Dynamic programming-2
0-1 Knapsack problem

How to pack the bag to maximize total amount without overloading your luggage ?

Prof.Dr.M. C. Okur

Reminder :Elements of Dynamic Programming

For implementing dynamic programming we should be able to break the original problem to smaller subproblems that have the same structure:

- Optimal Substructure
  - An optimal solution to a problem can be constructed efficiently from optimal solutions of its subproblems.
- Overlapping Subproblems
  - A problem has overlapping subproblems if finding its solution involves solving the same subproblem multiple times.

→ DP :Solves problems recursively by combining the solutions to similar smaller overlapping subproblems

Reminder : Memoization

- Memoization is another way to deal with overlapping subproblems in dynamic programming
- After computing the solution to a subproblem, store it in a table and in subsequent calls just do a table lookup .
- With memoization, we implement the algorithm recursively:
  - If we encounter a subproblem we have already seen, we look up the answer
  - If not,we compute the solution and add it to the list of subproblems we have seen.
- Most useful when the algorithm is easiest to implement recursively, especially if we do not need solutions to all subproblems.

Knapsack Problem

- We are given a set of n items, where each item i is specified by a weight w_i (or size s_i) and a value v_i $(benefit\ i)$.
- We are also given a limit W, the capacity of our knapsack.

Problem: How to pack the knapsack to achieve maximum total value of packed items?

Example1:

n = 4 (# of elements)

W = 5 (maximum capacity)

Elements (weight, value) = { (1, 100), (3, 240), (2, 140), (5, 150) }

Solution : (2,3) ,Benefit =240+140

Example2:

n=4, W=4 weights={5, 7, 5, 6}

no solution

Knapsack Problem Types

There are two versions of the problem:

1. "0-1 knapsack problem"
   Items are indivisible; you either take an item or not. Some special instances can be solved with dynamic programming

2. "Fractional knapsack problem"
   Items are divisible : you can take any fraction of an item.

→0-1 means take item or leave it (no fractions).We are going to consider this problem.

The 0-1 Knapsack Problem

- How to apply Dynamic Programming ? We have difficulties.
- Difficulty is in characterizing subsets :
  - Find best solution for first k units and use it later – no good, as optimal solution doesn't necessarily build on earlier solutions. Why ?
    - Find best solution for first k units, within quantity limit
    - Either use previous best at this limit, or new item plus previous best at reduced limit

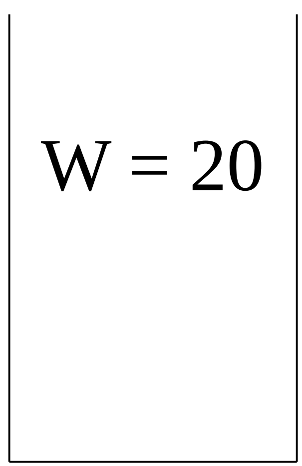
  → Previous solution for k has to be changed.

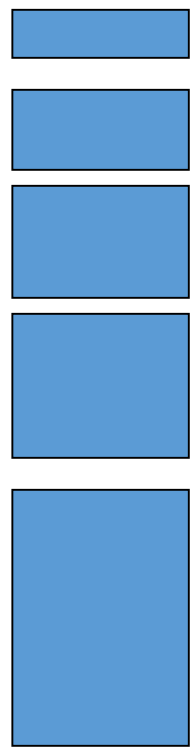
  → This is contrary to standard Dynamic Programming !

  We need a different formulation of subsets.

0-1 Knapsack Problem: Example

This is a knapsack
Max weight: W = 20





Items	Weights w_i	Benefit values b_i
	2	3
	3	4
	4	5
	5	8
	9	10

0-1 Knapsack problem

Problem: Find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is a *"0-1"* knapsack problem, because each item must be entirely accepted or rejected

Reminder : Solving 0-1 Knapsack Problem by Brute-force Approach

Let's first try to solve this problem with a straightforward Brute Force algorithm:

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with highest total value and with total size less than or equal to W
- Complexity will be $O(2^n)$

Example: n= 3, items A,B,C

Possible knapsack contents :

S={Empty, A,B, C, AB, AC, BC, ABC }

$|S| = 8 = 2^3$

Defining a Subproblem

- We can do better with an algorithm based on dynamic programming
- We need to identify the subproblems carefully.

DP solution should be composed of solutions to subproblems for subsets of k items (1,2,...,k).
→If items are labeled *1..n*, then a subproblem would be to find an optimal solution for
S_k = *{items labeled 1, 2, .. k}*

Defining a Subproblem

- This is a valid subproblem definition.
- But here is dynamic programming question : can we describe the final solution (S_n) in terms of subproblems (S_k) ?
- Unfortunately the answer is no, we can not do that. Why ?

Explanation follows….

Defining a Subproblem

Example:

- Assume that the best set of items for k=3: $S_k = \{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$.
- But,the best set of items from $S_{k+1} = \{I_0, I_1, I_{2,} I_3\}$ may be $\{I_0, I_2, I_3\} \neq \{I_0, I_1, I_2\}$.
- In this example, the optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution.
- Instead it builds upon the solution $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$

Defining a Subproblem : Example1

Consider the previous example :

n = 4 (# of elements)
W = 5 (maximum capacity)
Elements (weight, value) = {(1, 100), (3, 240), (2, 140), (5, 250)}
k=1,Solution :Item 4
k=k+1 =2 ,Solution :Items 2 and 3
→Item 4 is not part of k=2 solution.

Defining a Subproblem : Example2

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 3$ $b_4 = 4$	

Max weight: W = 20

Items in S_4: 1,2,3,4

Total weight=14

Maximum benefit=20

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_5 = 9$ $b_5 = 10$

Items in S_5 : 1,2,3,5

Total weight: 20

Maximum benefit: 26

S_5 S_4

Item #	Weight w_i	Benefit b_i
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

Solution for S_4 is not part of the solution for S_5!!!

Adding another Parameter to Subproblems

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is not valid and we need another one!
- Let's add another parameter: w, which will represent the exact total weight for each subset of items.

The problem is:

How to find an optimal solution for the combination (k,w)?

Warning : W and w are different! W represents the total capacity, w represents a quantity that must be optimized for each choice of k.

Recursive Formulation for Subproblems

- The subproblems will then be to compute the elements of a matrix *V[k,w]*.

→*We have to optimize:*

S_k = {items labeled 1, 2, .. k} in a knapsack of size w

Assuming we know V[i, j], where i=0,1, 2, … k-1, j=0,1,2, …w, how to derive V[k,w]?

V[k,w] = Matrix (Table) value for choices k and w.

→Find these values using previous entries in the table.

→This is Memoization!

Recursive Formulation for Subproblems

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

The best subset of S_k that has the total weight $\leq w$, either contains item k or not. Two cases :

- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be > w.
- Second case: $w_k \leq w$. Then the item k can be in the solution, and we choose *the case with greater (max) value*.

Recursive Formulation of Subproblems:Explanation

The formulation means that, the best subset of S_k that has total weight w is one of the two:

1) the best subset of S_{k-1} that has total weight w, **or**

2) the best subset of S_{k-1} that has total weight w-w_k plus the value of item k

→The best subset of S_k that has the total weight w, either contains item k or not(Notice the similarity with shortest paths solution).

- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be > w, which is unacceptable
- Second case: $w_k <= w$. Then item k can be in the solution, and we choose the case with greater value.

0-1 Knapsack Algorithm

Input: Sets of items S and w weights. Weights and values are stored in separate arrays. Output: Total value of best set items in a valid knapsack.

Algorithm Knapsack01

```
for w = 0 to W
  V[0,w] = 0
for i = 1 to n
  V[i,0] = 0
for i = 1 to n
  for w = 0 to W
        if w_i <= w     // item i can be part of the solution
                if b_i + V[i-1,w-w_i] > V[i-1,w]
                        V[i,w] = b_i + V[i-1,w- w_i]
                else
                        V[i,w] = V[i-1,w]
        else V[i,w] = V[i-1,w]       // w_i > w
```

Explanation of 0-1 Knapsack Algorithm

- Initializations for empty knapsack:
  First for-loop: We go through all the possible sizes of our knapsack until W and if item i is equal to 0, which is "V[0,w]", maximum value is 0. Because when i = 0, this means that we are not taking any item.
- Second for-loop: $\mathrm{i} = 1\ to\ n$. We go through all the items from 1 to n and if the knapsack's size is equal to 0, which is "V[i, 0]". Corresponding values is again 0. Because when w = 0, this means that we can't put anything in the knapsack
- The outer if and else conditions inside the third for loop ($w_i \leq w$) check if the knapsack can hold the current item or not.
- The inner if and else conditions check if the current value is bigger than the previous value so as to maximize the values the knapsack can hold.

Knapsack 0-1 Problem – Run Time

for w = 0 to W

  B[0,w] = 0

$O(W)$

for i = 1 to n

  B[i,0] = 0

$O(n)$

Repeat n times

for i = 1 to n

  for w = 0 to W

$O(W)$

      < the rest of the code >

What is the running time of this algorithm?

$T(n)= O(W)+O(n)+O(n*W)$

$= O(n*W)$

Run Time Comparison

- Remember that the brute-force algorithm takes $O(2^n)$
- When comparing complexities, we find that depending on W, either the dynamic programming algorithm is more efficient or the brute force algorithm could be more efficient.
- For example, for n=5, W=100000, brute force is preferable, but for n=30 and W=1000, the dynamic programming solution is preferable:

Case1: n=5 : $2^n = 32$, $n*W = 500000$

Case2: n=30 : $2^n = 1073741824$, $n*W = 30000$

Tracing 0-1 Knapsack Algorithm : Example(1)

Consider the following sample data:

Input :

n = 4 (# of elements)

W = 5 (max weight,capacity)

Elements : (weight, benefit)

(2,3), (3,4), (4,5), (5,6)

Output :

The table (matrix) V

Example (2)

Initialize first row and column to 0.

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

for $w = 0$ to W

$V[0,w] = 0$

Example (3)

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 = 1 to n

$V[i,0] = 0$

Recursive computations: use line 0 to calculate line 1, use line 1 to calculate line 2, etc. ... until all lines are calculated.

Example (4)

Note: All matrix computations can be simplified as :

$V(i, j) = \max\{ V(i-1, j), b_i + V(i-1, j - w_i) \}$

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					

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

if $w_i <= w$ // item i can be part of the solution

if $b_i + V[i-1,w-w_i] > V[i-1,w]$

$V[i,w] = b_i + V[i-1,w- w_i]$

else

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

else $V[i,w] = V[i-1,w]$ // $w_i > w$, so accept this.

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**			
2	0					
3	0					
4	0					

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

if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$ // Yes
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$ // $w_i > w$

Note: that **$V(i, j) = \max\{ V(i-1, j), b_i + V(i-1, j-w_i)\}$**

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**		
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

```
if w_i <= w // item i can be part of the solution
        if b_i + V[i-1,w-w_i] > V[i-1,w]     // yes
            V[i,w] = b_i + V[i-1,w- w_i]
        else
            V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

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**	
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i = 2$

if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w] // Yes
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

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					
3	0					
4	0					

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

if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$ // Yes
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$ // $w_i > w$

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	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

if w_i <= w // item i can be part of the solution

if $b_i + V[i-1,w-w_i] > V[i-1,w]$

$V[i,w] = b_i + V[i-1,w-w_i]$

else

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

else V[i,w] = V[i-1,w] // $w_i > w$

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**			
3	0					
4	0					

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

```
if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

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**		
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

if w_i <= w // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w-w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$ // $w_i > w$

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**	
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

```
if w_i <= w // item i can be part of the solution
        if b_i + V[i-1,w-w_i] > V[i-1,w]
            V[i,w] = b_i + V[i-1,w- w_i]
        else
            V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

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					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

if $w_i <= w$ // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w-w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$ // $w_i > w$

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**		
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w= 1..3$

if $w_i <= w$ // item i can be part of the solution

if $b_i + V[i-1,w-w_i] > V[i-1,w]$

$V[i,w] = b_i + V[i-1,w- w_i]$

else

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

else $V[i,w] = V[i-1,w]$ // $w_i > w$

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**	
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w= 4$

$w- w_i=0$

if w_i <= w // item i can be part of the solution

  if $b_i + V[i-1,w-w_i] > V[i-1,w]$

    $V[i,w] = b_i + V[i-1,w- w_i]$

  else

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

else $V[i,w] = V[i-1,w]$ // $w_i > w$

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					

$i=3$

$b_i=5$

$w_i=4$

$w= 5$

$w- w_i=1$

if w_i <= w // item i can be part of the solution

    if b_i + V[i-1,w-w_i] > V[i-1,w]

      V[i,w] = b_i + V[i-1,w- w_i]

    else

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

else V[i,w] = V[i-1,w] // w_i > w

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**	

i=4

b_i=6

w_i=5

w= 1..4

if w_i <= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

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

else V[i,w] = V[i-1,w] // w_i > w

Example (18)

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**

$i=4$

$b_i=6$

$w_i=5$

$w= 5$

$w- w_i=0$

if w_i <= w // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Comments

- The maximum value of benefit that can be obtained for optimal packing is 7.
- This algorithm only finds the max possible value (Total benefit)that can be carried in the knapsack
- To know the items that make this maximum value, an addition to this algorithm is necessary
- All we need is to trace back the computations for the maximum value in the table

How to find actual Knapsack Items

- All of the information we need is in the table.
- *V[n,W]* is the maximal value of items that can be placed in the Knapsack.
- Let i=n and k=W

if $V[i,k] \neq V[i-1,k]$ then

  mark the i^{th} item as in the knapsack

  $i = i-1,\ k = k-w_i$

else

  $i = i-1$ // Assume the i^{th} item is <u>not</u> in the knapsack

// Could it be in the optimally packed knapsack?

Finding the Items

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

$i=4$

$k= 5$

$b_i=6$

$w_i=5$

$V[i,k] = 7$

$V[i-1,k] =7$

i=n, k=W

while i,k > 0

    if $V[i,k] \neq V[i-1,k]$ then

        mark the i^{th} item as in the knapsack

        $i = i-1,\ k = k-w_i$

    else $\quad i = i-1$

Finding the Items (2)

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

$i=4$
$k= 5$
$b_i=6$
$w_i=5$
$V[i,k] = 7$
$V[i-1,k] =7$

while i,k > 0
    if $V[i,k] \neq V[i-1,k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        i = i–1

Finding the Items (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	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=3

k= 5

b_i=5

w_i=4

$V[i,k] = 7$

$V[i-1,k] = 7$

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1$, $k = k-w_i$

else

$\boldsymbol{i = i-1}$

Finding the Items (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	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=2$

$k= 5$

$b_i=4$

$w_i=3$

$V[i,k] = 7$

$V[i-1,k] =3$

$k - w_i=2$

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1,\ k = k-w_i$

else

i = i–1

Finding the Items (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	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=1$

$k= 2$

$b_i=3$

$w_i=2$

$V[i,k] = 3$

$V[i-1,k] =0$

$k - w_i=0$

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1$, $k = k-w_i$

else

$\boldsymbol{i = i-1}$

Finding the Items (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	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=0

k= 0

The optimal knapsack should contain {1, 2}

while i,k > 0

  if $V[i,k] \neq V[i-1,k]$ then

    mark the n^{th} item as in the knapsack

    $i = i-1$, $k = k-w_i$

  else

    $i = i-1$

Finding the Items (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	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

The optimal knapsack should contain items (1, 2)

while i,k > 0 // No,stop
    if $V[i,k] \neq V[i-1,k]$ then
        mark the n^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Knapsack Problem : Applications

- Applicable in most resource allocation problems where there are financial or other constraints :
  - Resource allocation with financial constraints
  - Construction and scoring of heterogeneous test sets (If each question has a different value , how to maximize the mark?)
  - Selection of capital investments
- Finding the least wasteful way to cut raw materials.
- Computer games : Pick the most valuable set of items

from a treasure to carry.

```
// A Dynamic Programming based c++ solution for 0-1 Knapsack
#include<iostream>
using namespace std;
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];
}
```

```
// Driver
int main()
{
    int val[] = {2,3,4,5};
    int wt[] = {3,4,5,6};
    int  W = 5;
    int n = sizeof(val)/sizeof(val[0]);
    //cout<<n<<endl;
    cout<<"Max. Knapsac Value = "<<knapSack(W, wt, val, n);
    return 0;
}

//Max. Knapsac Value = 7
```

Conclusion

- Dynamic programming is a technique for making a sequence of interrelated decisions.
- “Programming”: A tabular method (not writing computer code)
- It may be used to solve dynamic optimization problems, where the problem can be split into overlapping subproblems
- It requires formulating an appropriate recursive relationship for each individual subproblem
- When the solution can be recursively described in terms of optimal partial solutions, we can store these partial solutions and re-use them as necessary(Memoization).
- Running times are usullly better than Brute Force and other naïve algorithms . Example:

0-1 Knapsack problem: $O(W*n)$ vs. $O(2^n)$