Dynamic Programming | Set 10 (0-1 Knapsack Problem)

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

We strongly recommend that you click here and practice it, before moving on to the solution.

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

1) Optimal Substructure:
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.
Therefore, the maximum value that can be obtained from n items is max of following two values.
1) Maximum value obtained by n-1 items and W weight (excluding nth item).
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

2) Overlapping Subproblems
Following is recursive implementation that simply follows the recursive structure mentioned above.

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }
```

```c
// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
   // Base Case
   if (n == 0 || W == 0)
       return 0;

   // If weight of the nth item is more than Knapsack capacity W, then
   // this item cannot be included in the optimal solution
   if (wt[n-1] > W)
       return knapSack(W, wt, val, n-1);

   // Return the maximum of two cases:
   // (1) nth item included
   // (2) not included
   else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                    knapSack(W, wt, val, n-1)
                  );
}

// Driver program to test above function
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int  W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Run on IDE

Java

```java
/* A Naive recursive implementation of 0-1 Knapsack problem */
class Knapsack
{

    // A utility function that returns maximum of two integers
     static int max(int a, int b) { return (a > b)? a : b; }

     // Returns the maximum value that can be put in a knapsack of capacity W
     static int knapSack(int W, int wt[], int val[], int n)
     {
        // Base Case
    if (n == 0 || W == 0)
        return 0;

    // If weight of the nth item is more than Knapsack capacity W, then
    // this item cannot be included in the optimal solution
    if (wt[n-1] > W)
       return knapSack(W, wt, val, n-1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                     knapSack(W, wt, val, n-1)
                      );
      }
```

```
   // Driver program to test above function
   public static void main(String args[])
   {
        int val[] = new int[]{60, 100, 120};
        int wt[] = new int[]{10, 20, 30};
    int  W = 50;
    int n = val.length;
    System.out.println(knapSack(W, wt, val, n));
    }
}
/*This code is contributed by Rajat Mishra */
```

Run on IDE

Python

```
#A naive recursive implementation of 0-1 Knapsack Problem

# Returns the maximum value that can be put in a knapsack of
# capacity W
def knapSack(W , wt , val , n):

    # Base Case
    if n == 0 or W == 0 :
        return 0

    # If weight of the nth item is more than Knapsack of capacity
    # W, then this item cannot be included in the optimal solution
    if (wt[n-1] > W):
        return knapSack(W , wt , val , n-1)

    # return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    else:
        return max(val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1),
                   knapSack(W , wt , val , n-1))

# end of function knapSack

# To test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print knapSack(W , wt , val , n)

# This code is contributed by Nikhil Kumar Singh
```

Run on IDE

Output:

```
220
```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).

```
In the following recursion tree, K() refers to knapSack().  The two
parameters indicated in the following recursion tree are n and W.
The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

                       K(3, 2)         ---------> K(n, W)
                   /            \
                 /                \
            K(2,2)                  K(2,1)
          /       \                  /    \
        /           \              /        \
       K(1,2)      K(1,1)        K(1,1)     K(1,0)
       /  \         /   \          /   \
     /      \     /       \      /       \
K(0,2)  K(0,1)  K(0,1)  K(0,0)  K(0,1)   K(0,0)
Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.
```

Since suproblems are evaluated again, this problem has Overlapping Subprolems property. So the 0-1 Knapsack problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

C++

```
// A Dynamic Programming based solution for 0-1 Knapsack problem
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n+1][W+1];

   // Build table K[][] in bottom up manner
   for (i = 0; i <= n; i++)
   {
       for (w = 0; w <= W; 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];
       }
   }

   return K[n][W];
}

int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int  W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Run on IDE

Java

```
// A Dynamic Programming based solution for 0-1 Knapsack problem
class Knapsack
{

    // A utility function that returns maximum of two integers
    static int max(int a, int b) { return (a > b)? a : b; }

   // Returns the maximum value that can be put in a knapsack of capacity W
    static int knapSack(int W, int wt[], int val[], int n)
    {
         int i, w;
     int K[][] = new int[n+1][W+1];

     // Build table K[][] in bottom up manner
     for (i = 0; i <= n; i++)
     {
         for (w = 0; w <= W; 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];
         }
      }

      return K[n][W];
    }


    // Driver program to test above function
    public static void main(String args[])
    {
        int val[] = new int[]{60, 100, 120};
    int wt[] = new int[]{10, 20, 30};
    int  W = 50;
    int n = val.length;
    System.out.println(knapSack(W, wt, val, n));
    }
}
/*This code is contributed by Rajat Mishra */
```

Run on IDE

```python
# A Dynamic Programming based Python Program for 0-1 Knapsack problem
# Returns the maximum value that can be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]

    # Build table K[][] in bottom up manner
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif 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]

    return K[n][W]

# Driver program to test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))

# This code is contributed by Bhavya Jain
```

Run on IDE

Output:

```
220
```

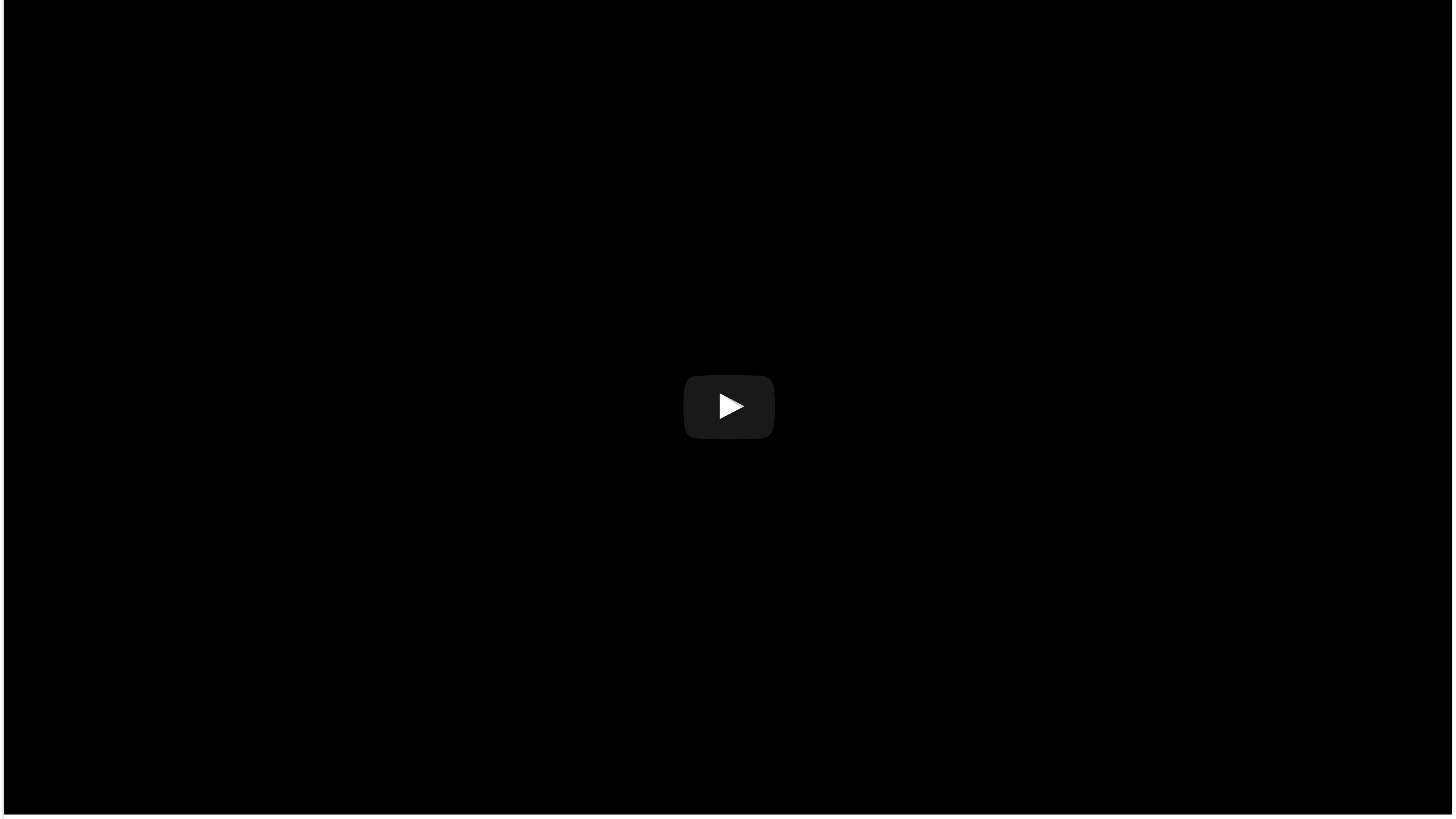
Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.

Asked in: Amazon, Flipkart, GreyOrange , Microsoft, Mobicip, Morgan Stanely, Oracle, Payu, Snapdeal, Visa

References:

http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf



Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   **Company Wise Coding Practice**

Dynamic Programming   knapsack

Recommended Posts:

Dynamic Programming | Set 11 (Egg Dropping Puzzle)
Dynamic Programming | Set 7 (Coin Change)
Dynamic Programming | Set 12 (Longest Palindromic Subsequence)
Dynamic Programming | Set 1 (Overlapping Subproblems Property)
Dynamic Programming | Set 9 (Binomial Coefficient)

(Login to Rate and Mark)

3.4 Average Difficulty : **3.4/5.0**
Based on **139** vote(s)

🟩 Add to TODO List
🟩 Mark as DONE

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Share this post!

183 Comments **GeeksforGeeks** **Dagny T**

Recommend 12 **Share** **Sort by Newest**

Join the discussion…

kaushik Lele · 3 days ago

In memoization; we create array of size N*W. If weights are large say 50K,100K,25123 etc. and required knpasack weight is 250K then this approach will be very time consuming.

Thus instead of "size" of inputs (size of input array) it is more dependent on "value" of inputs.

Can we make solution better to be actual-val-agnostic ? Is the recursive option only way ? Or there is enhanced version of memoization? If it is discussed in other post can you please share the link in post itself ?

· Reply · Share ›

Luckman · a month ago

The complexity of this solution is O(NW) in time and O(NW) in space. I try to provide a solution with O(NW) in time but O(W) in space. I am not sure if this works. I'd be happy if you can give me some opinions.

```
```

~~~~
public int knapsack(int[] weights, int[] values, int weight_limit){

int[] dp = new int[weight_limit+1];
Arrays.fill(dp, Integer.MIN_VALUE);
int ret_max = 0;
dp[0] = 0;
for(int i = 0; i < weights.length; i++){
for(int j = weight_limit; j >= 0; j--){
if(j-weights[i] >= 0 && dp[j-weights[i]] > Integer.MIN_VALUE){
dp[j] = Math.max(dp[j], dp[j-weights[i]] + values[i]);
if(dp[j] > ret_max){
ret_max = dp[j];
}
}
}
}
return ret_max;
}
```
~~~~

^ | ∨ • Reply • Share ›

suresh krishna • 2 months ago

can we take fractional values like half or oneforth of a given weight if there is space in bag

i mean there is space of 2 and there is a weight 5 can we take some part of 5

1 ^ | ∨ • Reply • Share ›

Shubham Surana ➦ suresh krishna • 2 months ago

Thats a different problem, this one is directly taking the thing whole or nothing.

Here : http://www.geeksforgeeks.or...

^ | ∨ • Reply • Share ›

Amir Khasru • 3 months ago

Now experts , if the number of each item is more than one how you will solve the problem by dynamic programming. Can any of you put an idea?

^ | ∨ • Reply • Share ›

Ashish K Rajput ➦ Amir Khasru • 3 months ago

We can solve this problem even when you are free to chose any item as many number of times. I guess this case will include your case as well which states if we have each item more than once.
The idea is simple again using DP. You need to find Max{T[i-1][j], val[i]+T`}
where T` would have to be calculated using below cases:

Get the value of j - wt[i] = w`. Now find T` corresponding to this w` using same formula as above and assigning same cases. For eg, for the new w`, consider the possibility if the same weight can be picked again--> find that value. Also find the value obtained after not picking that weight which would be T[i-1][j-wt[i]]. Select the maximum of these two values and add to the previous obtained value. This will get you T`. Put it in the formula we discussed previously,
Max{T[i-1][j], val[i]+T`}.
Fill out the whole table based on this.
Backtracking, however, might be a bit tricky here.
Let me know if any doubts in the above approach.

• Reply • Share ›

Shubham Surana • 3 months ago
what if the weights are not integers....any idea to do then

• Reply • Share ›

Aditya Aswal → Shubham Surana • 2 months ago
solve it greedily in that case.
Start adding the weights which give maximum profit per weight.

• Reply • Share ›

Shubham Surana → Aditya Aswal • 2 months ago
but how would you take index index in dp case coz here we take index of dp matrix as our required weight

• Reply • Share ›

Aditya Aswal → Shubham Surana • 22 days ago
I did't understand your question but here's how to do it greedily
value[n],weight[n] ;
use map<int,int> for storing (value[i]/weight[i],weight[i]);
start filling knapsack from the end

• Reply • Share ›

Sunilkumar • 3 months ago
The capital 'W' and lower case 'w' are confusing, can you please change the lower case 'w' to something else. The capital 'W' is fine.

• Reply • Share ›

doramamu • 4 months ago
where did I go wrong?

```
#include <iostream>
```

```
#include <stdio.h>
#include <malloc.h>

using namespace std;

int maxNum(int a, int b)
{
if (a>b)
return a;
else return b;
}

int knapSack(int W, int N, int wt[], int val[])
{
if (N == -1 || W == 0)
return 0;
```

see more

Reply · Share ›

Nikhil Kumar Singh doramamu · 4 months ago

Try running your code on http://www.practice.geeksfo... .. This is the online judge, will help to find out the error in your code in a much better way

Reply · Share ›

Ivawen · 5 months ago

but it does not tell which weights were finally chosen ? how do you implement that ?

Reply · Share ›

Nikhil Kumar Singh Ivawen · 4 months ago

In order to get the weights you can backtrack the matrix.

1 · Reply · Share ›

argha · 5 months ago

why k[n+1][w+1]?
if anyone knows share it will be very much helpful.

1 · Reply · Share ›

Ashwini Kumar argha · 4 months ago

Otherwise you got array out of bound Exception :-1

Reply · Share ›

Prashant Shubham argha · 5 months ago

The number of items can be zero also and so can be the weight chosen be zero so we need to include that zero value in matrix also. Thus, we take the matrix as one size larger than the original value. You can understand it in a way that there can also be a possibility when no weight is chosen or no item is picked. We need to consider these also so matrix is m[n+1][w+1].

Reply · Share ›

Mohammad Azeem · 5 months ago

Just to be clear, the problem/solution discussed here is for the case when duplicate items are forbidden, right?

Reply · Share ›

Ashish K Rajput → Mohammad Azeem · 3 months ago

The idea is simple again using DP. You need to find Max{T[i-1][j], val[i]+T`}
where T` would have to be calculated using below cases:
Get the value of j - wt[i] = w`. Now find T` corresponding to this w` using same formula as above and assigning same cases. For eg, for the new w`, consider the possibility if the same weight can be picked again--> find that value. Also find the value obtained after not picking that weight which would be T[i-1][j-wt[i]]. Select the maximum of these two values and add to the previous obtained value. This will get you T`. Put it in the formula we discussed previously,
Max{T[i-1][j], val[i]+T`}.
Fill out the whole table based on this.
Backtracking, however, might be a bit tricky here.
Let me know if any doubts in the above approach.

Reply · Share ›

Musa Paljoš → Mohammad Azeem · 4 months ago

Yes it is - Knapsack without repetitions

Reply · Share ›

Mohammad Azeem → Musa Paljoš · 4 months ago

Thanks :)

Reply · Share ›

Punit Sharma · 5 months ago

Converting the original recursive solution into dp will yield much better time complexity!

FAR BETTER THAN O(nW)

Check it here

Reply · Share ›

Ashwini Kumar ➦ Punit Sharma • 4 months ago

Very good.. really very good approximation. Salute you!

^ | ˅ • Reply • Share ›

Buddhi Prakash • 5 months ago

can it be solved by 1D dp?

^ | ˅ • Reply • Share ›

Erika Siregar • 5 months ago

Could someone please explain why the recursive function is
K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
instead of:
K[i][w] = max(val[i] + K[i-1][w-wt[i]], K[i-1][w]);

^ | ˅ • Reply • Share ›

Nikhil Kumar Singh ➦ Erika Siregar • 4 months ago

The reason for this is that although we have taken the input in val from 0 index, we have taken the base case on K[0][0] as if there are no item in the knapsack. If you want to make it clean. You can take the input in val and wt starting from 1 index. Then you can apply what your approach.

^ | ˅ • Reply • Share ›

Erika Siregar ➦ Erika Siregar • 5 months ago

I found the answer on the comments below.

^ | ˅ • Reply • Share ›

Md Farooq • 6 months ago

very nice explanation

^ | ˅ • Reply • Share ›

Ashish Jaiswal • 7 months ago

integer knapsack problem(duplicates permitted): O(c*n)
http://codepad.org/YiuuZOOC

^ | ˅ • Reply • Share ›

Saura_cr7 • 7 months ago

Time Complexity: O(nW)[Bottom-up dp]
http://ideone.com/JLBjAl

^ | ˅ • Reply • Share ›

Geo Anderson • 8 months ago

Does anyone know how to build a daily fantasy mma building lineup tool using knapsack?

Reply · Share ›

cherry me · 8 months ago

Hi, we are interested to develop knapsack multiple constraint in javascript or jquery.
we welcome any applicant to write us on: cherrywahin@gmail.com
thank you

Reply · Share ›

Vardaan Sangar · 8 months ago

1.different approach of recursion(Backtracking) in O(2^n) time O(1) space complexity
http://code.geeksforgeeks.o...
2.Dp Top down approach with O(Wn) time and space complexity.
http://code.geeksforgeeks.o...

1 · Reply · Share ›

Maliha Tasnim ➦ Vardaan Sangar · 5 months ago

why the time complexity is O(Wn) for top down approach in 0-1 knapsack problem??

Reply · Share ›

cherry me ➦ Vardaan Sangar · 8 months ago

hi Vardaan, are you familiar with multiple constraint knapsack

Reply · Share ›

Vardaan Sangar ➦ cherry me · 7 months ago

No
please tell the exact question

Reply · Share ›

cherry me ➦ Vardaan Sangar · 7 months ago

we want to develop an algorithm in javascript, jquery to do the following:

• every product have length, width, height AND weight
• we have multiple boxes (multiple sizes) . inner size of each box can be variable
• every box have a maximal weight
• some product can be mixed with others , some goes separate for example (pesticide cannot be mix in the same box with milk) but (water bottle can be mix in the same box with milk) for example
• when user add to cart and goes to checkout, we want to be able to

sort the items in cart in respective boxes the most optimal way. meaning each box holds which items in the cart.
• also in the backend, for our shipping calculation we need the know the number of boxes per each box size

• also if we can show each box how the items are positioned in 3D photo illustrative

Reply · Share ›

Lehar · 9 months ago

http://ideone.com/KQ56cw
Top down approach,
Prints the sequence.

Reply · Share ›

This comment was deleted.

jc Guest · 8 months ago

Hi,

From your algorithm I'm trying to get the list of kept items (your algo only outputs the max value). I need to keep it space optimized (for use on a PIC 8 bits uC) so the array of items kept, like the "dp" array, should stay of max n*2 length (or less if possible). I'm struggling. Any ideas?
Thanks in advance.

Reply · Share ›

Anurag Pasi · 9 months ago

topdown DP
http://ideone.com/EBV96N

Reply · Share ›

Shubham Chaudhary · 9 months ago

Topdown recursive + memoization:
https://ideone.com/3821A8

Reply · Share ›

Valkarie Andre Estevan Shubham Chaudhary · 8 months ago

Faad coder!!

Reply · Share ›

.NetGeek · 9 months ago

C# Implementation (Both Recursive & DP) : http://ideone.com/ZaOoNz

1 ^ | ∨ • Reply • Share ›

ameya • 9 months ago

cant the solution be with the value 5 * 10 = 50 weight where the value will become 5 * 60 = 300

^ | ∨ • Reply • Share ›

maverick ➦ ameya • 9 months ago

you have only 1 quantity of each item

1 ^ | ∨ • Reply • Share ›

H K • 10 months ago

using recursion with dynamic programming I think would be better. Here the loop runs (W + 1) * (n + 1), for the example, it would be 51 * 4 = 204. For recusion, its 11.
Actually, we don't need all values of K[i][w], only specific values to reach K[n][W].

^ | ∨ • Reply • Share ›

Kalyana Srinivas ➦ H K • 10 months ago

Hey! Can you share your code?

^ | ∨ • Reply • Share ›

H K ➦ Kalyana Srinivas • 10 months ago

```
int knapsack(int n, int remW) {
int val1, val2;
count++;
if (n == 0 | remW == 0)
return 0;
if (wt[n - 1] > remW) {
if (dp[n - 1][remW] != -1) {
val2 = dp[n - 1][remW];
}
else {
val2 = knapsack(n - 1, remW);
dp[n - 1][remW] = val2;
}
return val2;
}
if (dp[n - 1][remW - wt[n - 1]] != -1) {
val1 = dp[n - 1][remW - wt[n - 1]];
}
```

see more

Reply • Share ›

Load more comments

Subscribe Add Disqus to your site Add Disqus Add Privacy DISQUS