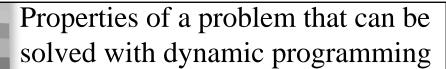
Dynamic programming
0-1 Knapsack problem

Review: Dynamic programming

- DP is a method for solving certain kind of problems
- DP can be applied when the solution of a problem includes solutions to subproblems
- We need to find a recursive formula for the solution
- We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory
- In the end we'll get the solution of the whole problem



- Simple Subproblems
 - We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the problems
 - The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
 - Optimal subproblems to unrelated problems can contain subproblems in common

3

0-1 Knapsack problem

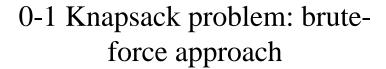
- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i , b_i and W are integer values)
- <u>Problem</u>: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem:						
	a picture Weight Benefit value					
	Items	$\mathbf{W_{i}}$	$\mathbf{b_{i}}$			
		2	3			
This is a knapsack		3	4			
Max weight: W = 20		4	5			
W = 20		5	8			
		9	10			
			5			

0-1 Knapsack problem

■ Problem, in other words, is to find $\max \sum_{i \in T} b_i$ 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.



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 $O(2^n)$

7

0-1 Knapsack problem: bruteforce 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 $S_k = \{items \ labeled \ 1, \ 2, ... \ k\}$



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....

9

Defining a Subproblem

$$\begin{vmatrix} w_1 = 2 \\ b_1 = 3 \end{vmatrix} \begin{vmatrix} w_2 = 4 \\ b_2 = 5 \end{vmatrix} \begin{vmatrix} w_3 = 5 \\ b_3 = 8 \end{vmatrix} \begin{vmatrix} w_4 = 3 \\ b_4 = 4 \end{vmatrix}$$

Max weight: W = 20

For S₄:

Total weight: 14; total benefit: 20

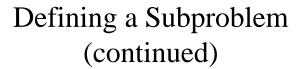
$w_1 = 2$	w ₂ =4	w ₃ =5	w ₄ =9
$b_1=3$	$b_2=5$	$b_3 = 8$	$b_4 = 10$

For S_5 :

Total weight: 20 total benefit: 26

		Item	Veight W _i	$\begin{array}{c} \text{Benefit} \\ b_i \end{array}$
		1 1	2	3
S_5	S_4	2	3	4
		3	4	5
		4	5	8
		5	9	10

Solution for S_4 is not part of the solution for $S_5!!!$ 10



- 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]

1

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

13

0-1 Knapsack Algorithm

Running time

for w = 0 to WO(W)

B[0,w] = 0

for i = 0 to n Repeat *n* times

B[i,0] = 0

for w = 0 to WO(W)

< the rest of the code >

What is the running time of this algorithm?

O(n*W)

Remember that the brute-force algorithm

takes O(2ⁿ)

Example

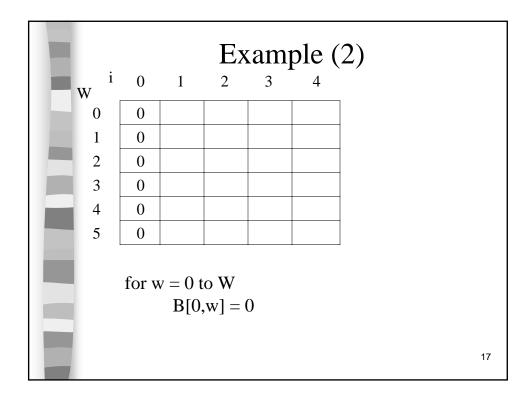
Let's run our algorithm on the following data:

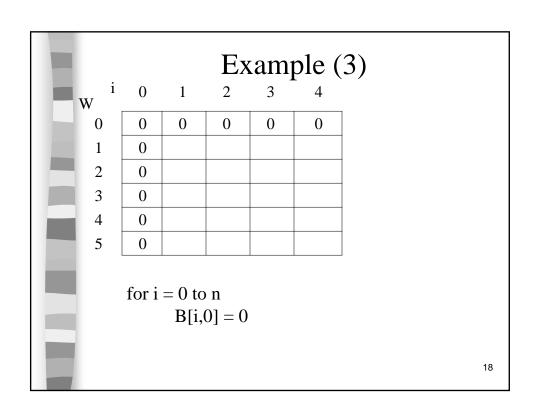
n = 4 (# of elements)

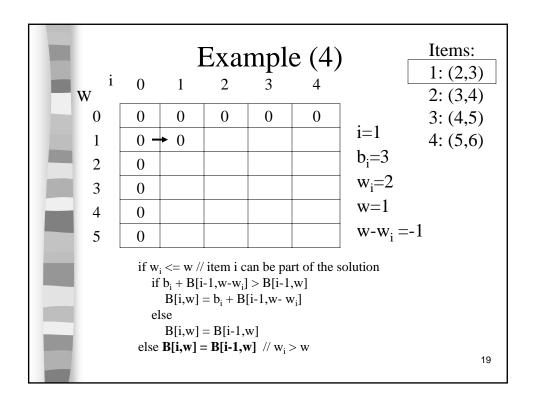
W = 5 (max weight)

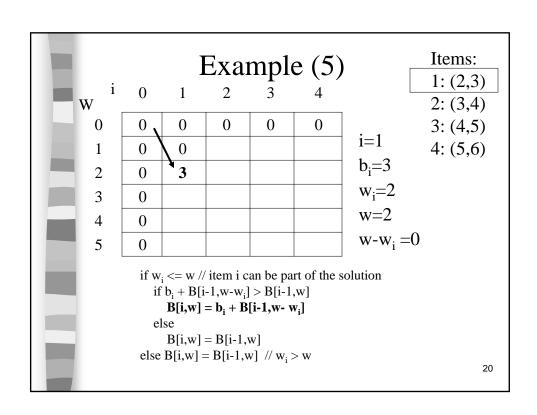
Elements (weight, benefit):

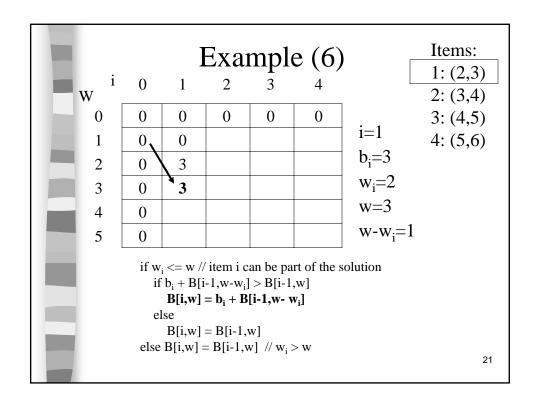
(2,3), (3,4), (4,5), (5,6)

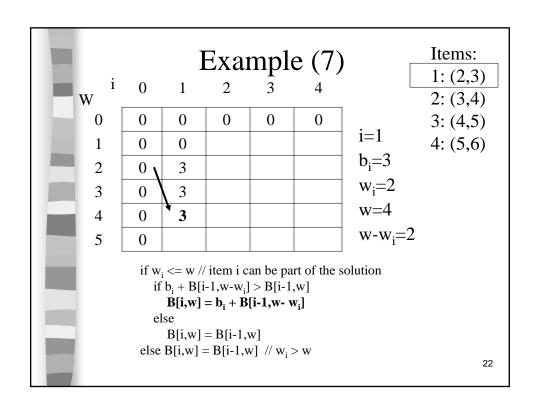


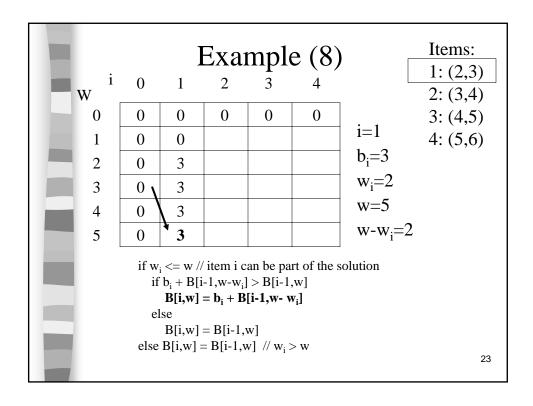


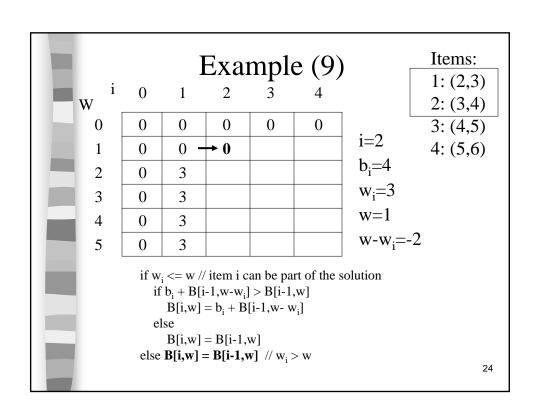


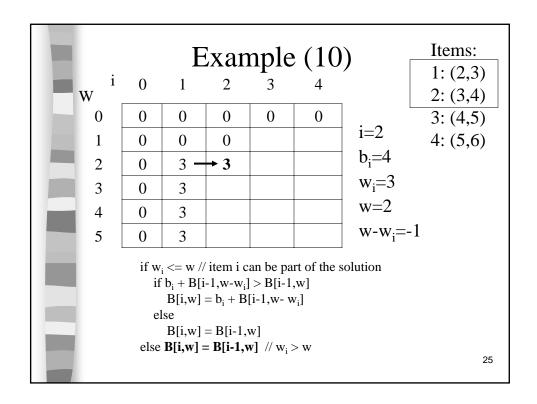


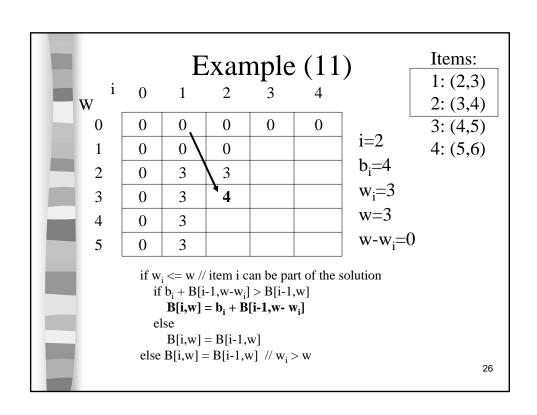


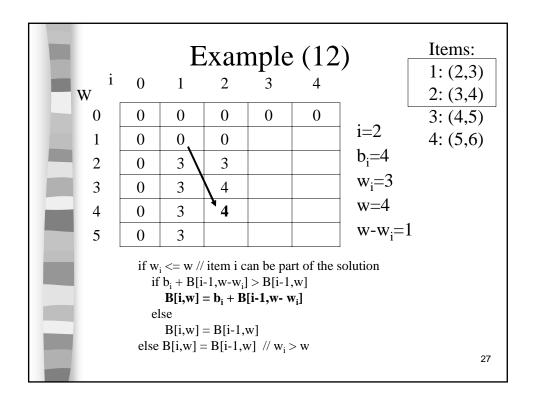


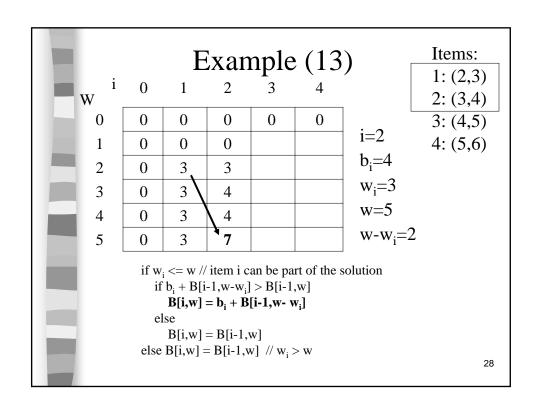


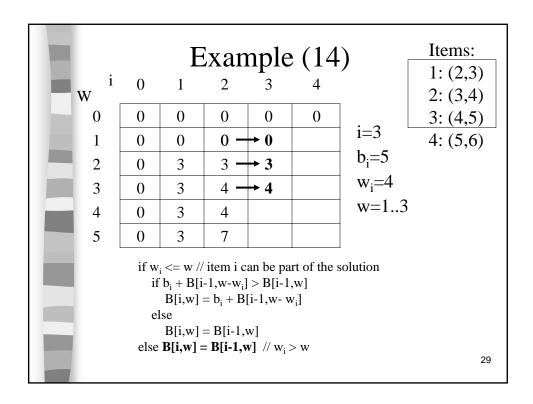


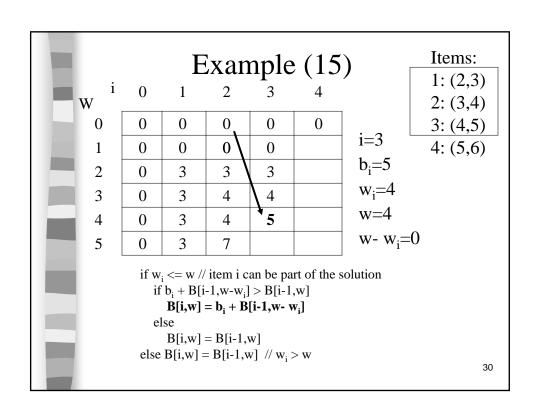


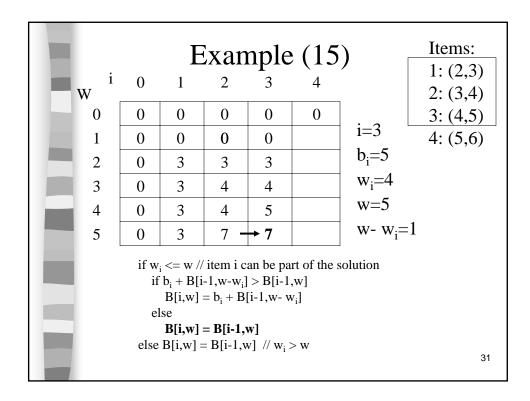


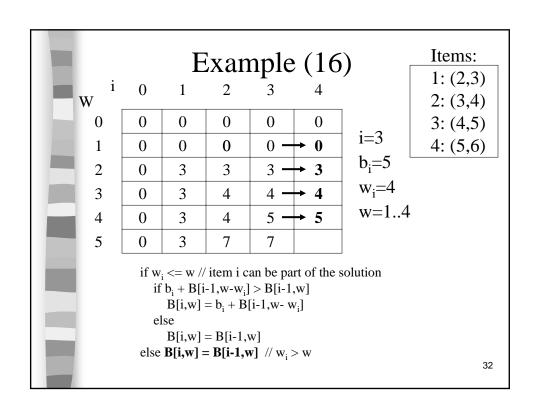


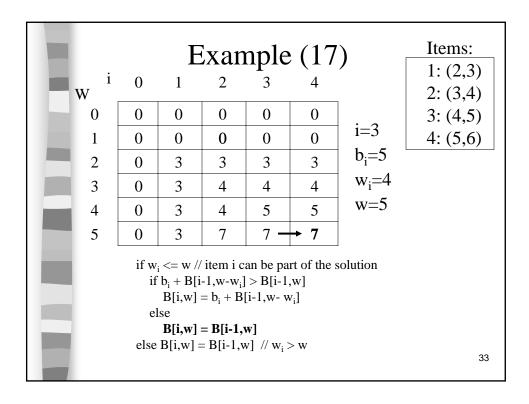












Comments

- 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(2ⁿ)**

