



CS 332 - Algorithms

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



Properties of a problem that can be solved with dynamic programming

- 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



Review: Longest Common Subsequence (LCS)

- Problem: how to find the longest pattern of characters that is common to two text strings X and Y
- Dynamic programming algorithm: solve subproblems until we get the final solution
- Subproblem: first find the LCS of *prefixes* of X and Y .
- this problem has *optimal substructure*: LCS of two prefixes is always a part of LCS of bigger strings

Review: Longest Common Subsequence (LCS) continued

- Define X_i , Y_j to be prefixes of X and Y of length i and j ; $m = |X|$, $n = |Y|$
- We store the length of $\text{LCS}(X_i, Y_j)$ in $c[i,j]$
- Trivial cases: $\text{LCS}(X_0, Y_j)$ and $\text{LCS}(X_i, Y_0)$ is empty (so $c[0,j] = c[i,0] = 0$)
- Recursive formula for $c[i,j]$:

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

$c[m,n]$ is the final solution



Review: Longest Common Subsequence (LCS)

- After we have filled the array $c[]$, we can use this data to find the characters that constitute the Longest Common Subsequence
- Algorithm runs in $O(m*n)$, which is *much* better than the brute-force algorithm: $O(n 2^m)$

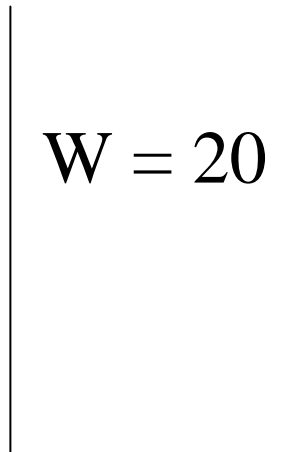







0-1 Knapsack problem

- 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: 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 can't do that.
Explanation follows....

Defining a Subproblem

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



Max weight: $W = 20$

For S_4 :

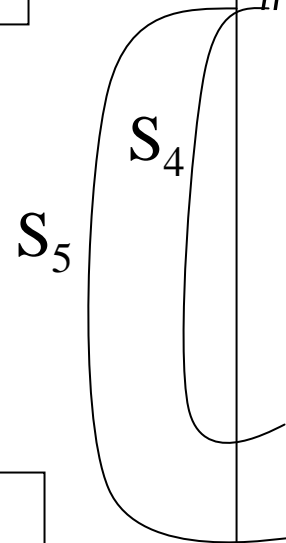
Total weight: 14;
total benefit: 20

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

For S_5 :

Total weight: 20
total benefit: 26

Item #	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 exact 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: $w_k \leq w$. 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 $w_i \leq w$ // 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

$O(W)$

$B[0,w] = 0$

for $i = 0$ to n

Repeat n times

$B[i,0] = 0$

for $w = 0$ to W

$O(W)$

< the rest of the code >

What is the running time of this algorithm?

$O(n*W)$

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



Example

Let's run our algorithm on the following data:

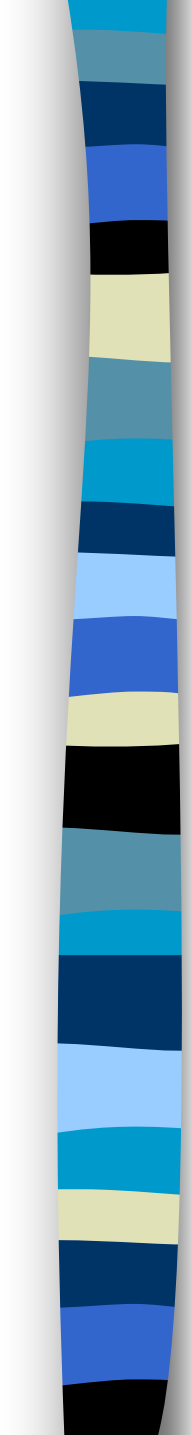
$n = 4$ (# of elements)

$W = 5$ (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)

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

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

if $w_i \leq w$ // 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)

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				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

if $w_i \leq w$ // 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)

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

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

if $w_i \leq w$ // 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)

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				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

if $w_i \leq w$ // 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)

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			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

if $w_i \leq w$ // 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)

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			

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i \leq w$ // 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)

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			

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

$$i=2$$

$$b_i=4$$

$$w_i=3$$

$$w=2$$

$$w-w_i=-1$$

if $w_i \leq w$ // 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)

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			

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i \leq w$ // 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)

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			

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i \leq w$ // 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)

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		

Items:

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

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

if $w_i \leq w$ // 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 (14)

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		

$i=3$

$b_i=5$

$w_i=4$

$w=1..3$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i \leq w$ // 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)

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		

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w - w_i=0$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i \leq w$ // 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)

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	

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w - w_i = 1$

if $w_i \leq w$ // 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 $w_i \leq w$ // 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)

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

$i=3$

$b_i=5$

$w_i=4$

$w=5$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i \leq w$ // 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. $O(2^n)$



The End

7/26/2007