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.
Last time

• DP algorithms for single-source shortest path and all-pairs shortest path.
• Floyd-Warshall solves APSP in time $O(n^3)$.

There was a question: Can we do better?
• If the graph is sparse, yes:
  • eg, Johnson’s algorithm runs in time $O(nm + n^2\log(n))$.
• There are algorithms that run in time $O(n^3/\log^c(n))$ for any constant $c$:
  • [R. Williams, 2013]
• But it’s conjectured that there is no algorithm that runs in time $O(n^{3-\epsilon})$, for any $\epsilon > 0$. 

This won’t be on the final 😊
Today

• Examples of dynamic programming:
  • Longest common subsequence
  • Knapsack problem
    • Two versions!
  • Independent sets in trees
    • If we have time...
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: GACAGCCTACAAGCGTTAGCTTGA

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

AGCCTAAGCGCTTTAGCTT
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 longest common subsequence
  • 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 $C[i,j] = \text{length}_\text{of}_\text{LCS}(X_i, Y_j)$
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 $C[i,j] = \text{length} \_ \text{of} \_ \text{LCS}(X_i, Y_j)$.

### Notation

- Denote this prefix $ACGC$ by $Y_4$.

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

- 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 $A$. 

![Diagram showing case 1 with $X_i$ and $Y_j$ prefixes and notation for $ACGC$ by $Y_4$.]
Two cases

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

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

\[
\begin{align*}
\text{Notation: denote this prefix ACGC by } Y_4 \\
\text{These are not the same}
\end{align*}
\]

- 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,
Recursive formulation of the optimal solution

\[ X_0 \]

\[
\begin{align*}
Y_j &= A C G C T T T A \\
\end{align*}
\]

\[ X_i \]

\[
\begin{align*}
A & C G G A \\
A & C G C T T T A \\
A & C G G T T T A \\
\end{align*}
\]

\[ Y_j \]

\[
\begin{align*}
A & C G C T T T A \\
A & C G T T T T A \\
A & C G C T T T A \\
\end{align*}
\]

- **Case 0**
  - If \( i = 0 \) or \( j = 0 \)

- **Case 1**
  - If \( X[i] = Y[j] \) and \( i, j > 0 \)

- **Case 2**
  - If \( X[i] \neq Y[j] \) and \( i, j > 0 \)

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

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

Running time: \( O(nm) \)

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

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

Let's consider two DNA sequences:

- **X**: A C G G A
- **Y**: A C T G

We will construct a matrix `C[i, j]` where `C[i, j]` represents the number of possible nucleotide insertions or deletions required to align sequence X with sequence Y.

### Table:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>G</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>G</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

### Formula:

The value of `C[i, j]` is determined by the following conditions:

- If `i = 0` or `j = 0`, then `C[i, j] = 0`.
- If `X[i] = Y[j]` and `i, j > 0`, then `C[i, j] = C[i-1, j-1] + 1`.
- If `X[i] ≠ Y[j]` and `i, j > 0`, then `C[i, j] = \max\{C[i, j-1], C[i-1, j]\}`.

For example, `C[1, 1] = 1` because both X[1] and Y[1] are C.

Thus, the matrix `C` represents the optimal number of insertions or deletions required to align the two sequences X and Y.
Example

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

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

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

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 2 & 2 \\
0 & 1 & 2 & 2 & 3 \\
0 & 1 & 2 & 2 & 3 \\
0 & 1 & 2 & 2 & 3 \\
\end{array}
\]

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

- 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!

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

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

A diagonal jump means that we found an element of the LCS!

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

\[
C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 
\end{cases}
\]
Once we’ve filled this in, we can work backwards.

A diagonal jump means that we found an element of the LCS!

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

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

A diagonal jump means that we found an element of the LCS!

\[
C[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
C[i - 1, j - 1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\
\max\{C[i, j - 1], C[i - 1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0
\end{cases}
\]
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 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 $\text{LCS}(X,Y)$ 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</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: 10
• Unbounded Knapsack:
  • Suppose I have infinite copies of all of the items.
  • What’s the most valuable way to fill the knapsack?

- Weight: 6 2 4 3 11
- Value: 20 8 14 13 35
- Total weight: 10
- Total value: 42

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

- Weight: 6 2 4 3 11
- Value: 20 8 14 13 35
- Total weight: 9
- Total value: 35
Some notation

Item: 🐢💡🍉 🔥
Weight: $W_1$ $W_2$ $W_3$ \ldots $W_n$
Value: $V_1$ $V_2$ $V_3$ \ldots $V_n$
Capacity: $W$
Recipe for applying Dynamic Programming

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

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

First solve the problem for small knapsacks

Then larger knapsacks

Then larger knapsacks
Optimal substructure

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

\[
\text{Value } v_i
\]
\[
\text{Weight } w_i
\]

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

\[
\text{Value } V - v_i
\]
\[
\text{Capacity } x - w_i
\]

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 \( x \).

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

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

Optimal way to fill the smaller knapsack

The value of item \( i \).

• (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($W$, $n$, weights, 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**($W$, $n$, weights, 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($W$, $n$, 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 $ITEMS[W]$

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

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

<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>4</td>
<td>6</td>
</tr>
</tbody>
</table>

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>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

ITEMS[1] = ITEMS[0] + Turtle
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 $ITEMS[W]$

<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

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

<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

<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>0</td>
<td>1</td>
<td>4</td>
<td>5</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>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


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

### Item:
- Weight: 1
- Value: 1

### Item:
- Weight: 2
- Value: 4

### Item:
- 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 \( \text{ITEMS}[W] \)

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

**ITEMS[3] = ITEMS[0] +**

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


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

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>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITEMS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>Turtle</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>Light</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</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
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*.

  First *j* items

  First *j-1* items

  Use only the first *j* items

  Use only the first *j-1* items.
Two cases

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

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

- First \( j \) items
- First \( j-1 \) items
- Weight \( w_j \)
- Value \( v_j \)
- Capacity \( x \)
- Value \( V \)
- Use only the first \( j \) items
- Capacity \( x - w_i \)
- Value \( V - v_i \)
- 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 \( x = 0,\ldots,W \)
  - \( K[0,i] = 0 \) for all \( i = 0,\ldots,n \)
  - for \( x = 1,\ldots,W \):
    - for \( j = 1,\ldots,n \):
      - \( K[x,j] = K[x, j-1] \) \hspace{1cm} \text{Case 1}
      - if \( w_j \leq x \):
        - \( K[x,j] = \max\{ K[x,j], K[x - w_j, j-1] + v_j \} \) \hspace{1cm} \text{Case 2}
    - return \( K[W,n] \)

Running time \( O(nW) \)
### Example

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

#### Diagram

**Current entry**

- Turtle
- Light bulb
- Watermelon

**Relevant previous entry**

- Turtle
- Light bulb

<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

---

- **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** $j = 1,\ldots,n$:
      - $K[x,j] = K[x-1,j]$  
      - **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 $x = 0,\ldots,W$
• $K[0,i] = 0$ 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>Current</th>
<th>Relevance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Turtle</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Lamp</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Watermelon</td>
<td>0</td>
</tr>
</tbody>
</table>

Weight: 1 2 3
Value: 1 4 6
Capacity: 3
### Example

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

- $K[x,0] = 0$ for all $x = 0,...,W$
- $K[0,i] = 0$ for all $i = 0,...,n$
- for $x = 1,...,W$:
  - for $j = 1,...,n$:
    - $K[x,j] = K[x, j-1]$
    - if $w_j \leq x$:
      - $K[x,j] = \max\{ K[x,j], K[x-w_j, j-1] + v_j \}$
- 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>0</td>
<td>0</td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<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>
<tr>
<td>Capacity:</td>
<td>3</td>
<td></td>
<td></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 \( K[W,n] \)
### 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**

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

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

#### Relevant Previous Entry

<table>
<thead>
<tr>
<th>Current Entry</th>
<th>Relevant Previous Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="turtle.png" alt="Turtle" /></td>
<td><img src="lightbulb.png" alt="Lightbulb" /></td>
</tr>
<tr>
<td><img src="watermelon.png" alt="Watermelon" /></td>
<td><img src="turtle.png" alt="Turtle" /></td>
</tr>
</tbody>
</table>

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

**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></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:**
- 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 $K[W,n]$. 
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 \( K[W,n] \)
### 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>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></td>
</tr>
</tbody>
</table>

**Item:**

<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>

**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>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></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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 \(K[W,n]\)

- \(W = 3\)
- \(n = 5\)
- \(w = [1, 2, 3, 1, 4]\)
- \(v = [1, 4, 6]\)

<table>
<thead>
<tr>
<th></th>
<th>Item:</th>
<th>Weight:</th>
<th>Value:</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j=1</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>j=2</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>j=3</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Current entry: tortoise
Relevant previous entry: light bulb

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 \(x = 0, \ldots, W\)
- \(K[0,i] = 0\) 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 \(K[W,n]\)
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>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>
<tr>
<td>j=1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>j=2</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>j=3</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

**Item:**
- Turtles
- Light bulbs
- Watermelon

**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>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 \( K[W,n] \)

**Item:**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>Weight:</strong></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>Value:</strong></td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td><strong>Capacity:</strong></td>
<td>3</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 $K[W,n]$

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>

Item:
- Turtle: 1
- Lightbulb: 2
- Watermelon: 3
- 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>
<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 $x = 0,...,W$
  - $K[0,i] = 0$ 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 $K[W,n]$

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 $u$.

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

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

• $A[u] = \max \begin{cases} 
\sum_{v \in u.\text{children}} A[v] \\
\text{weight}(u) + \sum_{v \in u.\text{children}} B[v]
\end{cases}$
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:
    • $A[u] = \text{weight}(u)$
    • $B[u] = 0$
  • 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 $A[T.\text{root}]$

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

• **MIS**\(_{(T)}\):
  • return **MIS_subtree**\(_{(T.\text{root})}\)
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 bottom-up 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!