- Bellman explains the reasoning behind the term dynamic programming in his autobiography, Eye of the Hurricane: An Autobiography (1984):

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical.

The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible.

Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

The Knapsack Problem

https://cs.pomona.edu/classes/cs140/

Outline

Topics and Learning Objectives

- Discuss the dynamic programming paradigm
- Solve the 0-1 Knapsack Problem

Assessments

- Knapsack Example

The Knapsack Problem

Input:

- A capacity W (a nonnegative integer) and
- n items, where each item has:
  - A value v_i, and (must be nonnegative values)
  - A size/weight w_i (must be nonnegative values and integral)

Output: a subset of the items called S

Where S maximizes this equation:

$$\sum_{i \in S} v_i$$

Subject to the constraint:

$$\sum_{i \in S} w_i \leq W$$

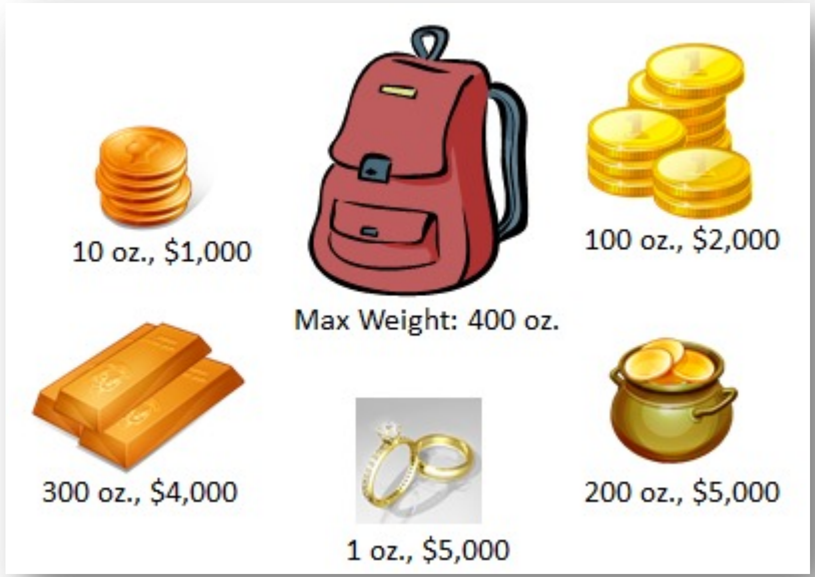
Knapsack Applications

Budgeting a resource

- Given a finite amount of time, schedule as much prioritized work as possible
- Different from the greedy scheduling problem
- Some tasks might not be completed

Burglarizing

- Steal as much value as possible
- Objects have different sizes and different values



(Image from Rensselaer)

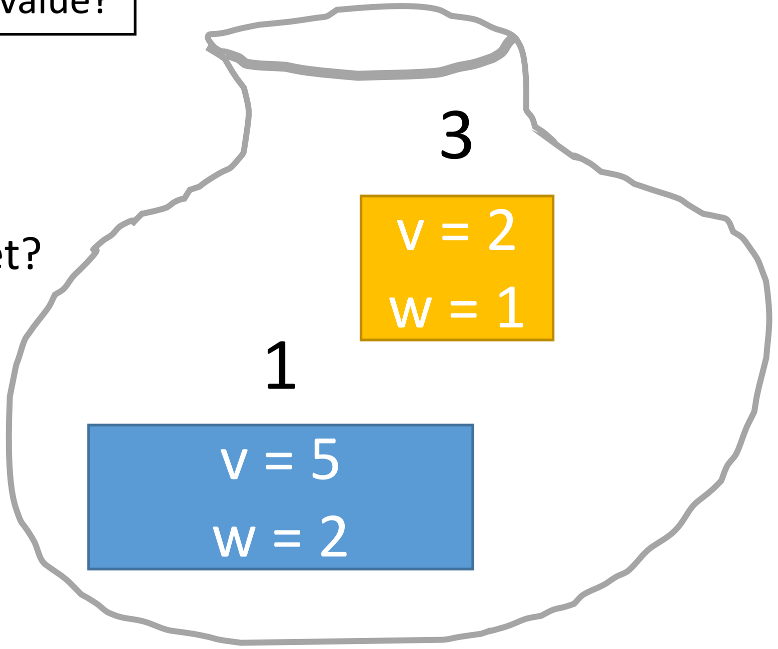
Knapsack Example, Does greedy work?

Item #s:	1	2	3	4

	v = 5
	w = 2
		v = 3
		w = 2
			v = 2
			w = 1
				v = 6
				w = 3

1. Greedy by highest value?

What is the optimal set?



S if W = 3?

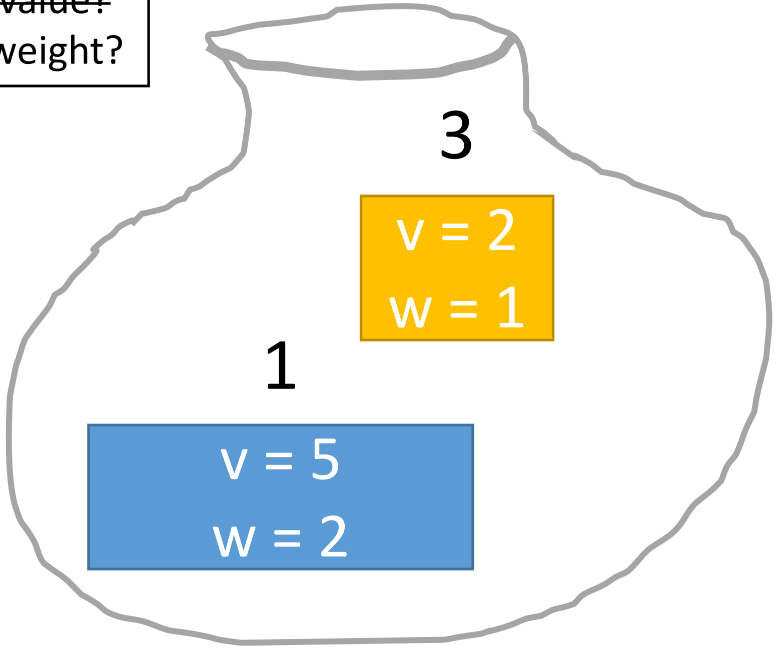
Knapsack Example, Does greedy work?

Item #s:

1	2	3	4

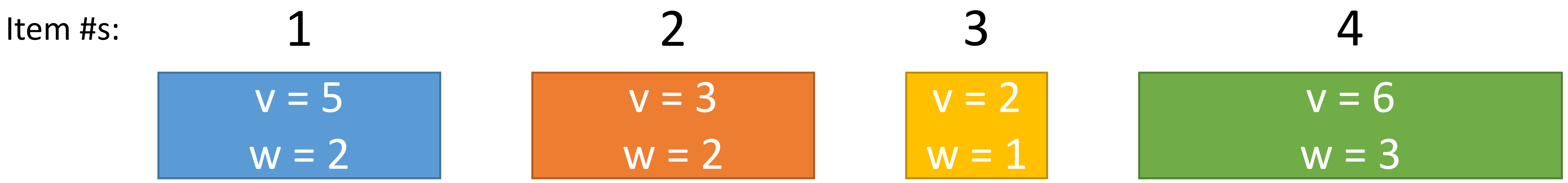
v = 5
w = 2
	v = 3
	w = 2
		v = 2
		w = 1
			v = 6
			w = 3

1. ~~Greedy by highest value?~~
2. Greedy by lowest weight?

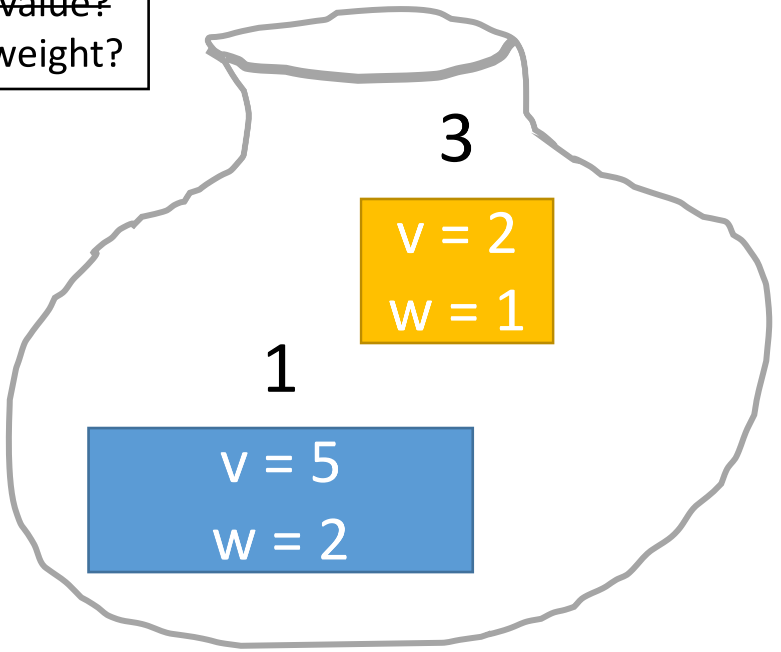


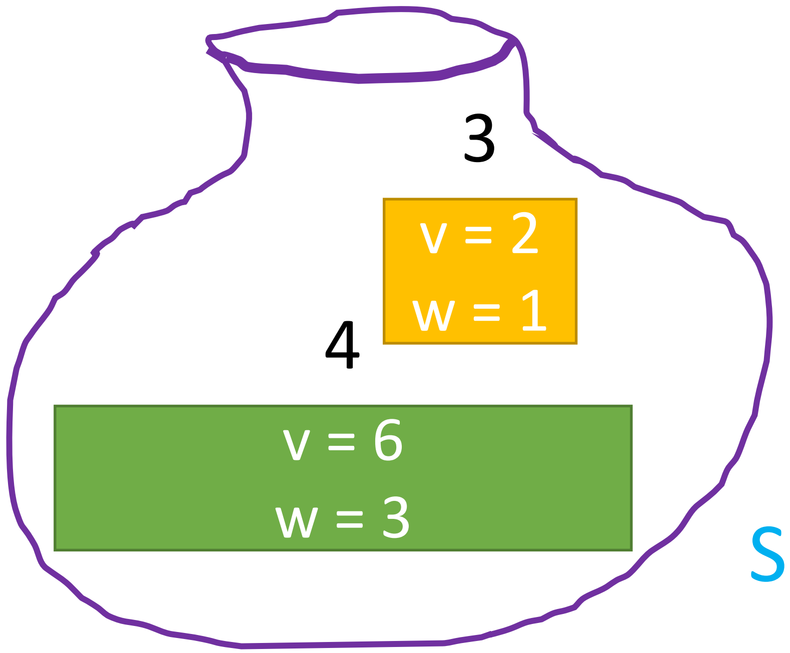
S if W = 3?

Knapsack Example, Does greedy work?



1. ~~Greedy by highest value?~~
2. Greedy by lowest weight?





S if W = 3?

S if W = 4?

This works if you are allowed to take fractions of items (Fractional Knapsack).

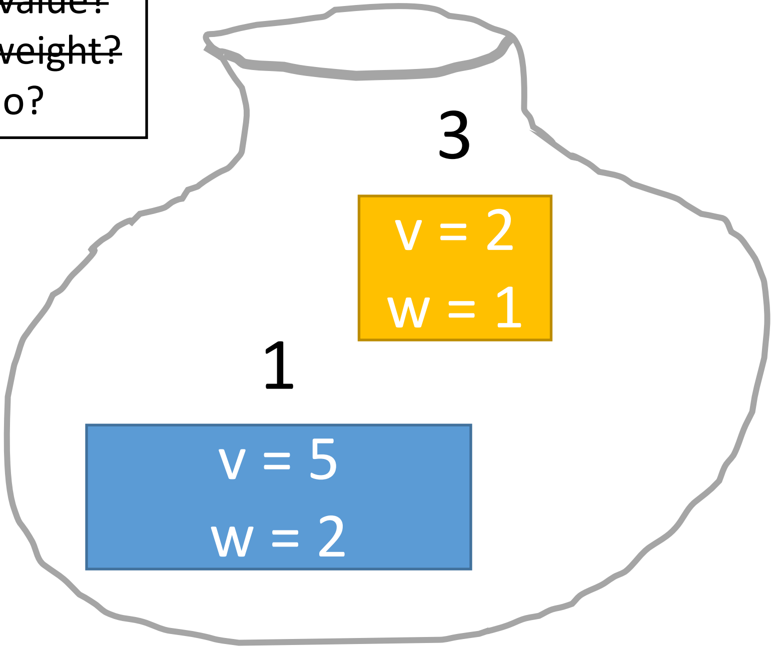
Knapsack Example, Does greedy work?

Item #s:	1	2	3	4

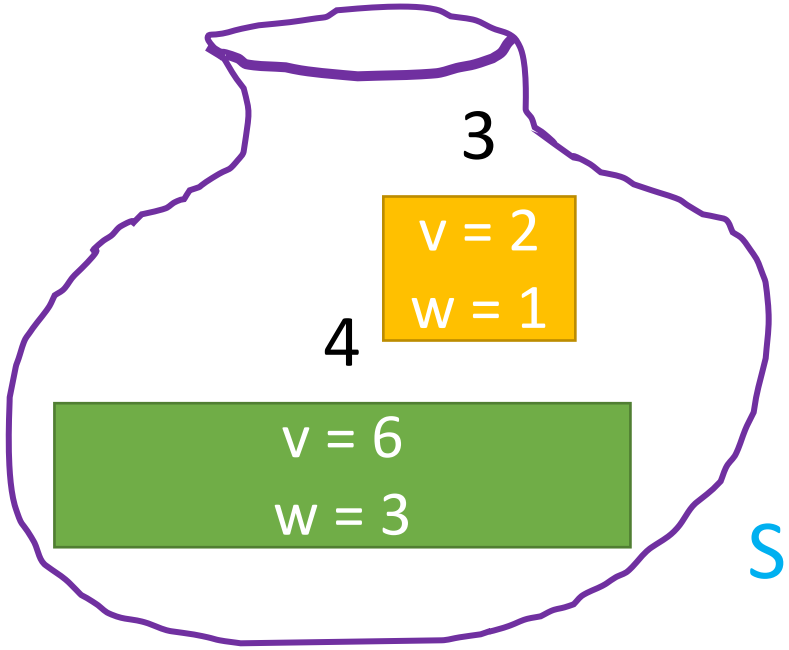
	v = 5
	w = 2
		v = 3
		w = 2
			v = 2
			w = 1
				v = 6
				w = 3

	5/2 = 2.5	3/2 = 1.5	2/1 = 2	6/3 = 2

1. ~~Greedy by highest value?~~
2. ~~Greedy by lowest weight?~~
3. Greedy by best ratio?



S if W = 3?



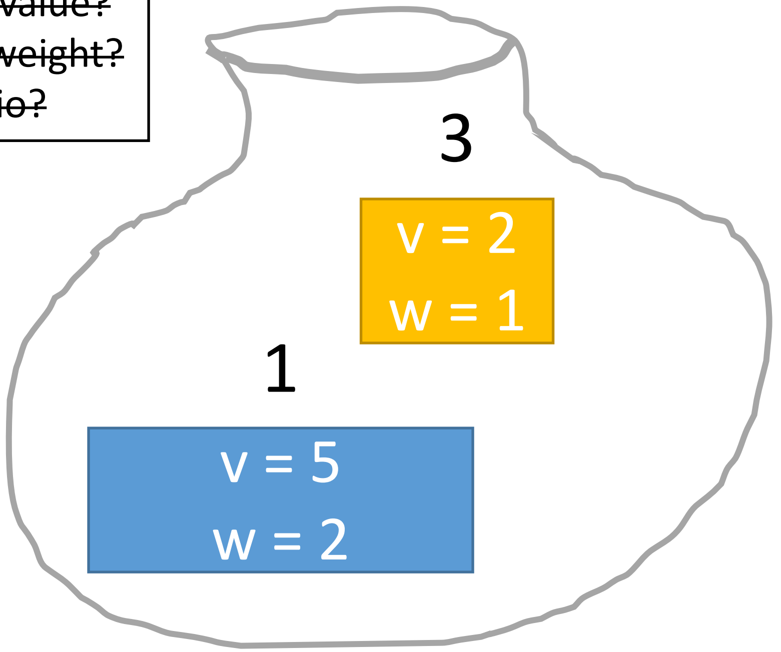
S if W = 4?

Knapsack Example, Does greedy work?

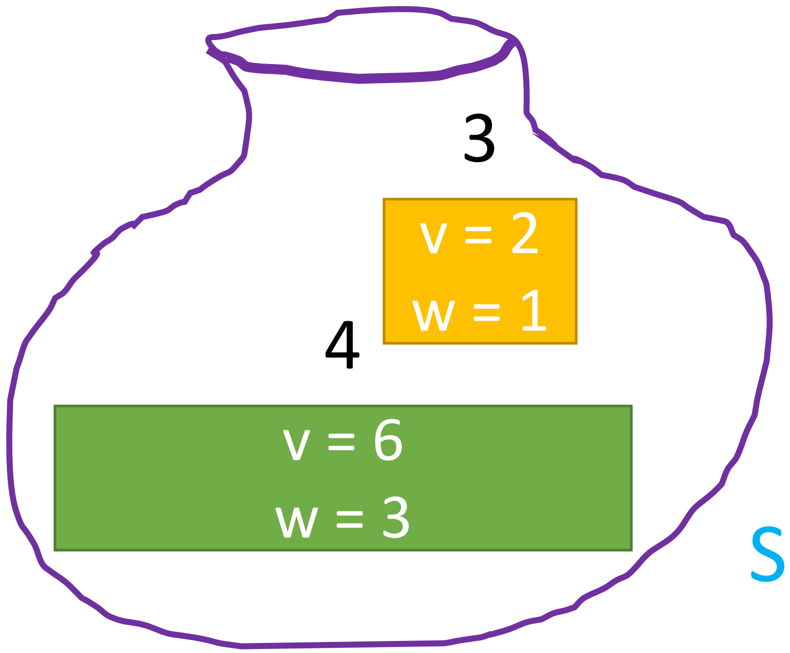
Item #s:

1	2	3	4
v = 5, w = 2	v = 3, w = 2	v = 2, w = 1	v = 6, w = 3
5/2 = 2.5	3/2 = 1.5	2/1 = 2	6/3 = 2

1. ~~Greedy by highest value?~~
2. ~~Greedy by lowest weight?~~
3. ~~Greedy by best ratio?~~



S if W = 3?



S if W = 4?

Dynamic Programming and Knapsacks

- Step 1: formulate a recurrence relationship based on the structure of the optimal solution.
- Another way to say this: look at the optimal solution as if it were a function of solutions to smaller problems.

- Let S be our optimal solution to the Knapsack Problem
- S doesn’t have any particular ordering but let's assume it does.
- Let’s label each of the n items and give them a sequence id: 1 … n

Dynamic Programming and Knapsacks

Case 1: suppose that item n is **not** in the optimal set S

- How does S relate to the first "n – 1" items?
- How do these first "n – 1" items relate to W?
- The set S must be optimal with respect to
  - Using only the first n – 1 items (imagine item n never existed) and
  - with respect to W.

Dynamic Programming and Knapsacks

Case 2: suppose that item n **is** in S

What can we say about S – {n}? Let's call this S'

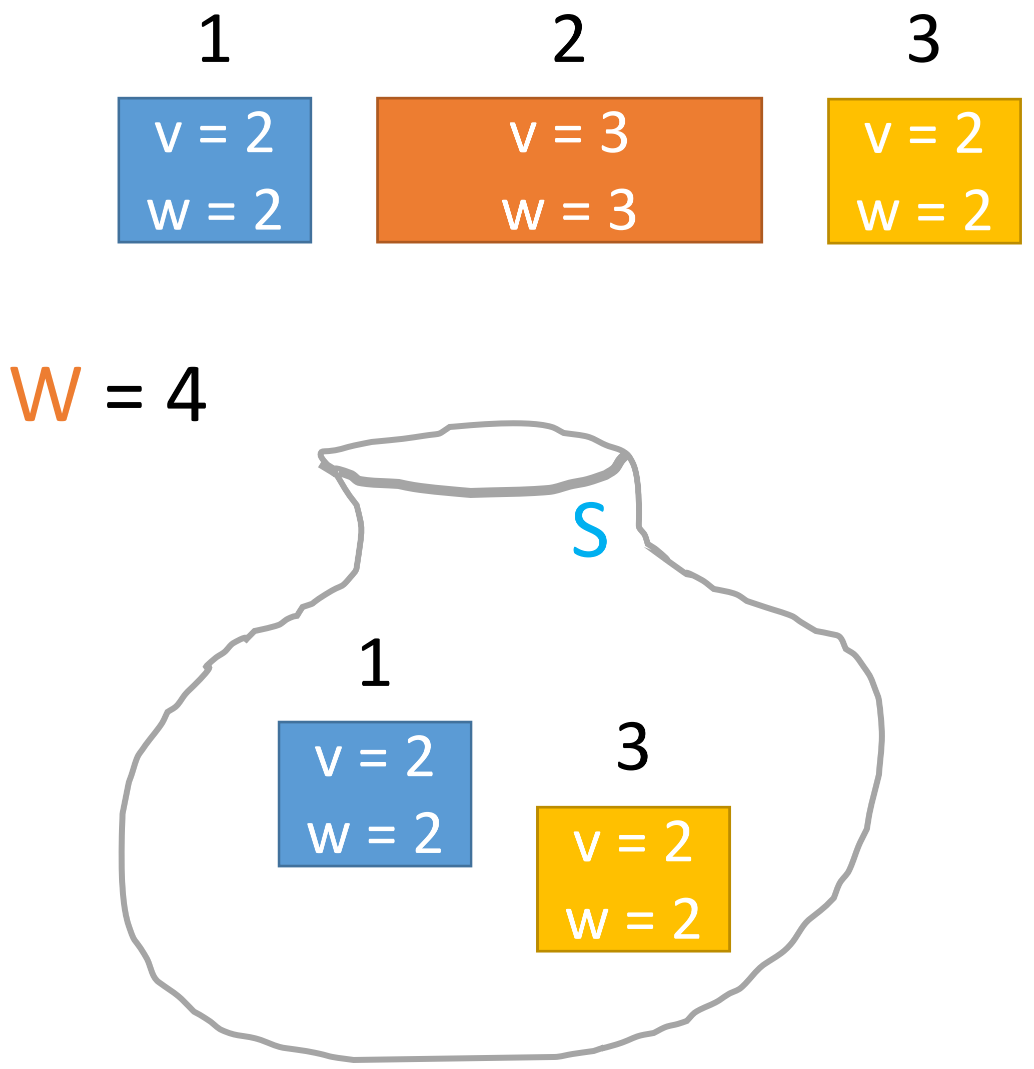
a. S' must be optimal with respect to the first "n – 1" items and W.

b. S' must be optimal with respect to the first "n – 1" items and W - w_n.

c. S' must be optimal with respect to the first "n – 1" items and W - v_n.

d. S' might not be feasible for W - w_n.

Case 2: suppose that item n is in S



What can we say about S – {n}?

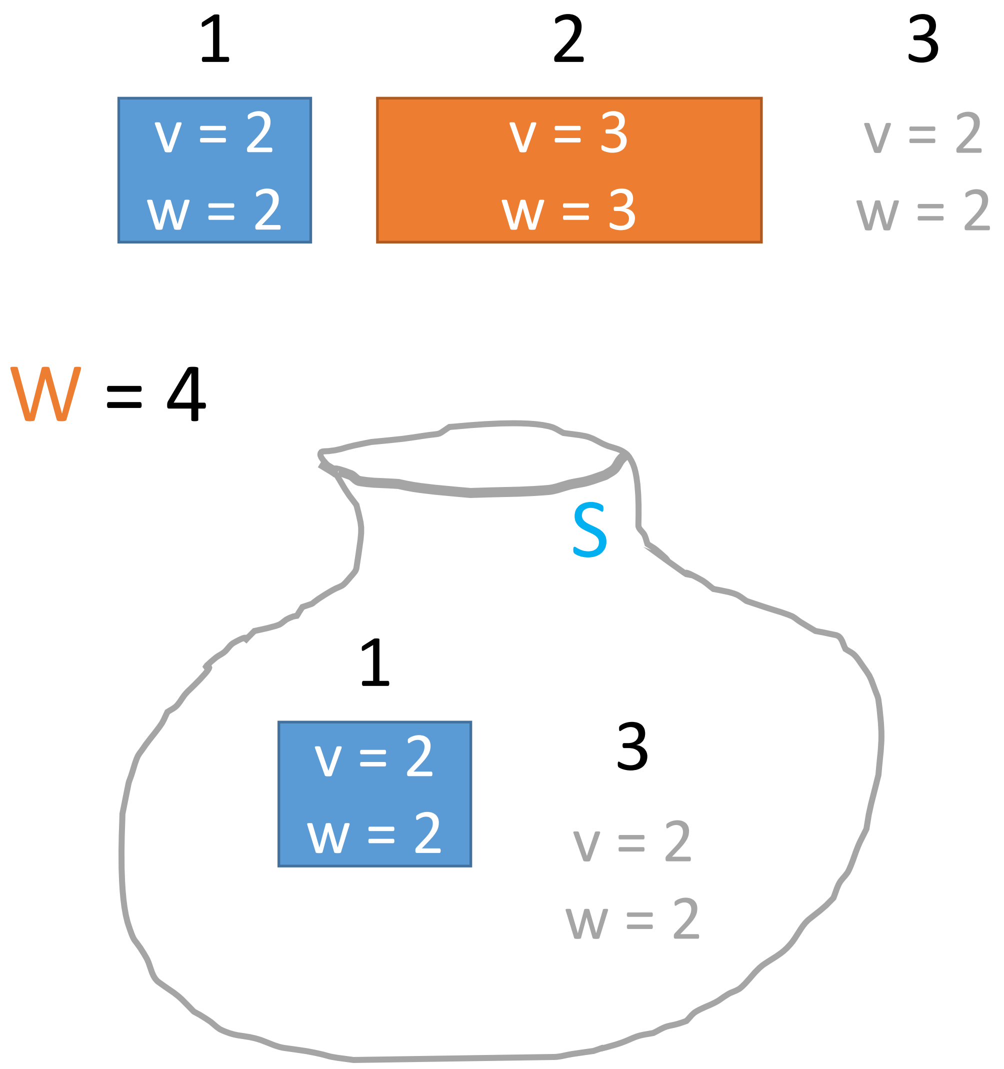
a. S' must be optimal with respect to the first "n – 1" and W.

b. S' must be optimal with respect to the first "n – 1" and W - w_n.

c. S' must be optimal with respect to the first "n – 1" and W - v_n.

d. S' might not be feasible for W - w_n.

Case 2: suppose that item n is in S



What can we say about S – {n}?

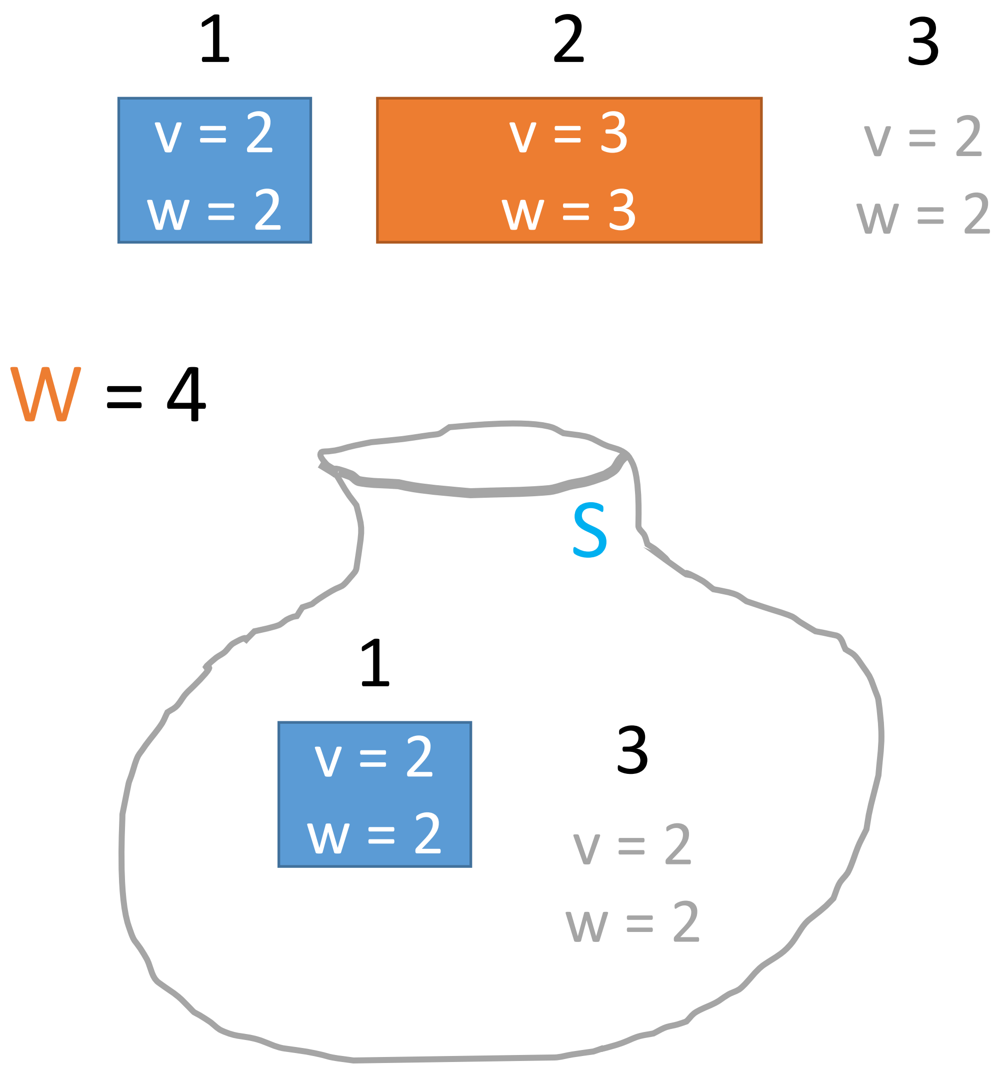
a. S’ must be optimal with respect to the first “n – 1” and W.

b. S’ must be optimal with respect to the first “n – 1” and W - w_n.

c. S’ must be optimal with respect to the first “n – 1” and W - v_n.

d. S’ might not be feasible for W - w_n.

Case 2: suppose that item n is in S



What can we say about S – {n}?

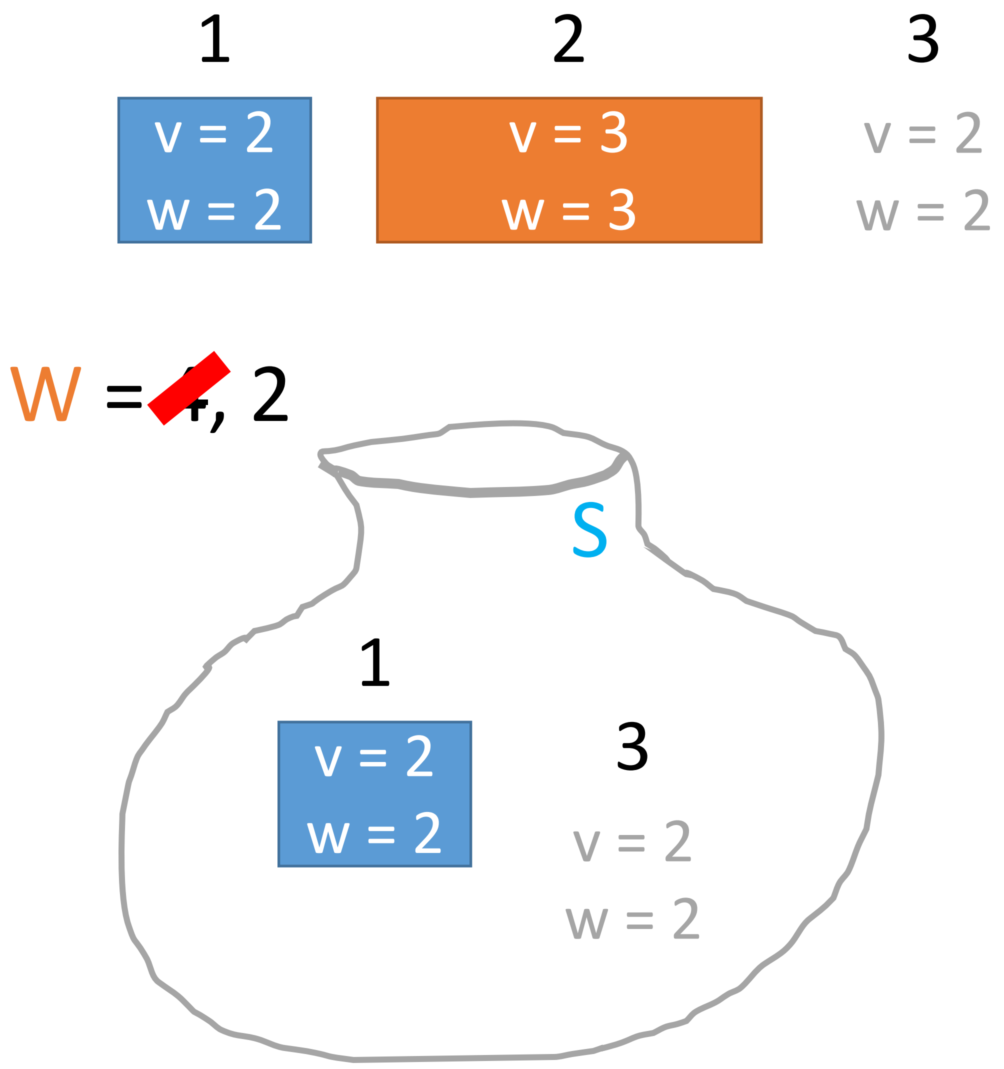
a. ~~S' must be optimal with respect to the first "n – 1" and W.~~

b. S' must be optimal with respect to the first "n – 1" and W - w_n.

c. S' must be optimal with respect to the first "n – 1" and W - v_n.

d. S' might not be feasible for W - w_n.

<u>Case 2</u>: suppose that item n is in S



What can we say about $S - \{n\}$?

a. S' must be optimal with respect to the first "$n - 1$" and W.

[✓ b. S' must be optimal with respect to the first "$n - 1$" and $W - w_n$.]

c. ~~S' must be optimal with respect to the first "$n - 1$" and $W - v_n$.~~

d. ~~S' might not be feasible for $W - w_n$.~~

Dynamic Programming and Knapsacks

Case 2: suppose that item n is in S

- What can we say about S – {n}?
- It must be optimal with respect to the first "n – 1" and W - w_n.

- Otherwise, there must exist some S* that is better than S – {n } for the first "n – 1" items.
- If S* is better for the first "n – 1", then it must be better than S + {n} when you add some arbitrary item that is not in S.
- This is a contradiction since we stated that S is the optimal solution

Examining the Cases to Create Subproblems

Case 1: suppose that item n is not in S

- → S must be optimal for the first "$n - 1$" items with respect to W.

Case 2: suppose that item n is in S

- → $S - \{n\}$ must be optimal with respect to the first "$n - 1$" and $W - w_n$.

Let $V_{i,x}$ be the value of the best solution such that:

- It only considers the first i items.
- It has a total size/weight $\leq x$.

$$V_{i,x} = \max(V_{(i-1),x} \text{ and } v_i + V_{(i-1),(x-wi)})$$

That was just part of step 1

As a reminder:

Step 1: formulate a recurrence relationship based on the structure of the optimal solution. And identify the subproblems.

What are our subproblems?

- All possible prefixes for the items: 1, 2, …, i
- All possible remaining capacities x can be: 0, 1, …, W

Both integer values only.

What’s next?

- Step 2: Quickly and correctly solve larger subproblems when provided solutions to the smaller subproblems.
- We’ll use a **bottom-up / tabular** approach (not top-down / memorized)

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]                Case 1
            withoutItem = table[i - 1][cap]              Case 2
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)
```

Dynamic Programming for Knapsacks

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)
```

Handle the edge case

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)
```

Exercise

Capacity Available						
	6					
	5					
	4					
	3					
	2					
	1					
	0					
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)
```

What do we return?

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

Capacity Available	6	0				
	5	0				
	4	0				
	3	0				
	2	0				
	1	0				
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

First Column

Capacity Available	6	0				
	5	0				
	4	0				
	3	0				
	2	0				
	1	0				
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

First Column

Capacity Available	6	0	3			
	5	0	3			
	4	0	3			
	3	0	0			
	2	0	0			
	1	0	0			
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

Second Column

Capacity Available	6	0	3			
	5	0	3			
	4	0	3			
	3	0	0			
	2	0	0			
	1	0	0			
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

Second Column

Capacity Available	6	0	3	3		
	5	0	3	3		
	4	0	3	3		
	3	0	0	2		
	2	0	0	0		
	1	0	0	0		
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

Third Column

Capacity Available	6	0	3	3		
	5	0	3	3		
	4	0	3	3		
	3	0	0	2		
	2	0	0	0		
	1	0	0	0		
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

Third Column

Capacity Available	6	0	3	3	7	
	5	0	3	3	6	
	4	0	3	3	4	
	3	0	0	2	4	
	2	0	0	0	4	
	1	0	0	0	0	
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
   table = [[0] * (capacity + 1)] * (n + 1)
   FOR i IN [1 ..= n]
      v = items[i].value
      w = items[i].weight
      FOR cap IN [0 ..= capacity]
         withoutItem = table[i - 1][cap]
         withItem = table[i - 1][cap - w] + v
         table[i, cap] = max(withoutItem, withItem)

   RETURN table[n][capacity]
```

Fourth Column

Capacity Available	6	0	3	3	7	
	5	0	3	3	6	
	4	0	3	3	4	
	3	0	0	2	4	
	2	0	0	0	4	
	1	0	0	0	0	
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

Fourth Column

Capacity Available						
	6	0	3	3	7	8
	5	0	3	3	6	8
	4	0	3	3	4	4
	3	0	0	2	4	4
	2	0	0	0	4	4
	1	0	0	0	0	0
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

Capacity Available	6	0	3	3	7	8
	5	0	3	3	6	8
	4	0	3	3	4	4
	3	0	0	2	4	4
	2	0	0	0	4	4
	1	0	0	0	0	0
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

Where is the answer?

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

Where is the answer?

Capacity Available	6	0	3	3	7	8
	5	0	3	3	6	8
	4	0	3	3	4	4
	3	0	0	2	4	4
	2	0	0	0	4	4
	1	0	0	0	0	0
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

What do we take?

Capacity Available	6	0	3	3	7	8
	5	0	3	3	6	8
	4	0	3	3	4	4
	3	0	0	2	4	4
	2	0	0	0	4	4
	1	0	0	0	0	0
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapsackReconstruct(items, capacity, table)
    S = {}
    cap = capacity
    FOR i IN [n ..= 1]
        v = items[i].value
        w = items[i].weight
        withoutItem = table[i - 1][cap]
        withItem = table[i - 1][cap - w] + v
        IF w ≤ cap && withItem ≥ withoutItem
            S = S + i
            cap = cap - w
    RETURN S
```

S

What do we take?

Capacity Available						
	6	0	3	3	7	8
	5	0	3	3	6	8
	4	0	3	3	4	4
	3	0	0	2	4	4
	2	0	0	0	4	4
	1	0	0	0	0	0
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapsackReconstruct(items, capacity, table)
    S = {}
    cap = capacity
    FOR i IN [n ..= 1]
        v = items[i].value
        w = items[i].weight
        withoutItem = table[i - 1][cap]
        withItem = table[i - 1][cap - w] + v
        IF w ≤ cap && withItem ≥ withoutItem
            S = S + i
            cap = cap - w
    RETURN S
```

S
4

What do we take?

Capacity Available						
	6	0	3	3	7	8
	5	0	3	3	6	8
	4	0	3	3	4	4
	3	0	0	2	4	4
	2	0	0	0	4	4
	1	0	0	0	0	0
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

n = 4, W = 6	v	w
Item 1	3	4
Item 2	2	3
Item 3	4	2
Item 4	4	3

```
FUNCTION KnapsackReconstruct(items, capacity, table)
    S = {}
    cap = capacity
    FOR i IN [n ..= 1]
        v = items[i].value
        w = items[i].weight
        withoutItem = table[i - 1][cap]
        withItem = table[i - 1][cap - w] + v
        IF w ≤ cap && withItem ≥ withoutItem
            S = S + i
            cap = cap - w
    RETURN S
```

What do we take?

Capacity Available						
	6	0	3	3	7	8
	5	0	3	3	6	8
	4	0	3	3	4	4
	3	0	0	2	4	4
	2	0	0	0	4	4
	1	0	0	0	0	0
	0	0	0	0	0	0
		0	1	2	3	4
		Number of Items Considered				

S
4
3

Running Time?

a. $O(n^2)$
b. $O(nW)$
c. $O(n^2W)$
d. $O(2^n)$

```
FUNCTION KnapSack(items, capacity)
    table = [[0] * (capacity + 1)] * (n + 1)
    FOR i IN [1 ..= n]
        v = items[i].value
        w = items[i].weight
        FOR cap IN [0 ..= capacity]
            withoutItem = table[i - 1][cap]
            withItem = table[i - 1][cap - w] + v
            table[i, cap] = max(withoutItem, withItem)

    RETURN table[n][capacity]
```

Correctness

A proof by induction can be constructed by examining the arguments of cases 1 and 2.

Question on assignment

Most Common Variants

- **0-1 Knapsack** – A thief robbing a store finds n items worth v1, v2, .., vn dollars and weight w1, w2, ..., wn pounds, where vi and wi are integers. The thief can carry at most W pounds in the knapsack. Which items should the thief take if they want to maximize value.

- **Fractional knapsack problem** – Same as above, but the thief happens to be at the bulk section of the store and can carry fractional portions of the items. For example, the thief could take 20% of item i for a weight of 0.2wi and a value of 0.2vi.

Preview for Dynamic Programming

How many ways can you return amount A using n kinds of coins?

All the ways returning amount A using all but the first kinds of coins (using the other (n – 1) kinds of coins)

+

All the ways returning amount (A – d) using n kinds of coins, where d is the denomination for the first kind of coin

Does this seem like a “hard” problem?