1][W+1]which will store the optimum weight for it's corresponding row and column in the K[][]matrix. wg[i][j] calculates the total weight for i no of parties and j amount of cost. For optimum value, Find the minimum value of the elements in the last row (nth row), and that minimum value is the optimal solution.

EDIT : No new array is required. Same can be done with K[][] array

You can check out my solution here - [RudraNilBasu/C-CPP-Codes](https://github.com/RudraNilBasu/C-CPP-Codes/blob/master/Competitive-Programming/SPOJ/PARTY.cpp" \t "_blank)  or more detailed discussion on 0/1 Knapsack - [Dynamic Programming | Set 10 ( 0-1 Knapsack Problem) - GeeksforGeeks](http://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/)  .

508 Views · [3 Upvotes](https://www.quora.com/api/mobile_expanded_voter_list?key=jpAB6AbO7HD&type=answer)

[Namit Sinha](https://www.quora.com/profile/Namit-Sinha), Build something that will live after i cast into void.

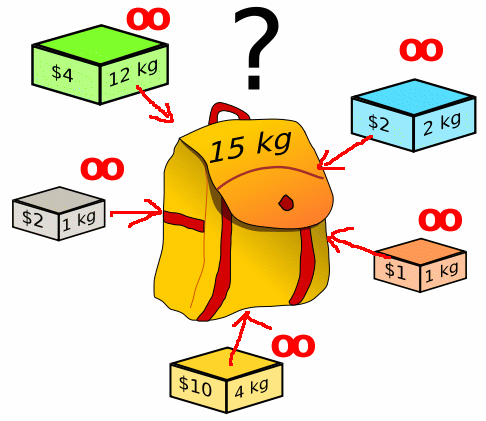
[Answered Jan 31, 2016](https://www.quora.com/I-am-not-able-to-visualize-the-2D-table-we-create-for-the-0-1-Knapsack-problem-Can-anybody-help/answer/Namit-Sinha)

**Don't think in code** **at first**that is the reason you are "not able to see how the elements are being filled in the 2D table" and "how to calculate the total weight which gives the optimal solution ".

There are many ways you arrive at Dynamic Programming(DP) solution. The one i find the easiest is following a step by step approach

1. First find a recursive solution.
2. Convert this in a Top-down algorithm.
3. Convert this to a Bottom-up Algorithm.

For example in a 0/1 Knapsack Problem say you have 5 items with their weights and price written on it.



Now you know that a item 'x' (say 4$ 12Kg block) is either selected or not for the final solution set (set of item whose weight is less than or equal to 15 but cost is maximized). So now your **problem is reduced to 2 sub problems** now we have to find making which choice would give us  higher value .

Picking 12Kg item for our knapsack (problem A in the figure) or discarding the 12Kg item(problem B) ?

This gives us 2 sub problems now you have to find which of the two choices will give you higher value for the knapsack the solution to the sub problems should not be in any way dependent on the last step (ie the value returned by problem A should not be affected by what steps we took on the first node) hence we will have to re-describe our two new problems.

That is easy to do because in problem A you have selected the 12 kg mass so the new knapsack has capacity 3 (15-12) so we need to find the best solution for knapsack of size 3 Kg (but now on you don't get to pick the 12 kg item because it is already taken care of, simply strike off item x from now).

For problem B you haven't picked the 12Kg item hence your knapsack still has the capacity of 15 kg.

**How will you use the solution of problem A and B to find the solution of the initial problem ?**

The solution of problem A and B is the maximum value of the knapsack possible using those configurations (size of the knapsack 3 or 15 in this case ...) so they are knapsack problem in themselves only without the 12 kg item.

you now have to find which of these values is greater **keeping in mind that after you selected 12kg mass you gained +4 value before generating problem A and +0 (nothing, because you discarded the 12kg item)**

**Final solution =  maximumOf ( Solution(A) + 4 , Solution(B) +0)**

if there is a list of items, the solution after picking/discarding the i'th  item is  given by solution[i] .

the weight and value of i'th item is represented by weight[i] and value[i] respectively

This Final Solution equation can be re-written as

**FinalSolution[i] = max ( solution(15 - weight[i])  + value[i] ,  solution(15) )**

**this is the crux of the DP solution.**

**Now what should we do if 15 - weight[i] is a negative number ? ?**

**Ans how will your solution function keep track of items you have considered ??**

If you notice the**'i'  in FinalSolution[i]** is Basically the number of items we currently have packed (considered to add) in the knapsack.

Here is a more expanded version of the tree each bubble represent a sub-problem with the knapsack size written on it

so at each step you need to know the item number you are on (because you will have to keep track of which items you have considered or not) and the knapsack capacity. These are the two numbers you see the in the table.

hence to keep track of the items already considered before we need to add another parameter to solution function.

solution(i , j) where i is the items already considered and j is the knapsack capacity.

now the final equation can be written as

**FinalSolution[i][j] = max ( solution(i - 1 , j - weight[i]) + value[i] , solution( i - 1 , j ) )**

hence i and j (items considered and knapsack size) are the only two dimensions required to describe a sub problem.

Now you can reduce every problem to sub problems and solve it recursively, but in doing so you will recompute solutions for many sub problems numerous times you can use memoization to improve this, basically create a 2D array (**lets call this 2d-array dp** ) which stores the answers for the sub problem and if we need this solution again we can simply query this 2D array.

This solution should work well, but it could be improved further by removing recursion by changing this solution to a bottom up algorithm. In this you will start computing answers for every element of dp array bottom up ie you first find the answers for every size of knapsack when you can only select a single item ( say item 5) for example you already know that if you can't pick any item or if the knapsack size is 0 the best value you can get is 0. So we first fill every element of dp with 0 then start our computation.

So now we will compute the best value for every knapsack size possible given that we can only select the 5th item then including  4th item then 3rd and so on. we do this by applying our finalSolution equation but taking care of negative values of **j - weight[i] .**

1. dp[number of items][maximum knapsack size] = {{0}}; //initialize to 0
3. **for**(**int** i = 1; i<=number of items **in** list; i++ ){
4. **for**(**int** j=1; j<=maximum possible weight of knapsack ; j++){
5. **if**(j >= weight[i])//check if j - weight[i] is non negative {
6. **int** notSelectedVal = dp[i-1][j];
7. **int** selectedVal = dp[i-1][j-weight[i]] + value[i];
8. dp[i][j] = max(notSelectedVal, selectedVal);
9. }
10. **else**{
11. dp[i][j] = dp[i-1][j]; //equivalent to notSelectedVal
12. }
13. }
14. }
16. // the answer is finally calculated at
17. // dp[number of items][maximum knapsack size]

Don't get confused by numbering it can be done from either side in description i have used numbering (ie 'i') from bottom but in code i have used it from top.

You can easily modify this algorithm to solve the problem at hand . I tried helping you with visualizing what is happening.

Oh god its 02:49 :P

684 Views · [3 Upvotes](https://www.quora.com/api/mobile_expanded_voter_list?key=jksq8my04mD&type=answer)

[Jayesh Patel](https://www.quora.com/profile/Jayesh-Patel-16)

[Jayesh Patel](https://www.quora.com/profile/Jayesh-Patel-16)

[Answered Sep 7, 2016](https://www.quora.com/I-am-not-able-to-visualize-the-2D-table-we-create-for-the-0-1-Knapsack-problem-Can-anybody-help/answer/Jayesh-Patel-16)

This link may be helpful, it explains in details both brute force recursive approach and Dynamic programming approach which solves problem occur in recursive approach along with Program.

It explains with pictures of Matrix data at each step for better understanding.  
[The Knapsack Problem](http://javabypatel.blogspot.in/2016/09/knapsack-problem-solution-recursive-iterative.html)

168 Views

[Shambhu Kumar](https://www.quora.com/profile/Shambhu-Kumar-40)

[Shambhu Kumar](https://www.quora.com/profile/Shambhu-Kumar-40), Sorry Shaktimaan!!!

[Answered Jan 31, 2016](https://www.quora.com/I-am-not-able-to-visualize-the-2D-table-we-create-for-the-0-1-Knapsack-problem-Can-anybody-help/answer/Shambhu-Kumar-40)

The underlying idea behind the Knapsack problem is constrained movement which is closely correlated to both - the superimposition of a 3D-reality onto a 2D-canvas and to the more general idea of optimization. So, imho, your problem can be solve if you focus your thoughts and actions at these two levels. Doing a course on Engineering Graphics will catalyze the process of trans-dimensional shift at the level of mind.

And one can improve one's optimization-skills while doing virtually anything and everything! Try using your elbows in order to switch off the lights. Try standing (and walking) with your heals elevated (with just air between you and the ground there). One can experiment with several kinds of lifestyles, with minimization of consumption and indulgences as an objective. In fact that *shall* give some of the best lessons on optimization.

Specific actions leading to similar abstractions are quite interconnected. And so temperament can travel across such actions more efficiently.

159 Views