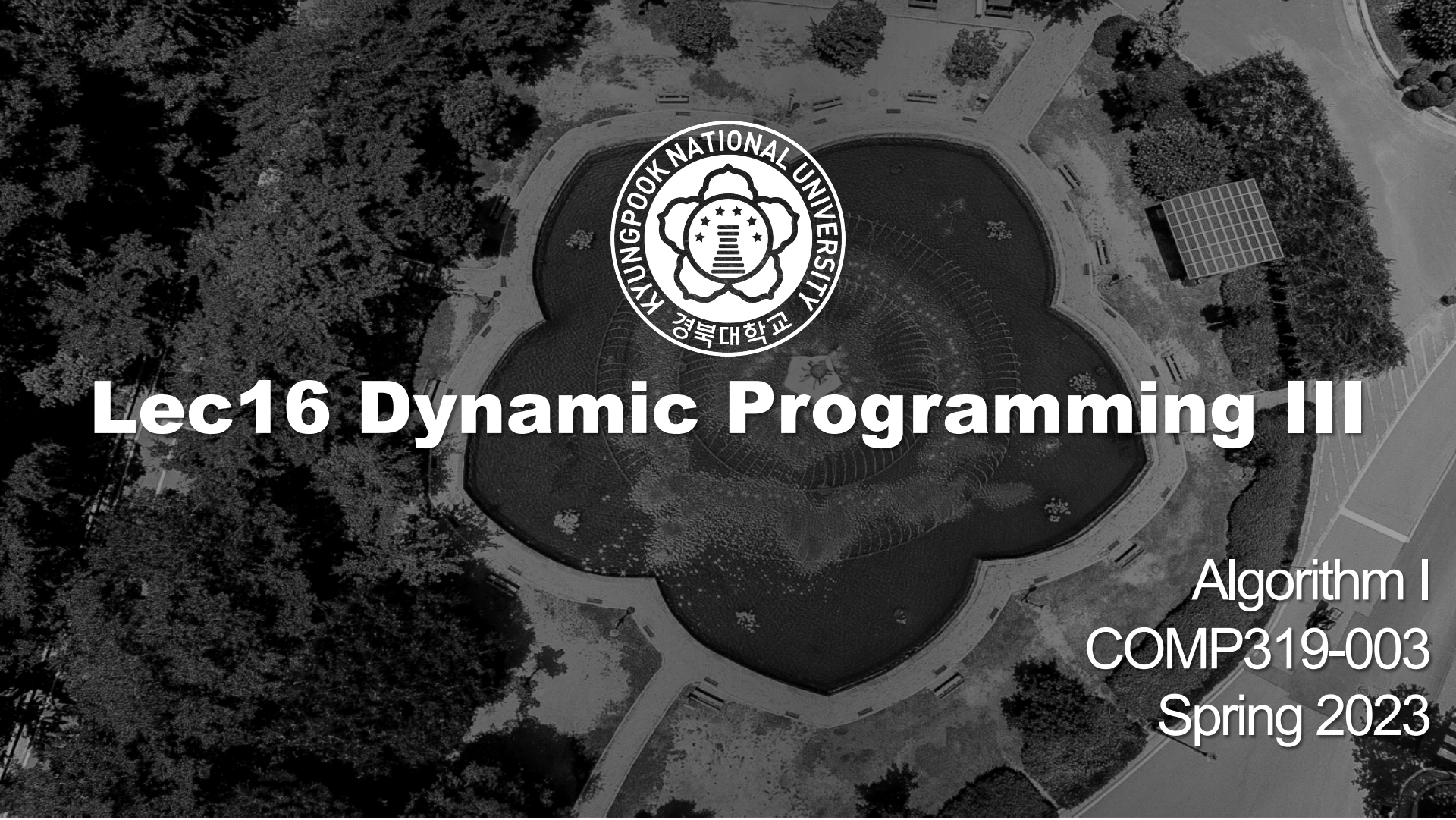


Lec16 Dynamic Programming III

Algorithm I
COMP319-003
Spring 2023

Instructor: Jiyeon Lee

School of Computer Science and Engineering

Kyungpook National University **(KNU)**

Last time

- Dynamic programming is an algorithm design paradigm.
- Basic idea:
  - Identify optimal sub-structure
    - Optimum to the big problem is built out of optima of small sub-problems
  - Take advantage of overlapping sub-problems
    - Only solve each sub-problem once, then use it again and again
  - Keep track of the solutions to sub-problems in a table as you build to the final solution.

Outline

1. More examples of DP:
   - Longest Common Subsequence
   - Knapsack Problem

- *Reading: CLRS 15*

Longest Common Subsequence

Longest Common Subsequence

- How similar are these two species?

DNA:
AGCCCTAAGGGCTACCTAGCTT

DNA:
GACAGCCTACAAGCGTTAGCTTG

Longest Common Subsequence

- How similar are these two species?

DNA:

AGCCCTAAGGGCTACCTAGCTT

DNA:

GACAGCCTACAAGCGTTAGCTTG

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

AGCCTAAGCTTAGCTT

Longest Common Subsequence

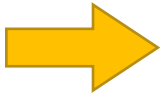
- Subsequence:
  - **BDFH** is a subsequence of **ABCDEFGH**
- If **X** and **Y** are sequences, a common subsequence is a sequence which is a subsequence of both.
  - **BDFH** is a common subsequence of **ABCDEFGH** and of **ABDFGHI**
- A longest common subsequence…
  - …is a common subsequence that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

We sometimes want to find these

- Applications in bioinformatics
- The UNIX command `diff`
- Merging in version control
  - svn, git, etc…

```
➜  ~ cat file1
A
B
C
D
E
F
G
[H                                                                    ]
➜  ~ cat file2
A
B
D
F
G
H
[I                                                                    ]
➜  ~ diff file1 file2
3d2
< C
5d3
< E
8a7
> I
➜  ~
```

Recipe for applying DP



Step 1: Identify optimal substructure.

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

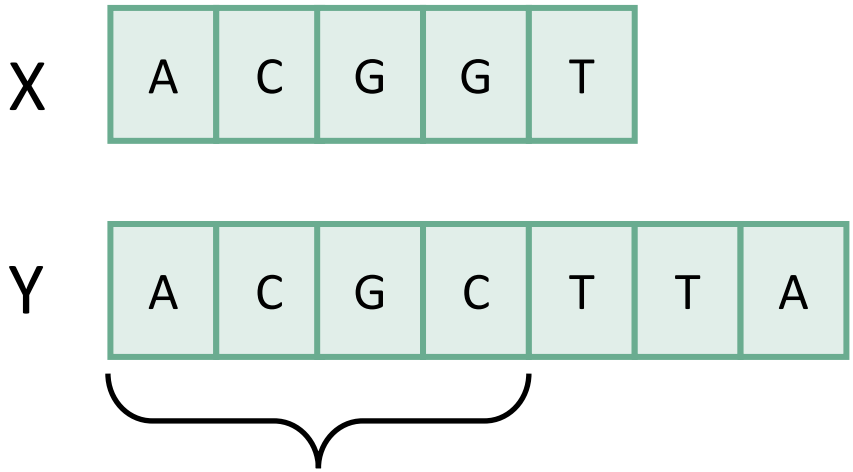
Step 3: Use dynamic programming to find the length of the longest common subsequence.

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.

Step 1: Optimal substructure

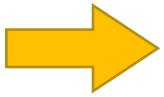
- Prefixes:



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

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i, j] = length_of_LCS(X_i, Y_j)
  - Examples: C[2, 3] = 2, C[4, 4] = 3

Recipe for applying DP



Step 1: Identify optimal substructure.

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

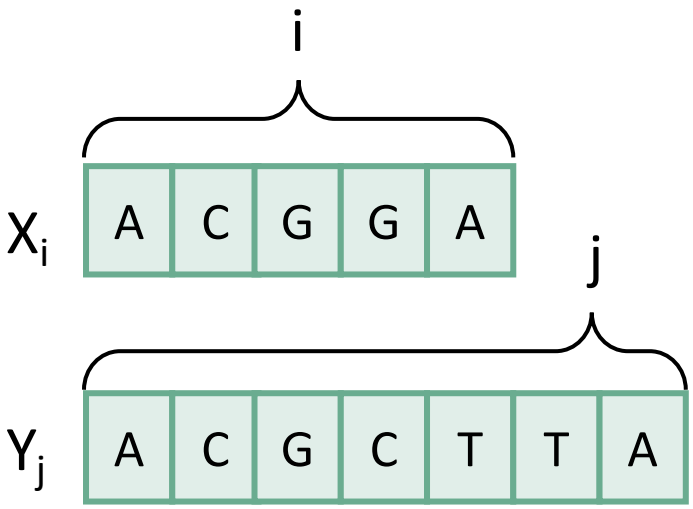
Step 3: Use dynamic programming to find the length of the longest common subsequence.

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.

Goal

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

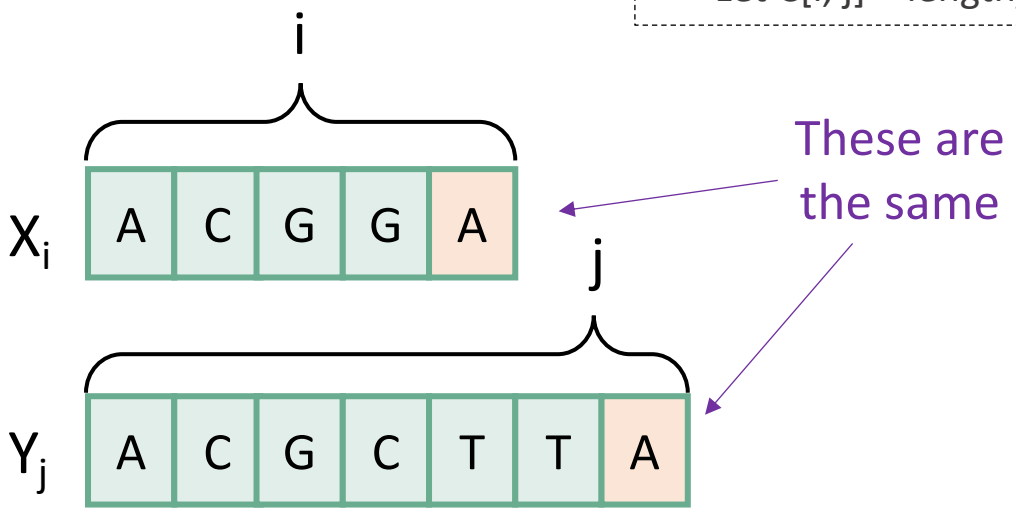


$$C[i, j] = \text{length\_of\_LCS}(X_i, Y_j)$$

Two cases

- **Case 1: X[i] = Y[j]**

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

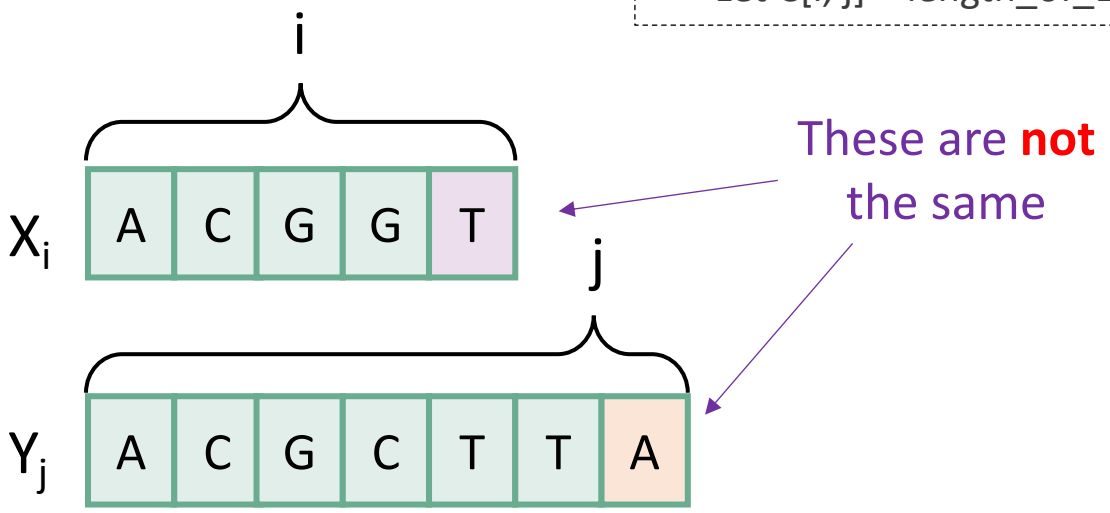


- Then, C[i, j] = 1 + C[i-1, j-1].
  - Because LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) followed by A

Two cases

- **Case 2: X[i] != Y[j]**

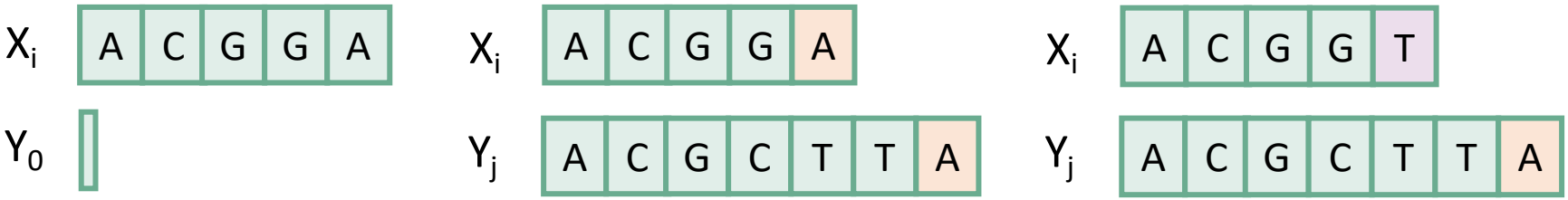
- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i, j] = length_of_LCS(X_i, Y_j)



- 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 T is not involved,
  - or LCS(X_i, Y_j) = LCS(X_i, Y_{j-1}) and A is not involved.

Recursive formulation

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \quad \text{Case 0} \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \quad \text{Case 1} \\ \max\{ C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \quad \text{Case 2} \end{cases}$$



Recipe for applying DP

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.

Longest Common Subsequence (LCS)

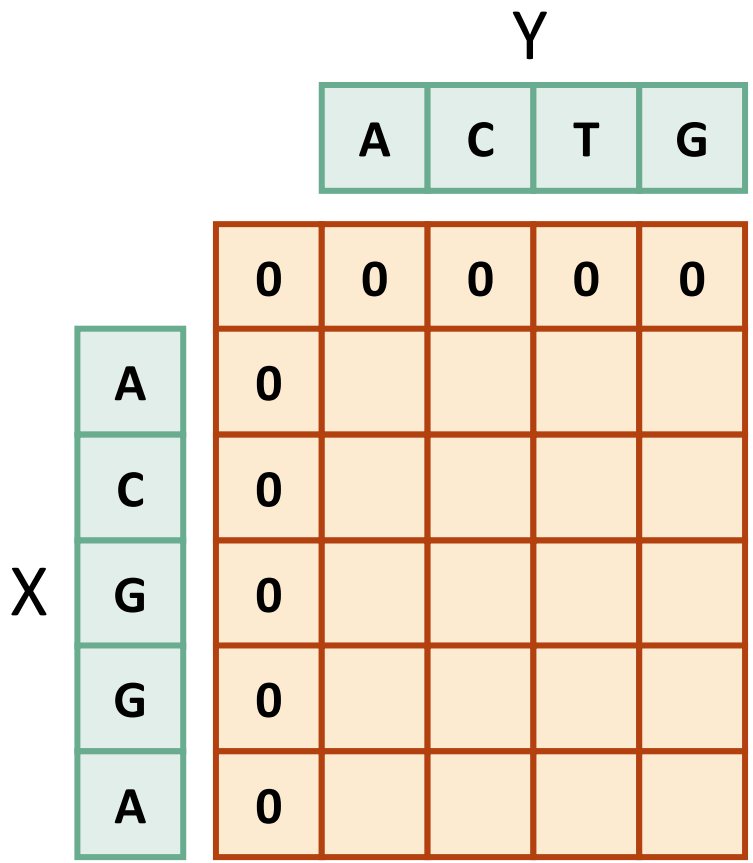
- **LCS**(X, Y):
  - C[i, 0] = C[0, j] = 0 for all i = 0 ,…, m, j=0, …n.
  - **For** i = 1,…,m and j = 1,…,n:
    - **If** X[i] = Y[j]:
      - C[i, j] = C[i-1, j-1] + 1
    - **Else:**
      - C[i, j] = max{ C[i, j-1], C[i-1, j] }
  - Return C[m, n]

- Running time: O(nm)

Example

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

- if X[i] = Y[i], C[i-1, j-1]+1
- else max{C[i, j-1], C[i-1, j]}



Example

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

- if X[i] = Y[i], C[i-1, j-1]+1
- else max{C[i, j-1], C[i-1, j]}

		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

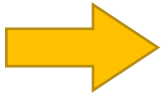
So the LCM of X and Y has length 3.

Recipe for applying DP

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.

Example

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

- if X[i] = Y[i], C[i-1, j-1]+1
- else max{C[i, j-1], C[i-1, j]}

		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

Example

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

- if X[i] = Y[i], C[i-1, j-1]+1
- else max{C[i, j-1], C[i-1, j]}

		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

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

Example

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

- if X[i] = Y[i], C[i-1, j-1]+1
- else max{C[i, j-1], C[i-1, j]}

		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

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

That 3 must have come from the 3 above it.

Example

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

- if X[i] = Y[i], C[i-1, j-1]+1
- else max{C[i, j-1], C[i-1, j]}

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

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

G

Example

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

- if X[i] = Y[i], C[i-1, j-1]+1
- else max{C[i, j-1], C[i-1, j]}

		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

- 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

Example

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

- if X[i] = Y[i], C[i-1, j-1]+1
- else max{C[i, j-1], C[i-1, j]}

		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

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

G

Example

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

- if X[i] = Y[i], C[i-1, j-1]+1
- else max{C[i, j-1], C[i-1, j]}

		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

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

C | G

Example

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

- if X[i] = Y[i], C[i-1, j-1]+1
- else max{C[i, j-1], C[i-1, j]}

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

This is the LCS!

A	C	G

Finding an LCS

- 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.
- So actually recovering the LCS from the table is much faster than building the table was.
- Altogether, we can find LCS(X, Y) in time O(mn).

Recipe for applying DP

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.

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?
  - Maybe a bit better. But doing much better (polynomially better) is an open problem.

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.

Knapsack Problem

0/1 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

0/1 Knapsack Problem

Q. What's the most valuable way to fill the knapsack?

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

Some notation

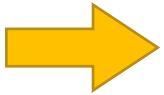
Item:    ... 

Weight: w_1 w_2 w_3 ... w_n

Value: v_1 v_2 v_3 v_n

Capacity: W 

Recipe for applying DP



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.

Optimal substructure

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



First solve the problem with few items

increase the size of the knapsacks

Then more items

Then yet more items


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

Optimal substructure

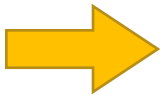
- Our sub-problems: indexed by x and j

First j items

Capacity x

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

Recipe for applying DP



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.

Relationship between sub-problems

- **Goal:** 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

- **Goal:** Want to write K[x, j] in terms of smaller sub-problems.
  - **Case 1**: Optimal solution for first j items does not use item j.
  - **Case 2**: Optimal solution for first j items does use item j.

item j

First j items

Capacity x

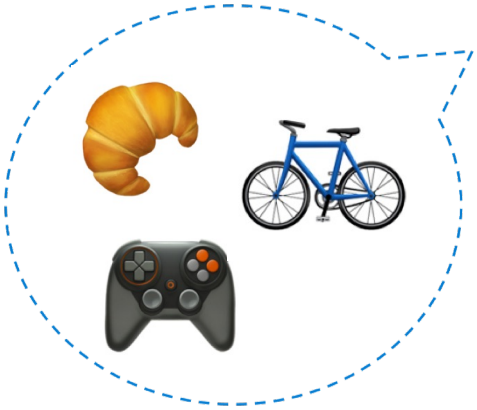
K[x, j] = optimal solution for a knapsack of size x using only the first j items.

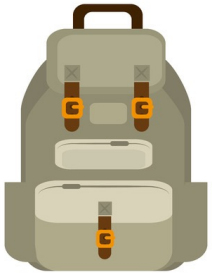
Two cases

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

item j

First j items

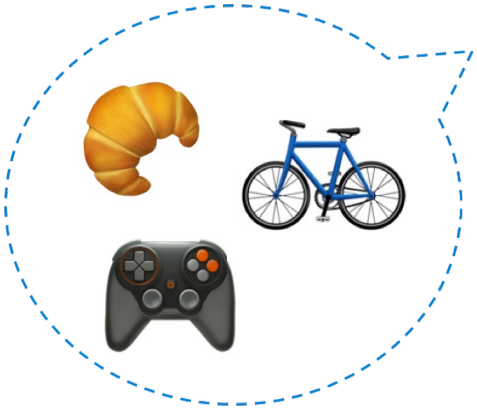




Capacity x, Value V
Use only the first j items

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

First j-1 items



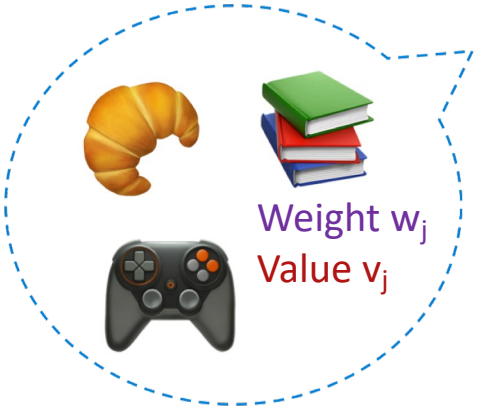
Capacity x, Value V
Use only the first j-1 items

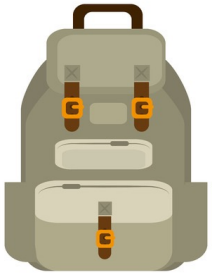
Two cases

- **Case 2**: Optimal solution for first j items does use item j.

item j

First j items

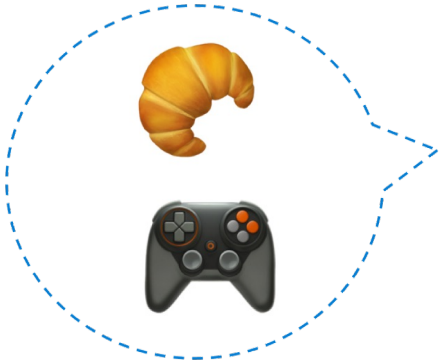




Capacity x, Value V
Use only the first j items

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

First j-1 items



Capacity x - w_j, Value V - v_j
Use only the first j-1 items

Recursive relationship

- Let K[x, j] be the optimal value for:
  - capacity x,
  - with j items.

$$K[x, j] = \max\{ K[x, j-1] , K[x - w_j, j-1] + v_j \}$$

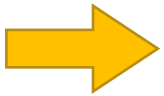
Case 1 Case 2

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

Recipe for applying DP

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.

Bottom-up DP algorithm

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

W: capacity, n: number of items,
w: set of weights, v: set of values

```
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]  (Case 1)
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }  (Case 2)
return K[W, n]
```

- Running time: O(nW)

Example

	x = 0	x = 1	x = 2	x = 3
j = 0	0	0	0	0
j = 1	0			
j = 2	0			
j = 3	0			

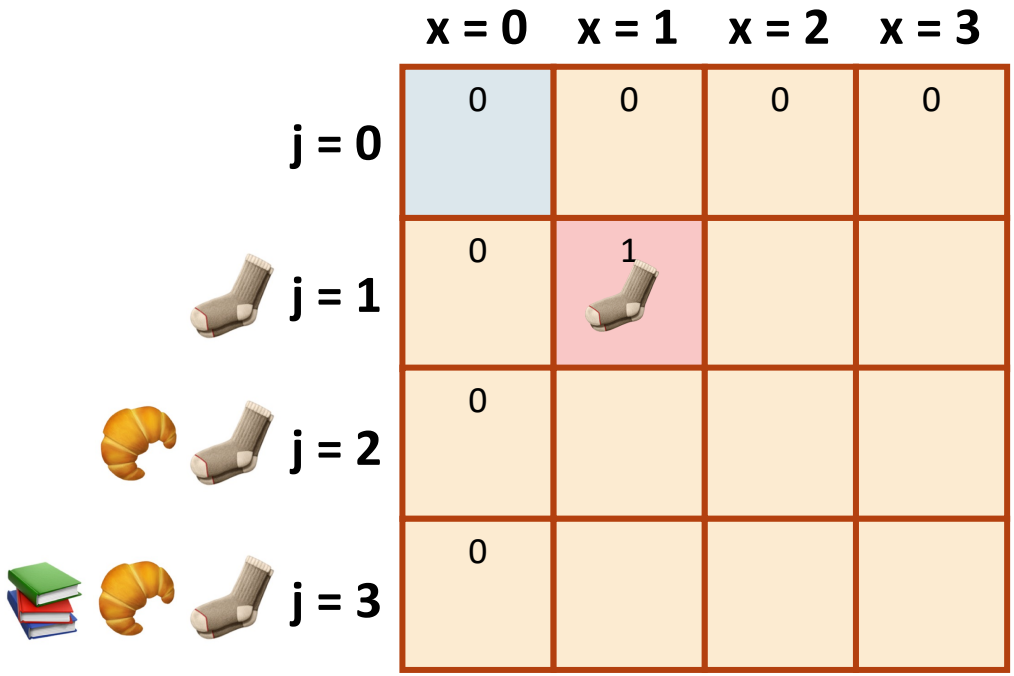
```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```

Example

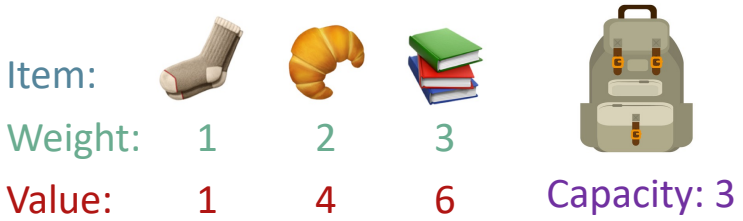
	x = 0	x = 1	x = 2	x = 3
j = 0	0	0	0	0
j = 1	0	0		
j = 2	0			
j = 3	0			

```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```

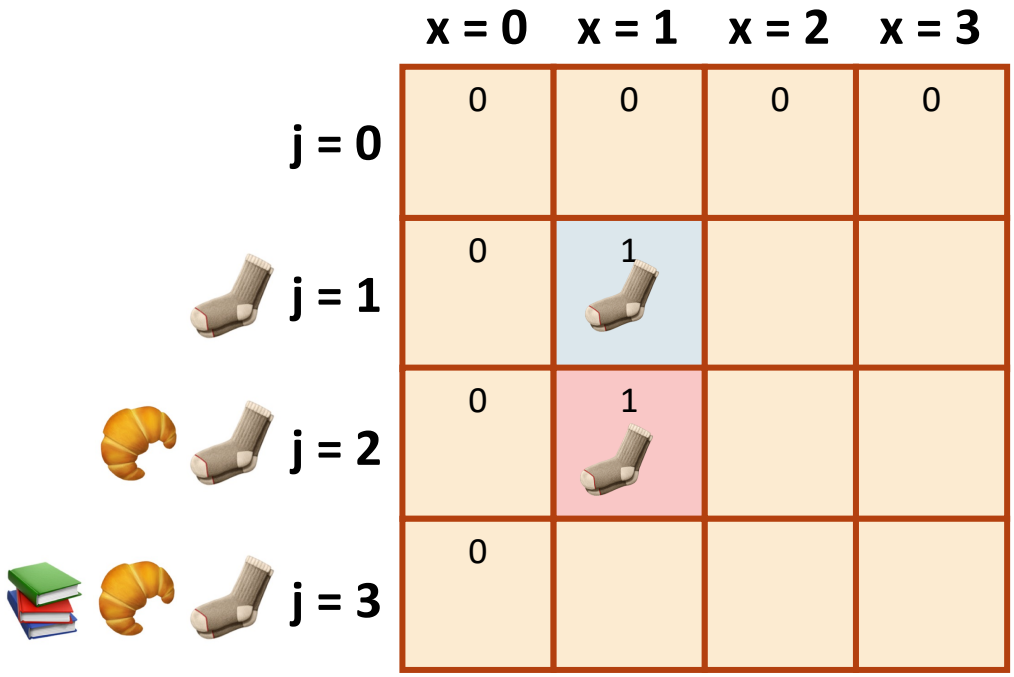
Example



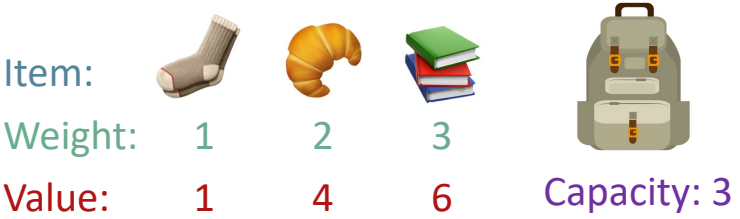

```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```



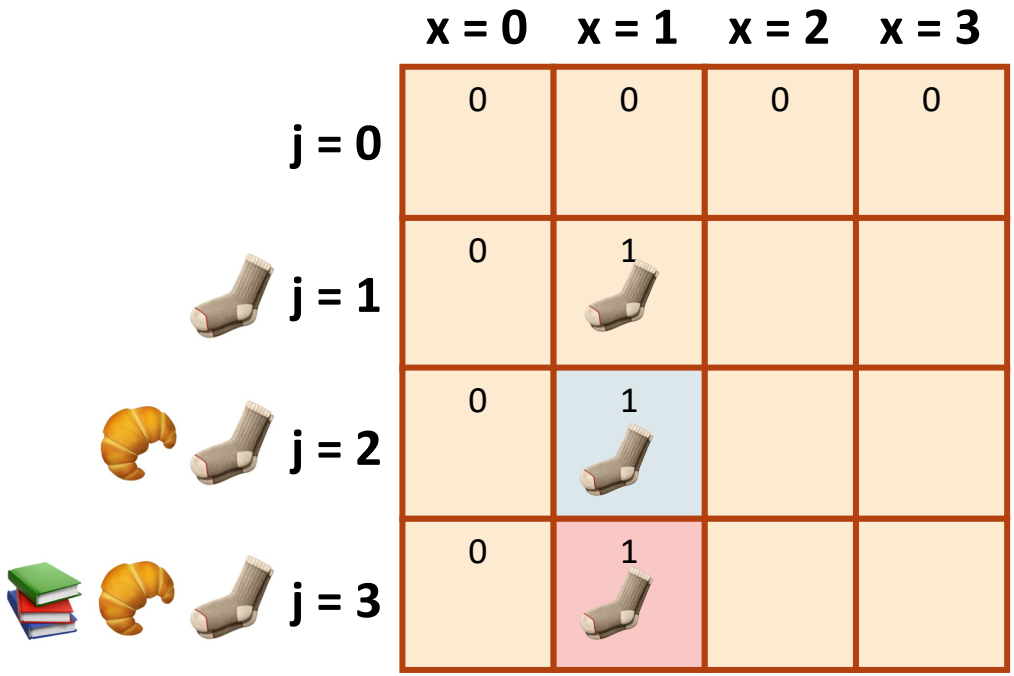
Example



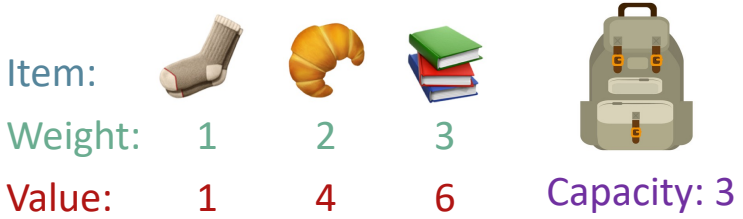

```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```



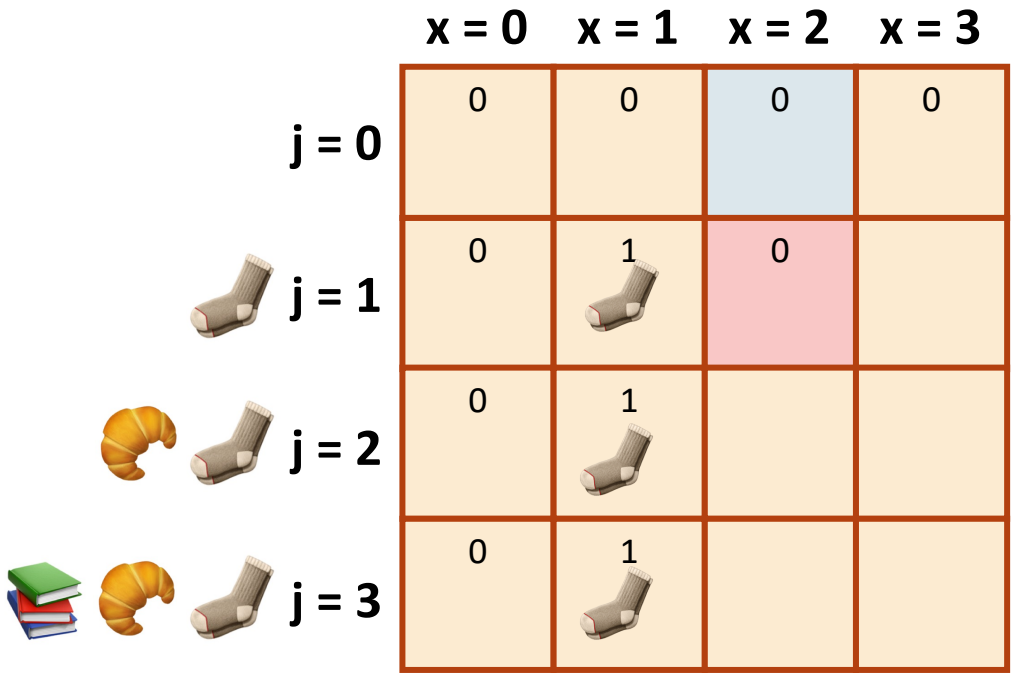
Example



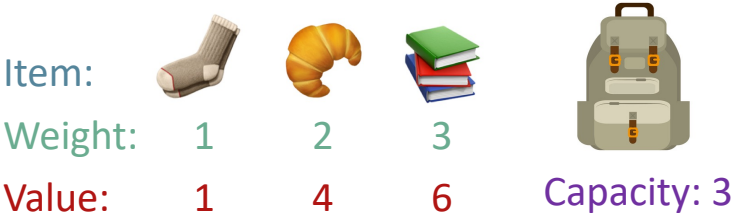

```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```



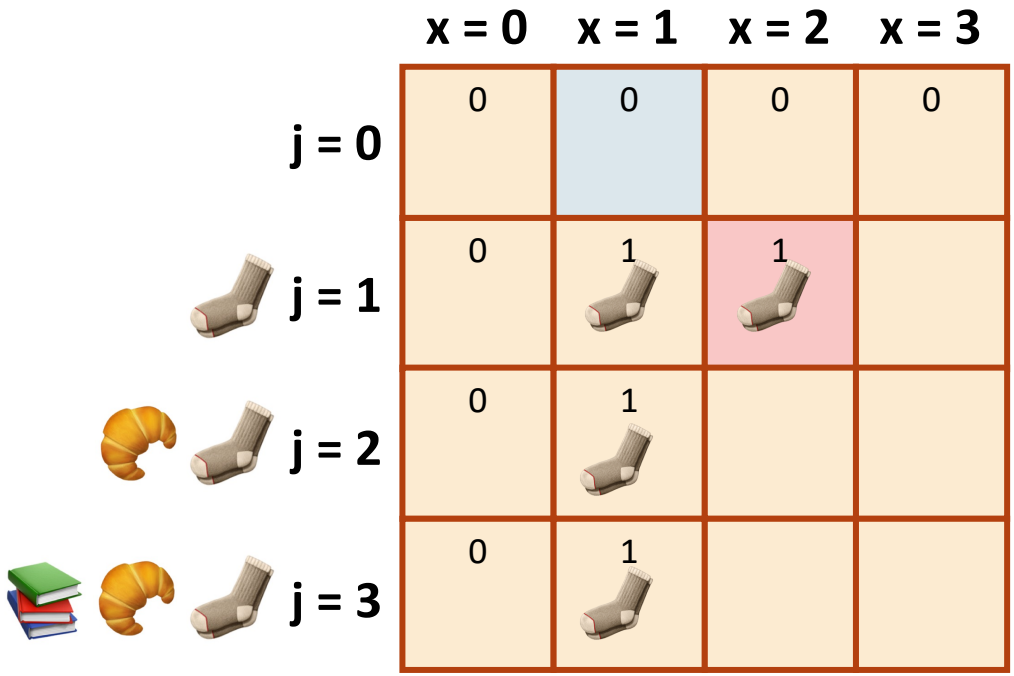
Example




```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```



Example

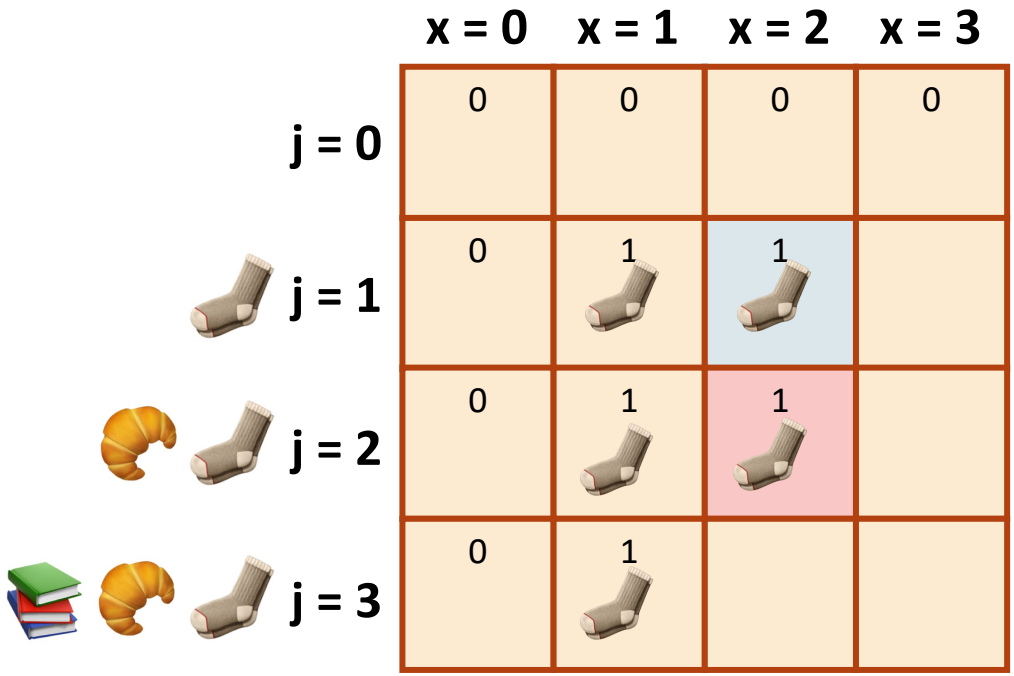



```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```

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

Capacity: 3

Example

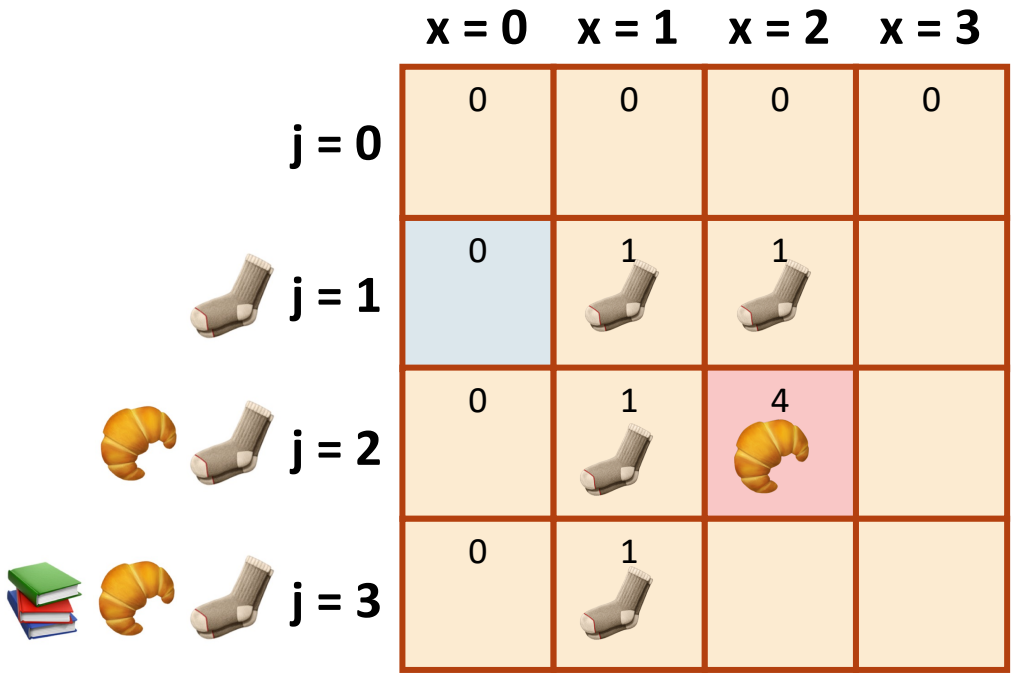



```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```

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

Capacity: 3

Example

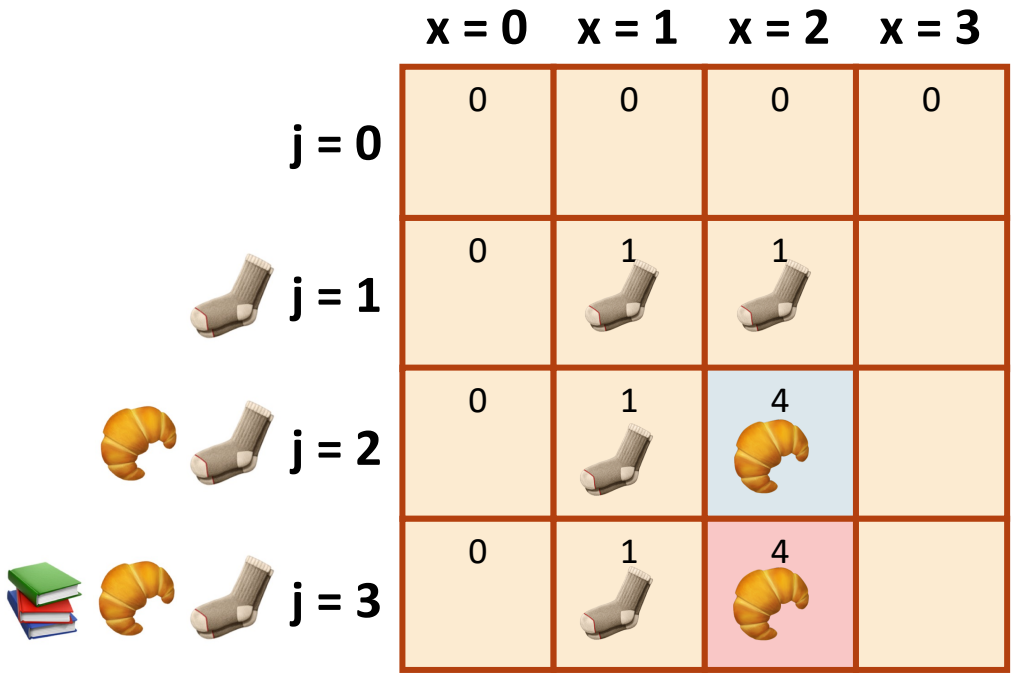



```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```

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

Capacity: 3

Example

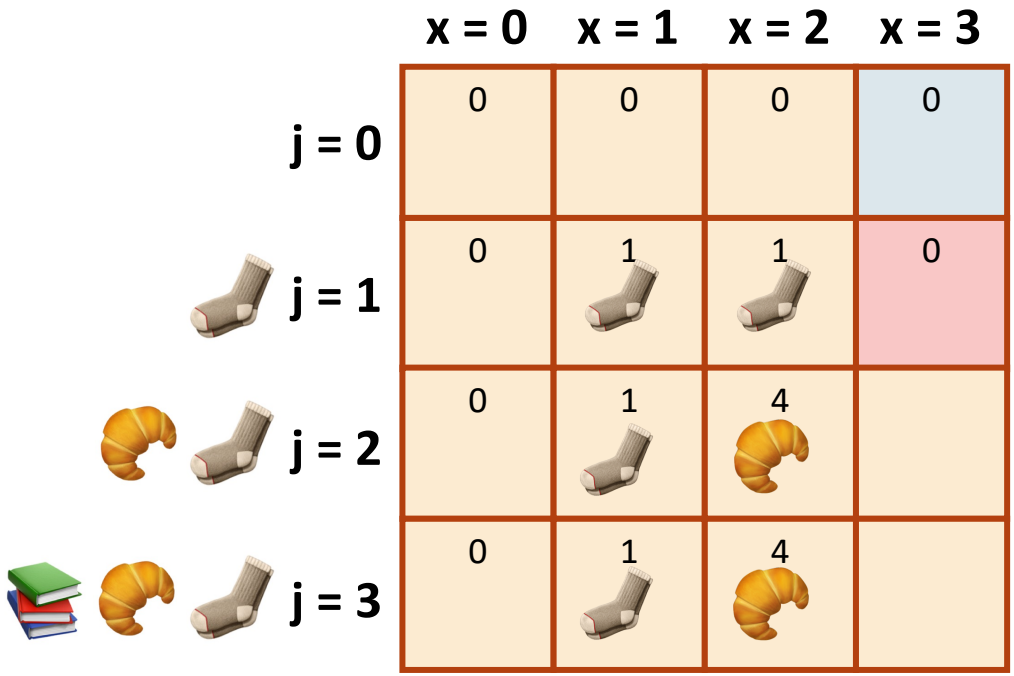



```
for x = 1, ..., W:
    for j = 1, ..., n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```

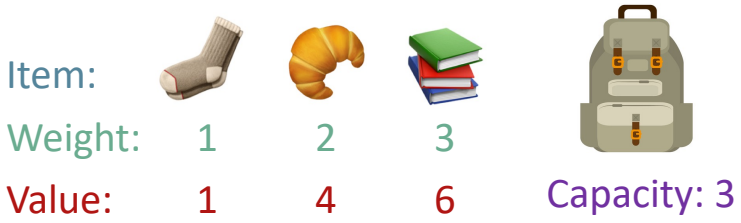
Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

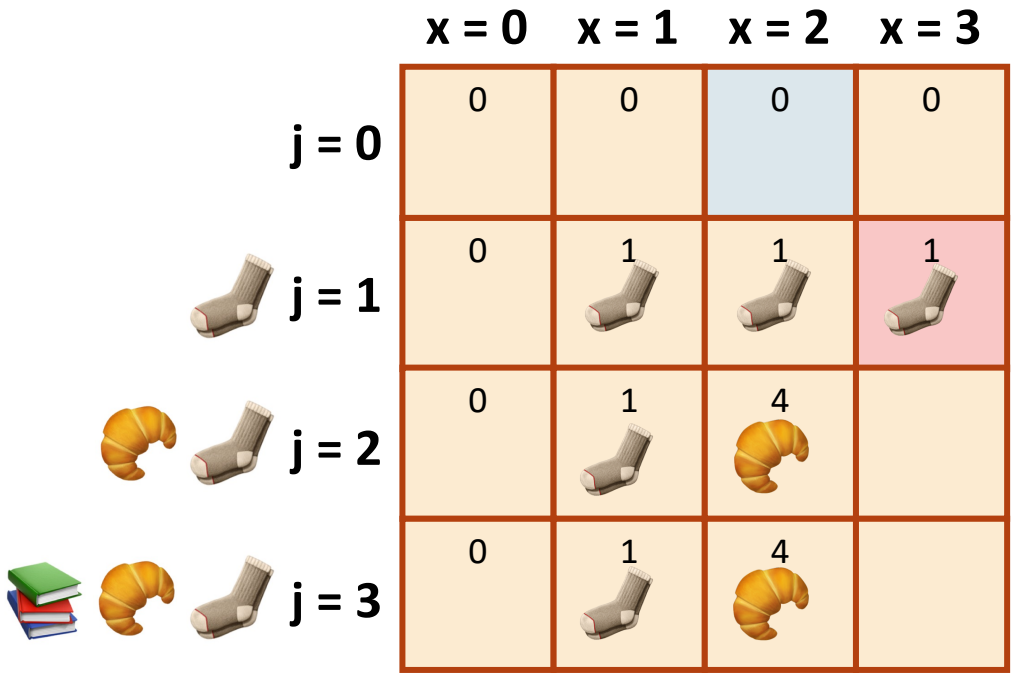
Example



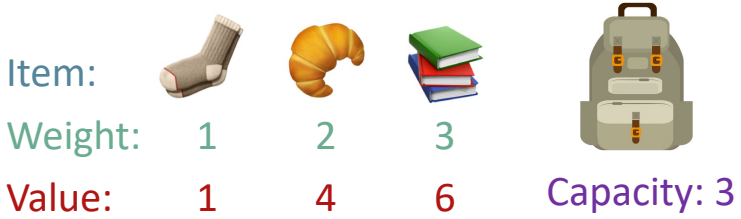

```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```



Example




```
for x = 1, ..., W:
    for j = 1, ..., n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```

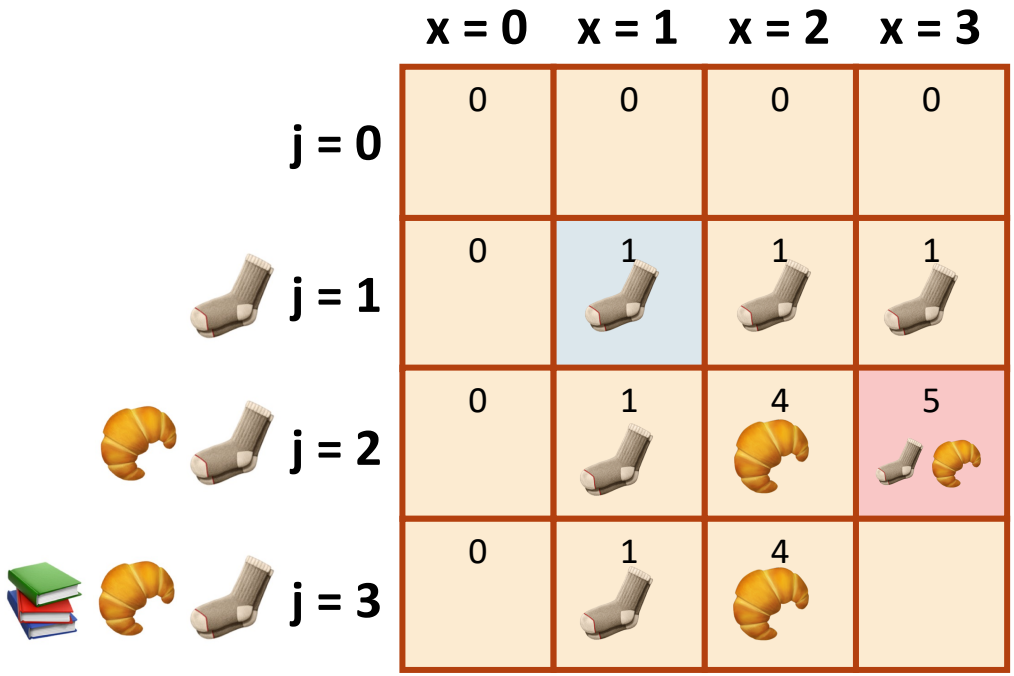


Example

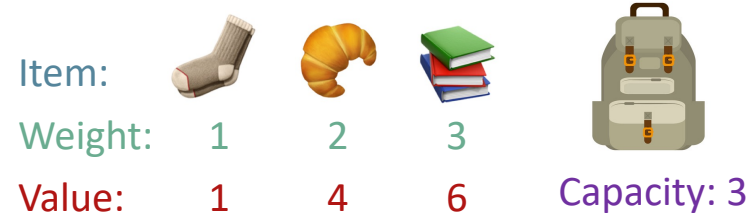
	x = 0	x = 1	x = 2	x = 3
j = 0	0	0	0	0
j = 1	0	1	1	1
j = 2	0	1	4	1
j = 3	0	1	4	

```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```

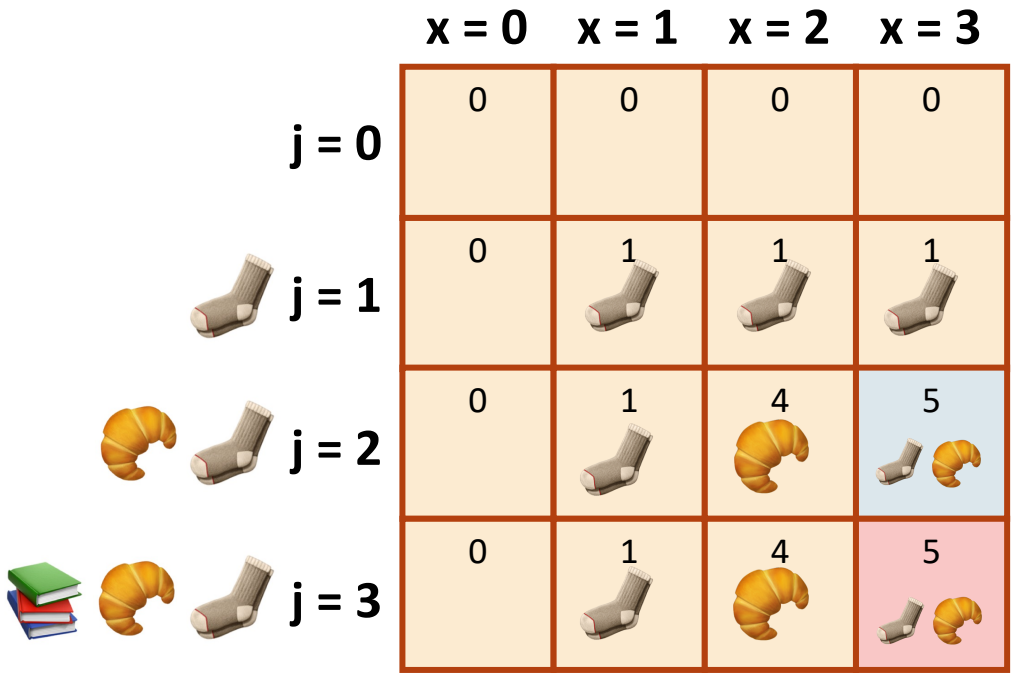
Example



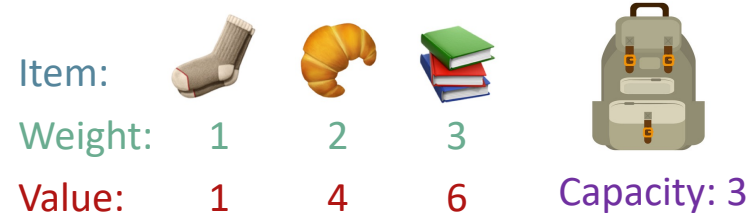

```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```



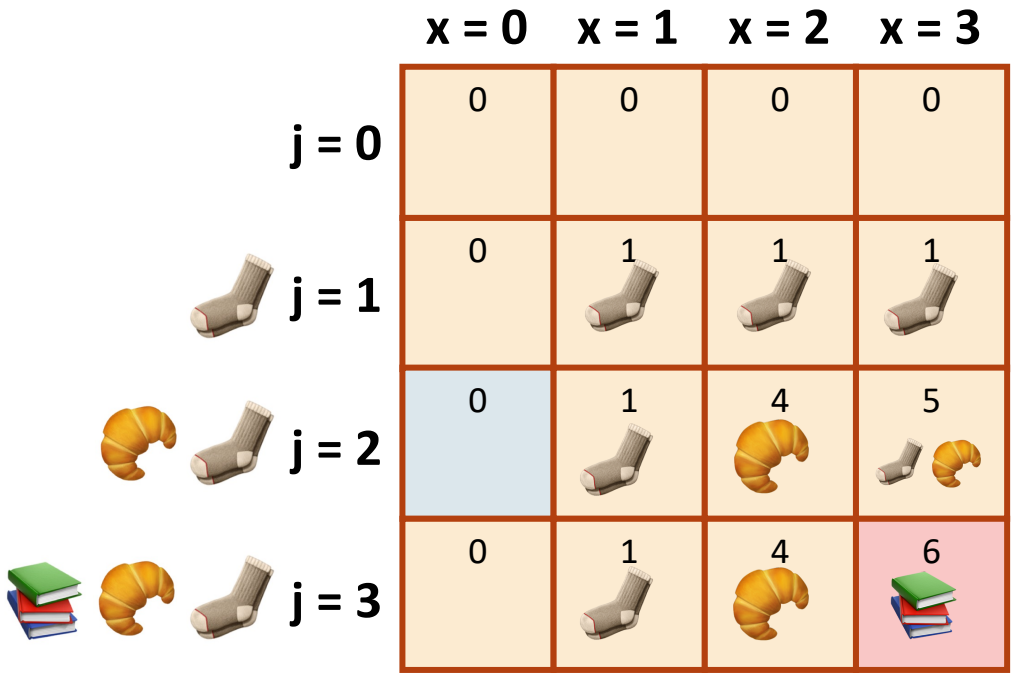
Example



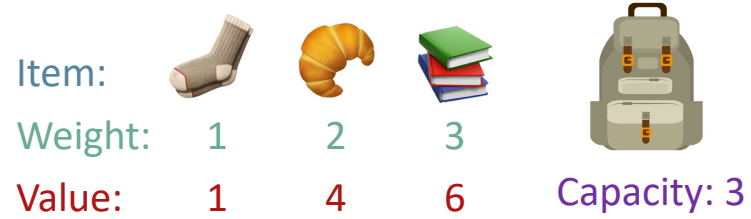

```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```



Example




```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```



Example

	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 a book in your knapsack!

```
for x = 1, …, W:
    for j = 1, …, n:
        K[x, j] = K[x, j – 1]
        if w_j ≤ x:
            K[x, j] = max{ K[x, j], K[x – w_j, j – 1] + v_j }
```

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.

Recap

- Today we saw examples of how to come up with dynamic programming algorithms.
  - Longest Common Subsequence
  - 0/1 Knapsack
- There is a **recipe** for dynamic programming algorithms.
- Sometimes coming up with the right substructure takes some creativity.
  - Check out additional examples/practice problems in CLRS! ☺

Next time

- Greedy algorithms!


Any Question?