ITCS 6114

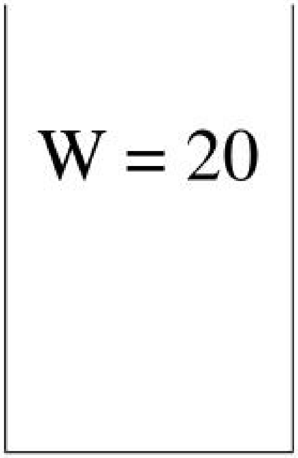
Dynamic programming
0-1 Knapsack problem

0-1 Knapsack problem

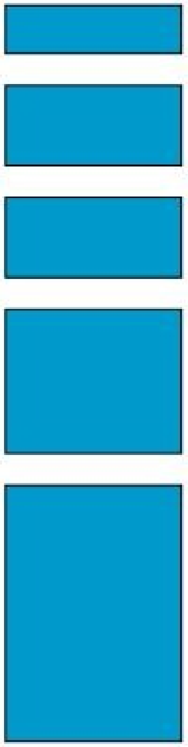
- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i , b_i and W are integer values)
- <u>Problem</u>: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem: a picture

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

0-1 Knapsack problem

- Problem, in other words, is to find

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

- The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected.
- Just another version of this problem is the "*Fractional Knapsack Problem*", where we can take fractions of items.

0-1 Knapsack problem: 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 the most total value and with total weight less or equal to W
- Running time will be $O(2^n)$

0-1 Knapsack problem: brute-force approach

- Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:

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

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 valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we <u>can't</u> do that. Explanation follows….

Defining a Subproblem

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

Max weight: W = 20

For S_4:

Total weight: 14;

total benefit: 20

$w_1 = 2$ $b_1 = 3$	w_2 $= 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 9$ $b_4 = 10$

For S_5:

Total weight: 20

total benefit: 26

S_5 S_4

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

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

Defining a Subproblem (continued)

- 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 flawed and we need another one!
- Let's add another parameter: *w*, which will represent the <u>exact</u> weight for each subset of items
- The subproblem then will be to compute $B[k,w]$

Recursive Formula for subproblems

- Recursive formula for subproblems:

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

- It 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 item k

Recursive Formula

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

- The best subset of S_k that has the total weight w, either contains item k or not.
- 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 the item k can be in the solution, and we choose the case with greater value

0-1 Knapsack Algorithm

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

Running time

```
for w = 0 to W
   B[0,w] = 0
for i = 0 to n
   B[i,0] = 0
   for w = 0 to W
      < the rest of the code >
```

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

$O(n*W)$

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 (2)

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

for w = 0 to W

$B[0,w] = 0$

Example (3)

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

for i = 0 to n

$B[i,0] = 0$

Example (4)

Items:

1: (2,3)

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

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0 →	0			
2	0				
3	0				
4	0				
5	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 + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (5)

Items:

1: (2,3)

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

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	**3**			
3	0				
4	0				
5	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 + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (6)

Items:

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

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	**3**			
4	0				
5	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 + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (7)

Items:

1: (2,3)

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

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	**3**			
5	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 $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example (8)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

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

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

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

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

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

else

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

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

Example (9)

Items:

1: (2,3)
2: (3,4)

3: (4,5)
4: (5,6)

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

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

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

Example (10)

Items:

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

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

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

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

Example (11)

Items:

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

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

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

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

Example (12)

Items:

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

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

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

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

Example (13)

Items:

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

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

$i=2$
$b_i=4$
$w_i=3$
$w=5$
$w\text{-}w_i=2$

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

Example (14)

Items:

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

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

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

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

Example (15)

Items:

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

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

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w- w_i=0$

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

Example (15)

Items:

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

4: (5,6)

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

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w- w_i=1$

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

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

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

  else

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

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

Example (16)

Items:

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

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

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

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

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

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

    else

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

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

Example (17)

Items:

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

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

$i=3$
$b_i=5$
$w_i=4$
$w=5$

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

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
- To know the items that make this maximum value, an addition to this algorithm is necessary
- Please see LCS algorithm from the previous lecture for the example how to extract this data from the table we built

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
- Running time (Dynamic Programming algorithm vs. naïve algorithm):
  - LCS: **$O(m*n)$** vs. **$O(n * 2^m)$**
  - 0-1 Knapsack problem: **$O(W*n)$** vs. **$O(2^n)$**