Lecture 13

*More dynamic programming!*

Longest Common Subsequences, Knapsack, and
(if time) independent sets in trees.
Announcements

• HW5 due Friday!
• HW6 released Friday!
Last time

- Not coding in an action movie.
Dynamic programming is an algorithm design paradigm.

Basic idea:

- Identify **optimal sub-structure**
  - Optimum to the big problem is built out of optima of small sub-problems
- Take advantage of **overlapping sub-problems**
  - Only solve each sub-problem once, then use it again and again
- Keep track of the solutions to sub-problems in a table as you build to the final solution.
Today

• Examples of dynamic programming:
  1. Longest common subsequence
  2. Knapsack problem
     • Two versions!
  3. Independent sets in trees
     • If we have time...
     • (If not the slides will be there as a reference)
The goal of this lecture

- For you to get really bored of dynamic programming
Longest Common Subsequence

• How similar are these two species?

DNA: AGCCCTAAGGGCTACCTAGCTT
DNA: GACAGCCTACAAGCGTTAGCTTG
Longest Common Subsequence

• How similar are these two species?

DNA: AGCCCTAAAGGGCTACCTAGCTT

DNA: GACAGCCTACAAGCGTTAGCTTT

• Pretty similar, their DNA has a long common subsequence:

AGCCTAAGCTTAGCTT
Longest Common Subsequence

• Subsequence:
  • BDFH is a subsequence of ABCDEFGH

• If X and Y are sequences, a common subsequence is a sequence which is a subsequence of both.
  • BDFH is a common subsequence of ABCDEFGH and of ABDFGHI

• A longest common subsequence...
  • ...is a common subsequence that is longest.
  • The longest common subsequence of ABCDEFGH and ABDFGHI is ABDFGH.
We sometimes want to find these

• Applications in bioinformatics

• The unix command **diff**

• Merging in version control
  • **svn**, **git**, etc...
Recipe for applying Dynamic Programming

• **Step 1:** Identify optimal substructure.

• **Step 2:** Find a recursive formulation for the length of the longest common subsequence.

• **Step 3:** Use dynamic programming to find the length of the longest common subsequence.

• **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.

• **Step 5:** If needed, code this up like a reasonable person.
Step 1: Optimal substructure

Prefixes:

- Notation: denote this prefix $ACGC$ by $Y_4$

- Our sub-problems will be finding LCS’s of prefixes to $X$ and $Y$.
- Let $C[i,j] = length\_of\_LCS(X_i, Y_j)$
Optimal substructure ctd.

• Subproblem:
  • finding LCS’s of prefixes of X and Y.

• Why is this a good choice?
  • There’s some relationship between LCS’s of prefixes and LCS’s of the whole things.
  • These subproblems overlap a lot.

To see this formally, on to...
Recipe for applying Dynamic Programming

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

Case 1: $X[i] = Y[j]$

- Our sub-problems will be finding LCS’s of prefixes to $X$ and $Y$.
- Let $C[i,j] = \text{length\_of\_LCS}(X_i, Y_j)$

Then $C[i,j] = 1 + C[i-1,j-1]$.

because $\text{LCS}(X_i, Y_j) = \text{LCS}(X_{i-1}, Y_{j-1})$ followed by $A$

\[\text{Notation: denote this prefix ACGC by } Y_4\]
Two cases

Case 2: \(X[i] \neq Y[j]\)

Notation: denote this prefix \(ACGC\) by \(Y_4\)

- Our sub-problems will be finding LCS’s of prefixes to \(X\) and \(Y\).
- Let \(C[i,j] = \text{length}_\text{of}_\text{LCS}(X_i, Y_j)\)

Then \(C[i,j] = \max\{ C[i-1,j], C[i,j-1] \}\).
- either \(\text{LCS}(X_i,Y_j) = \text{LCS}(X_{i-1},Y_j)\) and \text{T} is not involved,
- or \(\text{LCS}(X_i,Y_j) = \text{LCS}(X_i,Y_{j-1})\) and \text{A} is not involved,
Recursive formulation of the optimal solution

\[ C_{i,j} = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C_{i-1,j-1} + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\
\max\{ C_{i,j-1}, C_{i-1,j} \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0
\end{cases} \]

Case 0

Case 1

Case 2
Recipe for applying Dynamic Programming

• **Step 1:** Identify optimal substructure.
• **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
• **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
• **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
• **Step 5:** If needed, code this up like a reasonable person.
LCS DP OMG BBQ

• **LCS**(X, Y):
  - C[i,0] = C[0,j] = 0 for all i = 1,...,m, j=1,...n.
  - For i = 1,...,m and j = 1,...,n:
    - If X[i] = Y[j]:
      - C[i,j] = C[i-1,j-1] + 1
    - Else:
      - C[i,j] = max{ C[i,j-1], C[i-1,j] }

\[
C[i,j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\
\max\{ C[i,j-1], C[i-1,j] \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 
\end{cases}
\]
Example

\[
C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i - 1, j - 1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{ C[i, j - 1], C[i - 1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0
\end{cases}
\]
Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>Y</td>
<td>A</td>
<td>C</td>
<td>T</td>
<td>G</td>
</tr>
</tbody>
</table>

So the LCM of X and Y has length 3.

\[
C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i - 1, j - 1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{C[i, j - 1], C[i - 1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0
\end{cases}
\]
Recipe for applying Dynamic Programming

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

\[
C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i - 1, j - 1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{ C[i, j - 1], C[i - 1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0
\end{cases}
\]
Example

\[ C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0
\end{cases} \]
Example

\[ C[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i - 1, j - 1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{ C[i, j - 1], C[i - 1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases} \]

- Once we’ve filled this in, we can work backwards.
### Example

<table>
<thead>
<tr>
<th>X</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>G</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>A</td>
<td>C</td>
<td>T</td>
<td>G</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Once we’ve filled this in, we can work backwards.

That 3 must have come from the 3 above it.

\[
C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0
\end{cases}
\]
Example

\[ C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0
\end{cases} \]

- Once we’ve filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

This 3 came from that 2 – we found a match!
Example

- Once we’ve filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

\[
C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i - 1, j - 1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\
\max\{C[i, j - 1], C[i - 1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 
\end{cases}
\]

That 2 may as well have come from this other 2.
Example

\[ C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 
\end{cases} \]

• Once we’ve filled this in, we can work backwards.
• A diagonal jump means that we found an element of the LCS!
Example

\[
\begin{array}{|c|c|c|c|c|}
\hline
A & C & T & G \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|}
\hline
0 & 0 & 0 & 0 & 0 \\
\hline
0 & 1 & 1 & 1 & 1 \\
\hline
0 & 1 & 2 & 2 & 2 \\
\hline
0 & 1 & 2 & 2 & 3 \\
\hline
0 & 1 & 2 & 2 & 3 \\
\hline
0 & 1 & 2 & 2 & 3 \\
\hline
\end{array}
\]

- Once we’ve filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

\[
C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i - 1, j - 1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{C[i, j - 1], C[i - 1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 
\end{cases}
\]
### Example

<table>
<thead>
<tr>
<th></th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A C G G A</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A C T G</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
</tbody>
</table>

\[ C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i - 1, j - 1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{C[i, j - 1], C[i - 1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 
\end{cases} \]

- Once we’ve filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

This is the LCS!
This gives an algorithm to recover the actual LCS not just its length

• See lecture notes for pseudocode
• It runs in time $O(n + m)$
  • We walk up and left in an $n$-by-$m$ array
  • We can only do that for $n + m$ steps.
• So actually recovering the LCS from the table is much faster than building the table was.
• We can find $\text{LCS}(X,Y)$ in time $O(mn)$. 
Recipe for applying Dynamic Programming

• **Step 1:** Identify optimal substructure.
• **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
• **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
• **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
• **Step 5:** If needed, code this up like a reasonable person.
This pseudocode actually isn’t so bad

• If we are only interested in the length of the LCS:
  • Since we go across the table one-row-at-a-time, we can only keep two rows if we want.
  • If we want to recover the LCS, we need to keep the whole table.

• Can we do better than $O(mn)$ time?
  • A bit better.
    • By a log factor or so.
  • But doing much better (polynomially better) is an open problem!
    • If you can do it let me know :D
What have we learned?

• We can find LCS(X,Y) in time $O(nm)$
  • if $|Y| = n$, $|X| = m$

• We went through the steps of coming up with a dynamic programming algorithm.
  • We kept a 2-dimensional table, breaking down the problem by decrementing the length of $X$ and $Y$. 
Example 2: 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?

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>🐢</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>🔆</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>🍉</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>🌮</td>
<td>3</td>
<td>13</td>
</tr>
</tbody>
</table>

  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?

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>🔆</td>
<td>11</td>
<td>35</td>
</tr>
<tr>
<td>🍉</td>
<td>4</td>
<td>13</td>
</tr>
</tbody>
</table>

  Total weight: 9
  Total value: 35
Some notation

Item:
- Turtle
- Lightbulb
- Watermelon
- Fire truck

Weight:
- $w_1$
- $w_2$
- $w_3$
- $\ldots$
- $w_n$

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.

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 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.
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 $x$.

$$K[x] = \max_i \left\{ \begin{array}{c} \text{The maximum is over all } i \text{ so that } w_i \leq x. \\ \text{Optimal way to fill the smaller knapsack} \\ \text{The value of item } i. \end{array} \right\} + v_i$$

• (And $K[x] = 0$ if the maximum is empty).
  • That is, there are no $i$ so that $w_i \leq x$
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**\((W, n, \text{weights}, \text{values})\):
  • \(K[0] = 0\)
  • **for** \(x = 1, \ldots, W\):
    • \(K[x] = 0\)
    • **for** \(i = 1, \ldots, n\):
      • **if** \(w_i \leq x\):
        • \(K[x] = \max\{ K[x], K[x - w_i] + v_i \} \)
  • **return** \(K[W]\)

Running time: \(O(nW)\)

Why does this work?
Because our recursive relationship makes sense.
Can we do better?

- We only need \( \log(W) \) bits to write down the input \( W \) and to write down all the weights.

- 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**($W, n, \text{weights, values}$):
  - $K[0] = 0$
  - **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** $K[W]$
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

UnboundedKnapsack\((W, n, \text{weights, values})\):

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

<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: 4
Example

UnboundedKnapsack($W, n, \text{weights}, \text{values}$):

- $K[0] = 0$
- $\text{ITEMS}[0] = \emptyset$
- 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:
      - $\text{ITEMS}[x] = \text{ITEMS}[x - w_i] \cup \{ \text{item } i \}$
- return $\text{ITEMS}[W]$

ITEMS

<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>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ITEMS[1] = ITEMS[0] +

Item:  

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

Capacity: 4
Example

UnboundedKnapsack($W, n, \text{weights}, \text{values}$):
- $K[0] = 0$
- $ITEMS[0] = \emptyset$
- 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 $ITEMS[W]$

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

Item:
- Turtle
- Lightbulb
- Watermelon

Capacity: 4
Example

UnboundedKnapsack($W$, $n$, weights, values):
• $K[0] = 0$
• ITEMS[0] = Ø
• 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] U { item i }
• return ITEMS[W]

ITEMS[2] = ITEMS[0] + □

Item:

<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

0 1 2 3 4
K

0 1 4
ITEMS

□ □ □ □ □
Example

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 $ITEMS[W]$

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

Capacity: 4
Example

UnboundedKnapsack\((W, n, weights, values)\):
- \(K[0] = 0\)
- \(ITEMS[0] = \emptyset\)
- 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:
      - \(ITEMS[x] = ITEMS[x - w_i] \cup \{ \text{item } i \} \)
- return \(ITEMS[W]\)

<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></td>
</tr>
</tbody>
</table>

| ITEMS | | | | | |
|-------|---|---|---|---|
| Turtle | Light Bulb | Watermelon |   |   |

\(ITEMS[3] = ITEMS[0] + \) Watermelon

<table>
<thead>
<tr>
<th>Item:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turtle</td>
</tr>
<tr>
<td>Light Bulb</td>
</tr>
<tr>
<td>Watermelon</td>
</tr>
</tbody>
</table>

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

Capacity: 4
Example

UnboundedKnapsack\((W, n, \text{weights, values})\):
- \(K[0] = 0\)
- \(\text{ITEMS}[0] = \emptyset\)
- 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:
      - \(\text{ITEMS}[x] = \text{ITEMS}[x - w_i] \cup \{\text{item } i\}\)
  - return \(\text{ITEMS}[W]\)

\[
\begin{array}{c|c|c|c|c|c}
\text{K} & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 4 & 6 & 7 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{ITEMS} & \text{Turtle} & \text{Lightbulb} & \text{Watermelon} \\
\hline
0 & \text{Empty} & \text{Empty} & \text{Empty} \\
1 & \text{Empty} & \text{Empty} & \text{Empty} \\
2 & \text{Turtle} & \text{Empty} & \text{Empty} \\
3 & \text{Turtle} & \text{Lightbulb} & \text{Empty} \\
4 & \text{Turtle} & \text{Lightbulb} & \text{Watermelon} \\
\end{array}
\]

\[
\text{ITEMS}[4] = \text{ITEMS}[3] + \text{Turtle}
\]

Item:
- Turtle
- Lightbulb
- Watermelon

Weight:
- 1
- 2
- 3

Value:
- 1
- 4
- 6

Capacity: 4
• 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] $\cup$ \{item i\}
  • return ITEMS[W]


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

<table>
<thead>
<tr>
<th>Item:</th>
<th>Weight:</th>
<th>Value:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>35</td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>Item:</th>
<th>Weight:</th>
<th>Value:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $j$
Two cases

- **Case 1**: Optimal solution for \( j \) items does not use item \( j \).
- **Case 2**: Optimal solution for \( j \) items does use item \( j \).
Two cases

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

- Then this is an optimal solution for $j-1$ 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:
  - 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 $j = 1,\ldots,n$:
      • $K[x,j] = K[x, j-1]$ Case 1
      • if $w_j \leq x$:
        • $K[x,j] = \max\{ K[x,j], K[x - w_j, j-1] + v_j \}$ Case 2
    • return $K[W,n]$

Running time $O(nW)$
### 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>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></td>
<td></td>
</tr>
<tr>
<td>x=2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x=3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Capacity:** 3
Example

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

1. $K[x,0] = 0$ for all $x = 0,...,W$
2. $K[0,i] = 0$ for all $i = 0,...,n$
3. For $x = 1,...,W$: 
   - For $j = 1,...,n$:
     a. $K[x,j] = K[x, j-1]$
     b. If $w_j \leq x$:
        i. $K[x,j] = \max\{ K[x,j], K[x - w_j, j-1] + v_j \}$
4. 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>Item:</th>
<th>Weight:</th>
<th>Value:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Zero-One-Knapsack($W, n, w, v$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K[x,0] = 0$ for all $x = 0, ..., W$</td>
</tr>
<tr>
<td>$K[0,i] = 0$ for all $i = 0, ..., n$</td>
</tr>
<tr>
<td>for $x = 1, ..., W$:</td>
</tr>
<tr>
<td>for $j = 1, ..., n$:</td>
</tr>
<tr>
<td>$K[x,j] = K[x, j-1]$</td>
</tr>
<tr>
<td>if $w_j \leq x$:</td>
</tr>
<tr>
<td>$K[x,j] = \max{ K[x,j], K[x - w_j, j-1] + v_j }$</td>
</tr>
</tbody>
</table>

- return $K[W,n]$
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>0</td>
<td>0</td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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

**Current entry**

**Relevant previous entry**
### 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>0</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>

- **Current entry**
- **relevant previous entry**

**Item:**

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

**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

<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>

#### 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 $x = 0,...,W$
  • $K[0,i] = 0$ 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 $K[W,n]$

Example

<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>1</td>
</tr>
<tr>
<td>x=3</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Item:
- Weight:
  - Turtle: 1
  - Lamp: 2
  - Watermelon: 3
- Value:
  - Turtle: 1
  - Lamp: 4
  - Watermelon: 6

Capacity: 3

current entry
relevant previous entry
## 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

<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>4</td>
<td></td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Current Entry

- current entry
- relevant previous entry

### 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></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

- **current entry**
- **relevant previous entry**
### Example

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

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

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

<table>
<thead>
<tr>
<th>j=3</th>
<th>x=0</th>
<th>x=1</th>
<th>x=2</th>
<th>x=3</th>
</tr>
</thead>
<tbody>
<tr>
<td></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 \( x = 0, ..., W \)
- \( K[0,i] = 0 \) 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 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></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>

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

- **Current entry**
- **Relevant previous entry**

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

\[
\text{for } x = 1,\ldots,W: \quad \text{for } j = 1,\ldots,n:
\]

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

\[
\text{return } K[W,n]
\]

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>j=1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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></td>
</tr>
</tbody>
</table>

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

---

**Legend:**
- Current entry
- Relevant previous entry
### 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 \( K[W,n] \)

**Items:**
- Weight: 1
- Value: 1

**Current entry:**
- Turtle

**Relevant previous entry:**
- Watermelon

**Capacity:** 3
Example

```
<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>4</td>
</tr>
</tbody>
</table>
```

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

- **Capacity:** 3

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

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.

You do this one! (We did it on the slide in the previous example, just not in the pseudocode!)
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!
Example 3: Independent Set
if we still have time

An independent set is a set of vertices so that no pair has an edge between them.

• Given a graph with weights on the vertices...
  - What is the independent set with the largest weight?
Actually this problem is NP-complete. So we are unlikely to find an efficient algorithm.

- But if we also assume that the graph is a tree...

Problem:
find a maximal independent set in a tree (with vertex weights).
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

• **Subtrees** are a natural candidate.

• There are **two cases**:
  1. The root of this tree is in a not in a maximal independent set.
  2. Or it is.
Case 1:
the root is **not** in an maximal independent set

• Use the optimal solution from **these smaller problems**.
Case 2:
the root is in an maximal independent set

• Then its children can’t be.
• Below that, use the optimal solution from these smaller subproblems.
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 formulation: try 1

• Let $A[u]$ be the weight of a maximal independent set in the tree rooted at $u$.

• $A[u] =$
  
  $\max \begin{cases} 
  \text{weight}(u) + \sum_{v \in u.\text{grandchildren}} A[v] \\
  \sum_{v \in u.\text{children}} A[v] 
  \end{cases}$

When we implement this, how do we keep track of this term?
Recursive formulation: try 2

Keep two arrays!

• Let $A[u]$ be the weight of a maximal independent set in the tree rooted at $u$.

• Let $B[u] = \sum_{v \in u.\text{children}} A[v]$

• $A[u] = \max \left\{ \sum_{v \in u.\text{children}} A[v], \text{weight}(u) + \sum_{v \in u.\text{children}} B[v] \right\}$
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.
A top-down DP algorithm

**MIS_subtree(u):**
- **if** u is a leaf:
  - A[u] = weight(u)
  - B[u] = 0
- **else:**
  - for v in u.children:
    - MIS_subtree(v)
  - \( A[u] = \max \{ \sum_{v \in u \text{.children}} A[v], \text{weight}(u) + \sum_{v \in u \text{.children}} B[v] \} \)
  - \( B[u] = \sum_{v \in u \text{.children}} A[v] \)

**MIS(T):**
- MIS_subtree(T.root)
- return A[T.root]

**Running time?**
- We visit each vertex once, and at every vertex we do O(1) work:
  - Make a recursive call
  - Look stuff up in tables
- Running time is O(|V|)
Why is this different from divide-and-conquer?
That’s always worked for us with tree problems before...

• **MIS_subtree(u):**
  • **if** u is a leaf:
    • **return** weight(u)
  • **else:**
    • **for** v in u.children:
      • MIS_subtree(v)
    • **return** \( \max \{ \sum_{v \in \text{children}} \text{MIS_subtree}(v), \text{weight}(u) + \sum_{v \in \text{grandchildren}} \text{MIS_subtree}(v) \} \)

• **MIS(T):**
  • **return** MIS_subtree(T.root)

This is exactly the same pseudocode, except we’ve ditched the table and are just calling MIS_subtree(v) instead of looking up A[v] or B[v].
Why is this different from divide-and-conquer?
That’s always worked for us with tree problems before...

How often would we ask about the subtree rooted here?

Once for **this node** and once for **this one**.

But we then ask about **this node** twice, here and here.

This will blow up exponentially without using dynamic programming to take advantage of *overlapping subproblems*.
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**.

You do this one!
What have we learned?

• We can find maximal independent sets in trees in time $O(|V|)$ using dynamic programming!

• For this example, it was natural to implement our DP algorithm in a top-down way.
Recap

• Today we saw examples of how to come up with dynamic programming algorithms.
  • Longest Common Subsequence
  • Knapsack two ways
  • (If time) maximal independent set in trees.

• There is a **recipe** for dynamic programming algorithms.
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.
Recap

• Today we saw examples of how to come up with dynamic programming algorithms.
  • Longest Common Subsequence
  • Knapsack two ways
  • (If time) maximal independent set in trees.

• There is a **recipe** for dynamic programming algorithms.

• Sometimes coming up with the right substructure takes some creativity
  • You’ll get lots of practice on Homework 6! 😊
Next week

- Greedy algorithms!

Before next time

- Pre-lecture exercise: Greed is good!