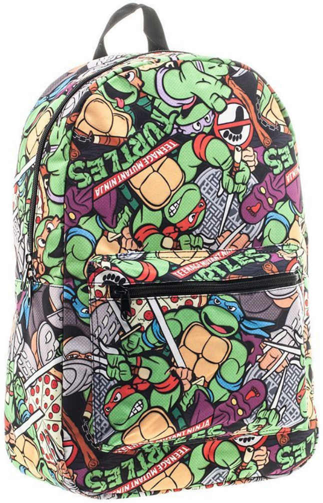
Knapsack Problem

- We have n items with weights and values:

Item:	🐢	💡	🍉	🌮	🚒
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

- And we have a knapsack:
  - it can only carry so much weight:



Capacity: 10

Capacity: 10

Item:	🐢	💡	🍉	🌮	🚒
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

- Unbounded Knapsack:
  - Suppose I have infinite copies of all of the items.
  - What’s the most valuable way to fill the knapsack?

🌮 🌮 💡 💡

Total weight: 10
Total value: 42

- 0/1 Knapsack:
  - Suppose I have only one copy of each item.
  - What’s the most valuable way to fill the knapsack?

💡 🍉 🌮

Total weight: 9
Total value: 35

Some notation

Item:	🐢	💡	🍉		🚒
Weight:	w_1	w_2	w_3	…	w_n
Value:	v_1	v_2	v_3		v_n

Capacity: W

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Optimal substructure

- Sub-problems:
  - Unbounded Knapsack with a smaller knapsack.
  - K[x] = value you can fit in a knapsack of capacity x

First solve the problem for small knapsacks

Then larger knapsacks

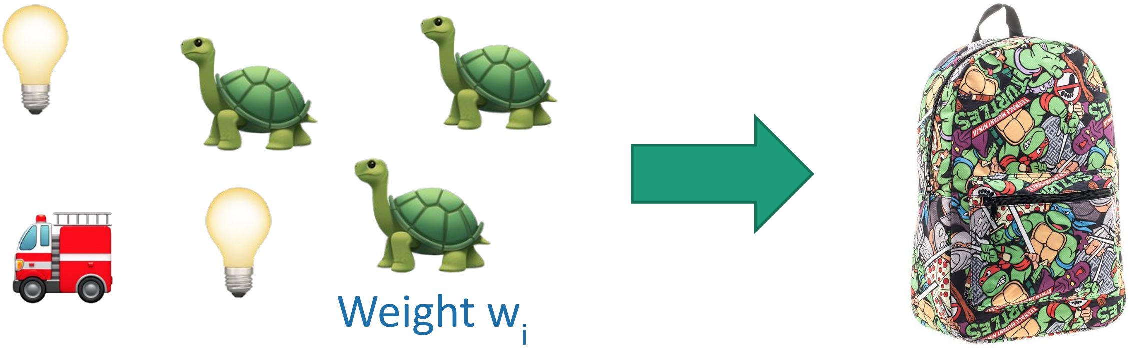
Then larger knapsacks

Optimal substructure

item i

- Suppose this is an optimal solution for capacity x:

Say that the optimal solution contains at least one copy of item i.

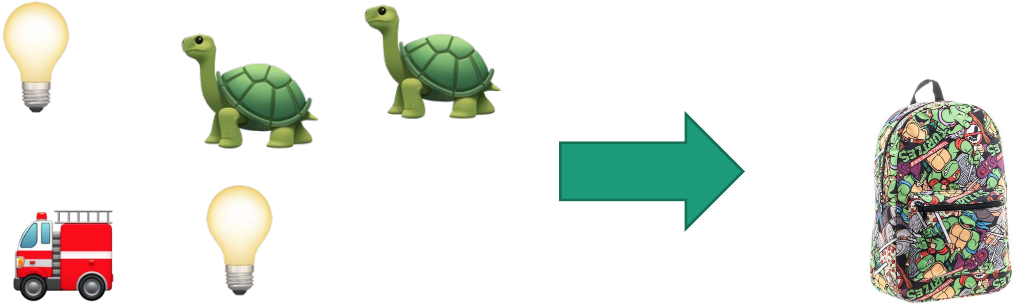


Weight w_i
Value v_i

Capacity x
Value V

- Then this is optimal for capacity x - w_i:

Why?



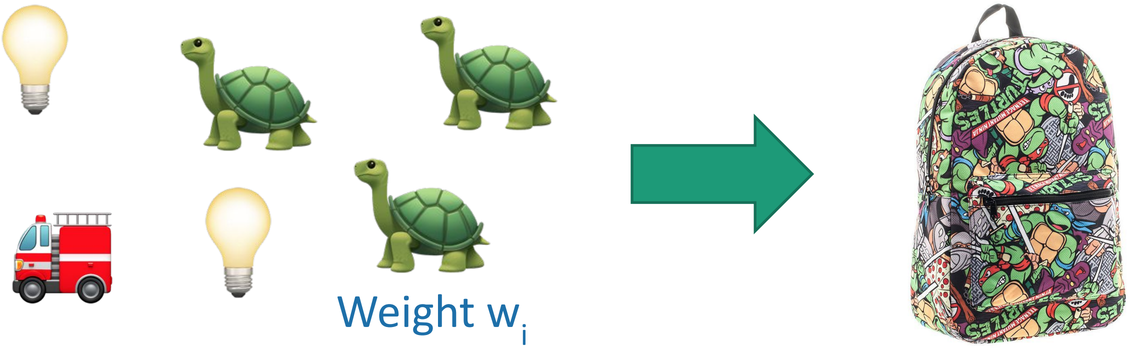
Capacity x – w_i
Value V - v_i

Optimal substructure

item i

- Suppose this is an optimal solution for capacity x:

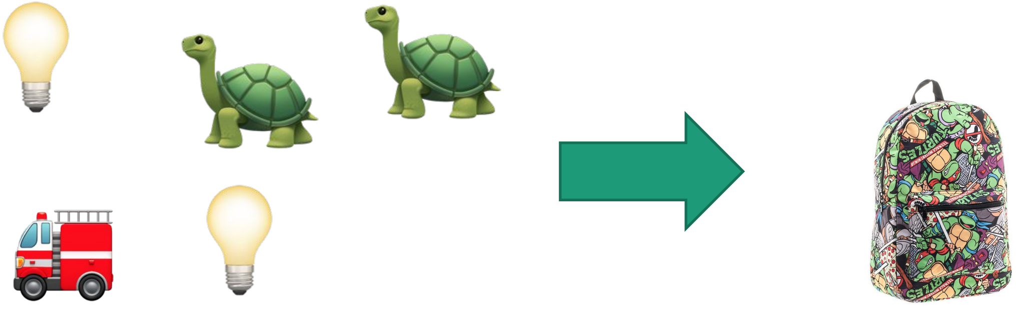
Say that the optimal solution contains at least one copy of item i.



Weight w_i
Value v_i

Capacity x
Value V

- Then this is optimal for capacity x - w_i:



If I could do better than the second solution, then adding a turtle to that improvement would improve the first solution.

Capacity x – w_i
Value V - v_i

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Recursive relationship

- Let K[x] be the optimal value for capacity x.

$K[x] = \max_i \{$ $+$ $\}$

The maximum is over all i so that

Optimal way to fill the smaller knapsack

The value of item i.

$$K[x] = \max_i \{ K[x - w_i] + v_i \}$$

- (And K[x] = 0 if the maximum is empty).
  - That is, if there are no i so that

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Let’s write a bottom-up DP algorithm

```
UnboundedKnapsack(W, n, weights, values):
    K[0] = 0
    for x = 1, …, W:
        K[x] = 0
        for i = 1, …, n:
            if $w_i \leq x$:
                $K[x] = \max\{\ K[x],\ K[x - w_i] + v_i\ \}$
    return K[W]
```

Running time: O(nW)

K[x] = max$_i$ { 🎒 + 🐢 }

= max$_i$ { K[x – w$_i$] + v$_i$ }

Why does this work?

Because our recursive relationship makes sense.

Can we do better?

- Writing down W takes log(W) bits.
- Writing down all n weights takes at most nlog(W) bits.
- Input size: nlog(W).
  - Maybe we could have an algorithm that runs in time O(nlog(W)) instead of O(nW)?
  - Or even $O(\ n^{1000000}\ \log^{1000000}(W)\ )$?

- Open problem!
  - (But probably the answer is **no**…otherwise P = NP)

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Let’s write a bottom-up DP algorithm

```
• UnboundedKnapsack(W, n, weights, values):
    • K[0] = 0
    • for x = 1, …, W:
        • K[x] = 0
        • for i = 1, …, n:
            • if $w_i \leq x$:
                • $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
    • return K[W]
```

K[x] = $\max_i$ { + 🐢 }

= $\max_i \{ K[x - w_i] + v_i \}$

Let’s write a bottom-up DP algorithm

```
- UnboundedKnapsack(W, n, weights, values):
    - K[0] = 0
    - ITEMS[0] = ∅
    - for x = 1, …, W:
        - K[x] = 0
        - for i = 1, …, n:
            - if $w_i \leq x$:
                - $K[x] = \max\{\ K[x],\ K[x - w_i] + v_i\ \}$
                - If K[x] was updated:
                    - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
    - return ITEMS[W]
```

$K[x] = \max_i \{$ 🎒 + 🐢 $\}$

$= \max_i \{ K[x - w_i] + v_i \}$

Example

$$K[x] = \max\{\ K[x],\ K\left[x\ - w_i\right] + v_i\ \}$$

	0	1	2	3	4
K	0				
ITEMS					

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

$$K[x] = \max\{\ K[x],\ K[x - w_i] + v_i\ \}$$

	0	1	2	3	4
K	0	1			
ITEMS		🐢			

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

ITEMS[1] = ITEMS[0] + 🐢

Capacity: 4

Example

$$K[x] = \max\{\ K[x],\ K\left[x\ - w_i\right] + v_i\ \}$$

	0	1	2	3	4
K	0	1	2		
ITEMS		🐢	🐢 🐢		

ITEMS[2] = ITEMS[1] + 🐢

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

$$K[x] = \max\{\ K[x],\ K[x - w_i] + v_i\ \}$$

	0	1	2	3	4
K	0	1	4		
ITEMS		🐢	💡		

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

ITEMS[2] = ITEMS[0] + 💡

Capacity: 4

Example

$$K[x] = \max\{\ K[x],\ K\left[x\ - w_i\right] + v_i\ \}$$

	0	1	2	3	4
K	0	1	4	5	
ITEMS		🐢	💡	💡 🐢	

ITEMS[3] = ITEMS[2] + 🐢

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

$$K[x] = \max\{\ K[x],\ K[x - w_i] + v_i\ \}$$

	0	1	2	3	4
K	0	1	4	6	
ITEMS		🐢	💡	🍉	

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

ITEMS[3] = ITEMS[0] + 🍉

Capacity: 4

Example

$$K[x] = \max\{\ K[x],\ K\left[x\ - w_i\right] + v_i\ \}$$

	0	1	2	3	4
K	0	1	4	6	7
ITEMS		🐢	💡	🍉	🍉 🐢

ITEMS[4] = ITEMS[3] + 🐢

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

$$K[x] = \max\{\ K[x],\ K\left[x - w_i\right] + v_i\ \}$$

	0	1	2	3	4
K	0	1	4	6	8
ITEMS		🐢	💡	🍉	💡 💡

ITEMS[4] = ITEMS[2] + 💡

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

(Pass)

What have we learned?

- We can solve unbounded knapsack in time O(nW).
  - If there are n items and our knapsack has capacity W.
- We again went through the steps to create DP solution:
  - We kept a one-dimensional table, creating smaller problems by making the knapsack smaller.

Capacity: 10

Item:	🐢	💡	🍉	🌮	🚒
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

- Unbounded Knapsack:
  - Suppose I have infinite copies of all of the items.
  - What’s the most valuable way to fill the knapsack?

🌮 🌮 💡 💡

Total weight: 10
Total value: 42

- 0/1 Knapsack:
  - Suppose I have only one copy of each item.
  - What’s the most valuable way to fill the knapsack?

💡 🍉 🌮

Total weight: 9
Total value: 35

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Optimal substructure: try 1

- Sub-problems:
  - Unbounded Knapsack with a smaller knapsack.

First solve the
problem for
small knapsacks

Then larger
knapsacks

Then larger
knapsacks

This won’t quite work…

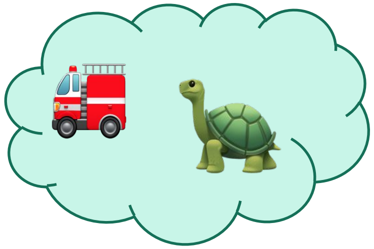
- We are only allowed **one copy of each item**.
- The sub-problem needs to “know” what items we’ve used and what we haven’t.

Optimal substructure: try 2

- Sub-problems:
  - 0/1 Knapsack with fewer items.

First solve the problem with few items

Then more items



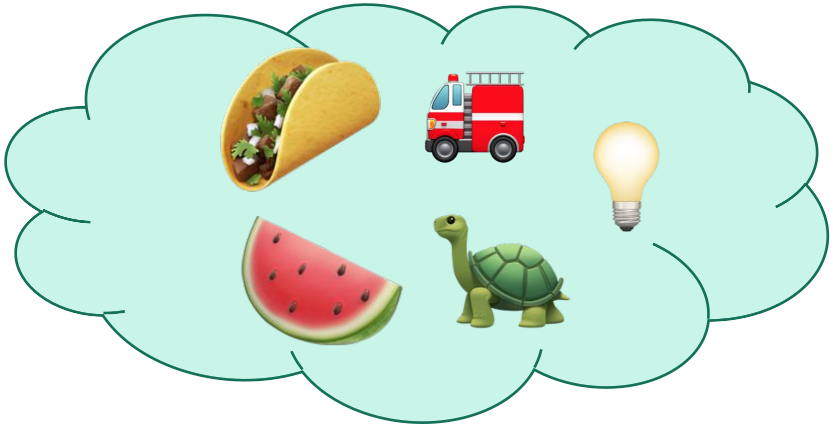
Then yet more items

We'll still increase the size of the knapsacks.

(We'll keep a two-dimensional table).

Our sub-problems:

- Indexed by x and j



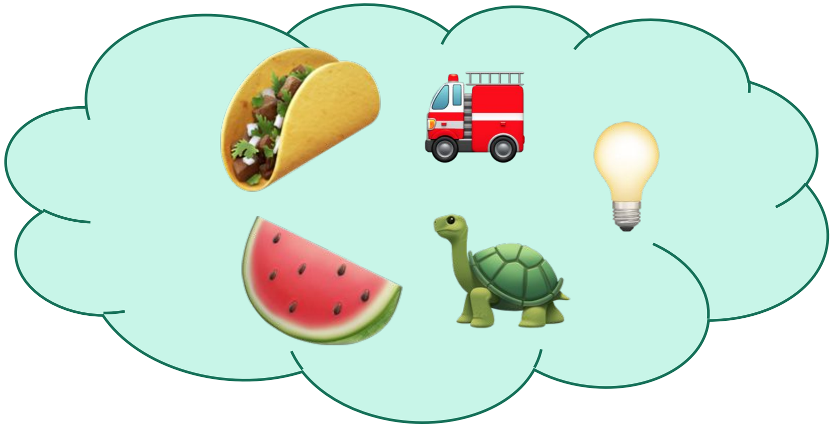
First j items

Capacity x

K[x,j] = optimal solution for a knapsack of size x using only the first j items.

Relationship between sub-problems

- Want to write K[x,j] in terms of smaller sub-problems.



First j items

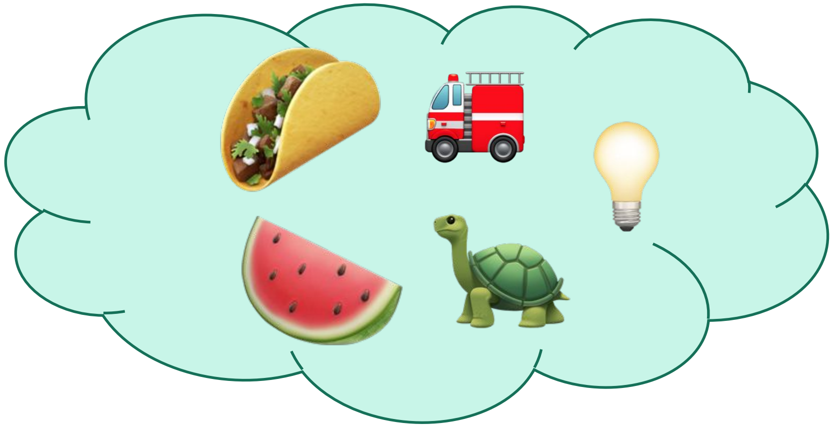
Capacity x

K[x,j] = optimal solution for a knapsack of size x using only the first j items.

Two cases

item j

- **Case 1**: Optimal solution for j items does not use item j.
- **Case 2**: Optimal solution for j items does use item j.



First j items

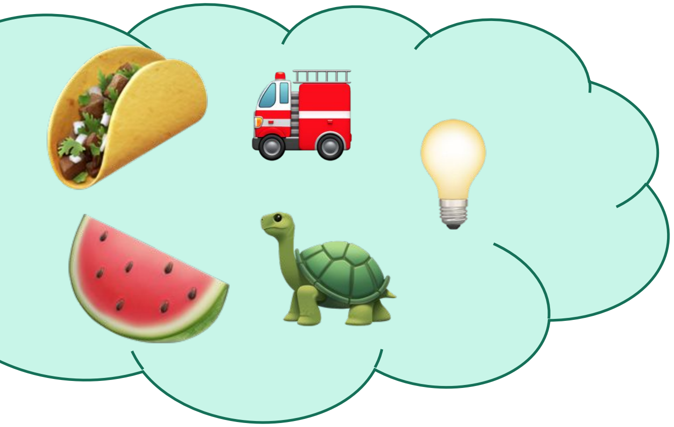
Capacity x

K[x,j] = optimal solution for a knapsack of size x using only the first j items.

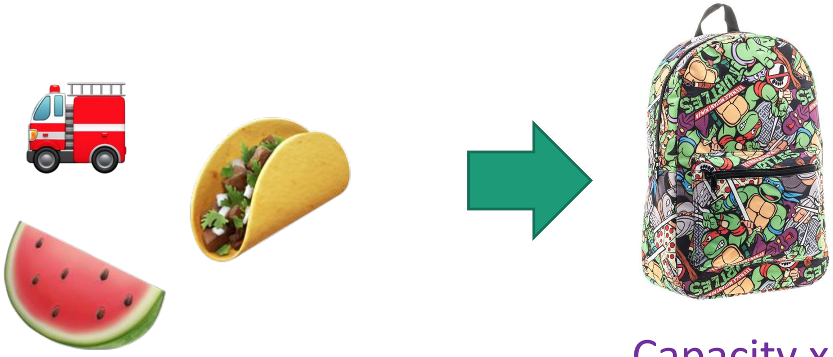
Two cases

item j

- **Case 1**: Optimal solution for j items does not use item j.



First j items



Capacity x

Value V

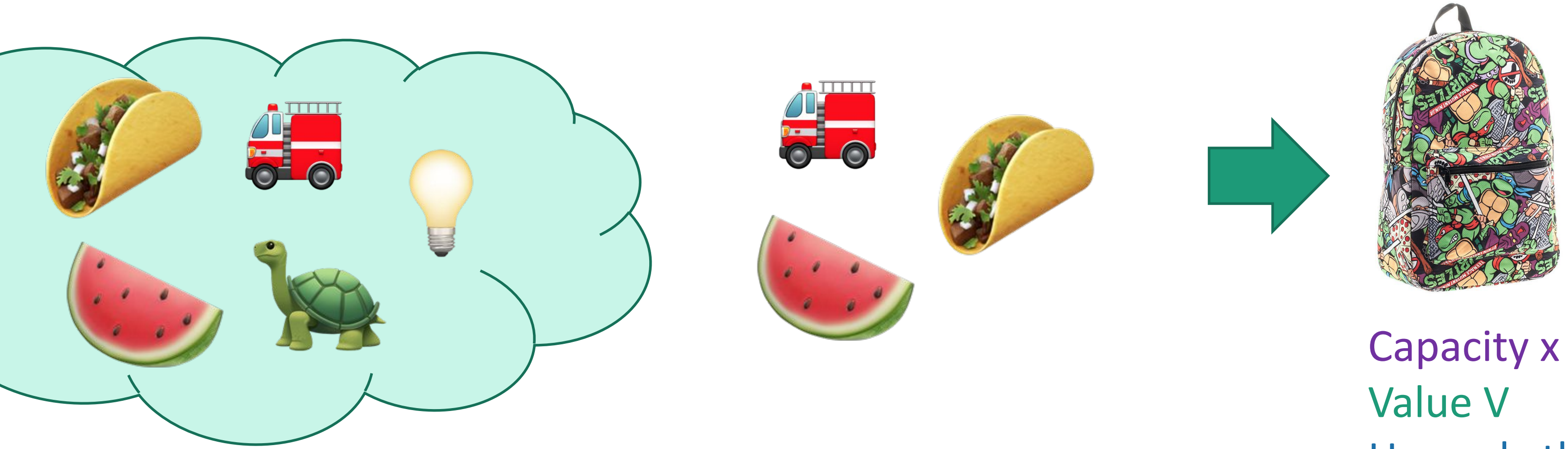
Use only the first j items

What lower-indexed
problem should we solve
to solve this problem?

Two cases

item j

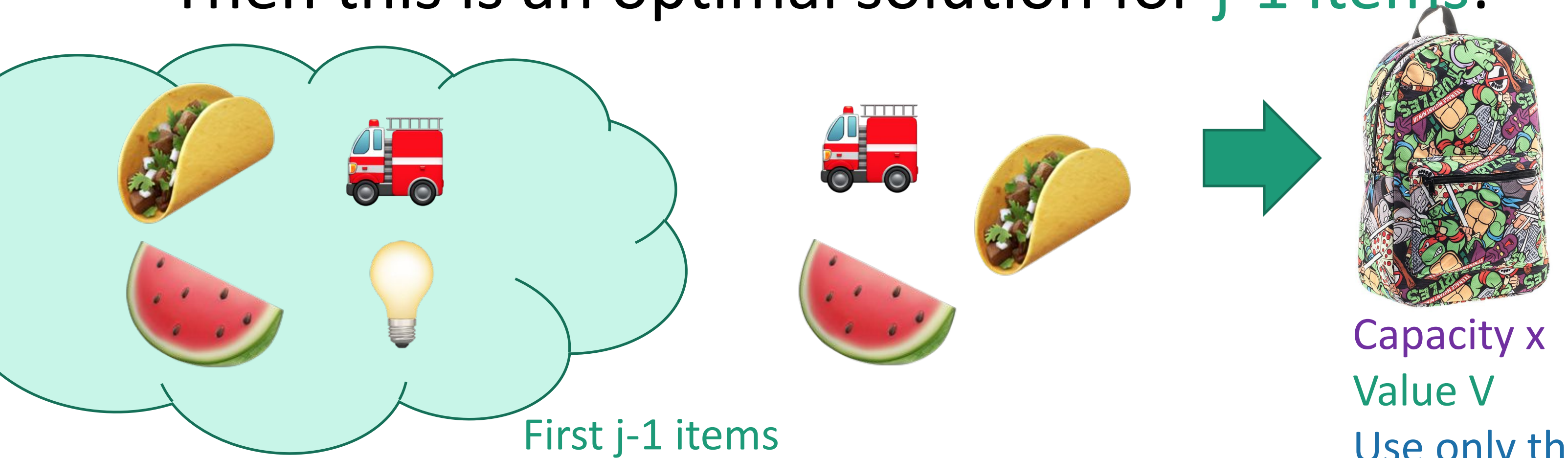
- **Case 1**: Optimal solution for j items does not use item j.



First j items

Capacity x
Value V
Use only the first j items

- Then this is an optimal solution for j-1 items:



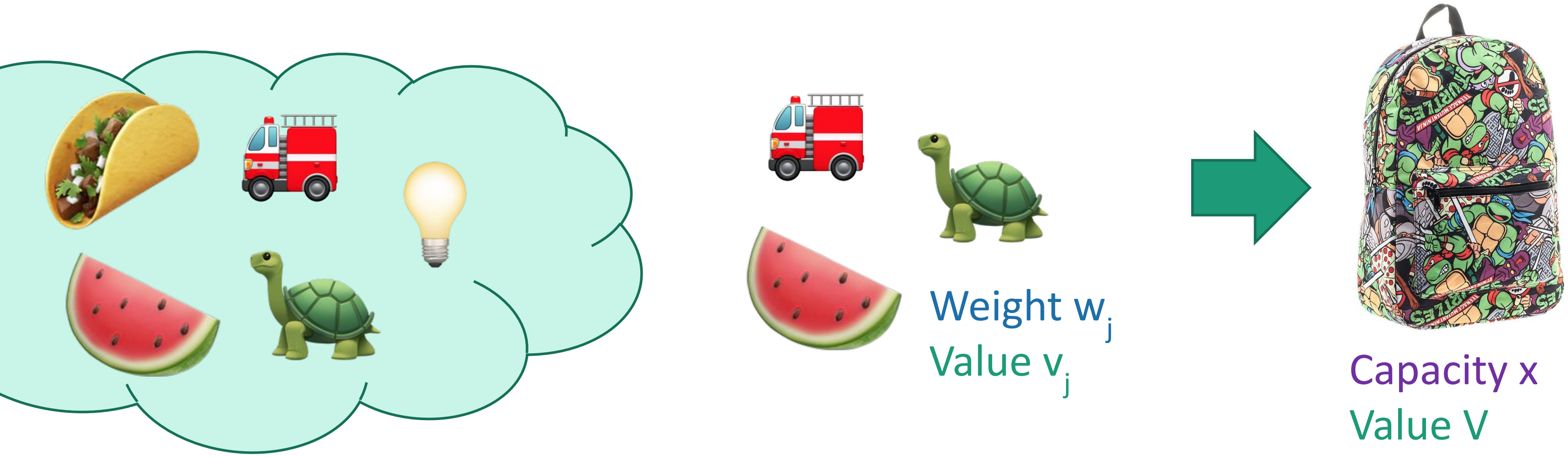
First j-1 items

Capacity x
Value V
Use only the first j-1 items.

Two cases

item j

- **Case 2**: Optimal solution for j items uses item j.



Weight w_j

Value v_j

Capacity x

Value V

First j items

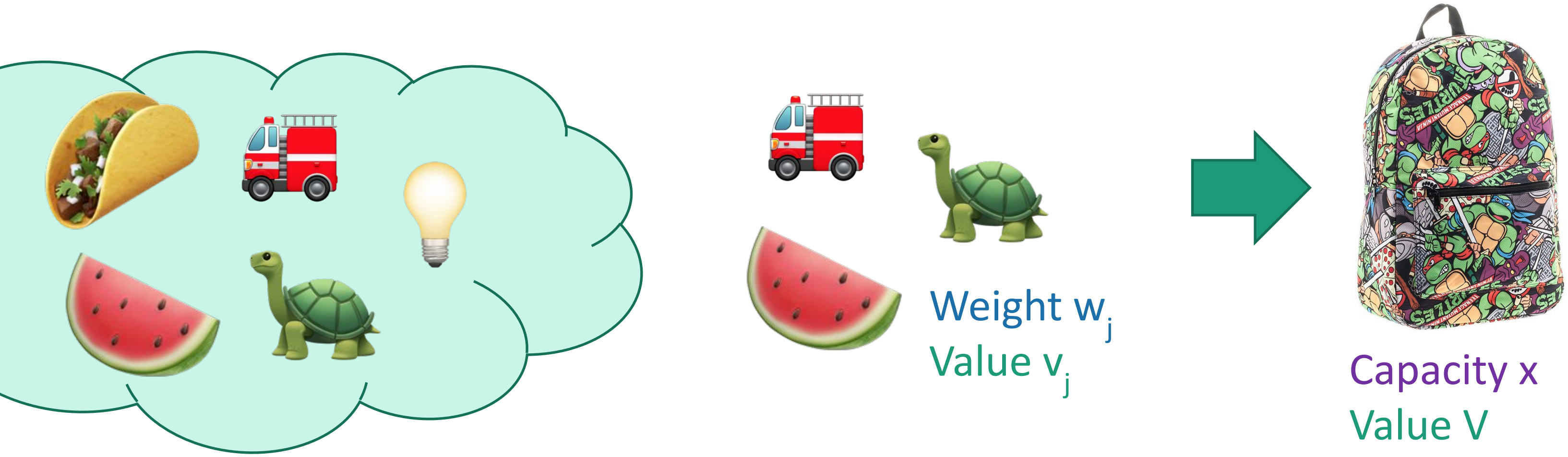
Use only the first j items

What lower-indexed
problem should we solve
to solve this problem?

Two cases

item j

- **Case 2**: Optimal solution for j items uses item j.



Weight w_j

Value v_j

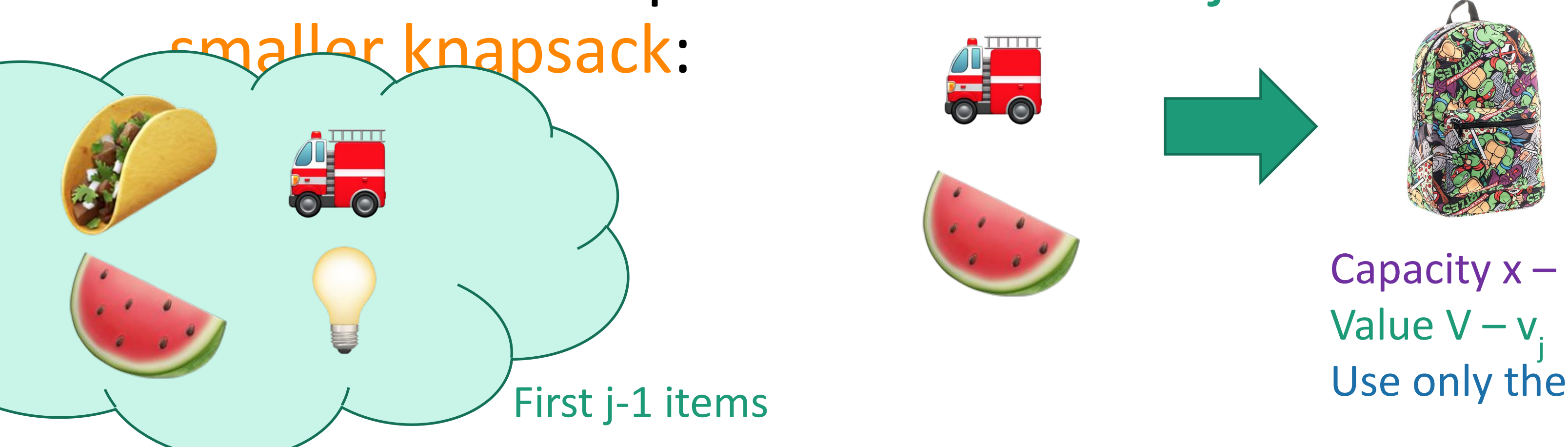
Capacity x

Value V

Use only the first j items

First j items

- Then this is an optimal solution for j-1 items and a smaller knapsack:



Capacity $x - w_j$

Value $V - v_j$

Use only the first j-1 items.

First j-1 items

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Recursive relationship

- Let K[x,j] be the optimal value for:
  - capacity x,
  - with j items.

$$K[x,j] = \max\{\ \underbrace{K[x, j-1]}_{\text{Case 1}}\ ,\ \underbrace{K[x - w_{j,}\ j-1] + v_j}_{\text{Case 2}}\ \}$$

- (And K[x,0] = 0 and K[0,j] = 0).

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Bottom-up DP algorithm

```
Zero-One-Knapsack(W, n, w, v):
    K[x,0] = 0 for all x = 0,…,W
    K[0,i] = 0 for all i = 0,…,n
    for x = 1,…,W:
        for j = 1,…,n:
                                        Case 1
            K[x,j] = K[x, j-1]
            if $w_j$  x:
                                                                Case 2
                K[x,j] = max{ K[x,j], K[x – $w_j$, j-1] + $v_j$ }
    return K[W,n]
```

Running time O(nW)

Example

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
🐢 j=1	0			
💡 🐢 j=2	0			
🍉 💡 🐢 j=3	0			

```
Zero-One-Knapsack(W, n, w, v):
    K[x,0] = 0 for all x = 0,…,W
    K[0,i] = 0 for all i = 0,…,n
    for x = 1,…,W:
        for j = 1,…,n:
            K[x,j] = K[x, j-1]
            if $w_j$  x:
                K[x,j] = max{ K[x,j],
                    K[x – $w_j$, j-1] + $v_j$ }
    return K[W,n]
```

🟧 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	0		
💡🐢	j=2	0			
🍉💡🐢	j=3	0			

```
- Zero-One-Knapsack(W, n, w, v):
    - K[x,0] = 0 for all x = 0,…,W
    - K[0,i] = 0 for all i = 0,…,n
    - for x = 1,…,W:
        - for j = 1,…,n:
            - K[x,j] = K[x, j-1]
            - if $w_j$  x:
                - K[x,j] = max{ K[x,j],
                          K[x – $w_j$, j-1] + $v_j$ }
    - return K[W,n]
```

🟥 current entry	🟦 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

```
• Zero-One-Knapsack(W, n, w, v):
    • K[x,0] = 0 for all x = 0,…,W
    • K[0,i] = 0 for all i = 0,…,n
    • for x = 1,…,W:
        • for j = 1,…,n:
            • K[x,j] = K[x, j-1]
            • if $w_j$  x:
                • K[x,j] = max{ K[x,j],
                      K[x – $w_j$, j-1] + $v_j$ }
    • return K[W,n]
```

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢		
💡 🐢	j=2	0			
🍉 💡 🐢	j=3	0			

🟥 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢		
💡 🐢	j=2	0	1 🐢		
🍉 💡 🐢	j=3	0			

🟥 current entry

🟩 relevant previous entry

- Zero-One-Knapsack(W, n, w, v):

```
• K[x,0] = 0 for all x = 0,…,W
• K[0,i] = 0 for all i = 0,…,n
• for x = 1,…,W:
    • for j = 1,…,n:
        • K[x,j] = K[x, j-1]
        • if $w_j$ x:
            • K[x,j] = max{ K[x,j], K[x – $w_j$, j-1] + $v_j$ }
• return K[W,n]
```

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

```
• Zero-One-Knapsack(W, n, w, v):
    • K[x,0] = 0 for all x = 0,…,W
    • K[0,i] = 0 for all i = 0,…,n
    • for x = 1,…,W:
        • for j = 1,…,n:
            • K[x,j] = K[x, j-1]
            • if $w_j$  x:
                • K[x,j] = max{ K[x,j],
                              K[x – $w_j$, j-1] + $v_j$ }
    • return K[W,n]
```

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
🐢 j=1	0	1 🐢		
💡 🐢 j=2	0	1 🐢		
🍉 💡 🐢 j=3	0	1 🐢		

🟥 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	0	
💡 🐢	j=2	0	1 🐢		
🍉 💡 🐢	j=3	0	1 🐢		

- Zero-One-Knapsack(W, n, w, v):

```
    • K[x,0] = 0 for all x = 0,…,W
    • K[0,i] = 0 for all i = 0,…,n
    • for x = 1,…,W:
        • for j = 1,…,n:
            • K[x,j] = K[x, j-1]
            • if $w_j$ x:
                • K[x,j] = max{ K[x,j],
                      $K[x – w_j, j-1] + v_j$ }
    • return K[W,n]
```

current entry

relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

```
• Zero-One-Knapsack(W, n, w, v):
    • $K[x,0] = 0$ for all $x = 0,\ldots,W$
    • $K[0,i] = 0$ for all $i = 0,\ldots,n$
    • for $x = 1,\ldots,W$:
        • for $j = 1,\ldots,n$:
            • $K[x,j] = K[x, j-1]$
            • if $w_j$  $x$:
                • $K[x,j] = \max\{ K[x,j], K[x - w_j, j-1] + v_j \}$
    • return $K[W,n]$
```

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	
💡 🐢	j=2	0	1 🐢		
🍉 💡 🐢	j=3	0	1 🐢		

🟥 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

```
• Zero-One-Knapsack(W, n, w, v):
    • K[x,0] = 0 for all x = 0,…,W
    • K[0,i] = 0 for all i = 0,…,n
    • for x = 1,…,W:
        • for j = 1,…,n:
            • K[x,j] = K[x, j-1]
            • if $w_j$ x:
                • K[x,j] = max{ K[x,j],
                          K[x – $w_j$, j-1] + $v_j$ }
    • return K[W,n]
```

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	
💡 🐢	j=2	0	1 🐢	1 🐢	
🍉 💡 🐢	j=3	0	1 🐢		

🟥 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉	
Weight:	1	2	3	
Value:	1	4	6	Capacity: 3

Example

```
Zero-One-Knapsack(W, n, w, v):
    K[x,0] = 0 for all x = 0,…,W
    K[0,i] = 0 for all i = 0,…,n
    for x = 1,…,W:
        for j = 1,…,n:
            K[x,j] = K[x, j-1]
            if $w_j$  x:
                K[x,j] = max{ K[x,j], K[x – $w_j$, j-1] + $v_j$ }
    return K[W,n]
```

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	
💡 🐢	j=2	0	1 🐢	4 💡	
🍉 💡 🐢	j=3	0	1 🐢		

🟥 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

```
• Zero-One-Knapsack(W, n, w, v):
    • K[x,0] = 0 for all x = 0,…,W
    • K[0,i] = 0 for all i = 0,…,n
    • for x = 1,…,W:
        • for j = 1,…,n:
            • K[x,j] = K[x, j-1]
            • if $w_j$ x:
                • K[x,j] = max{ K[x,j],
                               $K[x - w_j, j-1] + v_j$ }
    • return K[W,n]
```

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	
💡 🐢	j=2	0	1 🐢	4 💡	
🍉 💡 🐢	j=3	0	1 🐢	4 💡	

🟥 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

```
Zero-One-Knapsack(W, n, w, v):
    K[x,0] = 0 for all x = 0,…,W
    K[0,i] = 0 for all i = 0,…,n
    for x = 1,…,W:
        for j = 1,…,n:
            K[x,j] = K[x, j-1]
            if $w_j$  x:
                K[x,j] = max{ K[x,j],
                    K[x – $w_j$, j-1] + $v_j$ }
    return K[W,n]
```

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0 🟦
🐢	j=1	0	1 🐢	1 🐢	0 🟧
💡 🐢	j=2	0	1 🐢	4 💡	
🍉 💡 🐢	j=3	0	1 🐢	4 💡	

🟧 current entry

🟦 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

```
• Zero-One-Knapsack(W, n, w, v):
    • K[x,0] = 0 for all x = 0,…,W
    • K[0,i] = 0 for all i = 0,…,n
    • for x = 1,…,W:
        • for j = 1,…,n:
            • K[x,j] = K[x, j-1]
            • if $w_j$  x:
                • K[x,j] = max{
                  K[x,j],
                  K[x – $w_j$, j-1] + $v_j$ }
    • return K[W,n]
```

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
🐢 j=1	0	1 🐢	1 🐢	1 🐢
💡 🐢 j=2	0	1 🐢	4 💡	
🍉 💡 🐢 j=3	0	1 🐢	4 💡	

🟥 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

```
• Zero-One-Knapsack(W, n, w, v):
    • K[x,0] = 0 for all x = 0,…,W
    • K[0,i] = 0 for all i = 0,…,n
    • for x = 1,…,W:
        • for j = 1,…,n:
            • K[x,j] = K[x, j-1]
            • if $w_j$ x:
                • K[x,j] = max{ K[x,j],
                      $K[x – w_j, j-1] + v_j$ }
    • return K[W,n]
```

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	1 🐢
💡 🐢	j=2	0	1 🐢	4 💡	1 🐢
🍉 💡 🐢	j=3	0	1 🐢	4 💡	

🟥 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

```
• Zero-One-Knapsack(W, n, w, v):
    • K[x,0] = 0 for all x = 0,…,W
    • K[0,i] = 0 for all i = 0,…,n
    • for x = 1,…,W:
        • for j = 1,…,n:
            • K[x,j] = K[x, j-1]
            • if $w_j$  x:
                • K[x,j] = max{ K[x,j], K[$x – w_j$, j-1] + $v_j$ }
    • return K[W,n]
```

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
🐢 j=1	0	🟩 1 🐢	1 🐢	1 🐢
💡 🐢 j=2	0	1 🐢	4 💡	🟧 5 🐢 💡
🍉 💡 🐢 j=3	0	1 🐢	4 💡	

🟧 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

```
• Zero-One-Knapsack(W, n, w, v):
    • K[x,0] = 0 for all x = 0,…,W
    • K[0,i] = 0 for all i = 0,…,n
    • for x = 1,…,W:
        • for j = 1,…,n:
            • K[x,j] = K[x, j-1]
            • if $w_j$  x:
                • K[x,j] = max{ K[x,j], $K[x – w_j, j-1] + v_j$ }
    • return K[W,n]
```

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	1 🐢
💡 🐢	j=2	0	1 🐢	4 💡	5 🐢 💡
🍉 💡 🐢	j=3	0	1 🐢	4 💡	5 🐢 💡

🟥 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

```
Zero-One-Knapsack(W, n, w, v):
    K[x,0] = 0 for all x = 0,…,W
    K[0,i] = 0 for all i = 0,…,n
    for x = 1,…,W:
        for j = 1,…,n:
            K[x,j] = K[x, j-1]
            if $w_j$ x:
                K[x,j] = max{ K[x,j], K[x – $w_j$, j-1] + $v_j$ }
    return K[W,n]
```

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	1 🐢
💡 🐢	j=2	0	1 🐢	4 💡	5 🐢 💡
🍉 💡 🐢	j=3	0	1 🐢	4 💡	6 🍉

🟧 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	1 🐢
💡 🐢	j=2	0	1 🐢	4 💡	5 🐢 💡
🍉 💡 🐢	j=3	0	1 🐢	4 💡	6 🍉

```
• Zero-One-Knapsack(W, n, w, v):
    • K[x,0] = 0 for all x = 0,…,W
    • K[0,i] = 0 for all i = 0,…,n
    • for x = 1,…,W:
        • for j = 1,…,n:
            • K[x,j] = K[x, j-1]
            • if $w_j$  x:
                • K[x,j] = max{
                  K[x,j],
                  $K[x – w_j, j-1] + v_j$ }
    • return K[W,n]
```

So the optimal solution is to put one watermelon in your knapsack!

🟧 current entry

🟩 relevant previous entry

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

What have we learned?

- We can solve 0/1 knapsack in time O(nW).
    - If there are n items and our knapsack has capacity W.

- We again went through the steps to create DP solution:
    - We kept a two-dimensional table, creating smaller problems by restricting the set of allowable items.

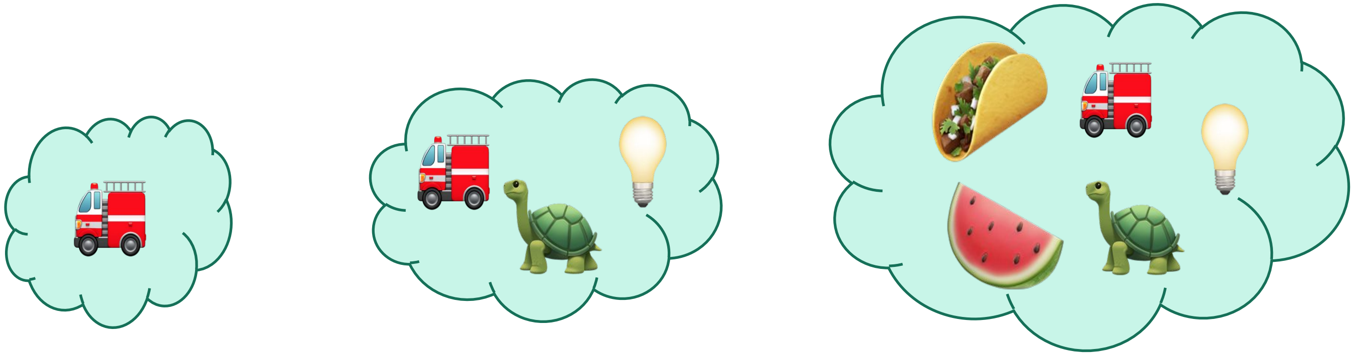
Question

- How did we know which substructure to use in which variant of knapsack?

Answer in retrospect:

This one made sense for unbounded knapsack because it doesn't have any memory of what items have been used.

vs.



In 0/1 knapsack, we can only use each item once, so it makes sense to leave out one item at a time.

Operational Answer: try some stuff, see what works!