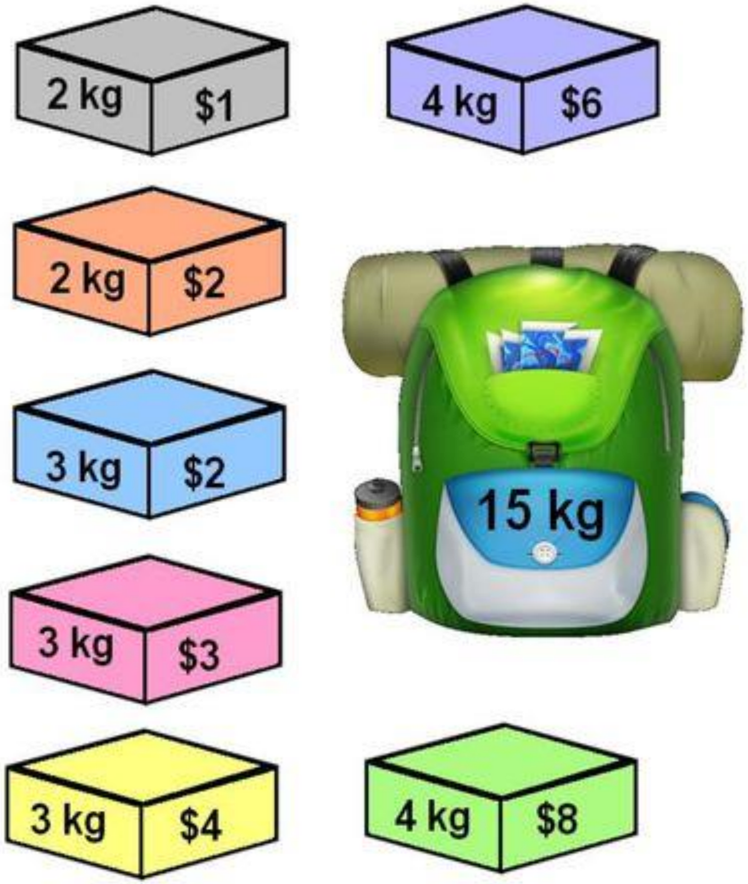
0-1 Knapsack problem

Knapsack Problem

You are a mischievous child camping in the woods. You would like to steal items from other campers, but you can only carry so much mass in your knapsack. You see seven items worth stealing. Each item has a certain mass and monetary value. How do you maximize your profit so you can buy more video games later?

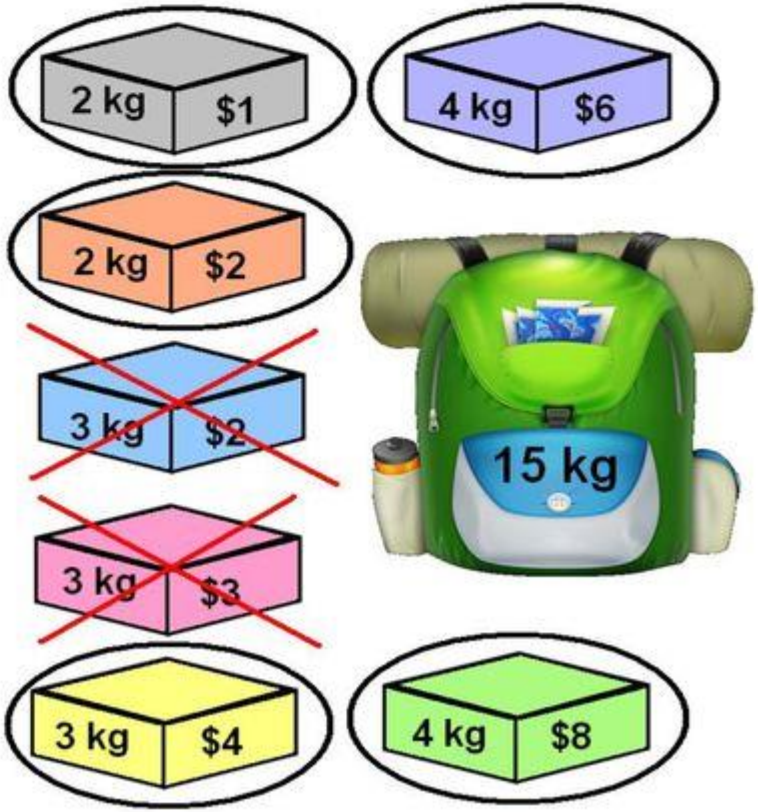
Knapsack Problem



- Value and mass of each item is given
- Maximize profit
- Subject to mass constraint of knapsack: 15 kg

- Being a smart kid, you apply dynamic programming.

Solution



Sensitivity Analysis

- Valuables are not very valuable when camping, pick richer people to steal from.

- Suddenly a valuable item of $15 that weighs 11kg is available, will the optimal solution change?

  Yes, the maximum profit becomes $23

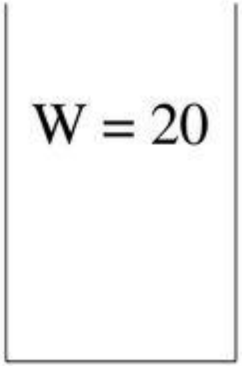
Knapsack problem

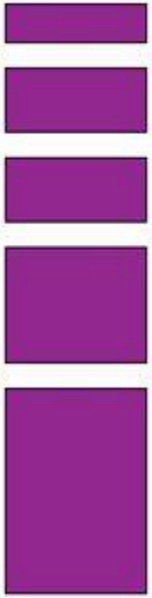
Given some items, pack the knapsack to get **the maximum total value**. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W. So we must consider weights of items as well as their values.

Item #(i)	Weight(w_i)	Value(b_i)
1	1	8
2	3	6
3	5	5

0-1 Knapsack problem

This is a knapsack
Max weight: W = 20





Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	8
	9	10

Knapsack problem

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 Knapsack problem

- Given a knapsack with maximum capacity ***W***, and a set *S* consisting of *n* items
- Each item *i* has some weight w_i and benefit value b_i (all w_i, b_i and W are integer values)

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

0-1 Knapsack problem

- Problem, in other words, is to find

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

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

Brute-force approach

Let's first solve this problem with a straightforward algorithm

- **Since there are *n* items, there are 2^n possible combinations of items.**
- **We go through all combinations and find the one with maximum value and with total weight less or equal to *W***
- **Running time will be $O(2^n)$**

Dynamic programming approach

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

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

- This is a reasonable subproblem definition.
- The question is:

  Can we describe the final solution (S_{K+1}) in terms of subproblems (S_k)?

- Unfortunately, we can't do that.

  Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

Knapsack 0-1 Problem

- Let's illustrate that point with an example:

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

So our definition of a subproblem is flawed and we need another one!

- **<u>The maximum weight the knapsack can hold is 20.</u>**

- The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$
- BUT the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$.
  - In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$.
    - (Instead it builds upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12 or less.)

Defining a Subproblem

- **Let's add another parameter: w, which will represent the maximum weight for each subset of items**

- **The subproblem then will be to compute *V[k,w], i.e.,* to find an optimal solution for *S_k = {items labeled 1, 2, .. k} in a knapsack of size w.***

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

Recursive Formula

$$V[i,w] = \begin{cases} V[i-1,w] & \text{if } w_i > w \\ \max\{V[i-1,w], V[i-1,w-w_i]+b_i\} & \text{else} \end{cases}$$

- The best subset of S_i that has the total weight $\leq$ *w,* either contains item *i* or not.
- **First case: w_i>*w***. Item *i* can't be part of the solution, since if it was, the total weight would be > *w*, which is unacceptable.
- **Second case:** $w_i \leq w$. Then the item *i* can be in the solution, and we choose *the case with greater value*.

0-1 Knapsack Algorithm

```
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 = 1 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
```

Running Time

What is the running time of this algorithm?

O(n*W)

```
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 = 1 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
```

$O(W)$

$O(n)$

Repeat n times

$O(W)$

Remember that the brute-force algorithm takes $O(2^n)$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example

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

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

Example

Items:

i	w_i	b_i
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

```
for i = 1 to n
        for w = 1 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
```

$\boxed{V[1,1] = V[0,1] = 0}$

Example

Items:

i	w_i b_i
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					

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

```
for i = 1 to n
    for w = 1 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
```

3+V[0,0] > V[0,2]
3+0 > 0
V[1,2] = 3

Example

Items:

i	w_i b_i
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					

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

```
for i = 1 to n
        for w = 1 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
```

$3+V[0,1] > V[0,3]$

$3+0 > 0$

$V[1,3] = 3$

Example

Items:

i	w_i b_i
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

for i = 1 to n

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

3+V[0,2] > V[0,4]

3+0 > 0

V[1,4] = 3

Example

Items:

i	w_i	b_i
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					

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=3$

```
for i = 1 to n
    for w = 1 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
```

$3+V[0,3] > V[0,5]$

$3+0 > 0$

$V[1,5] = 3$

Example

Items:

i	w_i	b_i
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					

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

```
for i = 1 to n
    for w = 1 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
```

$V[2,1] = V[1,1] = 0$

Example

Items:

i	w_i	b_i
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					

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

for i = 1 to n

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

$V[2,2] = V[1,2]$
$= 3$

Example

Items:

i	w_i	b_i
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					

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

```
for i = 1 to n
    for w = 1 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
```

4+V[1,0] > V[1,3]
4+0 > 3
V[2,3] = 4

Example

Items:

i	w_i	b_i
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					

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

```
for i = 1 to n
      for w = 1 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
```

$4+V[1,1] > V[1,4]$

$4+0 > 3$

$V[2,4] = 4$

Example

Items:

i	w_i b_i
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

```
for i = 1 to n
    for w = 1 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
```

4+V[1,2] > V[1,5]
 4+3 > 3
V[2,5] = 7

Example

Items:

i	w_i	b_i
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

for i = 1 to n

for w = 1 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

V[3,3] = V[2,3]

= 4

Example

Items:

i	w_i	b_i
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					

$i=3$

$b_i=5$

$w_i=4$

$w= 4$

$w- w_i=0$

for i = 1 to n

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

5+V[2,0] > V[2,4]

5+0 > 4

V[3,4] = 5

Example

Items:

i	w_i	b_i
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					

$i=3$

$b_i=5$

$w_i=4$

$w= 5$

$w- w_i=1$

for i = 1 to n

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

5+V[2,1] > V[2,5]

5+0 > 7

V[3,4] = 7

Example

Items:

i	w_i	b_i
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

```
for i = 1 to n
    for w = 1 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
```

V[4,4] = V[3,4]
= 5

Example

Items:

i	w_i	b_i
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

$i=4$

$b_i=6$

$w_i=5$

$w= 5$

$w- w_i=0$

for i = 1 to n

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

6+V[3,1] > V[3,5]

6+0 > 7

V[4,5] = 7

Exercise

1. a. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

, capacity $W = 6$.

How to find out which items are in the optimal subset?

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
  - i.e., the value in V[n,W]

To know the items that make this maximum value, an addition to this algorithm is necessary

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:

i	w_i	b_i
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
$V[i,k]$ =
$V[i-1,k]$ =

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
        $i = i-1$

Finding the Items

Items:

i	w_i	b_i
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] = V[4,5]=7

V[i–1,k]=V[3,5] =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

        $i = i-1$

Finding the Items

Items:

i	w_i b_i
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

$V[i,k]$ = V[3,5]=7

$V[i-1,k]$=V[2,5] =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

$i = i-1$

Finding the Items

Items:

i	w_i	b_i
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

$V[i,k]$ = V[2,5]=7

$V[i-1,k]$=V[1,5] =3

$k - w_i$= 5-3=2

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

    $i = i-1$

i =2-1 =1
k =5-3 =2

Finding the Items

Items:

i	w_i	b_i
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

$V[i,k]$ = V[2,5]=7

$V[i-1,k]$=V[1,5] =3

$k - w_i$= 5-3=2

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

        $i = i-1$

i =2-1 =1
k =5-3 =2

Finding the Items

Items:

i	w_i	b_i
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

$V[i,k]$ = V[1,2]=3

$V[i-1,k]$=V[0,2] =0

$k - w_i$=0

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

        $i = i-1$

I =1-1 =0
K =2-2 =0

Finding the Items

Items:

i	w_i	b_i
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] = V[1,2]=3

V[i−1,k]=V[0,2] =0

k − w_i=0

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
    $i = i-1$

I =1-1 =0
K =2-2 =0

Finding the Items

Items:

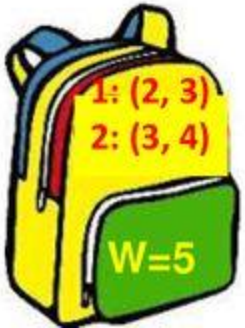
i	w_i	b_i
1:	(2, 3)	
2:	(3, 4)	
3:	(4, 5)	
4:	(5, 6)	

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}



i=n, k=W

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

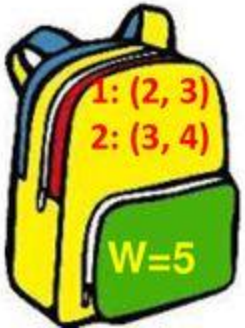
Items:

i	w_i b_i
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 {1, 2}

i=n, k=W
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
    $i = i-1$



Memorization (Memory Function Method)

- *Goal:*
    - *Solve only subproblems that are necessary and solve it only once*
- *Memorization* is another way to deal with overlapping subproblems in dynamic programming
- With memorization, we implement the algorithm ***recursively***:
    - If we encounter a new subproblem, we compute and store the solution.
    - If we encounter a subproblem we have seen, we look up the answer
- Most useful when the algorithm is easiest to implement recursively
    - Especially if we do not need solutions to all subproblems.

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be *recursively* described in terms of partial solutions, we can store these partial solutions and re-use them as necessary (memorization)
- Running time of dynamic programming algorithm vs. naïve algorithm:
  - 0-1 Knapsack problem: **$O(W*n)$** vs. **$O(2^n)$**