

# **CSCE 310J**

## **Data Structures & Algorithms**

---

### **Dynamic programming**

### **0-1 Knapsack problem**

**Dr. Steve Goddard**  
***goddard@cse.unl.edu***

***<http://www.cse.unl.edu/~goddard/Courses/CSCE310J>***

# CSCE 310J

## Data Structures & Algorithms

---

- ◆ Giving credit where credit is due:
  - » Most of slides for this lecture are based on slides created by Dr. David Luebke, University of Virginia.
  - » Some slides are based on lecture notes created by Dr. Chuck Cusack, UNL.
  - » I have modified them and added new slides.

# Dynamic Programming

---

- ◆ Another strategy for designing algorithms is *dynamic programming*
  - » A metatechnique, not an algorithm (like divide & conquer)
  - » The word “programming” is historical and predates computer programming
- ◆ Use when problem breaks down into recurring small subproblems

# Dynamic programming

---

- ◆ It is used when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*).
- ◆ Algorithm finds solutions to subproblems and stores them in memory for later use.
- ◆ More efficient than “*brute-force methods*”, which solve the same subproblems over and over again.

# Summarizing the Concept of Dynamic Programming

---

## ◆ Basic idea:

- » Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
- » Overlapping subproblems: few subproblems in total, many recurring instances of each
- » Solve bottom-up, building a table of solved subproblems that are used to solve larger ones

## ◆ Variations:

- » “Table” could be 3-dimensional, triangular, a tree, etc.

## **Knapsack problem (Review)**

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 value.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

# Knapsack problem

---

There are two versions of the problem:

1. “0-1 knapsack problem” and
  2. “Fractional knapsack problem”
- 
1. Items are indivisible; you either take an item or not. Solved with *dynamic programming*
  2. Items are divisible: you can take any fraction of an item. Solved with a *greedy algorithm*.
    - ❖ We have already seen this version

## 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:  $W = 20$

$W = 20$

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.
- ◆ In the “*Fractional Knapsack Problem*,” 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 maximum 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

$S_k = \{\text{items labeled } 1, 2, \dots, k\}$

## 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 reasonable 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.

## 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;

Maximum benefit: 20

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

**For  $S_5$ :**

Total weight: 20

Maximum benefit: 26

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

$S_4$   
 $S_5$

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:

- 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 = 1 to n
    B[i,0] = 0
for i = 1 to n
    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  $i = 1$  to  $n$

$B[i,0] = 0$

for  $i = 1$  to  $n$

Repeat  $n$  times

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)

---

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

for  $w = 0$  to  $W$

$B[0,w] = 0$

## Example (3)

---

$i \backslash W$	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  $n$   
     $B[i,0] = 0$

## Example (4)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i \backslash W$	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  $w_i \leq w$  // 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)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i \backslash W$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	<b>3</b>			
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

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)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i \backslash W$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	<b>3</b>		
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

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)

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	<b>3</b>	
2	0					
3	0					
4	0					

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)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i \backslash W$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	<b>3</b>
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=3$

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)

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  $w_i \leq w$  // 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)

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	<b>3</b>			
3	0					
4	0					

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)

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	<b>4</b>		
3	0					
4	0					

i=2

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

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 (12)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i \backslash W$	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	<b>4</b>	
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

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 (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	<b>7</b>
3	0					
4	0					

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)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i \backslash W$	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					

$i=3$

$b_i=5$

$w_i=4$

$w=11.3$

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)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i \backslash W$	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	<b>5</b>	
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w - w_i = 0$

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 (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  $w_i \leq w$  // 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)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i \backslash W$	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  $w_i \leq w$  // 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 (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  $w_i \leq w$  // 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
  - » I.e., the value in  $B[n,W]$
- ◆ 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.
- ◆  $B[n, W]$  is the maximal value of items that can be placed in the Knapsack.
- ◆ Let  $i=n$  and  $k=W$ 
  - if  $B[i, k] \neq B[i-1, k]$  then
    - mark the  $i^{\text{th}}$  item as in the knapsack
    - $i = i-1, k = k-w_i$
  - else
    - $i = i-1$  // Assume the  $i^{\text{th}}$  item is not 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)

$i \backslash W$	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

$i=4$

$k=5$

$b_i=6$

$w_i=5$

$B[i,k] = 7$

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

$i=n, k=W$

while  $i, k > 0$

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

mark the  $i^{\text{th}}$  item as in the knapsack

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

else

$i = i-1$



## Finding the Items (2)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i \backslash W$	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

$i=4$

$k=5$

$b_i=6$

$w_i=5$

$B[i,k] = 7$

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

$i=n, k=W$

while  $i, k > 0$

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

mark the  $i^{\text{th}}$  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)

$i \backslash W$	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

$i=3$

$k=5$

$b_i=6$

$w_i=4$

$B[i,k] = 7$

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

$i=n, k=W$

while  $i, k > 0$

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

mark the  $i^{\text{th}}$  item as in the knapsack

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

else

$i = i-1$

## Finding the Items (4)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i \backslash W$	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

$i=2$

$k=5$

$b_i=4$

$w_i=3$

$B[i,k] = 7$

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

$k - w_i = 2$

$i=n, k=W$

while  $i, k > 0$

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

mark the  $i^{\text{th}}$  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)

$i \backslash W$	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

$i=1$

$k=2$

$b_i=3$

$w_i=2$

$B[i,k] = 3$

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

$k - w_i = 0$

$i=n, k=W$

while  $i, k > 0$

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

mark the  $i^{\text{th}}$  item as in the knapsack

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

else

$i = i-1$

## Finding the Items (6)

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i \backslash W$	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

$i=0$   
 $k=0$

$i=n, k=W$

while  $i, k > 0$

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

mark the  $n^{\text{th}}$  item as in the knapsack

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

else

$i = i-1$

The optimal  
knapsack  
should contain  
{1, 2}

## Finding the Items (7)

$i \backslash W$	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

$i=n, k=W$

while  $i, k > 0$

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

mark the  $n^{\text{th}}$  item as in the knapsack

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

else

$i = i-1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

The optimal  
knapsack  
should contain  
{1, 2}

## Review: The Knapsack Problem And Optimal Substructure

---

- ◆ Both variations exhibit optimal substructure
- ◆ To show this for the 0-1 problem, consider the most valuable load weighing at most  $W$  pounds
  - » *If we remove item  $j$  from the load, what do we know about the remaining load?*
  - » A: remainder must be the most valuable load weighing at most  $W - w_j$  that thief could take, excluding item  $j$

# Solving The Knapsack Problem

---

- ◆ The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - » *Do you recall how?*
  - » Greedy strategy: take in order of dollars/pound
- ◆ The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
  - » Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds
    - ❖ *Suppose item 2 is worth \$100. Assign values to the other items so that the greedy strategy will fail*