Knapsack Problem

• We have n items with weights and values:

<table>
<thead>
<tr>
<th>Item:</th>
<th>Weight:</th>
<th>Value:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turtle</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>Light bulb</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Watermelon</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>Taco</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>Fire truck</td>
<td>11</td>
<td>35</td>
</tr>
</tbody>
</table>

• And we have a knapsack:
  • it can only carry so much weight:
    | Capacity: 10 |
• 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:
- Turtle
- Light bulb
- Watermelon
- Fire truck

Weight:
- $W_1$
- $W_2$
- $W_3$
- $\ldots$
- $W_n$

Value:
- $V_1$
- $V_2$
- $V_3$
- $\ldots$
- $V_n$

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.
  • \( K[x] = \text{value you can fit in a knapsack of capacity } x \)

First solve the problem for small knapsacks

Then larger knapsacks

Then larger knapsacks
Optimal substructure

- Suppose this is an optimal solution for capacity $x$:

- Then this is optimal for capacity $x - w_i$.

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

Why?
Optimal substructure

• Suppose this is an optimal solution for capacity $x$:

<table>
<thead>
<tr>
<th>Weight $w_i$</th>
<th>Value $v_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$V$</td>
</tr>
</tbody>
</table>

  Then this is optimal for capacity $x - w_i$:

<table>
<thead>
<tr>
<th>Weight $w_i$</th>
<th>Value $v_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$V - v_i$</td>
</tr>
</tbody>
</table>

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

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

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

The maximum is over all $i$ so that

Optimal way to fill the smaller knapsack

The value of item $i$.

• (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: \( O(nW) \)

\[ K[x] = \max_i \{ \text{bag} + \text{turtle} \} = \max_i \{ K[x - w_i] + v_i \} \]

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 $n\log(W)$ bits.
• Input size: $n\log(W)$.
  • Maybe we could have an algorithm that runs in time $O(n\log(W))$ 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, { backpack + turtle }
= max, { K[x - w_i] + v_i }
Let’s write a bottom-up DP algorithm

- **UnboundedKnapsack**($W$, $n$, **weights**, **values**):
  - $K[0] = 0$
  - $ITEMS[0] = \emptyset$
  - for $x = 1, \ldots, W$:
    - $K[x] = 0$
    - for $i = 1, \ldots, n$:
      - if $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If $K[x]$ was updated:
          - $ITEMS[x] = ITEMS[x - w_i] \cup \{ \text{item } i \}$
  - return $ITEMS[W]$

\[
K[x] = \max_i \{ \text{bag} + \text{turtle} \} \\
= \max_i \{ K[x - w_i] + v_i \}
\]
**Example**

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>K</strong></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ITEMS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Item:**
- Turtle: 1
- Light bulb: 2
- Watermelon: 3

**Weight:**
- Turtle: 1
- Light bulb: 2
- Watermelon: 3

**Value:**
- Turtle: 1
- Light bulb: 4
- Watermelon: 6

**Capacity:** 4
## Example

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITEMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ITEMS[1] = ITEMS[0] +

**Item:**
- Turtle: 1
- Lightbulb: 2
- Watermelon: 3

**Weight:**
- Turtle: 1
- Lightbulb: 2
- Watermelon: 3

**Value:**
- Turtle: 1
- Lightbulb: 4
- Watermelon: 6

**Capacity:** 4
Example

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

<table>
<thead>
<tr>
<th>ITEMS</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITEMS[0]</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITEMS[1]</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITEMS[2]</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Item:
- Turtle: 1
- Lightbulb: 2
- Watermelon: 3

Weight:
- Turtle: 1
- Lightbulb: 2
- Watermelon: 3

Value:
- Turtle: 1
- Lightbulb: 4
- Watermelon: 6

Capacity: 4
Example

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

<table>
<thead>
<tr>
<th>ITEMS</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ITEMS[2] = ITEMS[0] +

Item:

- Turtle
- Lightbulb
- Watermelon

Weight:

- 1
- 2
- 3

Value:

- 1
- 4
- 6

Capacity: 4
Example

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ITEMS</td>
<td>Turtles</td>
<td>Light Bulb</td>
<td>Light Bulb</td>
<td>Turtles</td>
<td></td>
</tr>
</tbody>
</table>


Item:
- Turtles
- Light Bulb
- Watermelon

Weight:
- 1
- 2
- 3

Value:
- 1
- 4
- 6

Capacity: 4
Example

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

<table>
<thead>
<tr>
<th>K</th>
<th>ITEMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Item:
- Turtle
- Light bulb
- Watermelon

Weight:
- 1
- 2
- 3

Value:
- 1
- 4
- 6

Capacity: 4

\[ \text{ITEMS}[3] = \text{ITEMS}[0] + \text{watermelon} \]
Example

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

<table>
<thead>
<tr>
<th>K</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

**ITEMS**

- Tortoise
- Light bulb
- Watermelon
- Tortoise


**Item:**
- Tortoise: 1
- Light bulb: 2
- Watermelon: 3

**Weight:**
- 1
- 2
- 3

**Value:**
- 1
- 4
- 6

**Capacity:** 4
Example

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

<table>
<thead>
<tr>
<th>ITEM</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turtle</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Light bulb</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Watermelon</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>


Capacity: 4

<table>
<thead>
<tr>
<th>K</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITEMS</td>
<td>Turtle</td>
<td>Light bulb</td>
<td>Watermelon</td>
<td>Light bulb</td>
<td>Light bulb</td>
</tr>
</tbody>
</table>
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.
• **Unbounded Knapsack:**
  • Suppose I have *infinite copies* of all of the items.
  • What’s the *most valuable way to fill the knapsack*?
    
    | Item: | Capacity: 10 |
    |------|-------------|
    | Weight: | Value: |
    | 6 | 20 | 6 |
    | 2 | 8 | 8 |
    | 4 | 14 | 14 |
    | 3 | 13 | 13 |
    | 11 | 35 | 35 |

    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*?
    
    | Item: | Capacity: 10 |
    |------|-------------|
    | Weight: | Value: |
    | 11 | 35 |
    | 13 | 35 |
    | 35 | 35 |

    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.

I can’t use any turtles…
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 \( j \)

\[ K[x,j] = \text{optimal solution for a knapsack of size } x \text{ using only the first } j \text{ items.} \]
Relationship between sub-problems

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

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

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

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

**Case 1**: Optimal solution for \( j \) items does not use item \( j \).

First \( j \) items

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

Capacity \( x \)
Value \( V \)
Use only the first \( j \) items
Two cases

• **Case 1**: Optimal solution for \( j \) items does not use item \( j \).

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

• Use only the first \( j \) items.

• Use only the first \( j-1 \) items.
Two cases

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

First $j$ items

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

Capacity $x$
Value $V$
Use only the first $j$ items
Two cases

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

• Then this is an optimal solution for *j*-1 items and a smaller knapsack:
  - Weight $w_j$
  - Value $v_j$
  - Capacity $x - w_j$
  - Value $V - v_j$

First *j* 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.

$$K[x,j] = \max \{ K[x, j-1], K[x - w_j, j-1] + v_j \}$$

Case 1

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 \(x = 0,\ldots,W\)
  - \(K[0,i] = 0\) for all \(i = 0,\ldots,n\)
  - for \(x = 1,\ldots,W\):
    - for \(j = 1,\ldots,n\):
      - Case 1
        - \(K[x,j] = K[x, j-1]\)
      - Case 2
        - if \(w_j \leq x\):
          - \(K[x,j] = \max\{ K[x,j], K[x - w_j, j-1] + v_j \}\)
  - return \(K[W,n]\)

Running time \(O(nW)\)
Example

<table>
<thead>
<tr>
<th></th>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $K[W,n]$

Item:

- Turtle: 1
- Light bulb: 2
- Watermelon: 3

Weight:

- Turtle: 1
- Light bulb: 2
- Watermelon: 3

Value:

- Turtle: 1
- Light bulb: 4
- Watermelon: 6

Capacity: 3
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 $W_j \geq x$:
      • $K[x,j] = \max\{K[x,j], K[x-w_j, j-1] + v_j\}$
  • return $K[W,n]$

Example

<table>
<thead>
<tr>
<th></th>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Item:**
- Turtle: 1
- Light bulb: 2
- Watermelon: 3

**Capacity:** 3
Example

<table>
<thead>
<tr>
<th></th>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Item:**
  - Turtles: 1
  - Lights: 2
  - Watermelon: 3

- **Weight:**
  - Turtles: 1
  - Lights: 2
  - Watermelon: 3

- **Value:**
  - Turtles: 1
  - Lights: 4
  - Watermelon: 6

- **Capacity:** 3

- **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 $W_j \leq x$:
        - $K[x,j] = \max\{K[x,j], K[x-w_j, j-1] + v_j\}$
  - return $K[W,n]$
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 $\sum_{j=1}^{n} w_j x$:
      - $K[x,j] = \max\{K[x,j], K[x - \sum_{j=1}^{n} w_j, j-1] + v_j\}$
- return $K[W,n]$
• 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 $w_j \leq x$:
        • $K[x,j] = \max\{K[x,j], K[x-w_j, j-1] + v_j\}$
  • return $K[W,n]$

Example

<table>
<thead>
<tr>
<th></th>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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 $w_j \leq x$
    - $K[x,j] = \max\{K[x,j], K[x-w_j, j-1] + v_j\}$
  - return $K[W,n]$

<table>
<thead>
<tr>
<th>Item:</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turtle</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Light</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Watermelon</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Capacity: 3
Example

<table>
<thead>
<tr>
<th></th>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 $w_j \leq x$:
        - K[x,j] = max{K[x,j], K[x – w_j, j-1] + v_j}
  - return K[W,n]

Item:
- Weight: 1
- Value: 1
- Capacity: 3
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 $w_j \leq x$:
      - $K[x,j] = \max\{K[x,j], K[x - w_j, j-1] + v_j\}$

- return $K[W,n]$

---

**Example**

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turtle</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Light</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Watermelon</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

- **Capacity**: 3

---

**Table**

<table>
<thead>
<tr>
<th>$j=0$</th>
<th>$j=1$</th>
<th>$j=2$</th>
<th>$j=3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x=0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$x=1$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$x=2$</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$x=3$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

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

- \( K[0, 0] = 0 \) 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 \( w_j \leq x \):
      - \( K[x, j] = \max\{ K[x, j], K[x - w_j, j-1] + v_j \} \)
- return \( K[W, n] \)
**Example**

<table>
<thead>
<tr>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

- **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 $w_j \leq x$:
        - $K[x,j] = \max\{K[x,j], K[x - w_j, j-1] + v_j\}$
  - return $K[W,n]$

<table>
<thead>
<tr>
<th>Item:</th>
<th>Weight:</th>
<th>Value:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turtle</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Light bulb</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Watermelon</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Capacity: 3
Example

<table>
<thead>
<tr>
<th></th>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

- **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 $w_j \leq x$
      - $K[x,j] = \max\{K[x,j], K[x – w_j, j-1] + v_j\}$
  - return $K[W,n]$

**Item:**
- **Weight:** 1 2 3
- **Value:** 1 4 6

**Capacity:** 3
Example

<table>
<thead>
<tr>
<th></th>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**Item:**
- Turtle: 1
- Light: 2
- Watermelon: 3

**Weight:**
- Turtle: 1
- Light: 2
- Watermelon: 3

**Value:**
- Turtle: 1
- Light: 4
- Watermelon: 6

**Capacity:** 3

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 \( W_j \times X \):
      - \( K[x,j] = \max \{ K[x,j], K[x - w_j, j-1] + v_j \} \)
- return \( K[W,n] \)
Example

<table>
<thead>
<tr>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Item:
- Weight: 1 2 3
- Value: 1 4 6

Capacity: 3

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 \( w_j \leq x \):
      - \( K[x,j] = \max\{ K[x,j], K[x-w_j, j-1] + v_j \} \)
- return \( K[W,n] \)
Example

<table>
<thead>
<tr>
<th></th>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

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 $w_j \leq x$:
      - $K[x,j] = \max\{K[x,j], K[x - w_j, j-1] + v_j\}$
- return $K[W,n]$

<table>
<thead>
<tr>
<th>Item:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight:</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Value:</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Capacity: 3
• 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 $w_j \leq x$
      • $K[x,j] = \max\{K[x,j],\ K[x - w_j, j-1] + v_j\}$
  • return $K[W,n]$
Weight: 1 2 3 4
Value: 6 5 4 2
Item: Watermelon Lightbulb

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 \( i = 0, \ldots, n \)

\[ K[x,j] = K[x, j-1] \]
\[ \text{if } w_j \leq x \]
\[ \max \{ K[x, j-1] + v_j \} \]
\[ \text{return } K[W,n] \]

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