Knapsack Problem

1. Motivation and Background

A 1998 study of the Stony Brook University Algorithm Repository showed that, out of 75 algorithm problems, the knapsack problem was the 18th most popular and the 4th most needed. Knapsack problems appear in real world decision making processes whenever there is a resource allocation with cost constraints. For example, given a fixed budget, we select things to buy based on the cost and value of each item.

The knapsack problem is referred to as a combinatorial optimization problem, where one is trying to find an optimal solution from a given finite set of objects. In this problem you are given a knapsack of a specific capacity, a number of items that are each assigned a value and weight. The goal of this problem is to find the maximum value you can achieve while still remaining under the weight that the knapsack can hold. Many different versions of this problem exist and each generally needs their own algorithm.

1.1 Fractional Vs 0-1 knapsack

- *Fractional :*
  The fractional knapsack is an algorithm that allows the program to take portions of an item so that only part of the weight and part of the value will be taken into the knapsack in order to optimize value. So let's say we have a set of items {1, 2, ..., n}. The items have values $\{v_1, v_2, ..., v_n\}$ and weights $\{w_1, w_2, ..., w_n\}$. This type of problem is solved using greedy algorithm, wherein, we first take the item having the maximum value to weight ratio, and then pick portions of the next item which has the second largest value to weight ratio and so on. This way, we will have a knapsack containing the maximum value possible with the overall weight within/equal to the required limit.
  For example, consider the following case wherein we need to choose from the items in table so that our maximum weight of knapsack doesn't exceed 100 pounds.

Item	Weight (pounds)
Gold	63
Silver	21
Copper	35

For this problem, our optimal choice using greedy algorithm would be 63 pounds of gold, 21 pounds of silver and the remaining 16 pounds of copper.

This type of knapsack is not used as often since its application is not practical in a real world situation.

- *0-1:*
  The 0-1 knapsack algorithm permits the user to either take the entire item or none of it. It is named so because if an item will be taken into the knapsack it will be given the label of 1 and if it is to be discarded it will be given a value of 0. Thus the value and weight of the items in this case are not divisible unlike the Fractional case. This application is seen in most algorithms as it is more realistic. The 0-1 knapsack problem can be solved using a method known as Dynamic Programming, which we will be seeing in detail in this lecture.

1.2 Unbounded Vs Bounded knapsack

In the unbounded knapsack problem, we are allowed to use duplicates of items in order to achieve a maximum value. It is not often used as in practical situations, as we don't always have access to unbounded resources. The bounded knapsack, on the other hand permits us to use only one of each item which is more practical and thus is applicable to real world circumstances.

2. Greedy Algorithm for the Fractional Knapsack

The Pseudocode for the Greedy Algorithm solution for the Fractional Knapsack problem can be given as follows:

```
SORT( items )
while ( weight < capacity )
        if ( capacity >= weight + weight of item )
                add current item to knapsack
                weight += weight of item
        elif ( capacity <= weight + weight of current item AND capacity != weight )
                add fractional amount of item, 'f' to knapsack
                weight += f *weight of  item
```

2.1 Proving a Greedy Algorithm

- Show that there exists an optimal solution such that it selects the first greedy choice.
- A problem exhibits optimal substructure if an optimal solution to the problem also contains within it optimal solutions to each of the subproblems.

2.1.1 Proof of Optimal Greedy Choice

Let $I = \{i_1, i_2, ..., i_n\}$ be the items available for selection
Let $O = \{o_1, o_2, ..., o_j\} \subseteq I$ be the optimal solution of a problem P.

Let $G = \{g_1, g_2, \ldots, g_k\} \subseteq$ I be the greedy solution where the items are ordered according to the sequence of greedy choices.

Case 1 - The entire unit
Let K be the Knapsack Capacity. Suppose G takes an entire unit of g_1 (implying that $K \geq w_{g1}$). If O also takes an entire unit of g_1, then there is nothing to be done. Suppose that O does not take an entire unit of g_1. Then we remove weight wg_1 from O and put a whole unit of g_1 into it giving a new solution O'. Since g_1 has the maximum value to weight ratio, O' is at least as good as O. Hence $g_1 \in O'$ is also an optimal solution to P.

Case 2 - A fraction of the unit
Suppose G takes a fraction f of g_1 (implying $K = f{*}wg_1$). If O also takes f amount of g_1, then we are done. Suppose O takes less than f amount of g_1 (O cannot take larger than f amount of g_1), then we remove weight $f{*}wg_1$ from O and put f amount of g_1 into it, yielding a new solution O'. O' is as good as O. Hence f amount of $g_1 \in O'$ is also an optimal solution to P.

2.1.2 Proof of Optimal Substructure

We have shown there is an optimal solution O' that selects one unit of g_1. After we select g_1 the weight constraint decreases to $K'' = K - wg_1$, the item set then becomes $I'' = I - \{g_1\}$.

Let P'' be a fractional knapsack problem such that the weight constraint is K'', and the item set is I''. Let $O'' = O' - g_1$. To prove the optimal substructure property, we need to show that O'' is an optimal solution to P'' (an optimal solution to the problem contains within it an optimal solution the sub-problem).

Proof
Suppose, on the contrary, that O'' is not an optimal solution of P''. Let Q be an optimal solution of P'', which is more valuable than O''. Let $R = Q \cup \{g_1\}$. Observe that R is a feasible selection for P.

On the other hand, the value of O' (an optimal solution to P) equals the value of $O'' + g_1$, which is less than the cost of R (since we assume the value of $O'' < Q$). Hence we find a selection R which is more valuable than the optimal solution O'. A contradiction! Hence O'' is an optimal solution for P''.

Therefore, after each greedy choice is made, we are left with a problem of the same form as the original problem. Since P'' needs to be solved optimally, we can show that there exists an optimal solution for P'' which selects g_2.

2.1.3 Analysis

The greedy choice is always made based on the highest value to weight ratio. By sorting the entire item list by this ratio, the run-time is bounded by the sorting method used *O(nlog(n))*.

3. Dynamic Programming Method for 0-1 Knapsack

Dynamic programming proves to be the optimum method to be used for solving Knapsack problems over Brute Force method or the Divide and Conquer algorithm. This is because when the number of items and the Knapsack weight limit are large, then the running time of the Brute force method becomes very high. Also, we can recall that the divide and conquer method first divides the problem into various subproblems, solves the subproblems and finally combines the solutions to solve the original problem. If the subproblems are dependent, then this algorithm tends to solve repeatedly common subsubproblems and thus does more work than what is actually required. Here is where Dynamic Programming comes to our rescue.

Dynamic Programming is used when the solution of a problem can be recursively described in terms of solutions to subproblems. This is referred to as the *Decomposition*. The solutions of the subproblems are stored in a table to be used later. This technique is referred to as Memoization. Thus there is a tradeoff between time and space.

3.1. Steps in dynamic programming[1]

Step 1: Decomposition of the problem

Decompose the problem into smaller problems and find a relation between the structure of the optimal solution of the original problem and the solutions to all the subproblems.

Step 2: Principle of Optimality (Proof of Correctness)

Express the solution of the original problem in terms of the optimal solutions of the subproblems thus recursively defining the value of an optimal solution.

Step 3: Bottom-up computation

Compute the value of the optimal solution in a bottom-up fashion using a table structure.

Step 4: Construction of optimal solution

Finally construct the value of the optimal solution of the original problem from the computed information.

3.2 Knapsack Problem Definition[1]

Given some items each having some weight w and worth some value v. Pack the knapsack to get the maximum total value keeping in mind that the total weight should not exceed a fixed number K.

Step 1: Decompose the problem into smaller problems

First, we construct an array $V[0...n, 0...K]$. For $1 \leq i \leq n$ and $0 \leq w \leq K$, the entry $V[i, w]$ will store the maximum value (combined) of any subset of items $\{1, 2,..., i\}$ of combined size at most w. As we compute all the values of the array, then the entry $V[n, w]$ will have the maximum value of all the items that can fit into the sack, which happens to be the solution of our problem.

Step 2: Principle of Optimality

Now, we need to recursively define the value of the optimal solution in terms of the optimal solutions to sub problems.
The initial settings are:

$$V[0, w] = 0 \text{ for } 0 \leq w \leq K \qquad \text{//No item}$$
$$V[i, w] = -\infty \text{ for } w < 0 \qquad \text{//Illegal}$$

In order to compute $V[i, w]$ recursively for an item i, we have two choices:
(i) Leave i:
If the optimal solution for items $\{1, 2, ... , i-1\}$ with the storage limit w is $V[i-1, w]$.

(ii) Include i:
Item i can be included only if $w_i \leq w$. If the optimal solution $V[i, w]$ includes i then leaving it will give us the optimal solution $V[i-1, w-w_i]$.

$$\text{So, } V[i, w] = V[i-1, w-w_i] + v_i$$

Thus, the recursively computed optimal solution can be given as:

$$V[i, w] = max(V[i-1, w], v_i + V[i-1, w-w_i]), \text{ for } 1 \leq i \leq n, 0 \leq w \leq K$$

1- Lecture notes of Prof. Dr. ir. RHJM (Ralph) Otten
http://www.es.ele.tue.nl/education/5MC10/

Step 3: Bottom up computation of V [i, w]

The formula given above is now used for computing *V [i, w]* for each entry in the table.

V[i, w]	w = 0	1	2	. . .	n
i =0	0	0	0	. . .	0
1	0	→	→	→	→
2	0	→	→	→	→
. . .	. . .	→	→	→	→
n	0	→	→	→	→

The pseudocode upto this step can be given as:

```
Knapsack(v, w, n, K)
{
for (w = 0 to K)
        $V [0, w] = 0$
for (i = 1 to n)
        for (w = 0 to K)
                if ( ($w_i \le w$)
                        $V [i, w] = \max(V [i-1, w],\ v_i + V [i-1, w-w_i])$
                else
                        $V [i, w] = V [i-1, w]$;
return $V [n, w]$
}
```

Step 4: Constructing the optimal solution

The algorithm described above just computes the maximum value of the Knapsack. It does not keep record of the subset of items that contribute to this maximum value. For this, we can store those items in a Boolean array *retain[i, w].* If an item is to be included in the Knapsack, then we store a 1 in the array, else store 0. After finding the maximum value of the Knapsack, we can go through the *retain[i, w]* array, for each item and output the item which has a 1 stored in the array. This gives us the items that are to be included in the Knapsack.

The algorithm for this part can be given as:

```
$R = K$;
for(i = n downto 1)
        if (retain[i, R] == 1)
                output i
                $R = R-w_i$
```

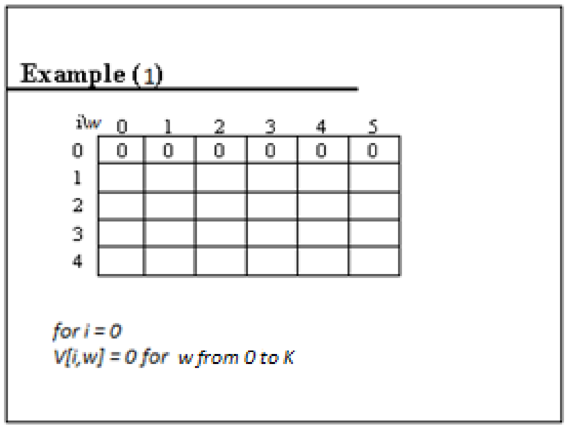
Thus the final algorithm is the combination of the previously given two functions.

```
Knapsack (v,w,n,K)
{
for (w = 0 to K)
        V [0, w] = 0
for (i = 1 to n)
{
        for (w = 0 to K)
        {
                if ( ($w_i \le w$) and ($v_i$ + V [i-1, w-$w_i$] > V [i-1, w]) )
                {
                        V [i, w] = $v_i$ + V [i-1, w-$w_i$]
                        retain[i, w] = 1
                }
                else
                {
                        V [i, w] = V [i-1, w]
                        retain[i, w] = 0
                }
        }
}
 R = K
for(i = n downto 1)             //Displaying the set of items included in the knapsack
{
        if (retain[i, R] == 1)
        {
                output i
                R = R-$w_i$
        }
}
return V [n, w]
}
```

Table 1 shows an example of the bottom up computation table for items with their values and weights as mentioned below:

Item	1	2	3	4
Value	3	4	5	6
Weight	2	3	4	5

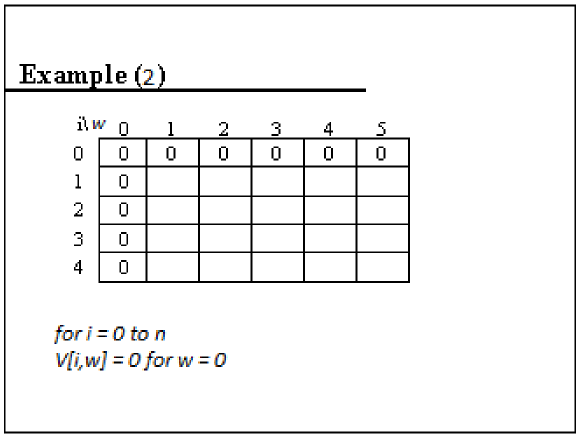
The maximum weight of the knapsack is 5. [2]



Example (1)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

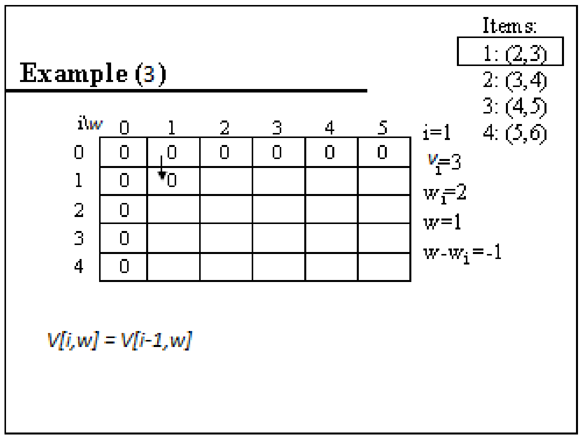
for i = 0
V[i,w] = 0 for w from 0 to K



Example (2)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

for i = 0 to n
V[i,w] = 0 for w = 0



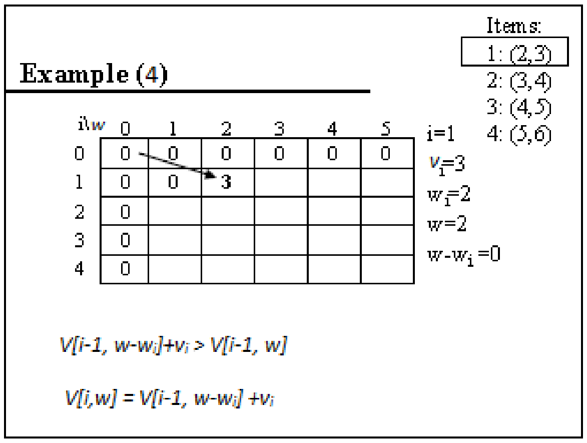
Example (3)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i=1$
$v_i=3$
$w_i=2$
$w=1$
$w-w_i=-1$

V[i,w] = V[i-1,w]



Example (4)

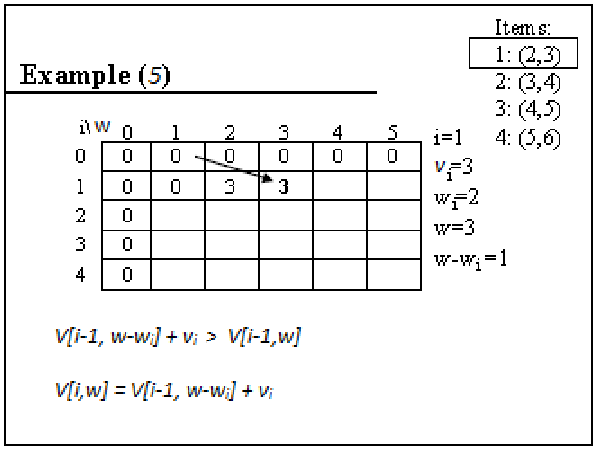
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	**3**			
2	0					
3	0					
4	0					

$i=1$
$v_i=3$
$w_i=2$
$w=2$
$w-w_i=0$

V[i-1, w-w$_i$]+v$_i$ > V[i-1, w]

V[i,w] = V[i-1, w-w$_i$] +v$_i$



Example (5)

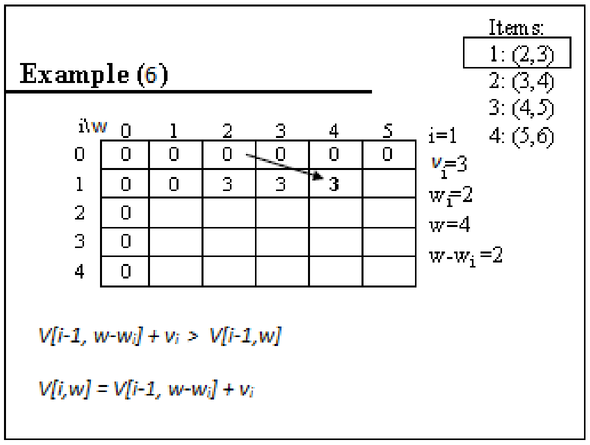
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	**3**		
2	0					
3	0					
4	0					

$i=1$
$v_i=3$
$w_i=2$
$w=3$
$w-w_i=1$

V[i-1, w-w$_i$] + v$_i$ > V[i-1,w]

V[i,w] = V[i-1, w-w$_i$] + v$_i$



Example (6)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

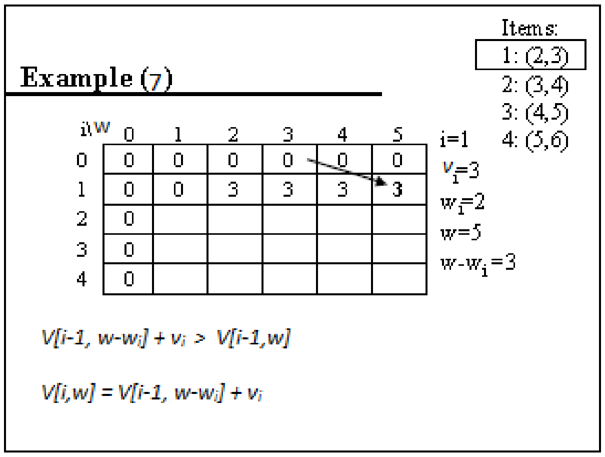
i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	**3**	
2	0					
3	0					
4	0					

$i=1$
$v_i=3$
$w_i=2$
$w=4$
$w-w_i=2$

V[i-1, w-w$_i$] + v$_i$ > V[i-1,w]

V[i,w] = V[i-1, w-w$_i$] + v$_i$

2- Lecture notes of Prof. Steve Goddard, http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf



Example (7)

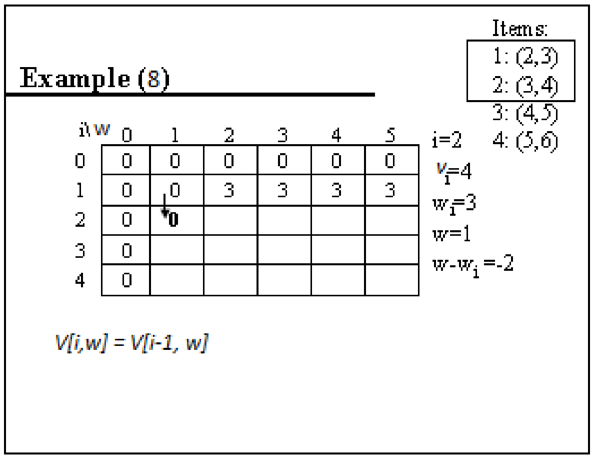
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	**3**
2	0					
3	0					
4	0					

$i=1$
$v_i=3$
$w_i=2$
$w=5$
$w-w_i=3$

V[i-1, w-w$_i$] + v$_i$ > V[i-1,w]

V[i,w] = V[i-1, w-w$_i$] + v$_i$



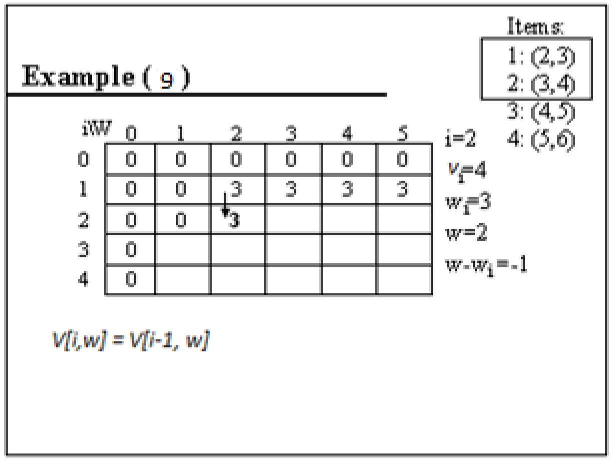
Example (8)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	**0**				
3	0					
4	0					

$i=2$
$v_i=4$
$w_i=3$
$w=1$
$w-w_i=-2$

V[i,w] = V[i-1, w]



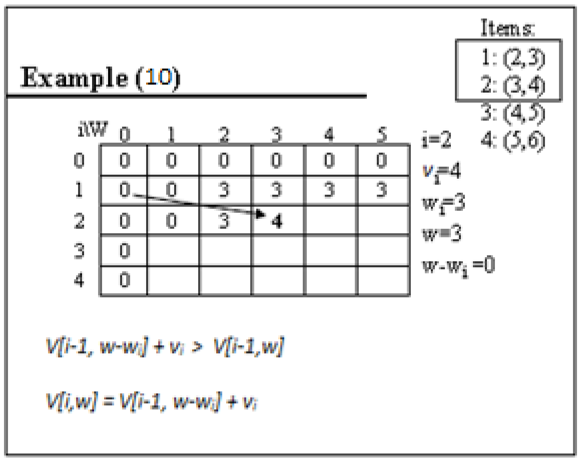
Example (9)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	**3**			
3	0					
4	0					

$i=2$
$v_i=4$
$w_i=3$
$w=2$
$w-w_i=-1$

V[i,w] = V[i-1, w]



Example (10)

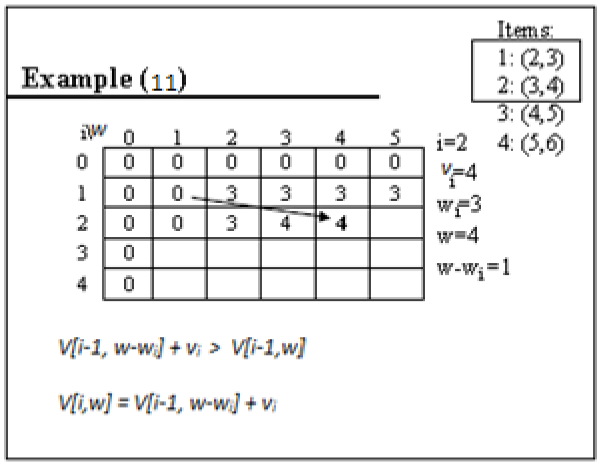
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	**4**		
3	0					
4	0					

$i=2$
$v_i=4$
$w_i=3$
$w=3$
$w-w_i=0$

V[i-1, w-w$_i$] + v$_i$ > V[i-1,w]

V[i,w] = V[i-1, w-w$_i$] + v$_i$



Example (11)

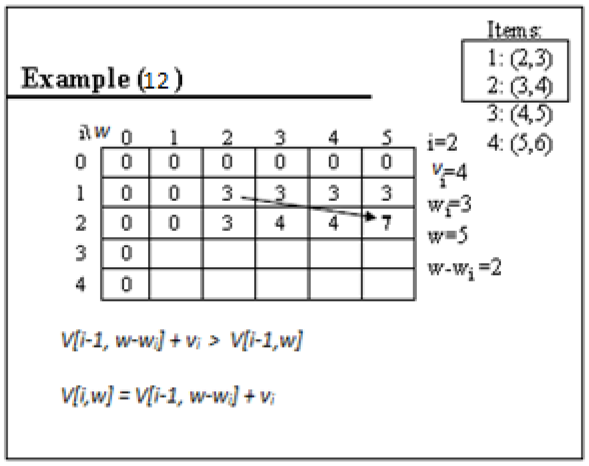
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	**4**	
3	0					
4	0					

$i=2$
$v_i=4$
$w_i=3$
$w=4$
$w-w_i=1$

V[i-1, w-w$_i$] + v$_i$ > V[i-1,w]

V[i,w] = V[i-1, w-w$_i$] + v$_i$



Example (12)

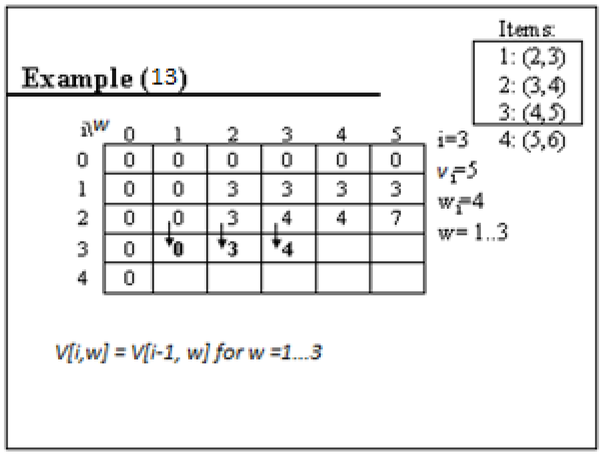
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	**7**
3	0					
4	0					

$i=2$
$v_i=4$
$w_i=3$
$w=5$
$w-w_i=2$

V[i-1, w-w$_i$] + v$_i$ > V[i-1,w]

V[i,w] = V[i-1, w-w$_i$] + v$_i$



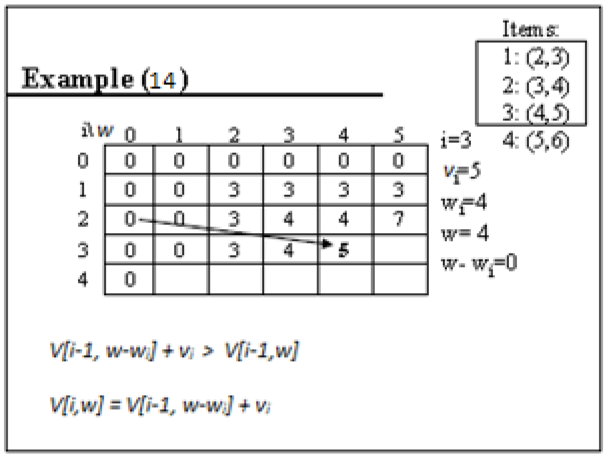
Example (13)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

$i=3$
$v_i=5$
$w_i=4$
$w=1..3$

V[i,w] = V[i-1, w] for w =1…3



Example (14)

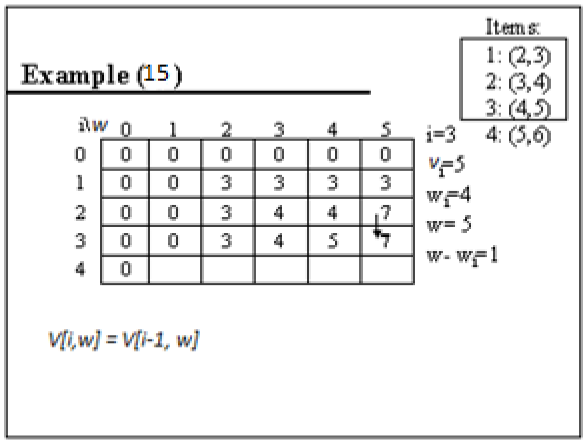
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i=3$
$v_i=5$
$w_i=4$
$w=4$
$w-w_i=0$

V[i-1, w-w$_i$] + v$_i$ > V[i-1,w]

V[i,w] = V[i-1, w-w$_i$] + v$_i$



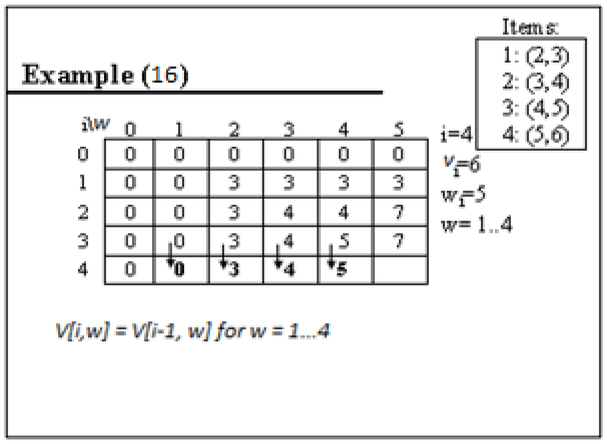
Example (15)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

$i=3$
$v_i=5$
$w_i=4$
$w=5$
$w-w_i=1$

V[i,w] = V[i-1, w]



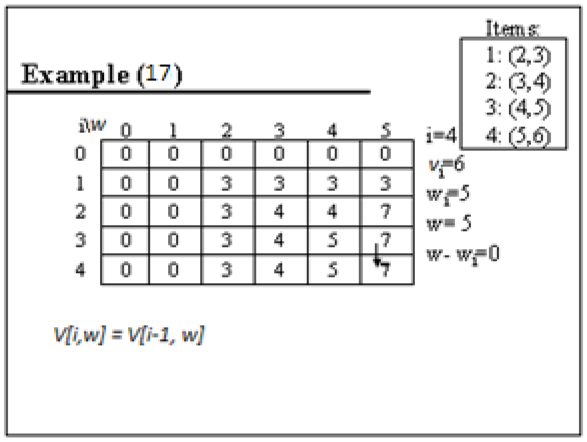
Example (16)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

$i=4$
$v_i=6$
$w_i=5$
$w=1..4$

V[i,w] = V[i-1, w] for w = 1…4



Example (17)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=4$
$v_i=6$
$w_i=5$
$w=5$
$w-w_i=0$

V[i,w] = V[i-1, w]

Retain array *retain[i, w] :*

i/w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	1	1	1	1
2	0	0	0	1	1	1
3	0	0	0	0	1	0
4	0	0	0	0	0	0

Optimal Knapsack items = {1, 2}

Maximum value of the knapsack = 7

Weight of the Knapsack =5

3.3 Proving the Dynamic Programming Algorithm

- **Decomposition**
  Construct an array $V[n*K]$ with n as the number of items and K as the capacity of the knapsack. For $1 \le i \le n$, and $0 \le w \le K$, the entry $V[i,w]$ will store the maximum value for any subset of items $\{1, 2, \ldots ,i\}$ of a combined size of at most w. If we can compute all of the entries of this array then the array entry $V[n,K]$ will contain the maximum value of items that can fit into our knapsack.

- **Recursively Define the Optimal Solution aka Principle of Optimality**
  Initial Settings:
  $V[0,w] = 0$ for $0 \le w \le K$ //no item
  $V[i,w] = -\infty$ for $w < 0$ //illegal item
  Recursive Step:
  $V[i,w] = max(V[i\text{-}1, w], v_i + V[i\text{-}1, w\text{-}w_i])$ for $1 \le i \le n$ and $0 \le w \le K$

- **Prove Correctness of Function**
  Lemma:
  $V[i,w]$ = Value of optimal solution for knapsack of capacity w and items from $\{1, 2, \ldots, i\}$.
  For $1 \le i \le n$ and $0 \le w \le K$
  $$V[i,w] = max(V[i-1, w]; v_i + V[i\text{-}1, w\text{-}w_i])$$

  Proof:
  To compute $V[i,w]$ we note that we only have two choices for a given item i:

Case 1: Leave Item
The best we can do with items $\{1, 2, ..., i-1\}$ and capacity limit w is $V[i-1, w]$. Therefore, adding the current item is either impossible given the capacity and weight, or the value added is not enough to replace an included item.

Case 2: Take Item
This is only possible if $w_i \leq w$. We gain v_i of value, but have used up w_i space in our knapsack. The best we can do with the remaining items $\{1, 2, ..., i-1\}$ and capacity $(w-w_i)$ is $V[i-1, w-w_i]$. In total, we get $v_i + V[i-1, w-w_i]$. This adds the current item to the highest possible value of the remaining capacity of the knapsack.

If $w_i > w$, then $v_i + V[i-1, w-w_i] = -\infty$ and so the lemma is correct in any case.

- **Analysis**
  Since each item $(0 \leq i \leq n)$ and weight $(1 \leq w \leq K)$ must be compared, the total run-time of the dynamic programming algorithm is $O(n*K)$. The total space required for the matrix is also $O(n*K)$.

References:

- http://www.swatijain.tripod.com/knapsack2.htm
- http://en.wikipedia.org/wiki/Knapsack_problem
- http://wiki.gametheorylabs.com/groups/kb/wiki/fa460/Knapsack_Problem.html
- http://www.es.ele.tue.nl/education/5MC10/
- http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf
- *Algorithms* by S. Dasgupta, C. H. Papadimitriou and U. V. Vazirani