Lecture 13

More dynamic programming!

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

• HW6 due Wednesday!
• HW7 out Wednesday!
• FAQ: What’s the best way to prepare for the final?
  • Practice problems!
    • If Section/HW aren’t enough for you, there are plenty in Algorithms Illuminated and in CLRS (which is available for free via the Stanford library).
    • We’ll also be posting multiple practice finals soon.
  • When you are reading the book or (re)watching lectures or section, try to guess what comes next.
    • If we state a lemma, close the book or pause the video, and try to prove the lemma.
    • If we’ve seen the intuition for an algorithm, try to write down pseudocode.
  • Try the HW on your own before collaborating.
Question from last time

• Does Bellman-Ford/Floyd-Warshall work on undirected graphs?
  • Yes, just do:

  ![Diagram](image_url)

• What about negative edge weights? Does that mean we just can’t handle negative edge weights in undirected graphs?
  • That’s right.

  ![Diagram](image_url)

I can still walk back and forth forever, and shortest paths might not be defined!
Last time

• Not coding in an action movie.
Last time

• 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)

• Yet more examples of DP in Algorithms Illuminated!
  • Weighted Independent Set in Paths
  • Sequence Alignment
  • Optimal Binary Search Trees
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?

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

DNA: AGCCTAAAGGGCTACCTAGCTT

DNA: GACAGCCTACAAGCGTTAGCTTTG

AGCCTAAGGCTTTAGCTT
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:

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

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

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

Examples:
- $C[2,3] = 2$
- $C[4,4] = 3$
Optimal substructure ctd.

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

• Why is this a good choice?
  • As we will see, there’s some relationship between LCS’s of prefixes and LCS’s of the whole things.
  • These subproblems overlap a lot.
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.
Goal

• Write $C[i,j]$ in terms of the solutions to smaller sub-problems

$$C[i,j] = \text{length_of_LCS}(X_i, Y_j)$$
Two cases
Case 1: $X[i] = Y[j]$

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

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 $A$
Two cases

Case 2: $X[i] \neq Y[j]$

- 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 $C[i,j] = \max\{ C[i-1,j], C[i,j-1] \}$.

- either $\text{LCS}(X_i, Y_j) = \text{LCS}(X_{i-1}, Y_j)$ and $T$ is not involved,
- or $\text{LCS}(X_i, Y_j) = \text{LCS}(X_i, Y_{j-1})$ and $A$ is not involved,
- (maybe both are not involved, that’s covered by the “or”).
Recursive formulation of the optimal solution

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

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

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

Running time: \(O(nm)\)
# Example

![Example Diagram](image)

The cost matrix $C[i, j]$ is defined as:

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

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

\[
\begin{array}{c c c c c}
A & C & T & G \\
A & 0 & 1 & 1 & 0 \\
C & 0 & 1 & 2 & 2 \\
G & 0 & 1 & 2 & 2 \\
A & 0 & 1 & 2 & 2 \\
\end{array}
\]

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

- 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

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!

The recurrence relation for the matrix $C[i,j]$ is defined as follows:

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

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

<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></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!
Once we’ve filled this in, we can work backwards. A diagonal jump means that we found an element of the LCS!
Example

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

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

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

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

This is the LCS!
Finding an LCS

• Good exercise to write out pseudocode for what we just saw!
  • Or you can find it in CLRS.

• Takes time $O(mn)$ to fill the table

• Takes time $O(n + m)$ on top of that to recover the LCS
  • We walk up and left in an $n$-by-$m$ array
  • We can only do that for $n + m$ steps.

• Altogether, we can find $LCS(X,Y)$ 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.
Our approach actually isn’t so bad

• If we are only interested in the length of the LCS we can do a bit better on space:
  • 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>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></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>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></td>
<td></td>
<td>35</td>
</tr>
</tbody>
</table>

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

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

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

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

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

Why?
1 minute think
1 minute pair+share

43
Optimal substructure

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

  - Capacity $x$
  - Value $V$
  - Weight $w_i$
  - Value $v_i$

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

  - Capacity $x - w_i$
  - Value $V - v_i$

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 $x$.

$$K[x] = \max_i \{ \text{Optimal way to fill the smaller knapsack} + \text{The value of item } i \}$$

The maximum is over all $i$ so that $w_i \leq x$.

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

• (And $K[x] = 0$ if the maximum is empty).
  • That is, if there are no $i$ so that $w_i \leq x$
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*, *weights*, *values*):
  • **K**[0] = 0
  • **for** *x* = 1, …, *W*:
    • **K**[*x*] = 0
    • **for** *i* = 1, …, *n*:
      • **if** *w*[*i*] ≤ *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?

• 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**\((W, n, \text{weights}, \text{values})\):
  • \(K[0] = 0\)
  • **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** \(K[W]\)
Let’s write a bottom-up DP algorithm

- **UnboundedKnapsack** \((W, n, \text{weights, values})\):
  - \(K[0] = 0\)
  - \(\text{ITEMS}[0] = \emptyset\)
  - \(\text{for } x = 1, \ldots, W:\)
    - \(K[x] = 0\)
    - \(\text{for } i = 1, \ldots, n:\)
      - 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 \(\text{ITEMS}[W]\)
Example

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

- $K[0] = 0$
- ITEMS[0] = ∅
- for $x = 1, \ldots, W$:
  - $K[x] = 0$
  - for $i = 1, \ldots, n$:
    - if $w_i \leq x$:
      - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
    - if $K[x]$ was updated:
      - ITEMS[x] = ITEMS[x − $w_i$] U { item $i$ }
- return ITEMS[W]

ITEMS[1] = ITEMS[0] +

<table>
<thead>
<tr>
<th>Item</th>
<th>1</th>
<th>2</th>
<th>3</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: 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 \(\text{ITEMS}[W]\)

<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>ITEMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(\text{ITEMS}[2] = \text{ITEMS}[1] + \)

Item:

- Weight: 1
- Value: 1

- Weight: 2
- Value: 4

- Weight: 3
- Value: 6

Capacity: 4
Example

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

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

<table>
<thead>
<tr>
<th>ITEMS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

ITEMS[2] = ITEMS[0] +

Item:

- Weight: 1
- Value: 1

- Weight: 2
- Value: 4

- Weight: 3
- Value: 6

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 \(\text{ITEMS}[W]\)

\[
\begin{array}{c|c|c|c|c|c}
K & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 1 & 4 & 5 & & \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
\text{ITEMS} & & & & & \\
\hline
\text{Turtle} & \text{Light bulb} & \text{Light bulb} & & & \\
\end{array}
\]

\[
\text{ITEMS}[3] = \text{ITEMS}[2] + \text{Turtle}
\]
Example

UnboundedKnapsack(W, n, weights, values):
• K[0] = 0
• ITEMS[0] = ∅
• for x = 1, ..., W:
  • K[x] = 0
  • for i = 1, ..., n:
    • if w_i ≤ 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]

ITEMS[3] = ITEMS[0] + 🔵

<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>ITEMS</td>
<td>🐢</td>
<td>🕯️</td>
<td>🍉</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Item:
- 🐢: 1
- 🕯️: 2
- 🍉: 3

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 $ITEMS[W]$

Example

UnboundedKnapsack(W, n, weights, values):

- K[0] = 0
- ITEMS[0] = ∅
- for x = 1, ..., W:
  - K[x] = 0
  - for i = 1, ..., n:
    - if w_i ≤ 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]


Item:
- Turtles
- Bulbs
- Watermelon

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.
• 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>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</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 $j$

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

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

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

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

Let $K[x,j]$ be the 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*.

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

First *j* items

Weight $w_j$
Value $v_j$

Capacity $x$
Value $V$
Use only the first *j* items

What lower-indexed problem should we solve to solve this problem?
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
  
  Capacity \( x \)
  Value \( V \)

  Use only the first \( j \) items

  First \( j-1 \) items

  Capacity \( x - w_j \)
  Value \( V - v_j \)

  Use only the 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,...,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]

Running time \(O(nW)\)
Example

<table>
<thead>
<tr>
<th></th>
<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>
<td>0</td>
</tr>
<tr>
<td>x=1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x=2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x=3</td>
<td></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 \(w_j \leq x\):
        - \(K[x,j] = \max\{ K[x,j], K[x – w_j, j-1] + v_j \}\)
  - **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>
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 \( K[W,n] \)

current entry
relevant previous entry

<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>0</td>
<td></td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Item:
- Turtle
- Light Bulb
- Watermelon

Weight:
- 1
- 2
- 3

Value:
- 1
- 4
- 6

Capacity: 3
### Zero-One-Knapsack($W, n, w, v$):

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

---

**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>Bulb</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Watermelon</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

**Capacity: 3**

- **Weight:**
  - Current entry
  - Relevant previous entry

---

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

Current entry | Relevant previous entry
---|---

**Item:**
- Turtle: Weight: 1, Value: 1
- Light bulb: Weight: 2, Value: 4
- Watermelon: Weight: 3, Value: 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>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>

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

- **Capacity:** 3

#### Zero-One-Knapsack($W, n, w, v$):
- $K[x,0] = 0$ for all $x = 0, ..., W$
- $K[0,i] = 0$ 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** $K[W,n]$
### 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></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
- Lightbulb: 2
- Watermelon: 3

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

**Value:**
- Turtle: 1
- Lightbulb: 4
- Watermelon: 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 \( i = 0,...,n \)
- \textbf{for} \( x = 1,...,W \):
  - \textbf{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 \( K[W,n] \)

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

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></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 ≤ 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 w_j ≤ 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 $w_j \leq x$:
      - $K[x,j] = \max\{ K[x,j], K[x – w_j, j-1] + v_j \}$
- 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>0</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:**
  - Turtles: 1
  - Lights: 2
  - Watermelons: 3

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

- **Value:**
  - Turtles: 1
  - Lights: 4
  - Watermelons: 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 $w_j \leq x$
      - $K[x,j] = \max\{ K[x,j], K[x – w_j, j-1] + v_j \}$
- **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:**
- 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 $w_j \leq x$:
        - $K[x,j] = \max\{K[x,j], K[x - w_j, j-1] + v_j\}$
  - 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>1</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**
### 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**

- 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><strong>j=0</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>j=1</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>j=2</strong></td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td><strong>j=3</strong></td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

- **Weight:**
  - Item: 1
  - Value: 1

- **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

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

```
<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>6</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 $K[W,n]$  

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.

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 **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 $u$.

- $A[u] = \max \left\{ \text{weight}(u) + \sum_{v \in u.\text{grandchildren}} A[v], \sum_{v \in u.\text{children}} A[v] \right\}$

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

\[
A[u] = \max \left\{ \sum_{v \in u.\text{children}} A[v], \quad \text{weight}(u) + \sum_{v \in u.\text{children}} B[v] \right\}
\]
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{ ∑_{v \in u.children} A[v], weight(u) + ∑_{v \in u.children} B[v] }
    • B[u] = ∑_{v \in u.children} A[v]

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

Initialize global arrays A, B that we will use in all of the recursive calls.

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:
    • return \( \max\left\{ \sum_{v \in u.\text{children}} \text{MIS\_subtree}(v), \text{weight}(u) + \sum_{v \in u.\text{grandchildren}} \text{MIS\_subtree}(v) \right\} \)

• **MIS(T):**
  • return MIS\_subtree(T.root)
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 got some practice on HW6 and you’ll get more on HW7! 😊
  • For even more practice check out additional examples/practice problems in Algorithms Illuminated or CLRS or section!
Next week

• Greedy algorithms!

Before next time

• Pre-lecture exercise: Greed is good!