# Lecture 13

***More dynamic programming!***

Longest Common Subsequences, Knapsack two ways

(and if time independent sets in trees).

# Announcements

- HW5 due tomorrow!

# Last time

Dynamic Programming!

- Not coding in an action movie.

# Last time

Dynamic Programming!

- 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.

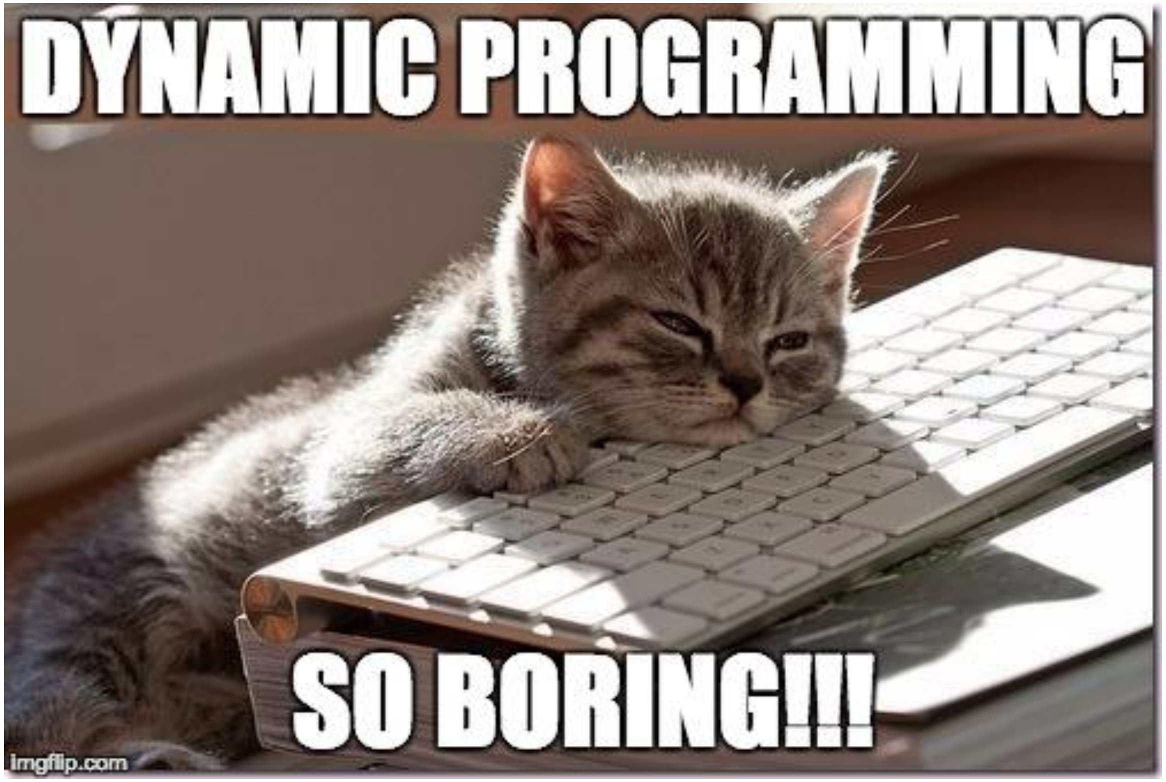
# Today

- Examples of dynamic programming:
    1. Longest common subsequence
    2. Knapsack problem
        - Two versions!
    3. Independent sets in trees
        - If we have time!
        - If not, that’s okay, and 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

from your pre-lecture exercise!

- How similar are these two species?

DNA:
AGCCCTAAGGGCTACCTAGCTT

DNA:
GACAGCCTACAAGCGTTAGCTTG

# Longest Common Subsequence

from your pre-lecture exercise!

- How similar are these two species?

DNA:

AGCCCTAAGGGCTACCTAGCTT

DNA:

GACAGCCTACAAGCGTTAGCTTG

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

AGCCTAAGCTTAGCTT

# Longest Common Subsequence

from your pre-lecture exercise!

- 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…

```
DN0a22a660:~ mary$ cat file1
A
B
C
D
E
F
G
H
DN0a22a660:~ mary$ cat file2
A
B
D
F
G
H
I
DN0a22a660:~ mary$ diff file1 file2
3d2
< C
5d3
< E
8a7
> I
DN0a22a660:~ mary$
```

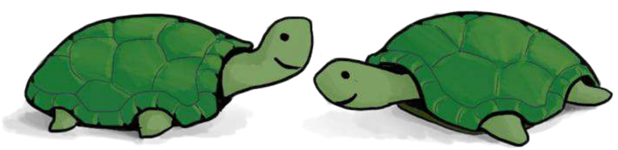
# Recipe for applying Dynamic Programming

- **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

- You thought about this on your pre-lecture exercise!
- Any thoughts?

Share what you thought of with your neighbor!

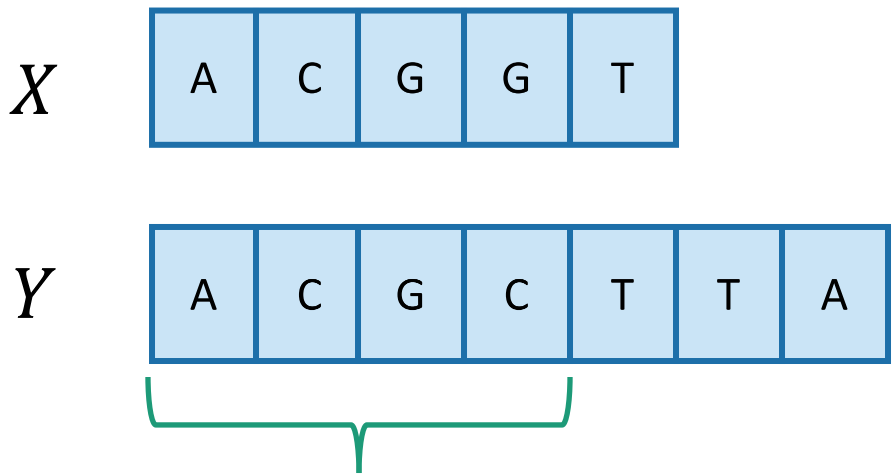


**What we want:**

- An optimal solution to the big problem builds on optimal solutions to the sub-problems
- We can use the same sub-problem again and again

# Step 1: Optimal substructure

Prefixes:



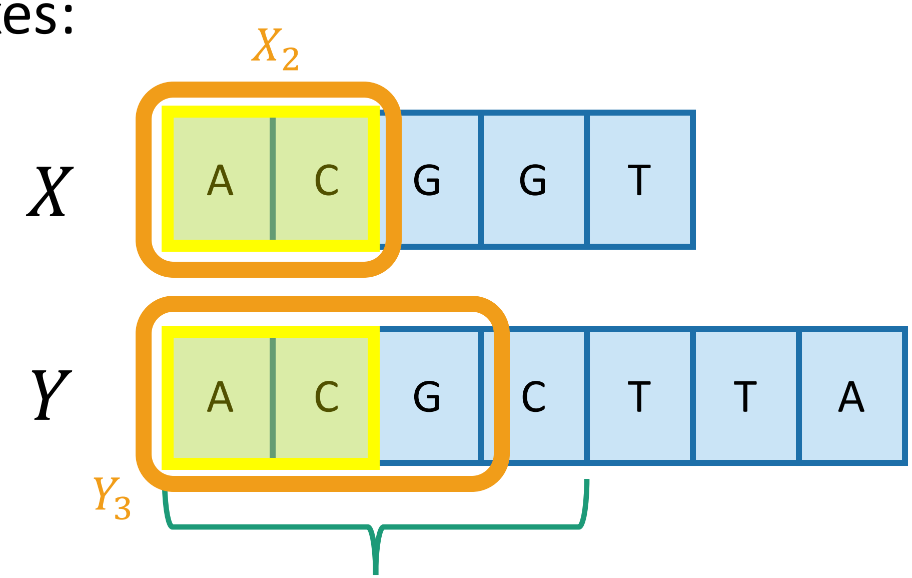

**Notation**: denote this prefix **ACGC** by $Y_4$

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] be the length of the LCS between $X_i$ and $Y_j$

# Step 1: Optimal substructure

Prefixes:

Example: C[2,3] = 2




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

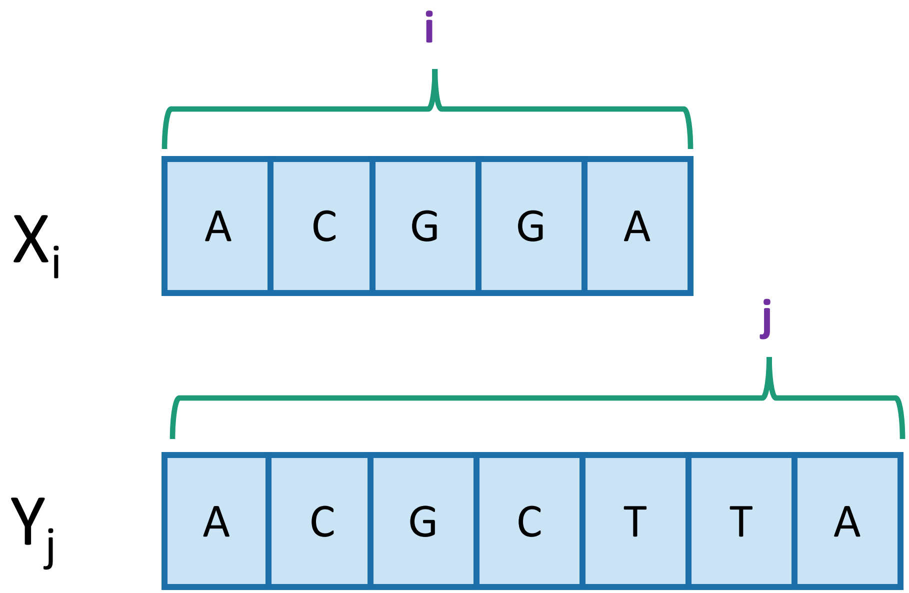
- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] be the length of the LCS between $X_i$ and $Y_j$

# 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

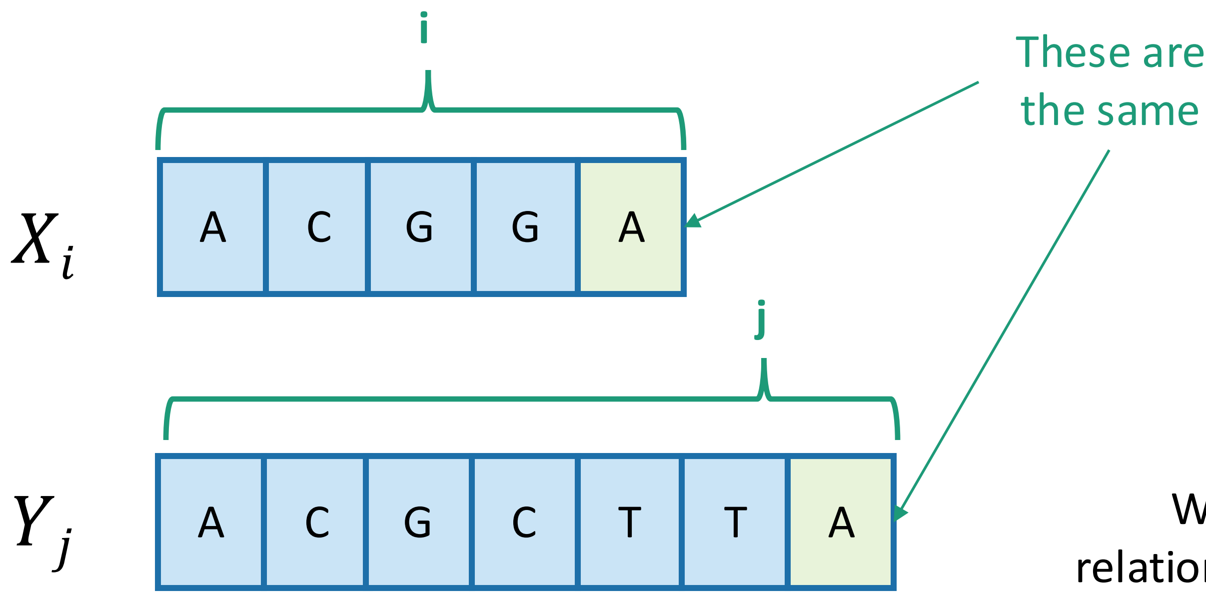



C[i,j] = length of LCS between $X_i$ and $Y_j$

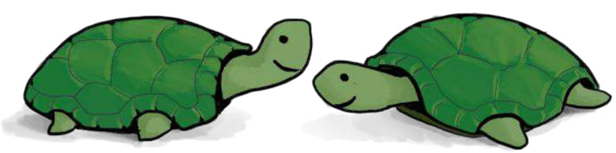
# Two cases

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- C[i,j] = length of LCS($X_i$,$Y_j$)

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




What is the relationship between C[i,j] and C[smaller] in this case?

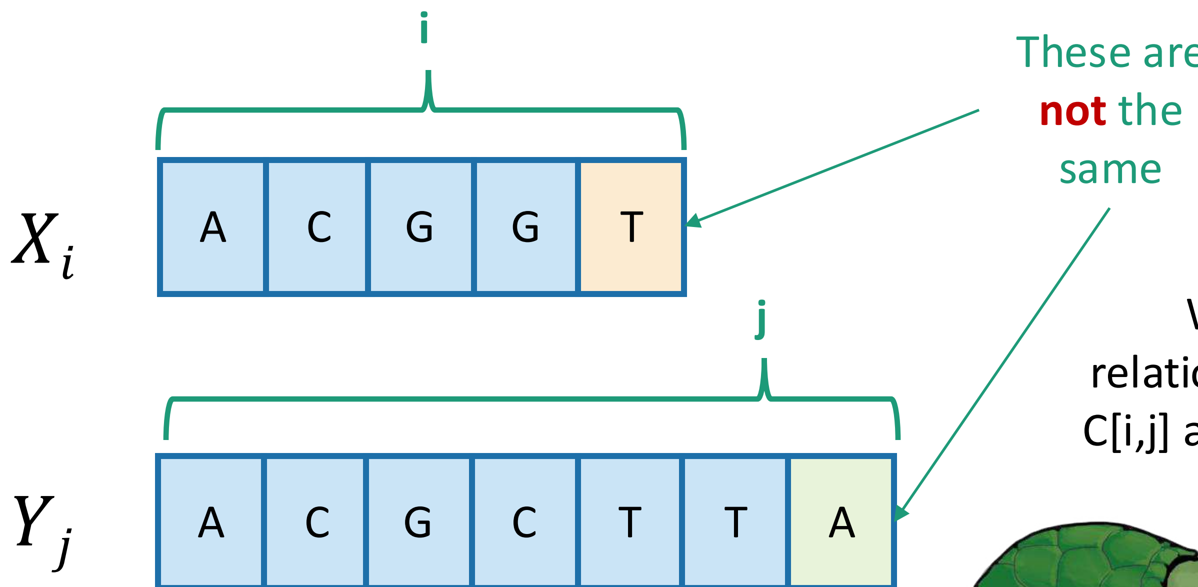


- Then C[i,j] = 1 + C[i-1,j-1].
  - because LCS($X_i$, $Yj$) = (LCS($X_{i-1}$, $Y_{j-1}$) followed by $\boxed{A}$ )

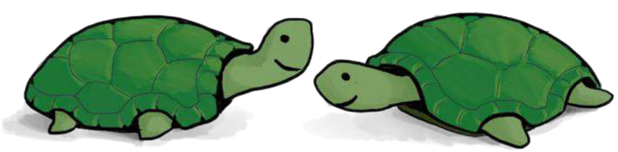
# Two cases

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- C[i,j] = length of LCS($X_i$,$Y_j$)

## Case 2: X[i] ≠Y[j]




What is the relationship between C[i,j] and C[smaller] in this case?



- Then C[i,j] = max{ C[i-1,j], C[i,j-1] }.
  - either LCS($X_i$, $Yj$) = LCS($X_{i-1}$, $Yj$) and $\boxed{\text{T}}$ is not involved,
  - or LCS($X_i$, $Yj$) = LCS($X_i$, $Y_{j-1}$) and $\boxed{\text{A}}$ is not involved,
  - (maybe both are not involved, that's covered by the "or").

# Recursive formulation
## of the optimal solution

$X_0$ | |
|---|

$Y_j$

| | | | | | | |
|---|---|---|---|---|---|---|
| A | C | G | C | T | T | A |

Case 0

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

Case 1

$X_i$

| | | | | |
|---|---|---|---|---|
| A | C | G | G | A |

$Y_j$

| | | | | | | |
|---|---|---|---|---|---|---|
| A | C | G | C | T | T | A |

Case 2

$X_i$

| | | | | |
|---|---|---|---|---|
| A | C | G | G | T |

$Y_j$

| | | | | | | |
|---|---|---|---|---|---|---|
| A | C | G | C | T | T | A |

# 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,…,m, j=0,…n.
  - **For** i = 1,…,m and j = 1,…,n:
    - **If** X[i] = Y[j]:
      - C[i,j] = C[i-1,j-1] + 1
    - **Else:**
      - C[i,j] = max{ C[i,j-1], C[i-1,j] }
  - Return C[m,n]

**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

| X | | | | | |
|---|---|---|---|---|---|
| | A | C | G | G | A |

| Y | | | | |
|---|---|---|---|---|
| | A | C | T | G |

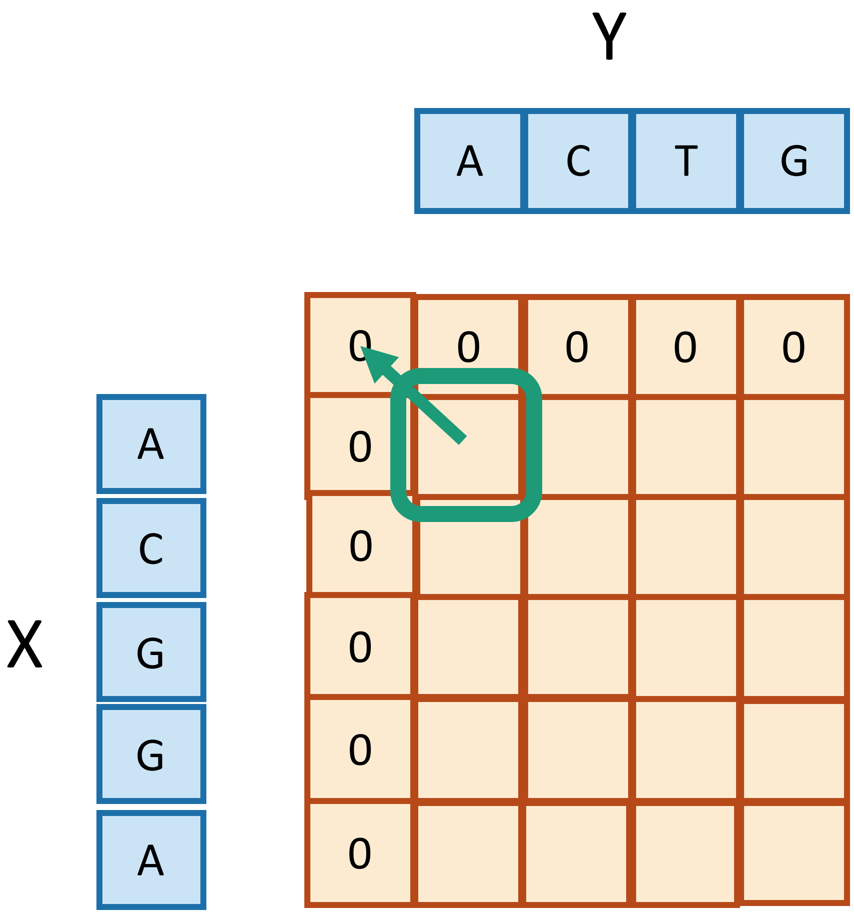
| | | Y: A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| X: A | 0 | | | | |
| C | 0 | | | | |
| G | 0 | | | | |
| G | 0 | | | | |
| A | 0 | | | | |

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

# Example

X: A C G G A

Y: A C T G

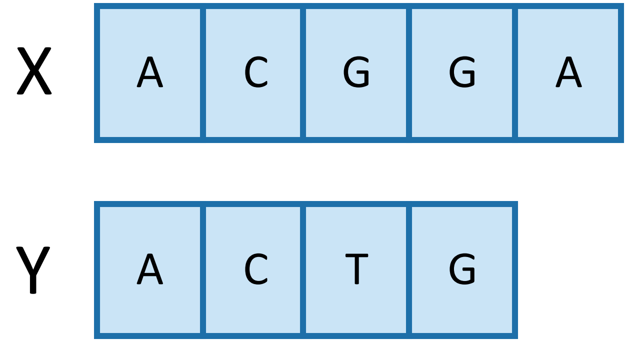


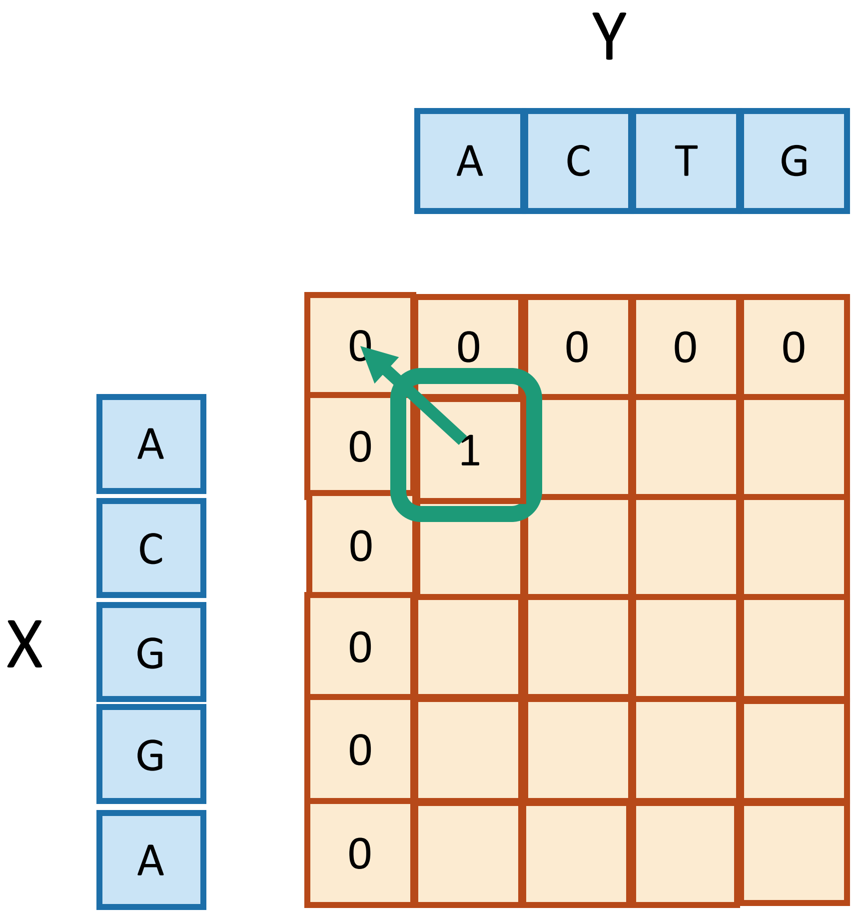

$X[i] = Y[j]$

So we want
$C[i-1, j-1] + 1$

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

# Example



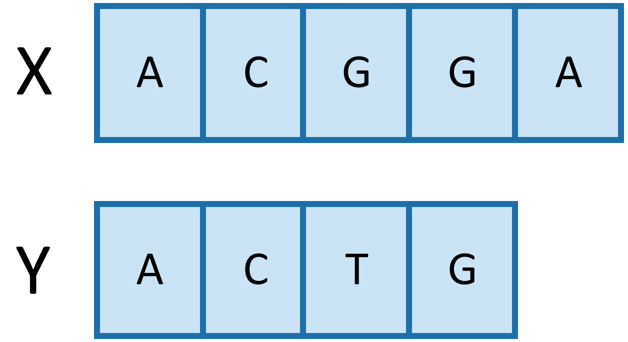

$$X[i] = Y[j]$$

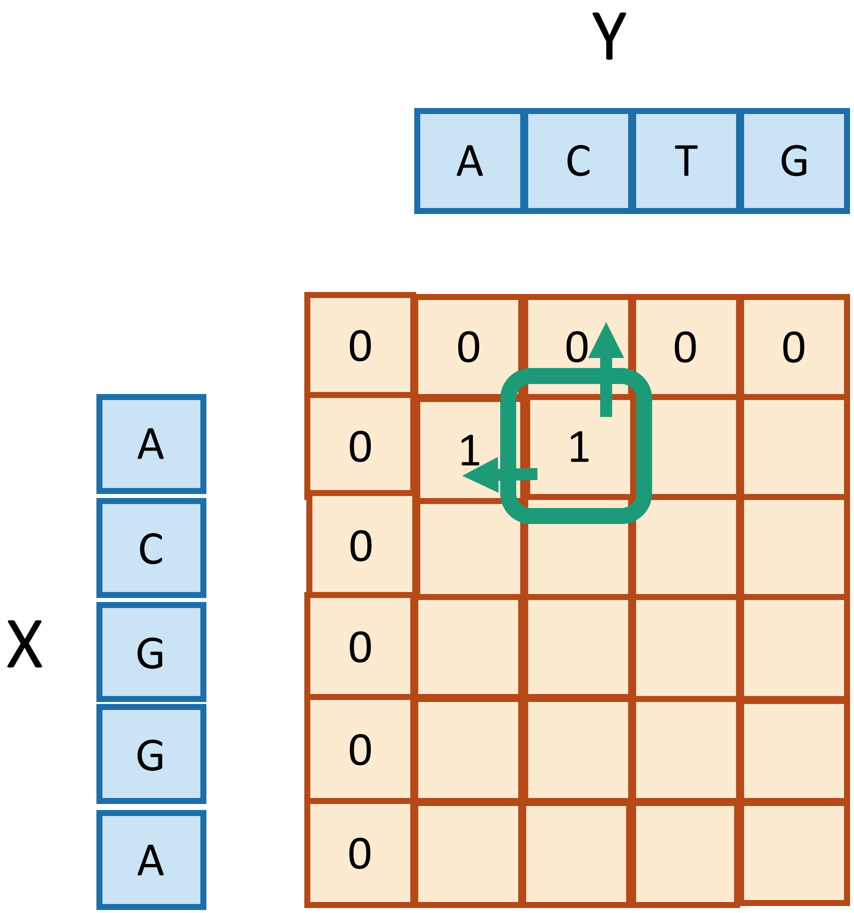
So we want

$$C[i-1, j-1] + 1$$

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

# Example



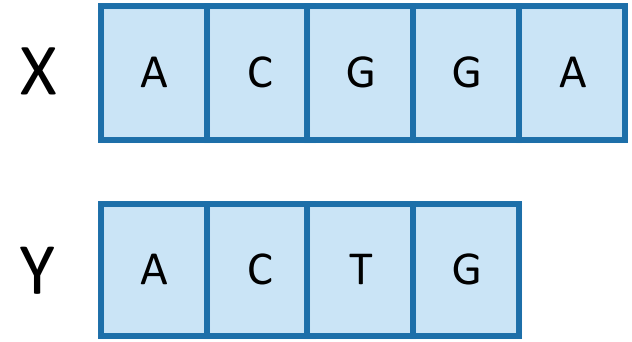

$X[i] \neq Y[j]$

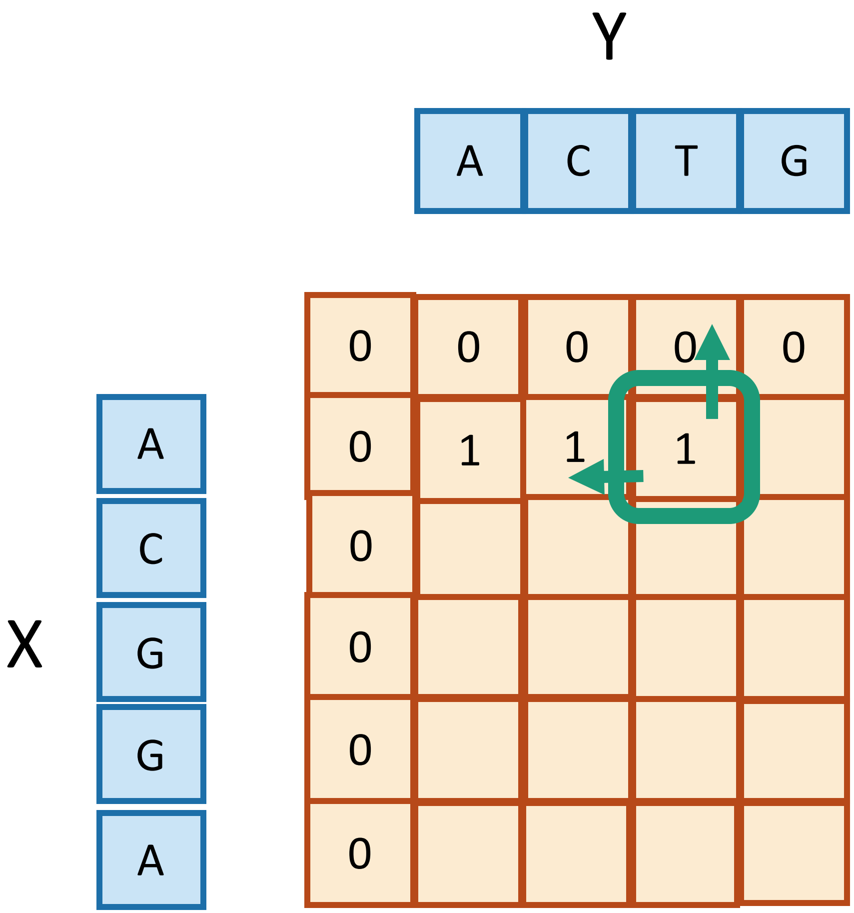
So we want

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

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

# Example




$X[i] \neq Y[j]$

So we want

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

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

# Example

| | A | C | G | G | A |
|---|---|---|---|---|---|
| X | A | C | G | G | A |

| | A | C | T | G |
|---|---|---|---|---|
| Y | A | C | T | G |

| | | Y: A | C | T | G |
|---|---|---|---|---|---|
| X | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | $\boxed{1}$ |
| C | 0 | | | | |
| G | 0 | | | | |
| G | 0 | | | | |
| A | 0 | | | | |

$X[i] \neq Y[j]$

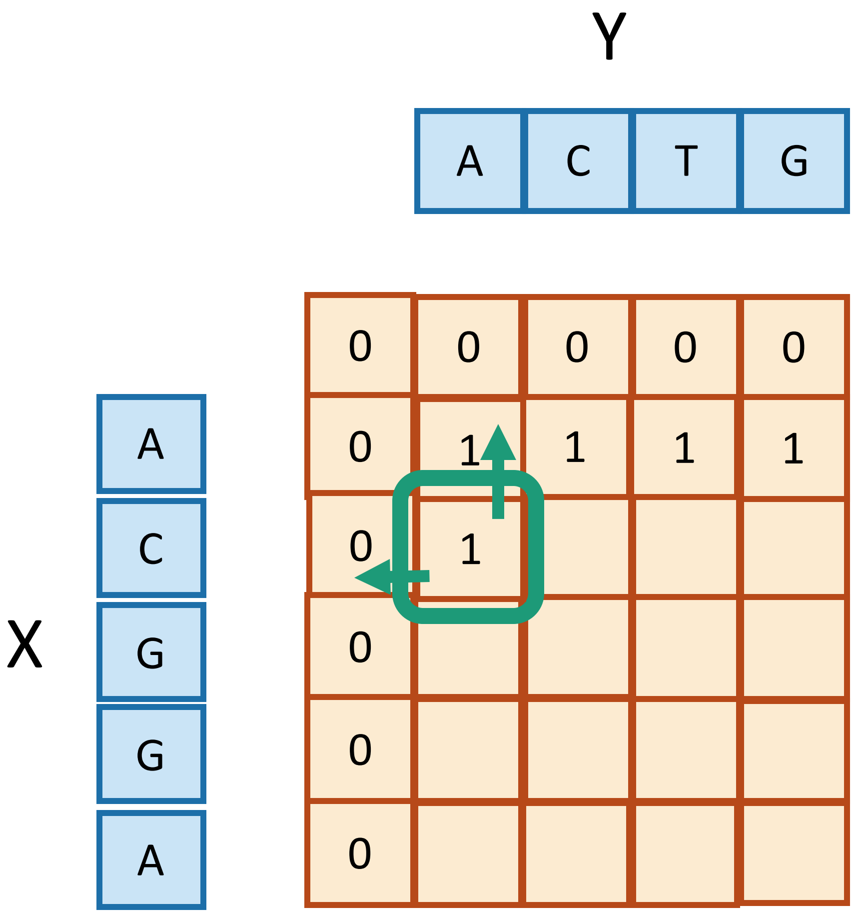
So we want

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

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

# Example

| | | | | | |
|---|---|---|---|---|---|
| X | A | C | G | G | A |
| Y | A | C | T | G | |




$X[i] \neq Y[j]$

So we want
$\max\{\, C[i, j-1], C[i-1, j]\}$

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

# Example

X: | A | C | G | G | A |

Y: | A | C | T | G |

| | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | $\boxed{2}$ | | |
| G | 0 | | | | |
| G | 0 | | | | |
| A | 0 | | | | |

$X[i] = Y[j]$

So we want

$C[i-1, j-1] + 1$

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

# Example

| | | | | | |
|---|---|---|---|---|---|
| X | A | C | G | G | A |

| | | | | |
|---|---|---|---|---|
| Y | A | C | T | G |

Y

| X | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | $\boxed{3}$ |

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

| X | | | | | |
|---|---|---|---|---|---|
| X | A | C | G | G | A |

| Y | | | | |
|---|---|---|---|---|
| Y | A | C | T | G |

Y

| X \ Y | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | | | | |
| C | 0 | | | | |
| G | 0 | | | | |
| G | 0 | | | | |
| A | 0 | | | | |

X

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

# Example

| X | A | C | G | G | A |
|---|---|---|---|---|---|

| Y | A | C | T | G |
|---|---|---|---|---|

Y

| X | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 3 |

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

# Example

| | | | | | |
|---|---|---|---|---|---|
| X | A | C | G | G | A |
| Y | A | C | T | G | |

| | | Y: A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| X: A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 3 |

- Once we've filled this in, we can work backwards.

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

# Example

| | | | | | |
|---|---|---|---|---|---|
| X | A | C | G | G | A |
| Y | A | C | T | G | |

| X \ Y | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 3 |

- 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

| | | | | | |
|---|---|---|---|---|---|
| X | A | C | G | G | A |
| Y | A | C | T | G | |

| X \ Y | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 3 |

- 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!

G

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

# Example

| | | | | | |
|---|---|---|---|---|---|
| X | A | C | G | G | A |
| Y | A | C | T | G | |

| | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 3 |

- 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.

G

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

# Example

| | | | | | |
|---|---|---|---|---|---|
| X | A | C | G | G | A |

| | | | | |
|---|---|---|---|---|
| Y | A | C | T | G |

Y

| | A | C | T | G |
|---|---|---|---|---|

| X | | | | | |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 3 |

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

G

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

# Example

| | | | | | |
|---|---|---|---|---|---|
| X | A | C | G | G | A |
| Y | A | C | T | G | |

Y

| X | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 3 |

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

| | |
|---|---|
| C | G |

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

# Example

X: | A | C | G | G | A |

Y: | A | C | T | G |

| X \ Y | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 3 |

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

| A | C | G |

**This is the LCS!**

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

# 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:

| | | | | | |
|---|---|---|---|---|---|
| Item: |  |  |  |  |  |
| Weight: | 6 | 2 | 4 | 3 | 11 |
| Value: | 20 | 8 | 14 | 13 | 35 |

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

Capacity: 10

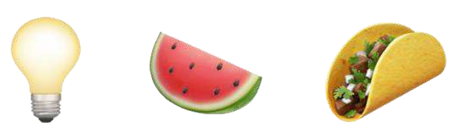
Capacity: 10

| | | | | | |
|---|---|---|---|---|---|
| Item: |  |  |  |  |  |
| Weight: | 6 | 2 | 4 | 3 | 11 |
| Value: | 20 | 8 | 14 | 13 | 35 |

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

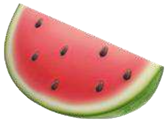
Total weight: 10
Total value: 42

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



Total weight: 9
Total value: 35

# Some notation

| | | | | | |
|---|---|---|---|---|---|
| Item: |  |  |  | |  |
| Weight: | $w_1$ | $w_2$ | $w_3$ | … | $w_n$ |
| Value: | $v_1$ | $v_2$ | $v_3$ | | $v_n$ |

Capacity: W

# Recipe for applying Dynamic Programming

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

# Optimal substructure

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

First solve the problem for small knapsacks

Then larger knapsacks

Then larger knapsacks

# What's the relationship between bigger and smaller problems?

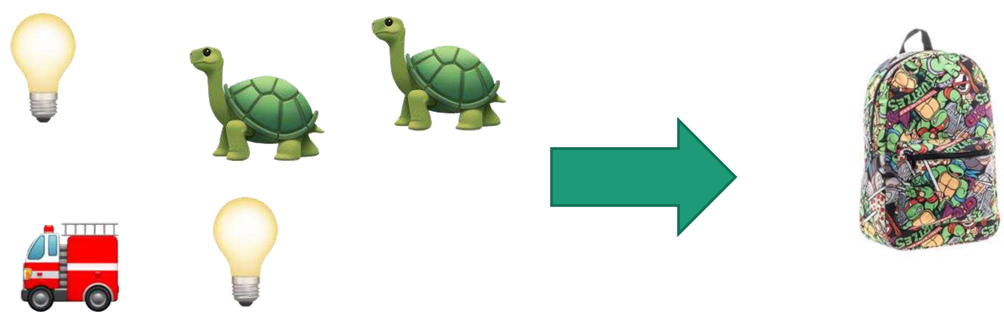
item i

- Suppose this is an optimal solution for capacity x (and it includes item i):

Weight $w_i$
Value $v_i$

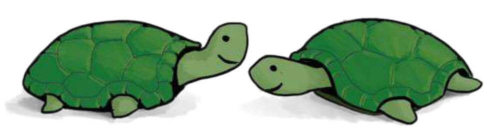
Capacity x
Value V

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



Capacity x – $w_i$
Value V - $v_i$

**Why?**



# What’s the relationship between bigger and smaller problems?

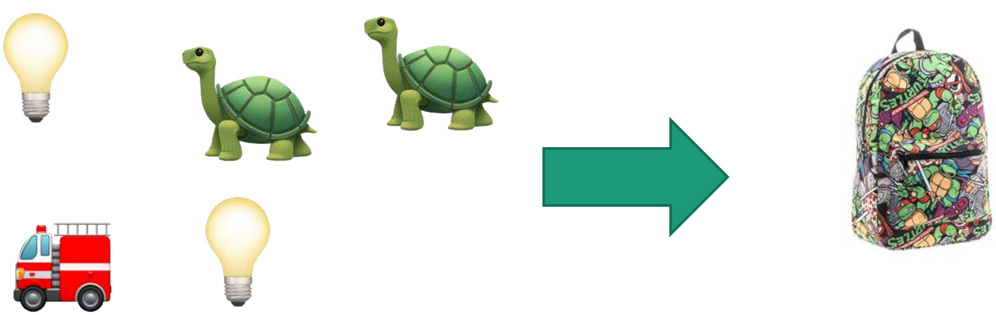
item i

- Suppose this is an optimal solution for capacity x (and it includes item i):

Weight $w_i$
Value $v_i$

Capacity x
Value V

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



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

Capacity x – $w_i$
Value V - $v_i$

# Recipe for applying Dynamic Programming

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

# Recursive relationship

- K[x] is the optimal value for capacity x.

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

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

Optimal way to fill a smaller knapsack

The value of item i.

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

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

# 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):
  - Initialize K of length W+1, K[0] = 0
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
  - **return** K[W]

Running time: O(nW)

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

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

# Can we do better than $O(nW)$?

- Input size: $O(n \log W)$ bits.
  - Writing down "W" takes $\log_2 W$ bits.
  - Writing down all n weights takes at most $n \log_2 W$ bits.

- Maybe we could have an algorithm that runs in time $O(n \log W)$ instead of $O(nW)$?
  - Or even $O\left((n \log W)^{10000000}\right)$?

- Getting time polynomial in $n \log W$ is an open problem!
  - But probably the answer is **no**…otherwise $P = NP$

# Recipe for applying Dynamic Programming

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

# Let's write a bottom-up DP algorithm

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

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

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

# Let’s write a bottom-up DP algorithm

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

**Idea:** Keep an another array, ITEMS, so that ITEMS[x] stores the optimal solution (a list of items) for capacity x.

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

= $\max_i$ { K[x – $w_i$] + $v_i$ }

Note: this syntax means that we are appending item i on the list ITEMS[x-$w_i$]

# Example

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | | | | |
| ITEMS | | | | | |

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

Item:   

Weight: 1 2 3

Value: 1 4 6

Capacity: 4

# Example

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

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | | | |
| ITEMS | |  | | | |

ITEMS[1] = ITEMS[0] + 

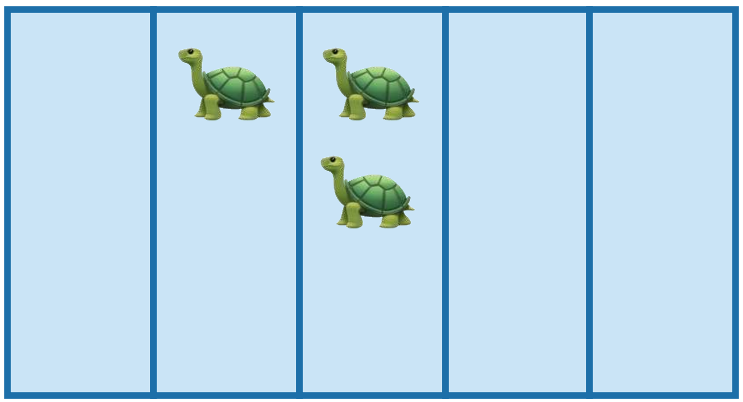
| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

Capacity: 4

# Example

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 2 | | |

ITEMS



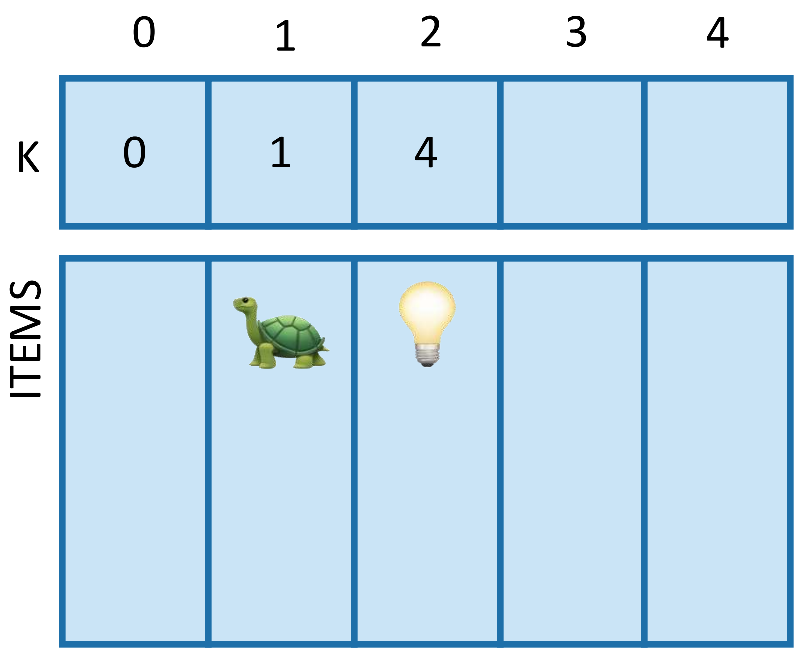
ITEMS[2] = ITEMS[1] + 

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

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

Capacity: 4

# Example




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

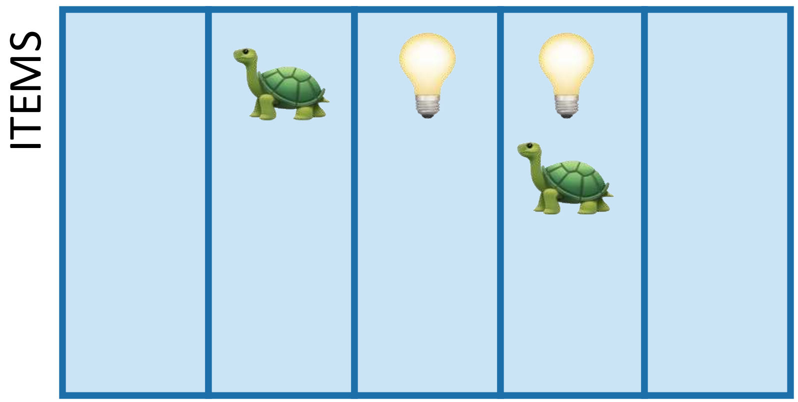
ITEMS[2] = ITEMS[0] + 

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

Capacity: 4

# Example

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 5 | |




ITEMS[3] = ITEMS[2] + 

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

Item: 

Weight: 1 2 3

Value: 1 4 6

Capacity: 4

# Example

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 6 | |
| ITEMS | |  |  |  | |

ITEMS[3] = ITEMS[0] + 

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

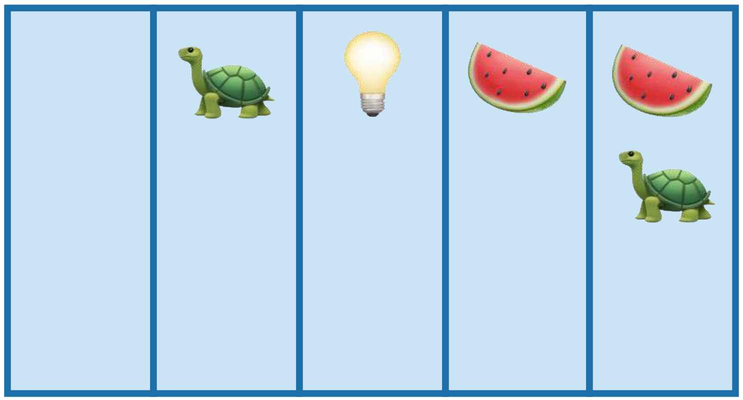
| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

Capacity: 4

# Example

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 6 | 7 |

ITEMS



ITEMS[4] = ITEMS[3] + 

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

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

Capacity: 4

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

# Example

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 6 | 8 |
| ITEMS | |  |  |  |   |

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

ITEMS[4] = ITEMS[2] + 

Capacity: 4

So the optimal solution is   , with value 8

# Recipe for applying Dynamic Programming

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

(Pass)

# What have we learned?

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

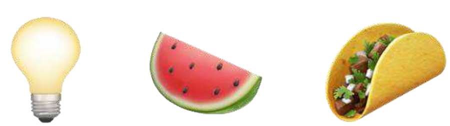
Capacity: 10

| Item: |  |  |  |  |  |
|---|---|---|---|---|---|
| Weight: | 6 | 2 | 4 | 3 | 11 |
| Value: | 20 | 8 | 14 | 13 | 35 |

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

Total weight: 10
Total value: 42

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



Total weight: 9
Total value: 35

# Recipe for applying Dynamic Programming

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

# Optimal substructure: try 1

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

First solve the problem for small knapsacks

Then larger knapsacks

Then larger knapsacks

# This won’t quite work…

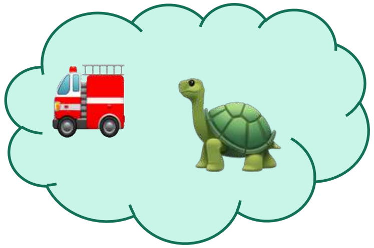
- We are only allowed **one copy of each item**.
- The sub-problem needs to “know” what items we’ve used and what we haven’t.

# Optimal substructure: try 2

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

First solve the problem with few items

Then more items



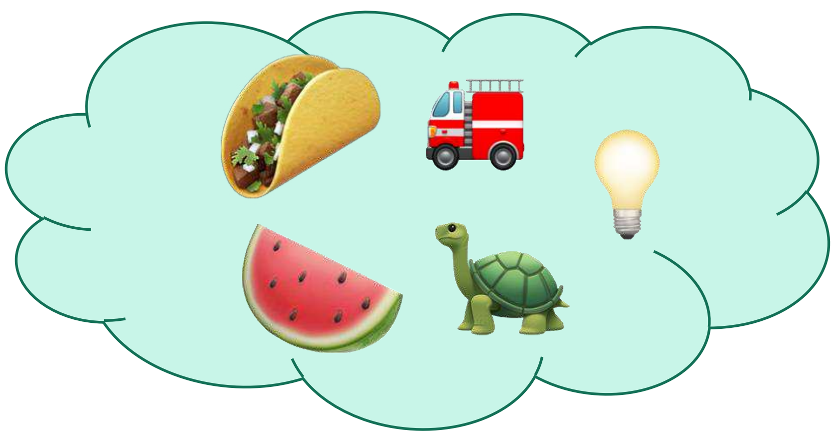
Then yet more items

We'll still increase the size of the knapsacks.

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

# Our sub-problems:

- Indexed by x and j



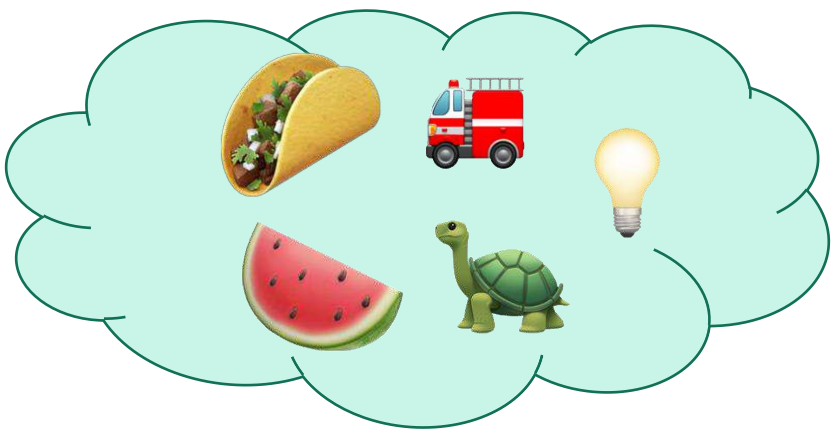
First j items

Capacity x

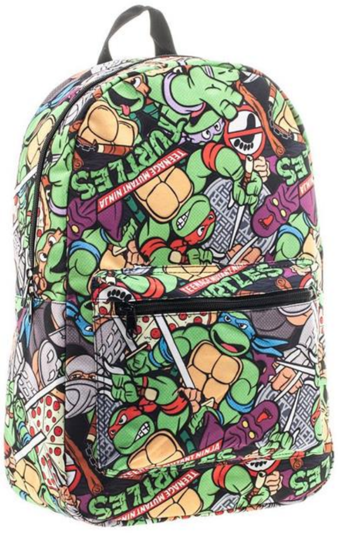
K[x,j] = optimal solution for a knapsack of size x using only the first j items.

# Relationship between sub-problems

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



First j items



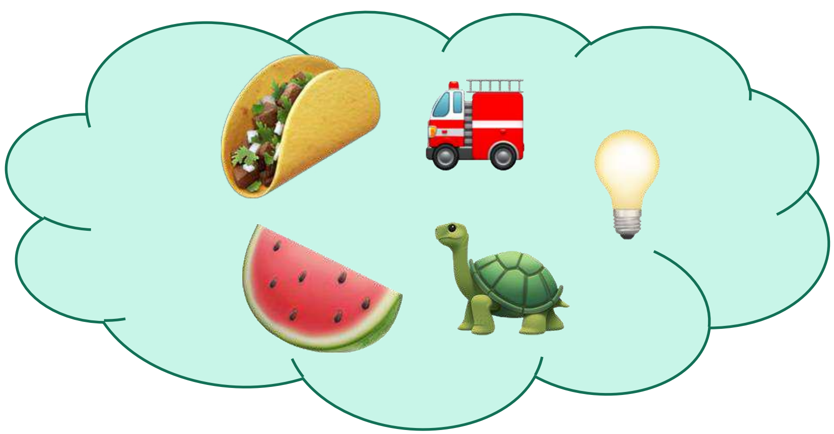
Capacity x

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

# Two cases

item j

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



First j items

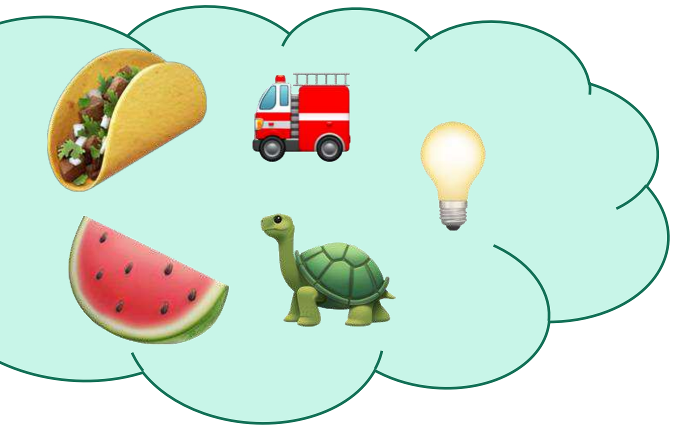
Capacity x

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

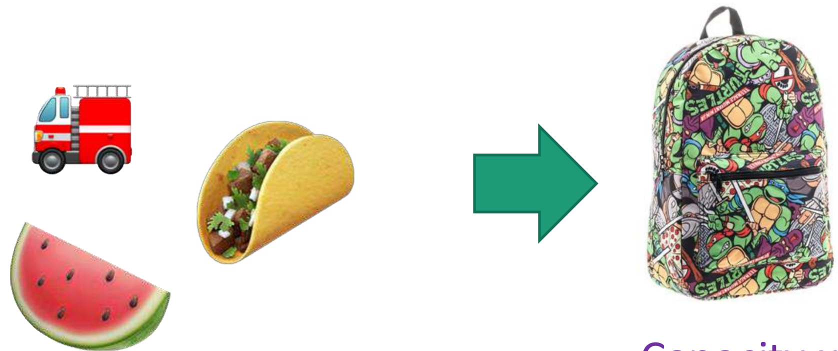
# Two cases

item j

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

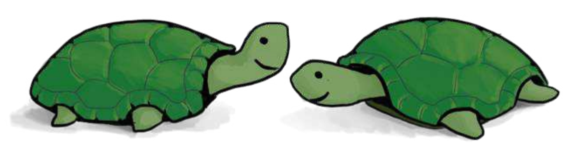


First j items



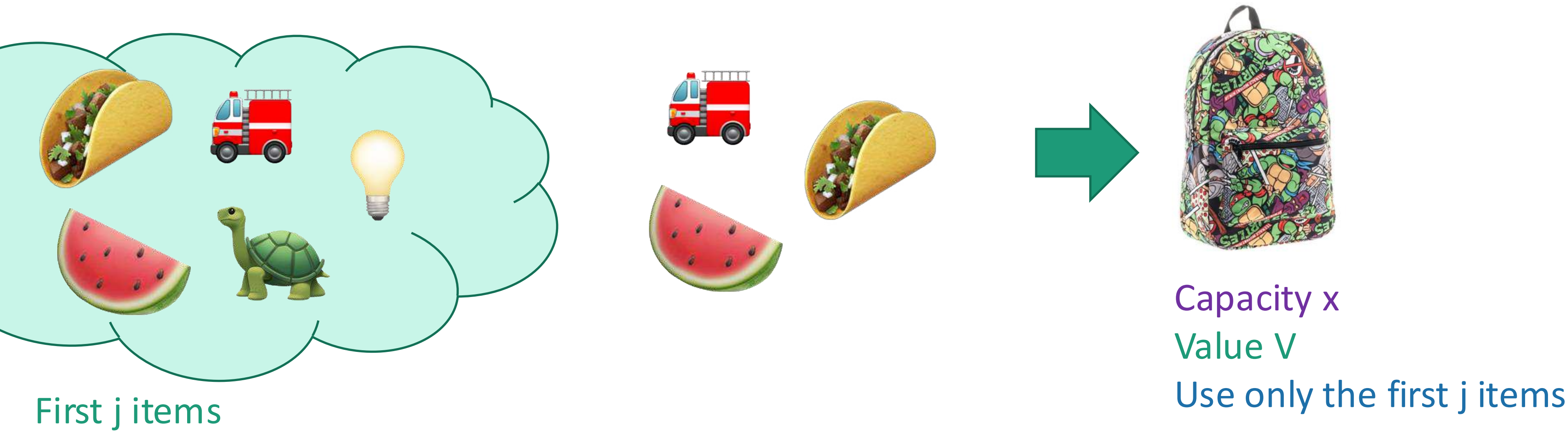
Capacity x
Value V
Use only the first j items

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

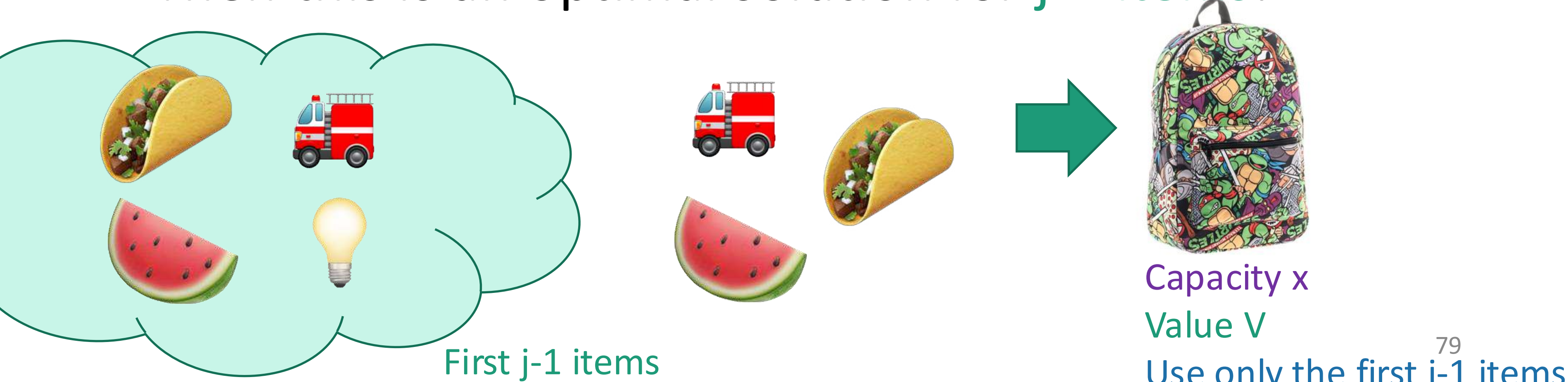


# Two cases


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

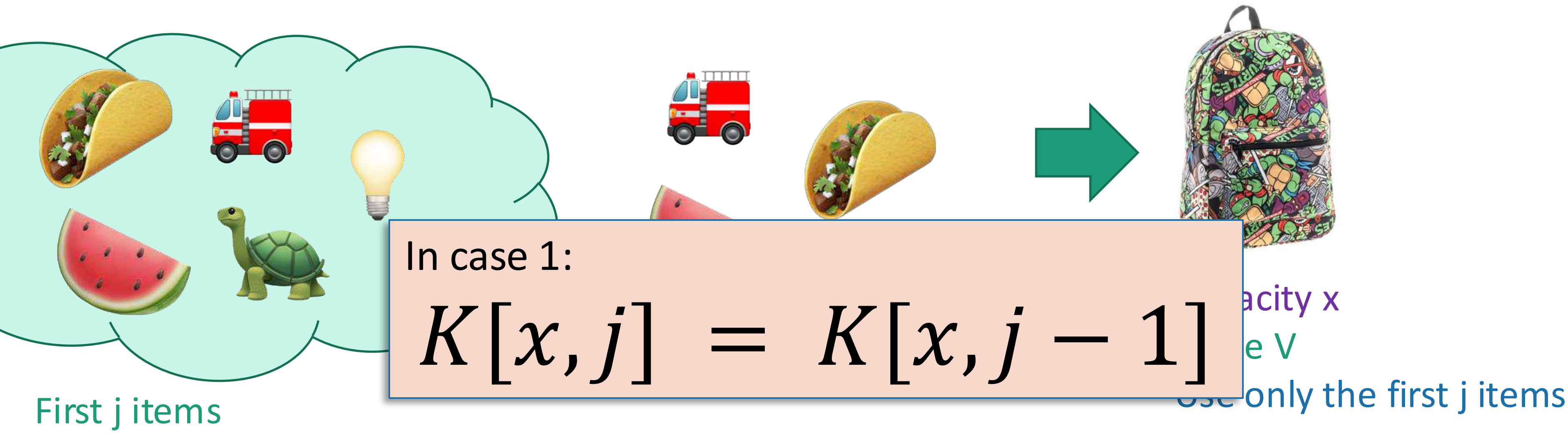



- Then this is an optimal solution for j-1 items:



# Two cases

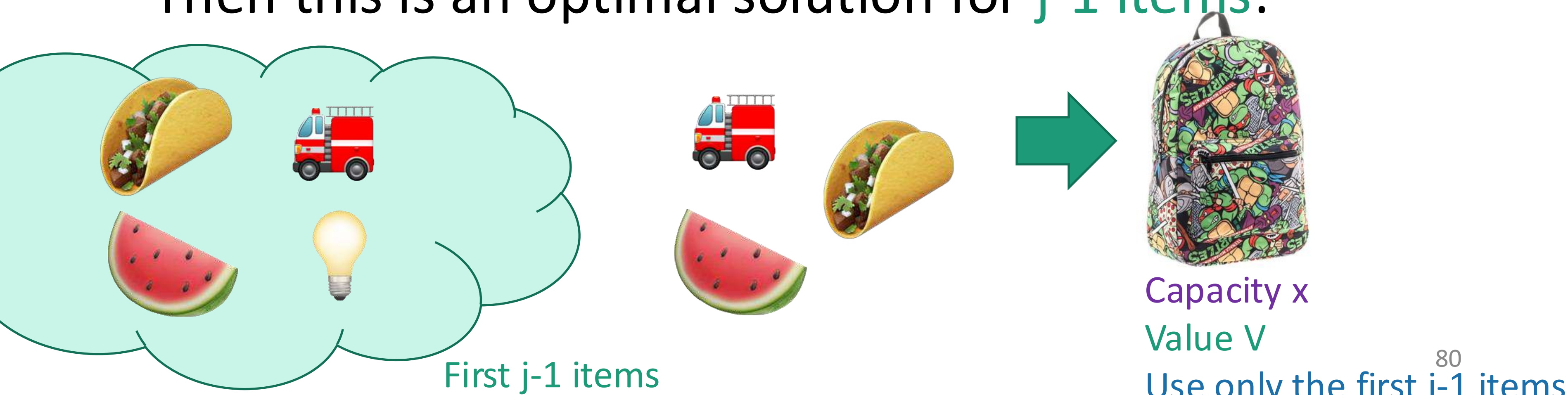

- **Case 1**: Optimal solution for j items does not use item j.




In case 1:

$$K[x,j] = K[x,j-1]$$

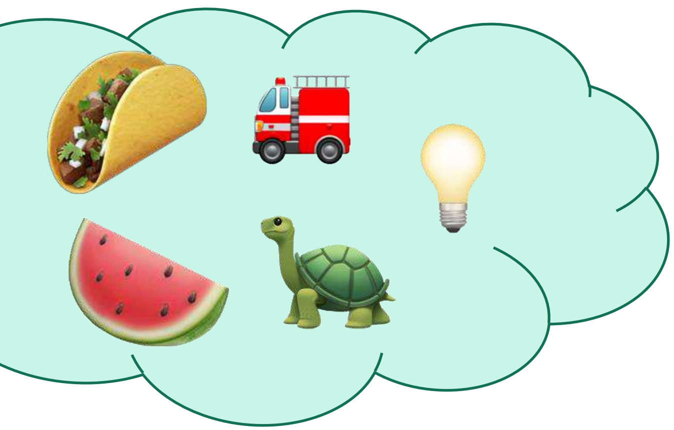
- Then this is an optimal solution for j-1 items:



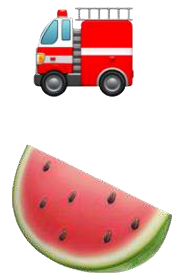
# Two cases

item j

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



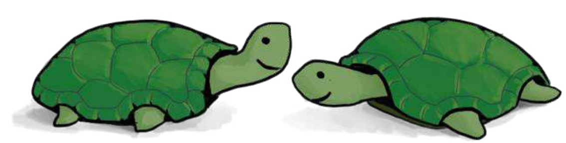
First j items



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?

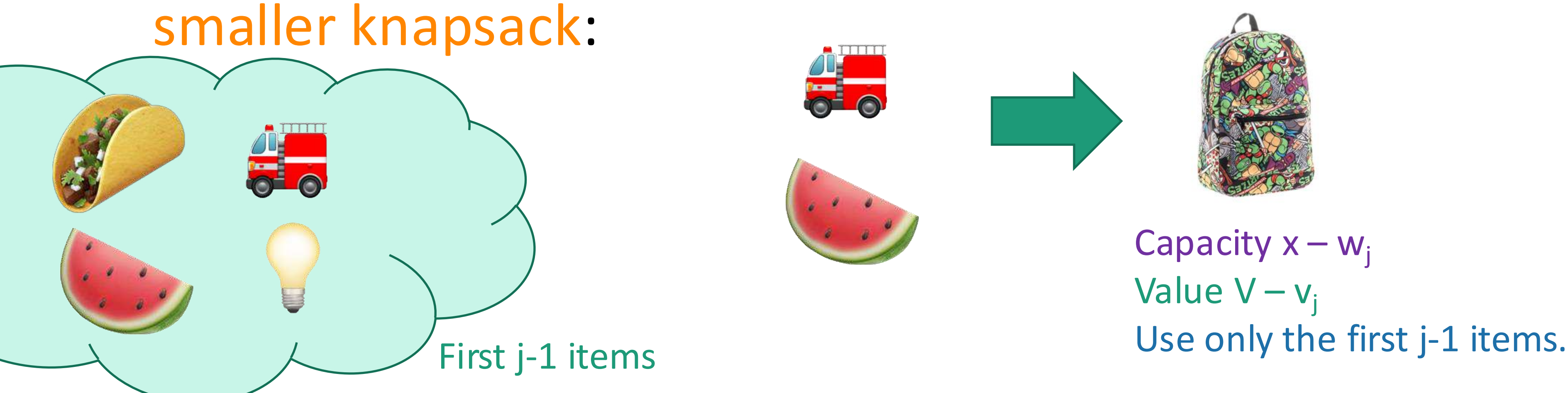


# Two cases

item j

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

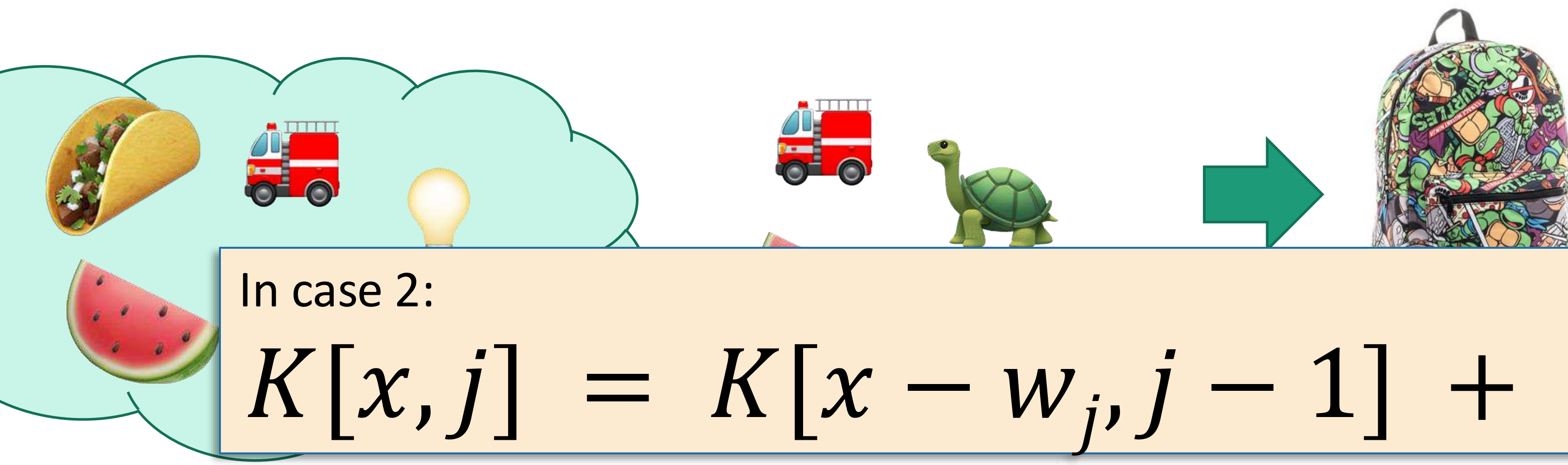

- Then this is an optimal solution for j-1 items and a smaller knapsack:



# Two cases

item j

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



In case 2:

$$K[x, j] = K[x - w_j, j - 1] + v_j$$

First j items

Use only the first j items

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

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.

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

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

# Recipe for applying Dynamic Programming

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

# Bottom-up DP algorithm

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:

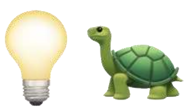
      Case 1
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:

        Case 2
        - K[x,j] = max{ K[x,j], K[x – $w_j$, j-1] + $v_j$ }
  - **return** K[W,n]

Running time O(nW)

# Example

- 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]

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
|  j=1 | 0 | | | |
|  j=2 | 0 | | | |
|  j=3 | 0 | | | |

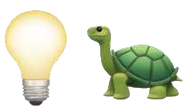
current entry

relevant previous entry

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

# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
|  j=1 | 0 | 0 | | |
|  j=2 | 0 | | | |
|  j=3 | 0 | | | |

current entry

relevant previous entry

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

# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | | |
| j=2 | 0 | | | |
| j=3 | 0 | | | |

current entry

relevant previous entry

Item:   

Weight: 1 2 3

Value: 1 4 6

Capacity: 3

# Example

- 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]

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

current entry

relevant previous entry

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

Capacity: 3

# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | | |
| j=2 | 0 | 1 | | |
| j=3 | 0 | 1 | | |

current entry

relevant previous entry

Item:

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

Capacity: 3

# Example

|  | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 0 | |
| j=2 | 0 | 1 | | |
| j=3 | 0 | 1 | | |

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

current entry

relevant previous entry

Item:

Weight: 1 2 3

Value: 1 4 6

Capacity: 3

# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | |
| j=2 | 0 | 1 | | |
| j=3 | 0 | 1 | | |

current entry

relevant previous entry

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

Capacity: 3

# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | |
| j=2 | 0 | 1 | 1 | |
| j=3 | 0 | 1 | | |

current entry

relevant previous entry

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

# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | |
| j=2 | 0 | 1 | 4 | |
| j=3 | 0 | 1 | | |

current entry

relevant previous entry

Item: 

Weight: 1 2 3

Value: 1 4 6

Capacity: 3

# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | |
| j=2 | 0 | 1 | 4 | |
| j=3 | 0 | 1 | 4 | |

current entry

relevant previous entry

| Item: | turtle | light bulb | watermelon |
|---|---|---|---|
| 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]

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 0 |
| j=2 | 0 | 1 | 4 | |
| j=3 | 0 | 1 | 4 | |

current entry

relevant previous entry

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

Capacity: 3

# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 1 |
| j=2 | 0 | 1 | 4 | |
| j=3 | 0 | 1 | 4 | |

current entry

relevant previous entry

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

Capacity: 3

# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 1 |
| j=2 | 0 | 1 | 4 | 1 |
| j=3 | 0 | 1 | 4 | |

current entry

relevant previous entry

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

# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 1 |
| j=2 | 0 | 1 | 4 | 5 |
| j=3 | 0 | 1 | 4 | |

current entry

relevant previous entry

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

Capacity: 3

# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 1 |
| j=2 | 0 | 1 | 4 | 5 |
| j=3 | 0 | 1 | 4 | 5 |

current entry

relevant previous entry

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

Capacity: 3

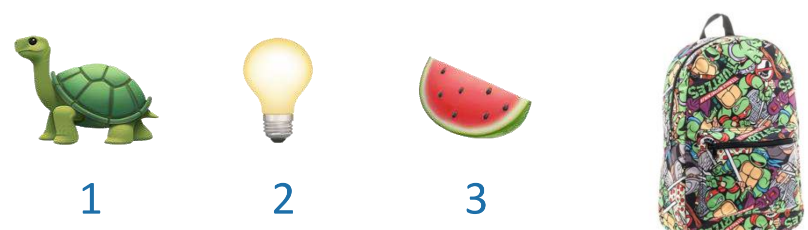
# Example

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

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 1 |
| j=2 | 0 | 1 | 4 | 5 |
| j=3 | 0 | 1 | 4 | 6 |

current entry

relevant previous entry



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

Capacity: 3

# Example

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

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

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

current entry

relevant previous entry

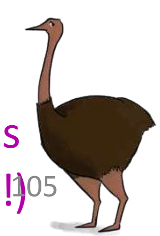
| Item: | 🐢 | 💡 | 🍉 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

Capacity: 3

# Recipe for applying Dynamic Programming

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

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.

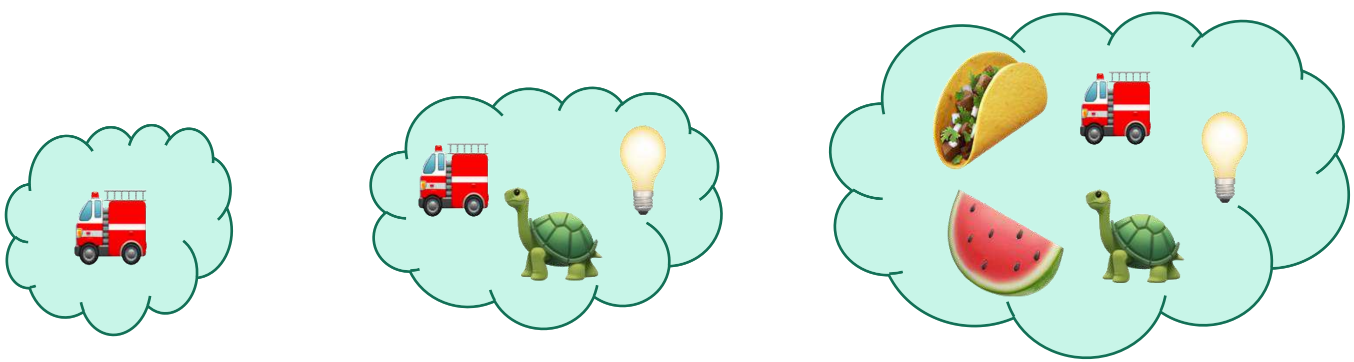
# Question

- 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.**

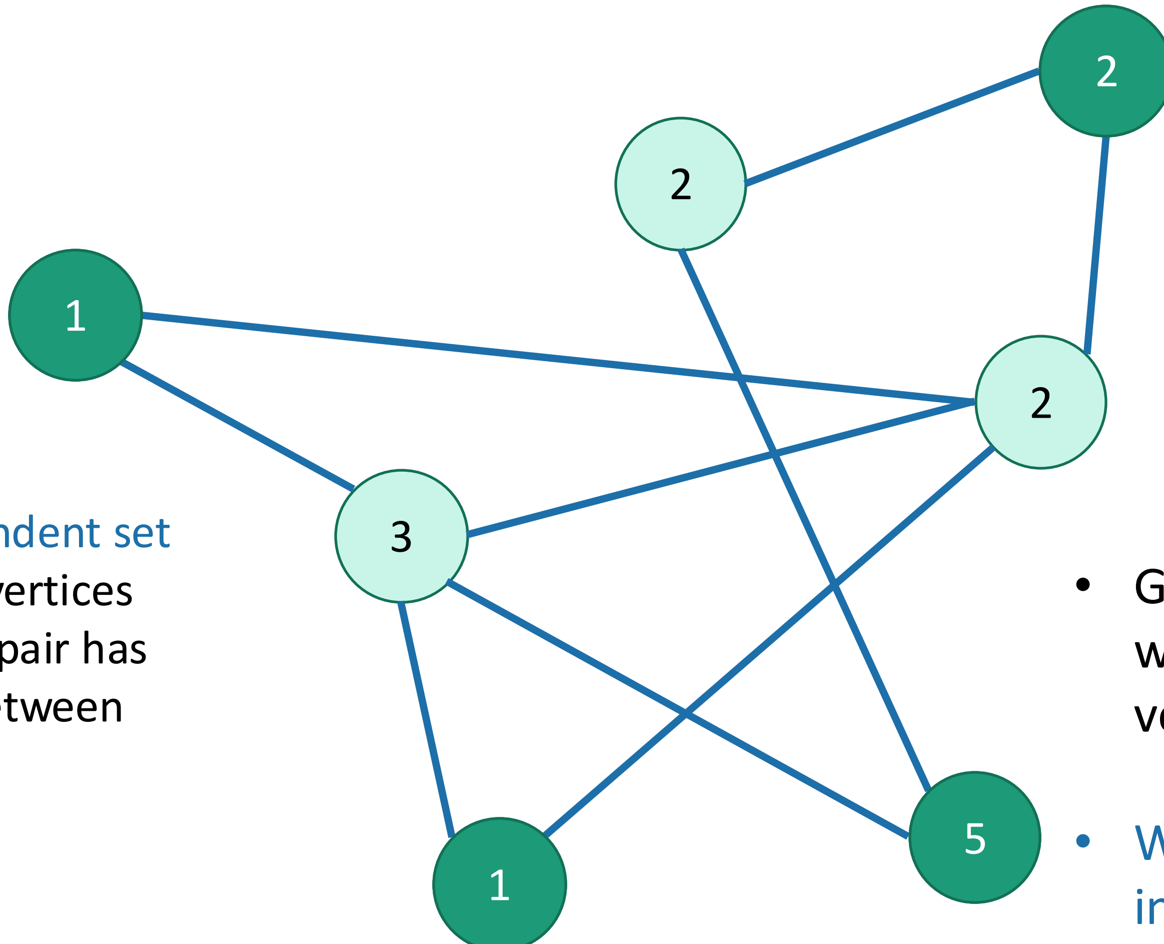


In 0/1 knapsack, we can only use each item once, so it makes sense to leave out one item at a time.

**Operational Answer**: try some stuff, see what works!

# Example 3: Independent Set

## if we still have time



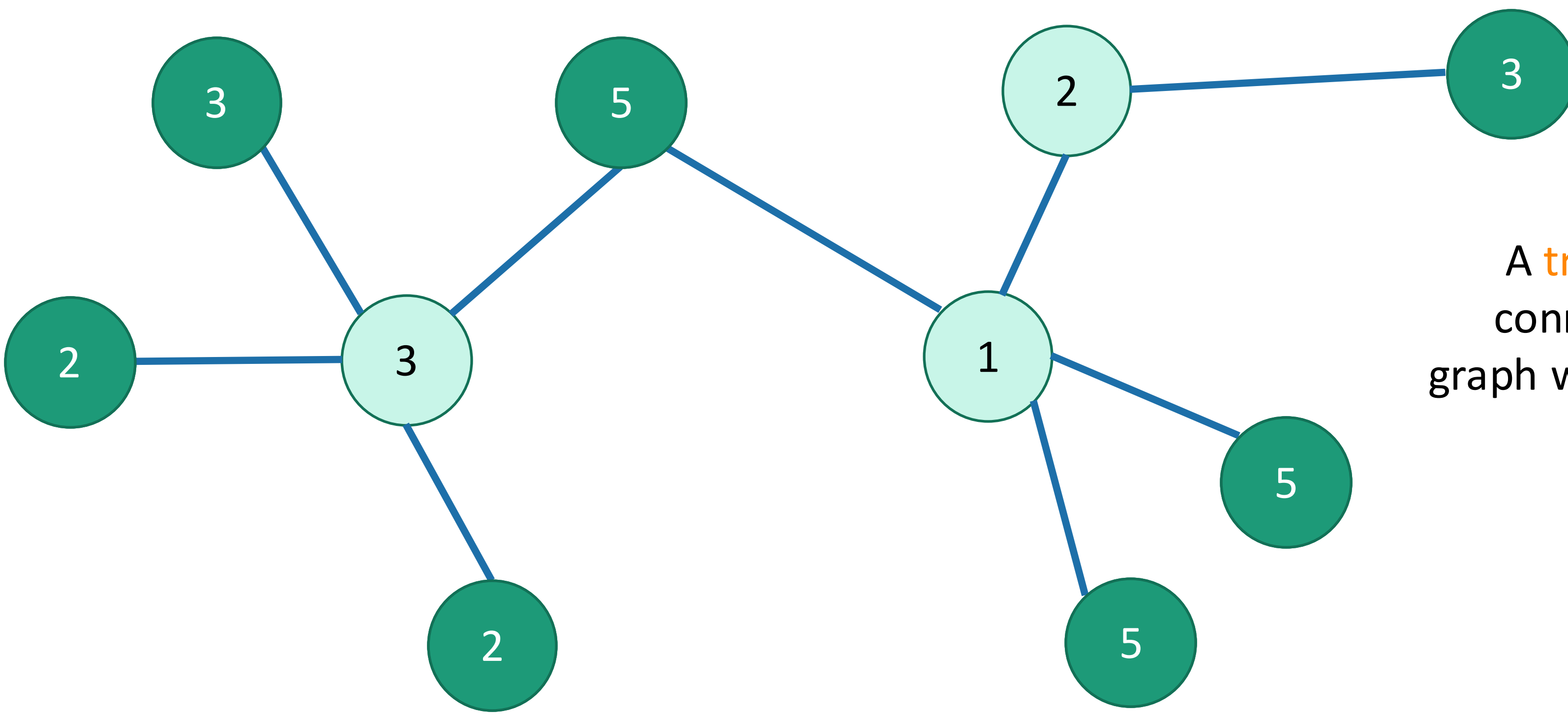

An independent set is a set of vertices so that no pair has an edge between them.

- 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…




A tree is a connected graph with no cycles.

**Problem**:

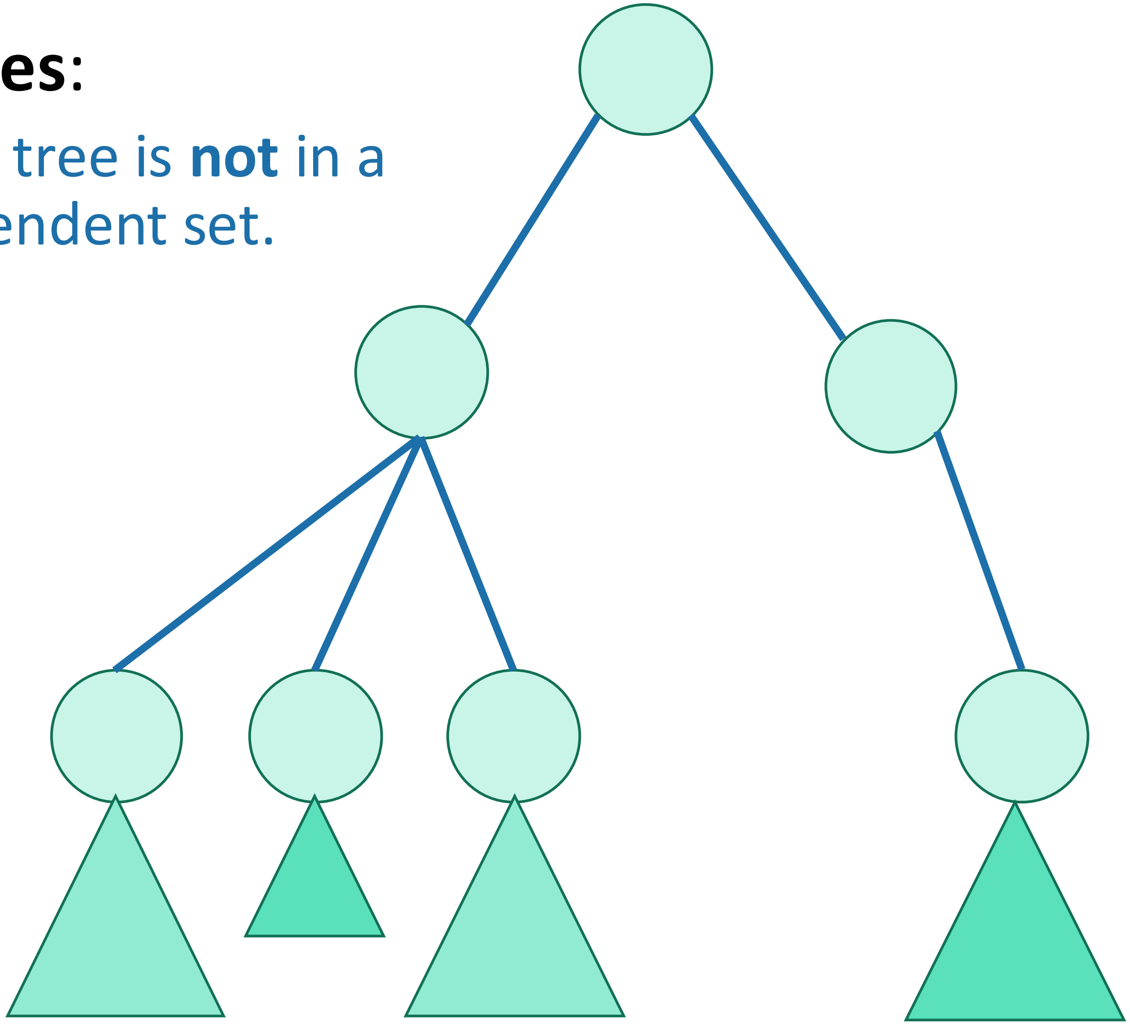
find a maximal independent set in a tree (with vertex weights).

# 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.

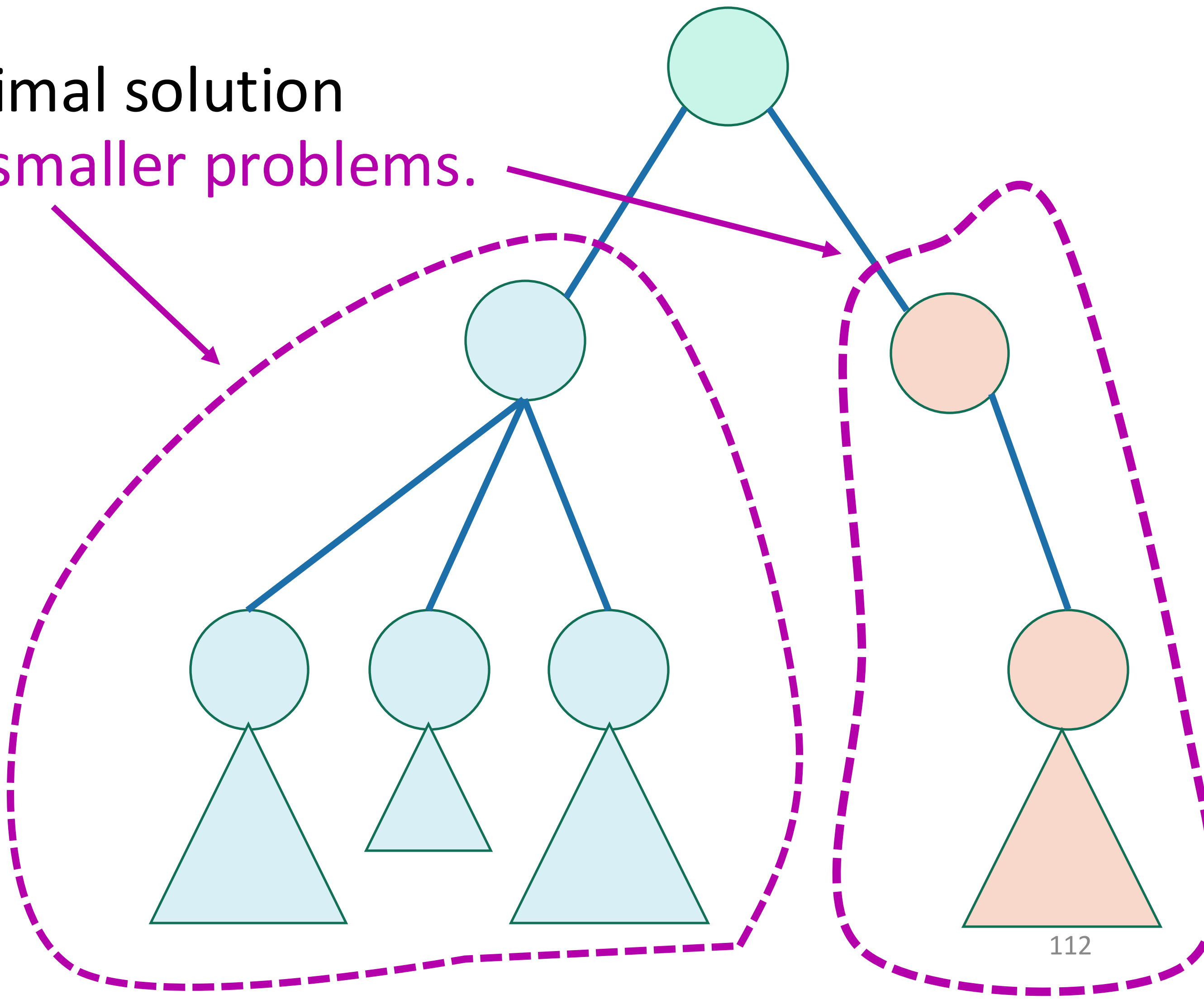
# Optimal substructure

- **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.



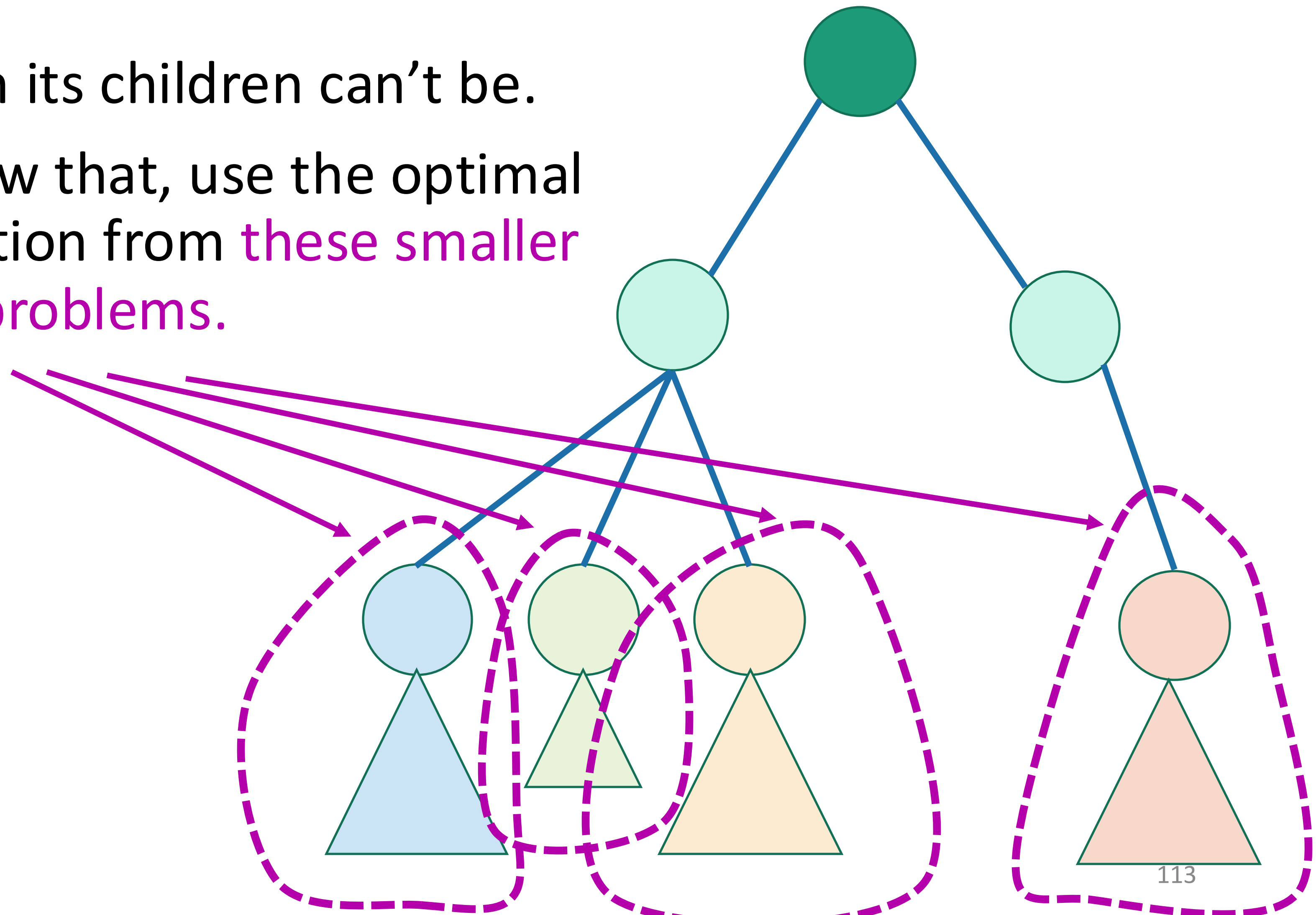
# Case 1:
## the root is **not** in an maximal independent set

- 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.

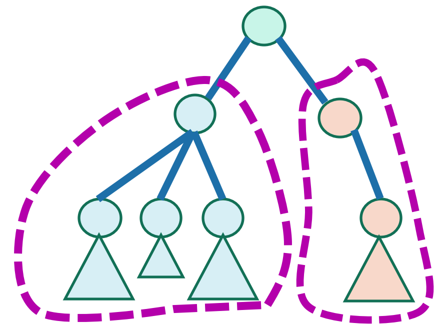


# Recipe for applying Dynamic Programming

- **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

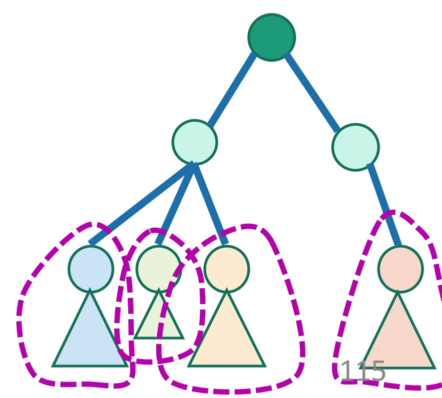
- Let A[u] be the weight of a maximal independent set in the tree rooted at u.



- $A[u] =$

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

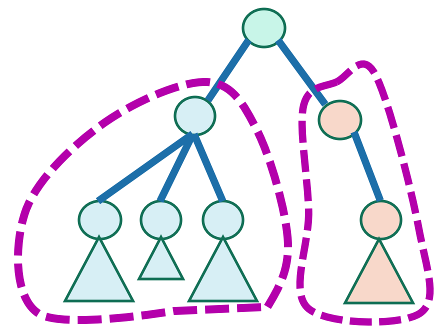
When we implement this, how do we keep track of this term?



# Slightly cleaner version

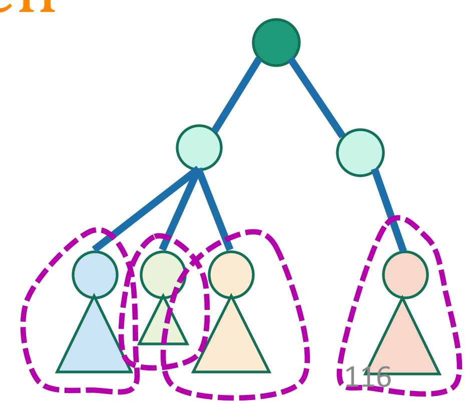
## Keep two arrays!

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



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

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



This saves on pointer-chasing, and also has a nice interpretation: B[v] is “the weight of a maximal independent set in the tree rooted at u, if u is not in the MIS.”

# Recipe for applying Dynamic Programming

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

# A top-down DP algorithm

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

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

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

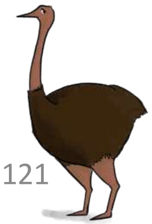
**Running time?**

- We visit each vertex once, and at every vertex we do O(1) work:
  - Make a recursive call
  - look stuff up in tables
- Running time is O(|V|)

# 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.
    - If not time, that's fine! Feel free to check out the slides if you want more examples of DP.
- 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
  - Lots of practice on HW6!!! ☺
  - For even more practice check out additional examples/practice problems in Algorithms Illuminated or CLRS or section!

# Next time

- Greedy algorithms!

# Before next time

- Pre-lecture exercise: Greed is good!