# CSE100: Design and Analysis of Algorithms
# Lecture 22 – More Dynamic Programming (cont)

**Apr 14th 2022**

Longest Common Subsequences, Knapsack, and (if time) Independent Sets in Trees

# Last Lecture (review)

**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 the slides will be there as a reference)

# Longest Common Subsequence (review)

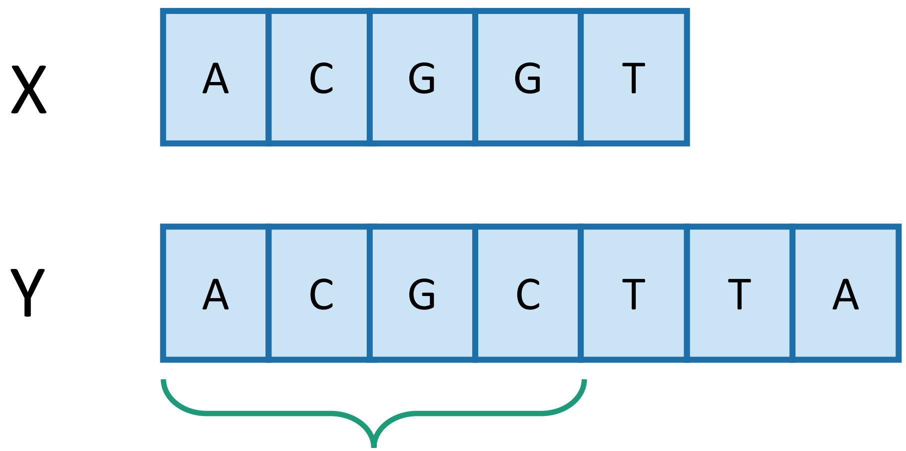
- Subsequence:
  - BDFH is a **subsequence** of ABCDEFGH
- If X and Y are sequences, a **common subsequence** is a sequence which is a subsequence of both.
  - BDFH is a **common subsequence** of ABCDEFGH and of ABDFGHI
- A **longest common subsequence**…
  - …is a common subsequence that is longest.
  - The **longest common subsequence** of ABCDEFGH and ABDFGHI is ABDFGH.

# Recipe for applying Dynamic Programming (review)

- **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 (review)

Prefixes:




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

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

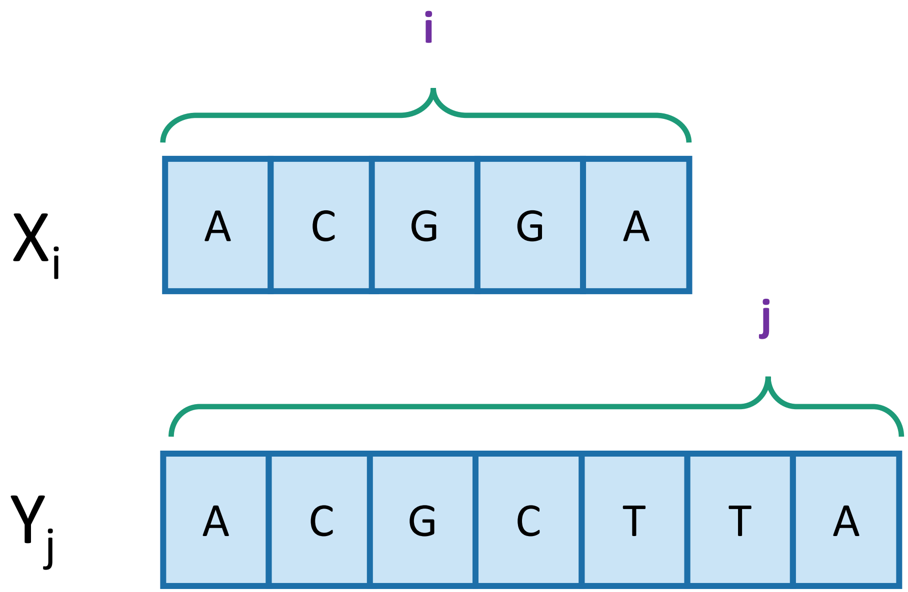
Examples: C[2,3] = 2
C[4,4] = 3

# Recipe for applying Dynamic Programming

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

# Goal

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

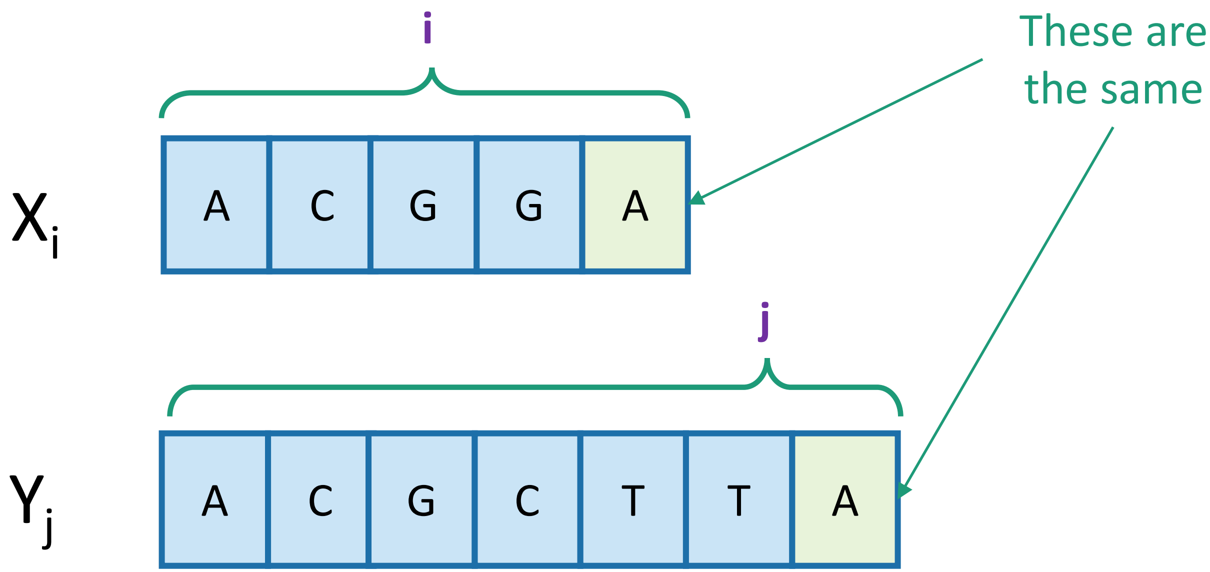



$$C[i, j] = \text{length\_of\_LCS}(X_i, Y_j)$$

# Two cases

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

Case 1: $X_i[i] = Yj[j]$

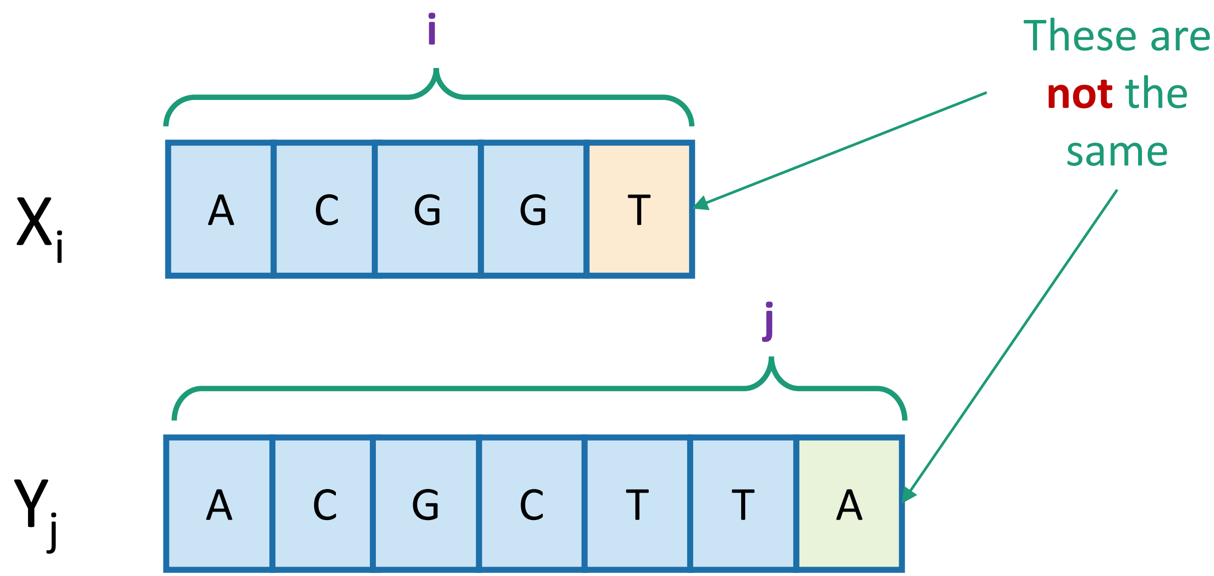



- Then C[i, j] = 1 + C[i-1, j-1].
  - because LCS($X_i$, $Y_j$) = LCS($X_{i-1}$, $Y_{j-1}$) followed by $\boxed{A}$

# Two cases

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

Case 2: $X_i[i] \neq Y_j[j]$




- Then C[i, j] = max{C[i-1, j], C[i, j-1]}.
  - either LCS($X_i$, $Y_j$) = LCS($X_{i-1}$, $Y_j$) and $\boxed{T}$ is not involved,
  - or LCS($X_i$, $Y_j$) = LCS($X_i$, $Y_{j-1}$) and $\boxed{A}$ 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

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

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** $X_i$[i] = $Y_j$[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)**

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

# Example

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

Y

| | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | | | | |
| C | 0 | | | | |
| G | 0 | | | | |
| G | 0 | | | | |
| A | 0 | | | | |

X

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

# 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 |

So the LCS of X and Y has length 3.

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

# 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 | A | C | G | G | A |

| | | | | |
|---|---|---|---|---|
| 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 | | | | |

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

# Example

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

Y

| | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| 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 |

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

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

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!

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 | |

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!

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

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

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!

| 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 | |

| | | 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.
- 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

- See CLRS for pseudocode
- 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.

# This pseudocode 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 us know :D

# What have we learned?

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

- 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] = value you can fit in a knapsack of capacity x

First solve the problem for small knapsacks

Then larger knapsacks

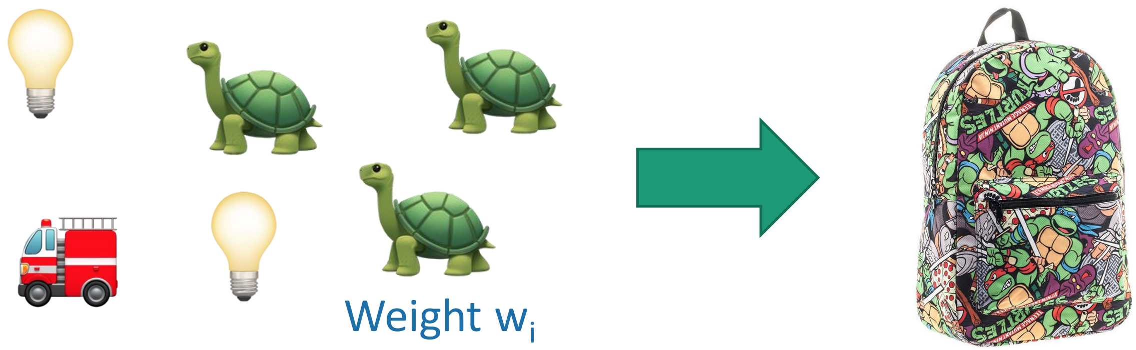
Then larger knapsacks

# Optimal substructure

item i

- Suppose this is an optimal solution for capacity x:

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



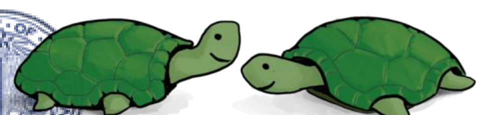
Weight $w_i$
Value $v_i$

Capacity x
Value V

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

**Why?**

Capacity x – $w_i$
Value V - $v_i$



# Optimal substructure

item i

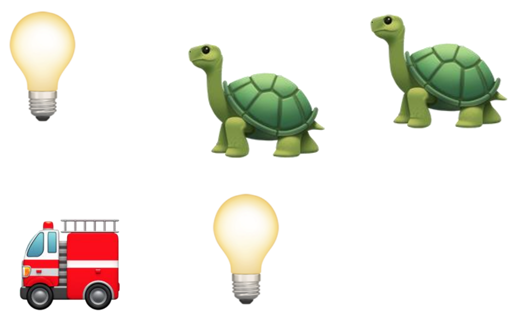
- Suppose this is an optimal solution for capacity x:

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

Weight $w_i$
Value $v_i$

Capacity x
Value V

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



Capacity x – $w_i$
Value V - $v_i$

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

# Recipe for applying Dynamic Programming

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

# Recursive relationship

- Let K[x] be the optimal value for capacity x.

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

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

Optimal way to fill the 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):
  - 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] = $\max_i$ {  +  }

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

Why does this work?

Because our recursive relationship makes sense.

# Can we do better?

- Writing down W takes log(W) bits.
- Writing down all n weights takes at most nlog(W) bits.
- Input size: nlog(W).
  - Maybe we could have an algorithm that runs in time O(nlog(W)) instead of O(nW)?
  - Or even O( $n^{1000000} \log^{1000000}(W)$ )?

- Open problem!
  - (But probably the answer is **no**…otherwise P = NP)

# Recipe for applying Dynamic Programming

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

# Let's write a bottom-up DP algorithm

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

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

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

# Let's write a bottom-up DP algorithm

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = ∅
  - **for** x = 1, …, W:
    - 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]

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

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

# Example

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

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

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

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

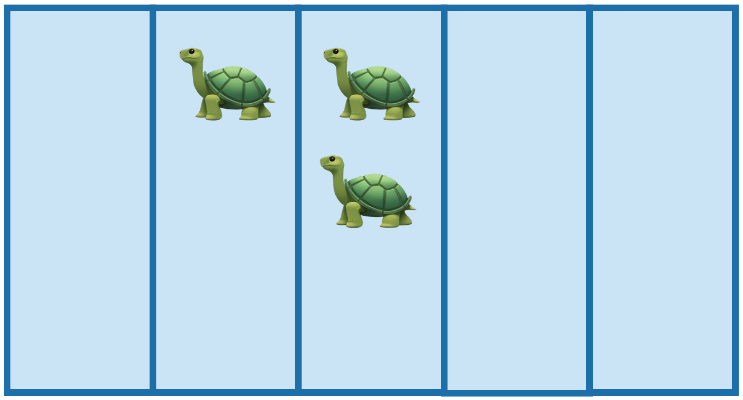
ITEMS[1] = ITEMS[0] + 

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

Item: 

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

Capacity: 4

# Example

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

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

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

ITEMS[2] = ITEMS[0] + 

Capacity: 4

# Example

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

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 5 | |
| ITEMS | | 🐢 | 💡 | 💡 🐢 | |

ITEMS[3] = ITEMS[2] + 🐢

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

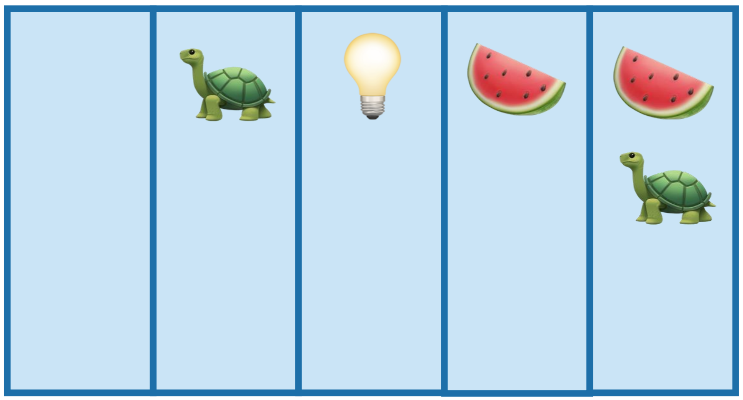
| Item: | 🐢 | 💡 | 🍉 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

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

| | 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

# 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:

| 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.

I can’t use
any turtles…

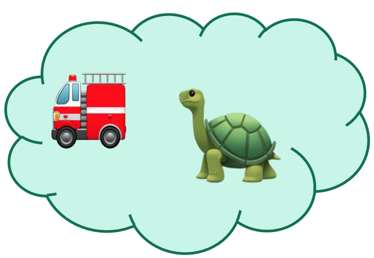
# Optimal substructure: try 2

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

We'll still increase the size of the knapsacks.

First solve the problem with few items

Then more items

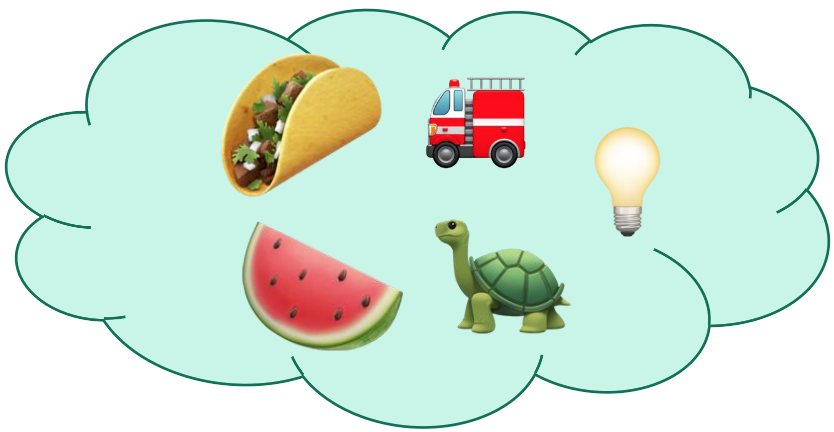


Then yet more items

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

# Our sub-problems:

- Indexed by x and j



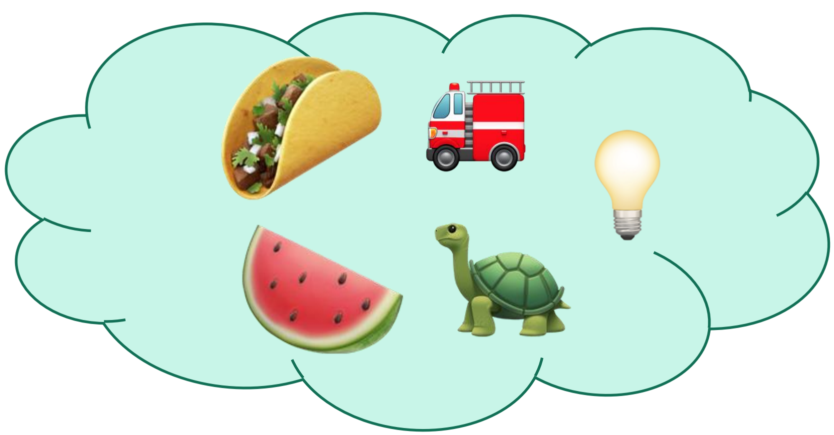
First j items

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

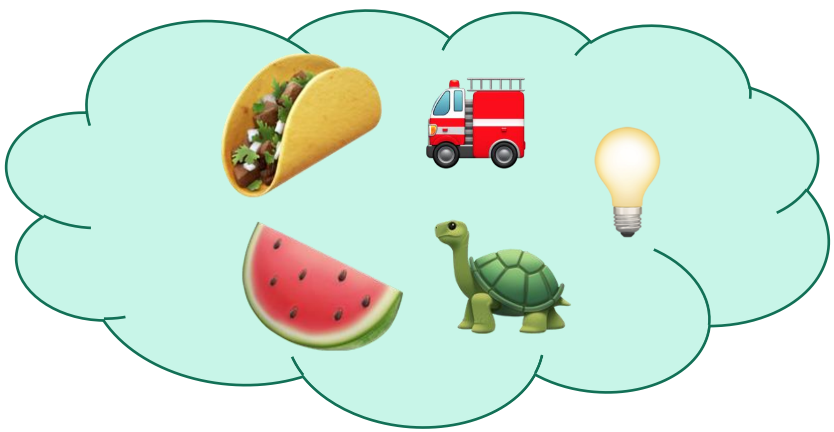
Capacity x

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

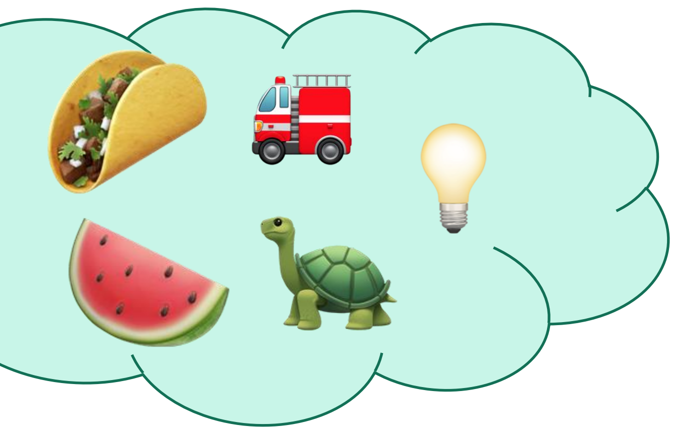
Capacity x

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

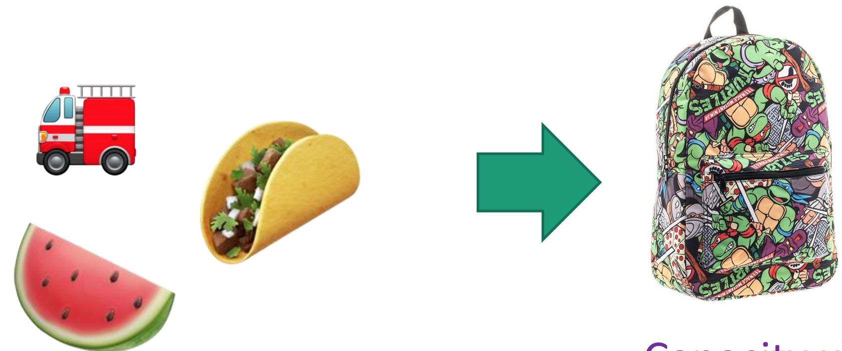
# Two cases

item j

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



First j items

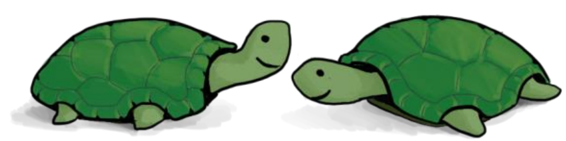


Capacity x

Value V

Use only the first j items

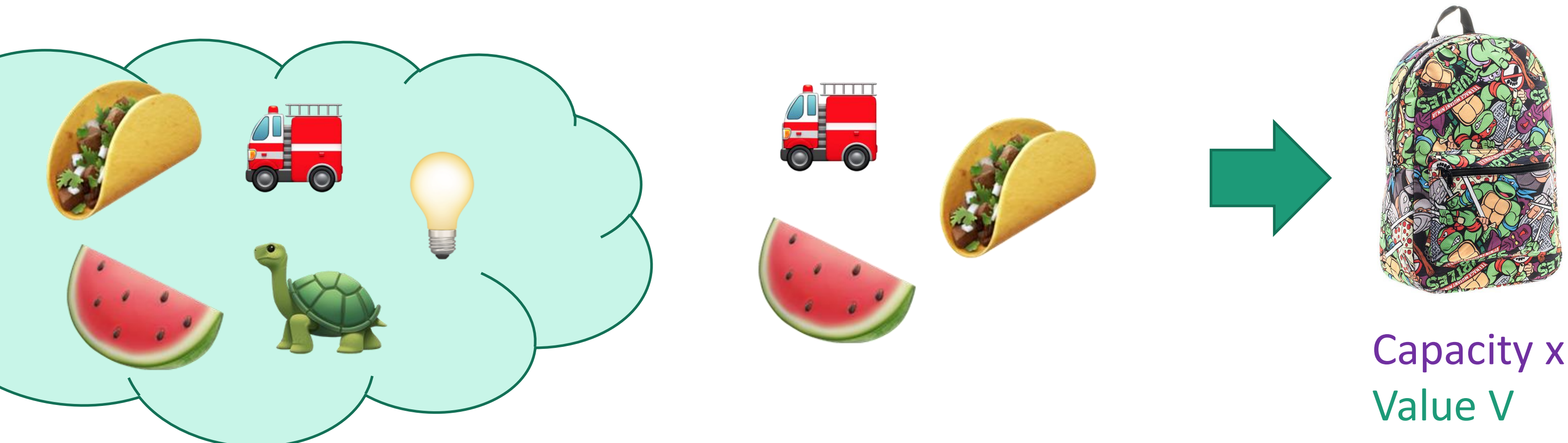
What lower-indexed
problem should we solve
to solve this problem?



# Two cases

item j

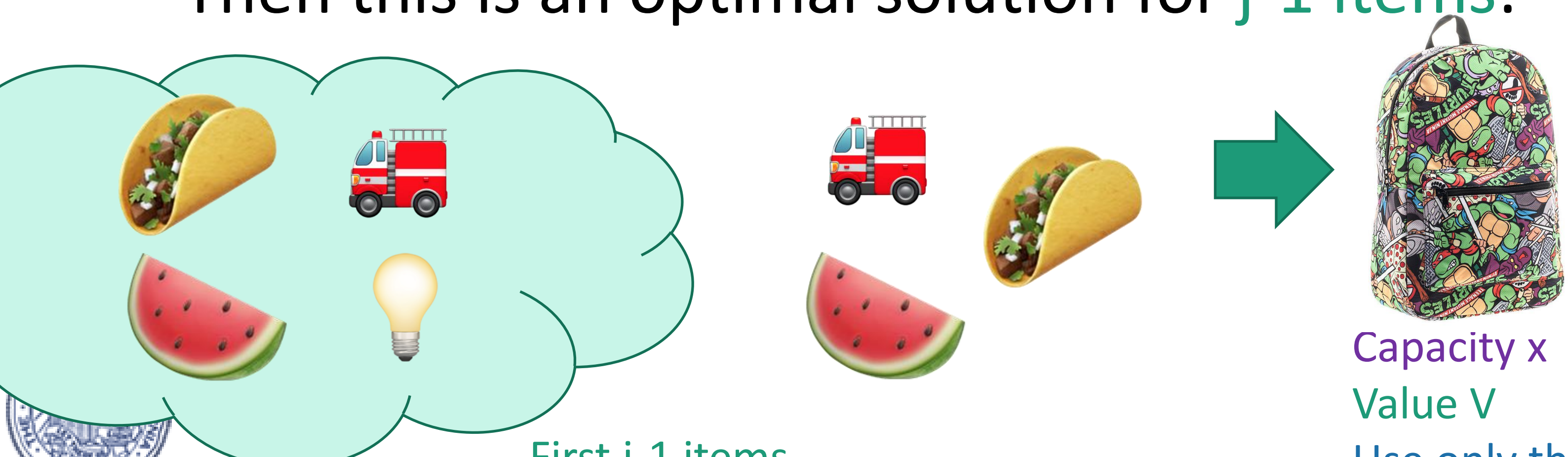
- **Case 1**: Optimal solution for j items does not use item j.



First j items

Capacity x
Value V
Use only the first j items

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



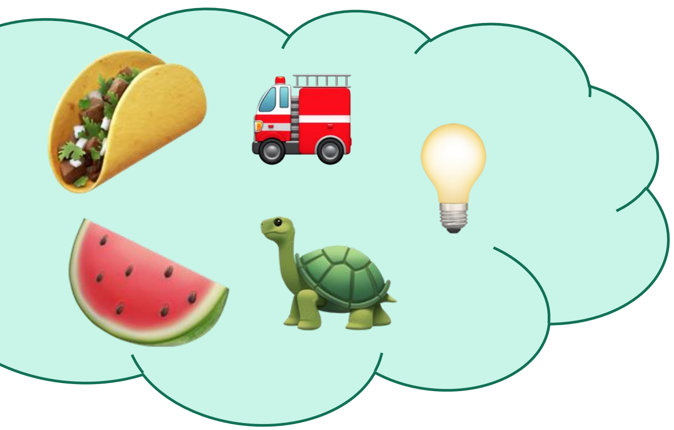
First j-1 items

Capacity x
Value V
Use only the first j-1 items.

# Two cases

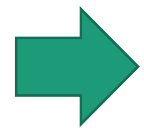
item j

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



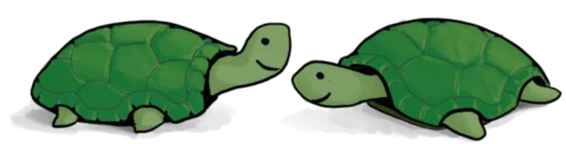
First j items

Weight $w_j$
Value $v_j$



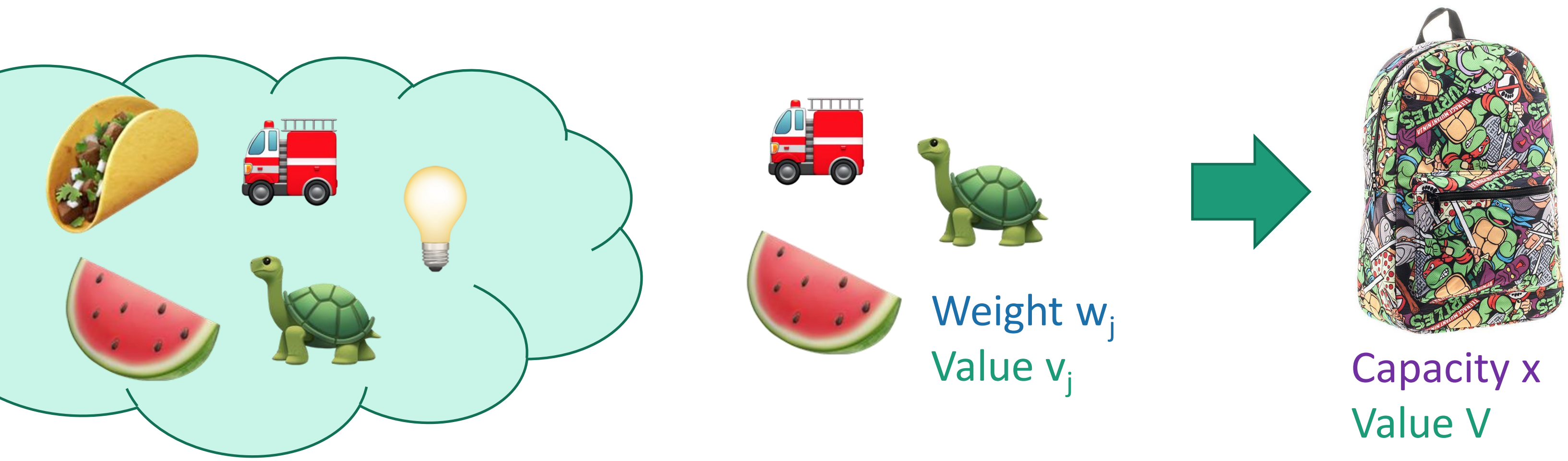
Capacity x
Value V
Use only the first j items

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

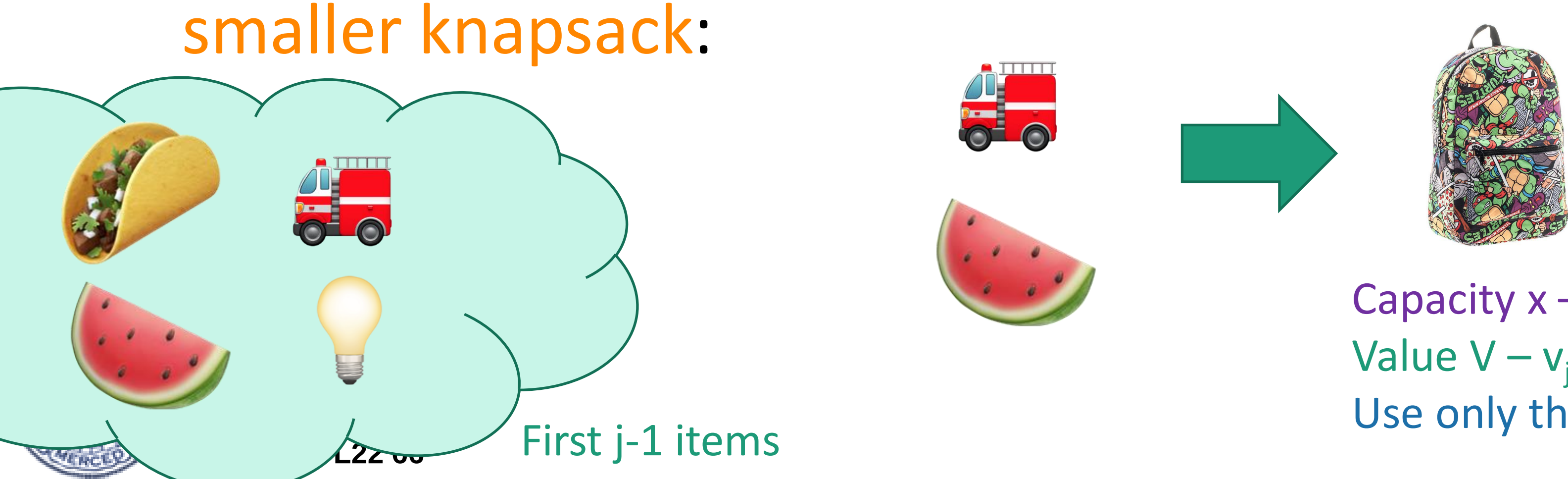


# Two cases


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




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

# 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.

# Next lecture

- Greedy algorithms!


**Gordon Geckko in Wall Street (1987)**