Module 7 – Dynamic Programming

SIT320 – Advanced Algorithms

Dr. Nayyar Zaidi

From Mod3b

- 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!

Longest Common Subsequence

How similar are these two species?



DNA:

AGCCCTAAGGGCTACCTAGCTT



DNA:
GACAGCCTACAAGCGTTAGCTTG

Longest Common Subsequence

How similar are these two species?





Pretty similar, their DNA has a long common subsequence:

AGCCTAAGCTTAGCTT

DNA:

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...

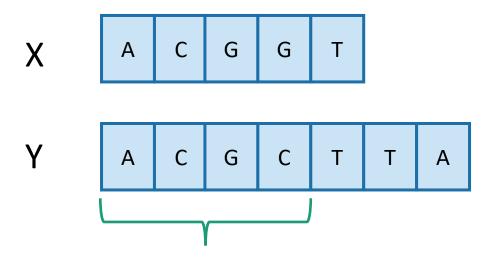
```
[DN0a22a660:~ mary$ cat file1
[DN0a22a660:~ mary$ cat file2
[DN0a22a660:~ mary$ diff file1 file2
3d2
5d3
DN0a22a660:~ mary$ ■
```

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

Prefixes:



Notation: denote this prefix **ACGC** by Y₄

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

Optimal substructure ctd.

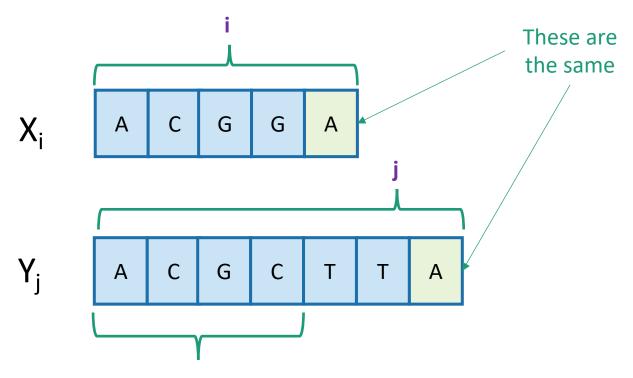
- Subproblem:
 - finding LCS's of prefixes of X and Y.
- Why is this a good choice?
 - There's some relationship between LCS's of prefixes and LCS's of the whole things.
 - These subproblems overlap a lot.

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



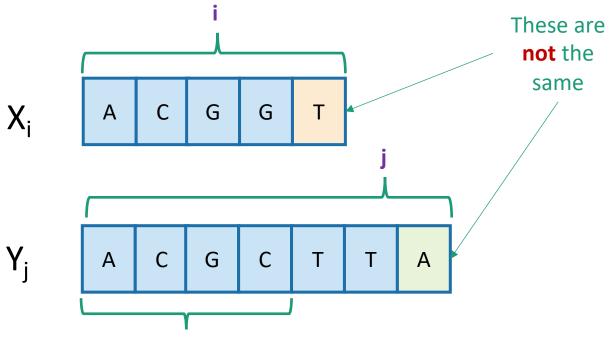
Notation: denote this prefix **ACGC** by Y₄

- 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

Case 2: 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_i)



Notation: denote this prefix **ACGC** by Y₄

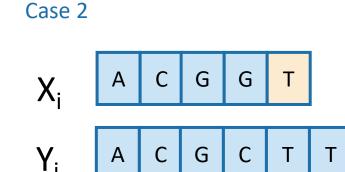
- Then C[i,j] = max{ C[i-1,j], C[i,j-1] }.
 - either $LCS(X_i,Y_j) = LCS(X_{i-1},Y_j)$ and \top is not involved,
 - or $LCS(X_i,Y_i) = LCS(X_i,Y_{i-1})$ and A is not involved,

Recursive formulation of the optimal solution

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

X_i A C G G A

Y_i A C G C T T A



Recipe for applying Dynamic Programming

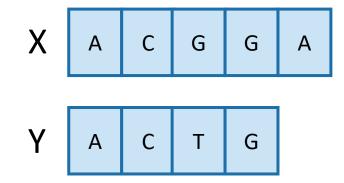
- Step 1: Identify optimal substructure.
- Step 2: Find a recursive formulation for the length of the longest common subsequence.
- Step 3: Use dynamic programming to find the length of the longest common subsequence.
- Step 4: If needed, keep track of some additional info so that the algorithm from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

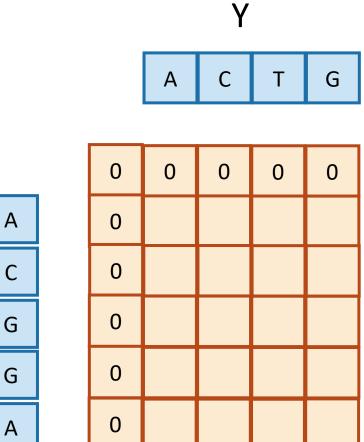
LCS DP omg bbq

- LCS(X, Y):
 - C[i,0] = C[0,j] = 0 for all i = 1,...,m, j=1,...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] }

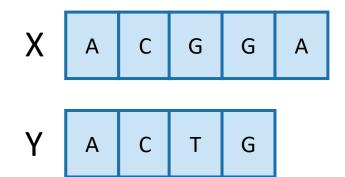
Running time: O(nm)

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





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



Υ							
Α	С	Т	G				

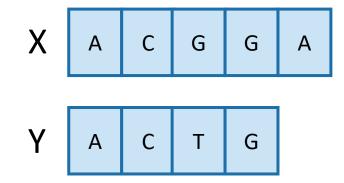
0	0	0	0	0
0	1	1	1	1
0	1	2	2	2
0	1	2	2	3
0	1	2	2	3
0	1	2	2	3

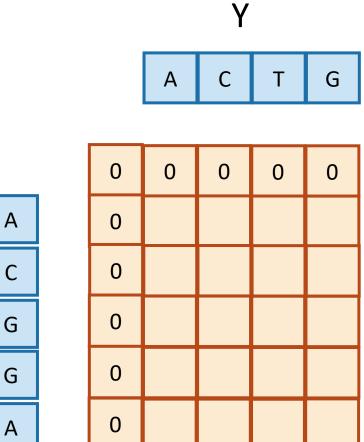
So the LCM of X and Y has length 3.

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

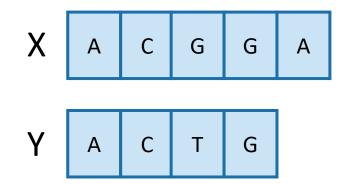
Recipe for applying Dynamic Programming

- 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.





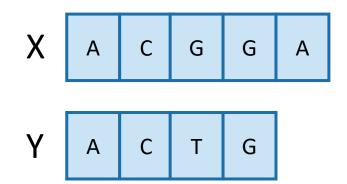
$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

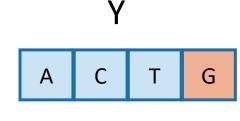


Υ					
Α	С	Т	G		

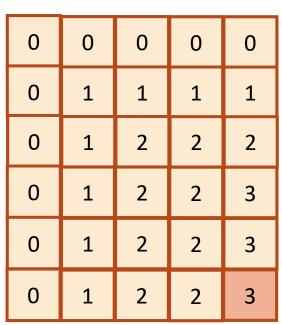
0	0	0	0	0
0	1	1	1	1
0	1	2	2	2
0	1	2	2	3
0	1	2	2	3
0	1	2	2	3

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

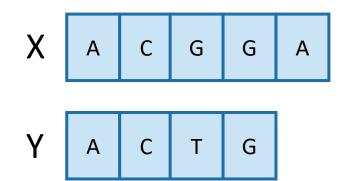


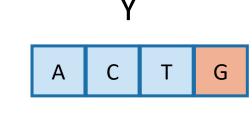


 Once we've filled this in, we can work backwards.



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



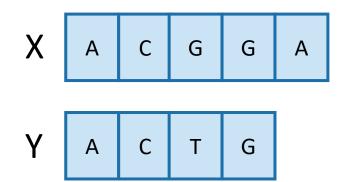


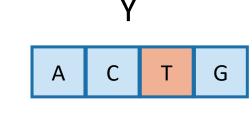
 Once we've filled this in, we can work backwards.

0	0	0	0	0
0	1	1	1	1
0	1	2	2	2
0	1	2	2	3
0	1	2	2	3
0	1	2	2	3

That 3 must have come from the 3 above it.

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

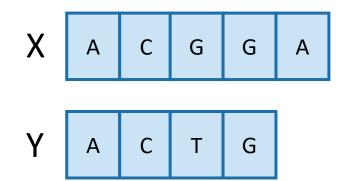


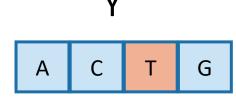


- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

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

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



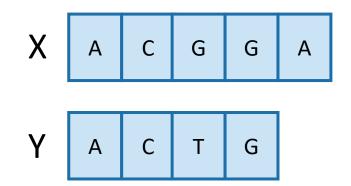


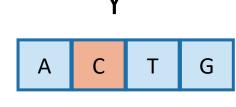
0	0	0	0	0
0	1	1	1	1
0	1	2	2	2
0	1	2	2	3
0	1	2	2	3
0	1	2	2	3

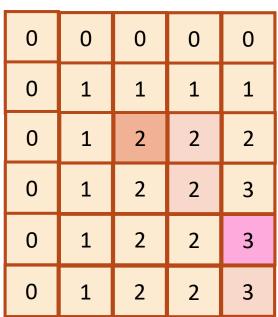
- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

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

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



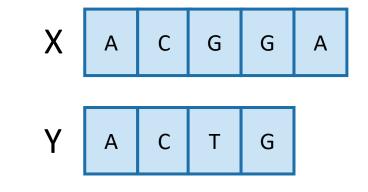


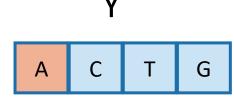


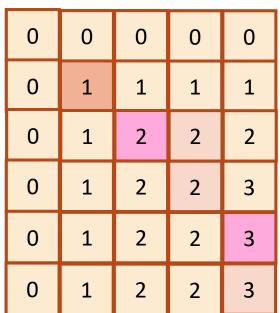
- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

G

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





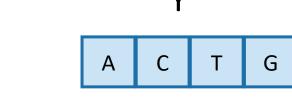


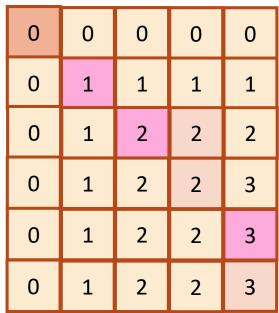
- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

C G

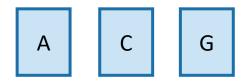
$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1],C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$







- 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!

Α

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

This gives an algorithm to recover the actual LCS not just its length

- See lecture notes for pseudocode
- It runs in time O(n + m)
 - We walk up and left in an n-by-m array
 - We can only do that for n + m steps.
- So actually recovering the LCS from the table is much faster than building the table was.
- We can find LCS(X,Y) in time O(mn).

Recipe for applying Dynamic Programming

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

This pseudocode actually isn't so bad

- If we are only interested in the length of the LCS:
 - Since we go across the table one-row-at-a-time, we can only keep two rows if we want.
 - If we want to recover the LCS, we need to keep the whole table.
- Can we do better than O(mn) time?
 - A bit better.
 - By a log factor or so.
 - But doing much better (polynomially better) is an open problem!
 - If you can do it let me know:D

What have we learned?

- We can find 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:
 <th

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



Capacity: 10



Capacity: 10









Weight:

Item:

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

Value:



 W_2

 V_2



 W_3

V₃



 \mathbf{W}_{n}



Capacity: W

Recipe for applying Dynamic Programming

• Step 1: Identify optimal substructure.



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

Optimal substructure

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







First solve the problem for small knapsacks

Then larger knapsacks

Then larger knapsacks

Optimal substructure



Suppose this is an optimal solution for capacity x:

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





Capacity x Value V

• Then this optimal for capacity x - w_i:





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

Recipe for applying Dynamic Programming

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

Recursive relationship

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

$$K[x] = \max_{i} \left\{ \begin{array}{c} + \\ \\ \end{array} \right\}$$
 The maximum is over all i so that $w_i \leq x$. Optimal way to fill the smaller knapsack

$$K[x] = \max_{i} \{ K[x - w_{i}] + v_{i} \}$$

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

Recipe for applying Dynamic Programming

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

Let's write a bottom-up DP algorithm

- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - for x = 1, ..., 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} \{ w_{i} + w_{i} \}
= \max_{i} \{ K[x - w_{i}] + v_{i} \}
```

Why does this work?
Because our recursive relationship makes sense.

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

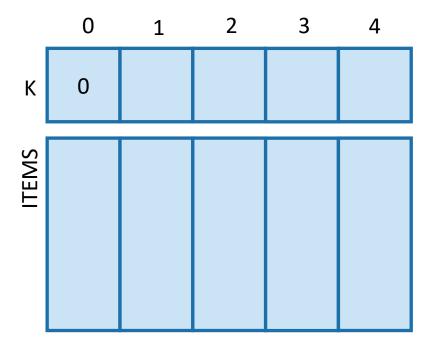
- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - return K[W]

```
K[x] = \max_{i} \{ w_{i} + w_{i} \}
= \max_{i} \{ K[x - w_{i}] + v_{i} \}
```

Let's write a bottom-up DP algorithm

- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0ITEMS[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] U { item i }
 - return ITEMS[W]

```
K[x] = \max_{i} \{ || \mathbf{K}[x - \mathbf{w}_{i}] + \mathbf{v}_{i} \}
= \max_{i} \{ || \mathbf{K}[x - \mathbf{w}_{i}] + \mathbf{v}_{i} \}
```



- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS $[0] = \emptyset$
 - for x = 1, ..., W:
 - K[x] = 0
 - **for** i = 1, ..., n:
 - if $w_i \leq x$:

Value:

- $K[x] = \max\{K[x], K[x w_i] + v_i\}$
- If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item i }
- return ITEMS[W]

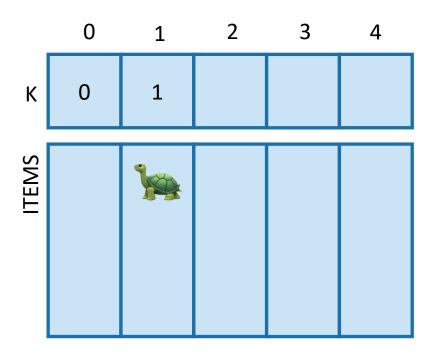




6

4

Capacity: 4



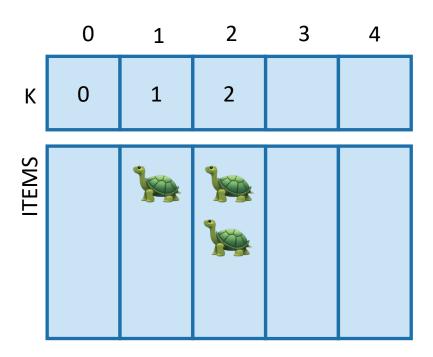
- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS[0] = Ø
 - for x = 1, ..., W:
 - K[x] = 0
 - for i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item i }
 - return ITEMS[W]



Value: 1 4 6



Capacity: 4



- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS[0] = Ø
 - for x = 1, ..., W:
 - K[x] = 0
 - for i = 1, ..., n:
 - if $w_i \leq x$:

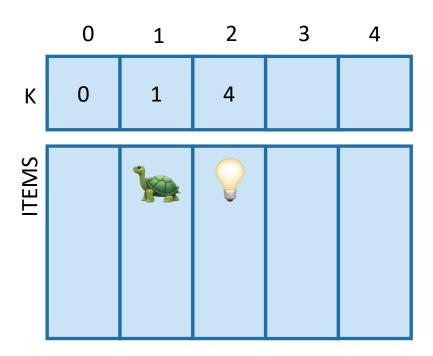
Value:

- $K[x] = \max\{K[x], K[x w_i] + v_i\}$
- If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item i }
- return ITEMS[W]



1 4 6





$$ITEMS[2] = ITEMS[0] +$$

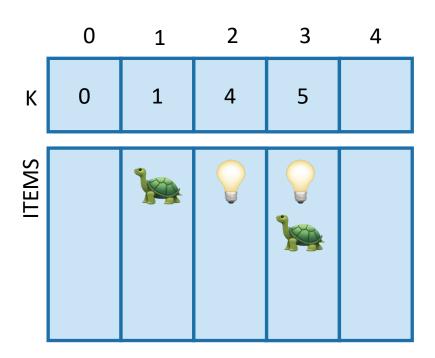
- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - $ITEMS[0] = \emptyset$
 - for x = 1, ..., W:
 - K[x] = 0
 - for i = 1, ..., n:
 - if $w_i \leq x$:

Value:

- $K[x] = \max\{K[x], K[x w_i] + v_i\}$
- If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item i }
- return ITEMS[W]



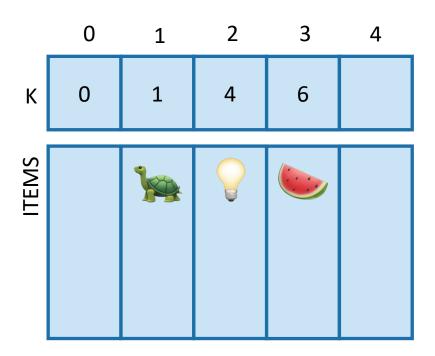




- UnboundedKnapsack(W, n, weights, values):
 - K[0] = 0
 - ITEMS $[0] = \emptyset$
 - for x = 1, ..., W:
 - K[x] = 0
 - for i = 1, ..., n:
 - if $w_i \leq x$:
 - $K[x] = \max\{K[x], K[x w_i] + v_i\}$
 - If K[x] was updated:
 - ITEMS[x] = ITEMS[x w_i] U { item i }
 - return ITEMS[W]







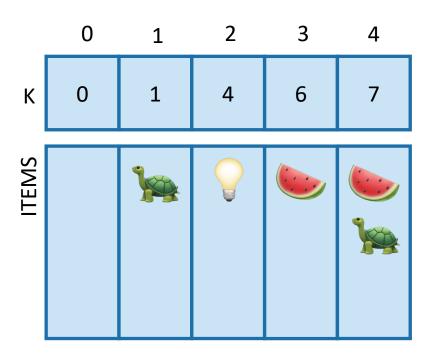
$$ITEMS[3] = ITEMS[0] +$$

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





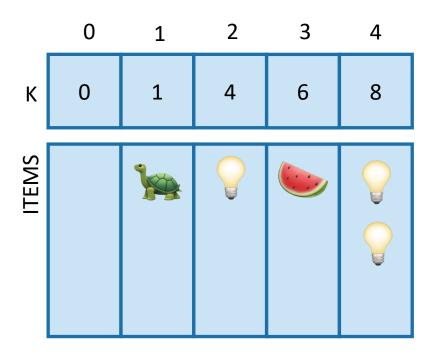
Capacity: 4



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







$$ITEMS[4] = ITEMS[2] +$$

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





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













Weight:

Item:

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.

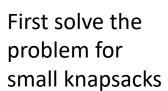


- Step 2: Find a recursive formulation for the value of the optimal solution.
- Step 3: Use dynamic programming to find the value of the optimal solution.
- Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- Step 5: If needed, code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

Optimal substructure: try 1

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







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:

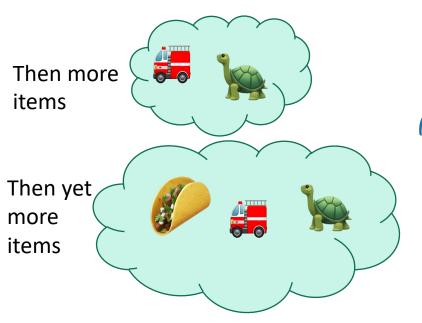
• 0/1 Knapsack with fewer items.

First solve the problem with few items









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

Two cases



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

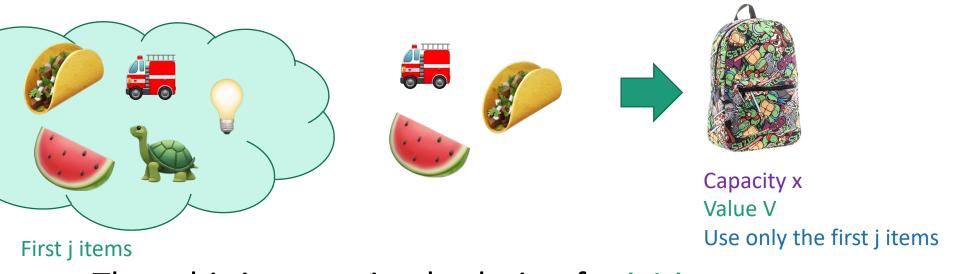


Capacity x

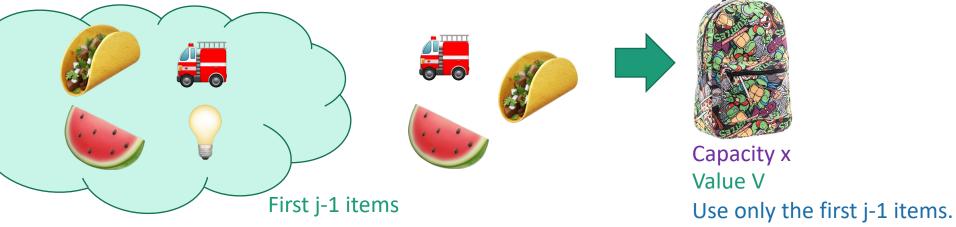
Two cases



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



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



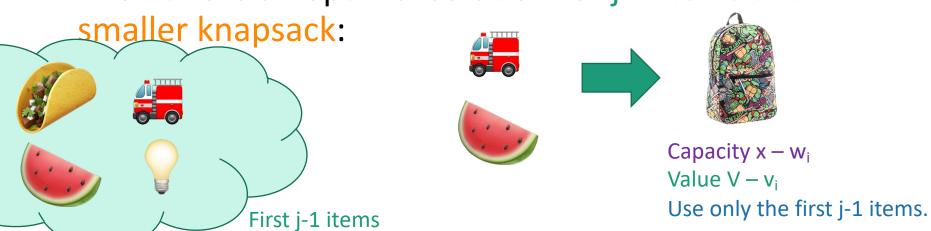
Two cases



• Case 2: Optimal solution for j items uses item j.



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



Recipe for applying Dynamic Programming

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

Recursive relationship

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

$$K[x,j] = max\{ K[x, j-1], K[x - w_{j,} j-1] + v_{j} \}$$
Case 1

Case 2

• (And K[x,0] = 0 and K[0,j] = 0).

Recipe for applying Dynamic Programming

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

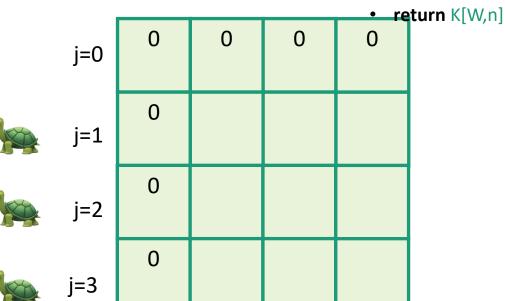
Bottom-up DP algorithm

```
Zero-One-Knapsack(W, n, w, v):
   • K[x,0] = 0 for all x = 0,...,W
   • K[0,i] = 0 for all i = 0,...,n
   • for x = 1,...,W:
       • for j = 1,...,n:
                               Case 1
           • K[x,j] = K[x, j-1]
           • if w_i \leq x:
                                                 Case 2
               • K[x,j] = max\{ K[x,j], K[x - w_i, j-1] + v_i \}
   return K[W,n]
```

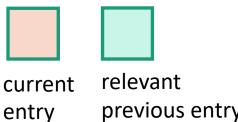
• Zero-One-Knapsack(W, n, w, v):

- K[x,0] = 0 for all x = 0,...,W
- K[0,i] = 0 for all i = 0,...,n
- for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:

x=1 $x=2$ $x=3$	$K[x,j] = max\{ K[x,j], K[x-w_j, j-1] + v_j \}$
-----------------	---



x=0



Example











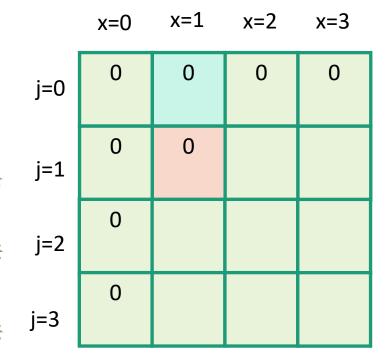




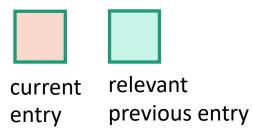


previous entry

4



- 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 $w_i \leq x$:
 - K[x,j] = max{ K[x,j],
 K[x w_j, j-1] + v_j }
 - return K[W,n]













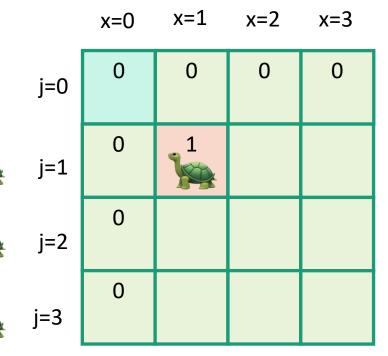


e: 1

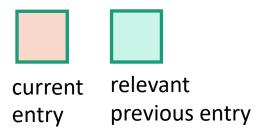
4

6

Capacity: 3



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















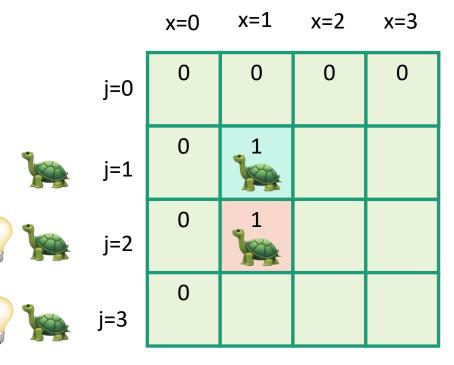




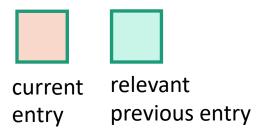
4

6

Capacity: 3



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















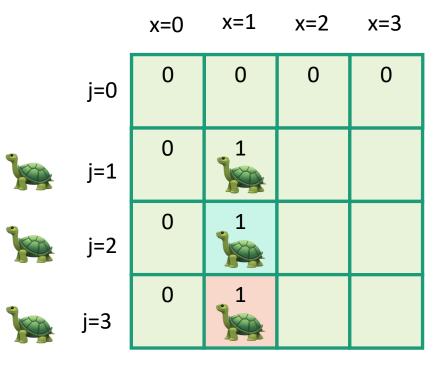
4



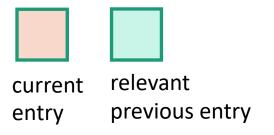




Capacity: 3



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













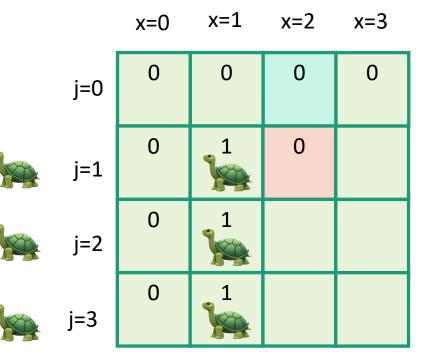
4



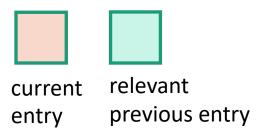




6



- 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 $w_i \le x$:
 - K[x,j] = max{ K[x,j],
 K[x w_j, j-1] + v_j }
 - return K[W,n]



Item:

Weight: Value:







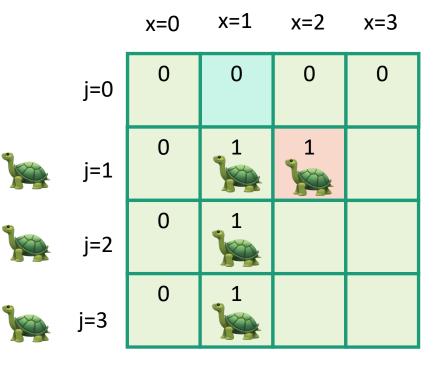
4



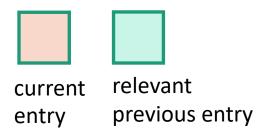


6





- 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 $w_i \le x$:
 - $K[x,j] = max\{ K[x,j],$ $K[x - w_j, j-1] + v_j$
 - return K[W,n]















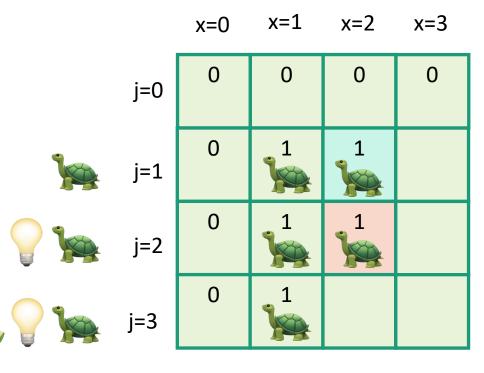




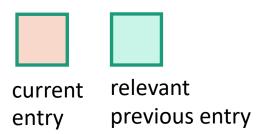
4

6

3



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











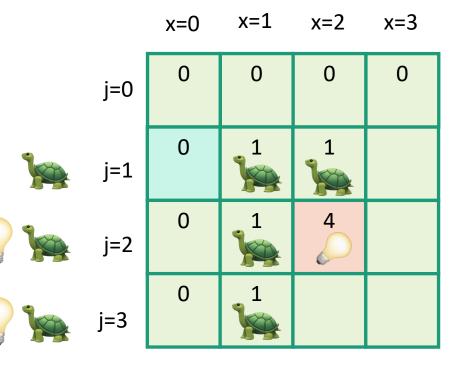




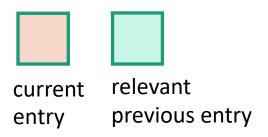


1 4

6



- 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 $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_j, j-1] + v_j$
 - return K[W,n]











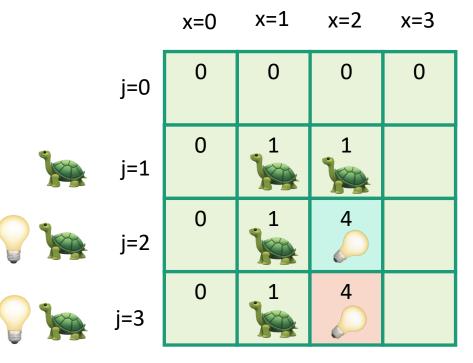


4

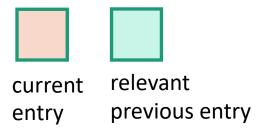








- Zero-One-Knapsack(W, n, w, v):
 - K[x,0] = 0 for all x = 0,...,W
 - K[0,i] = 0 for all i = 0,...,n
 - for x = 1,...,W:
 - **for** j = 1,...,n:
 - K[x,j] = K[x, j-1]
 - if $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_j, j-1] + v_j$
 - return K[W,n]











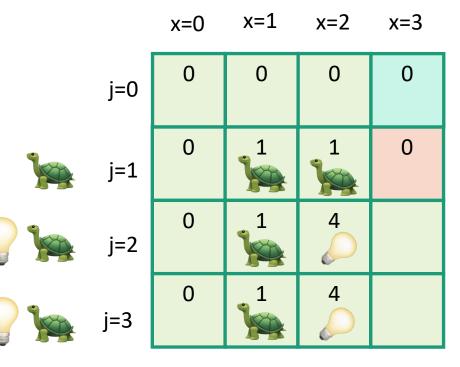




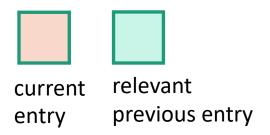


4

6



- 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 $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_j, j-1] + v_j$
 - return K[W,n]











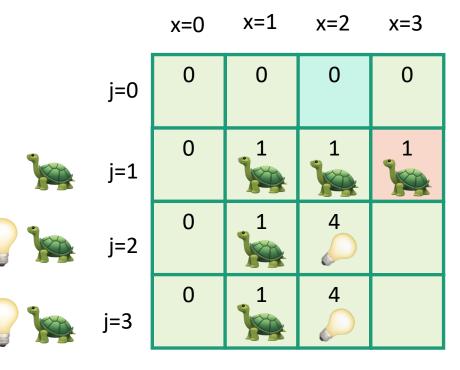
4



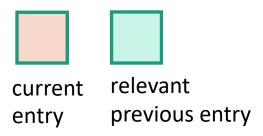
6







- 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 $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_j, j-1] + v_j$
 - return K[W,n]













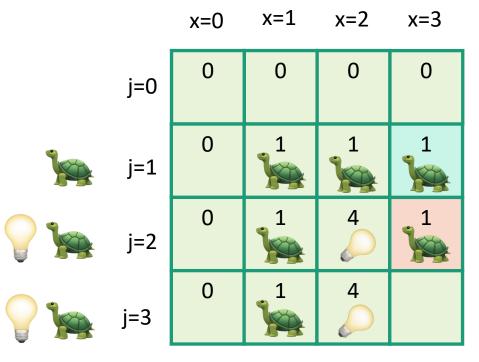
4



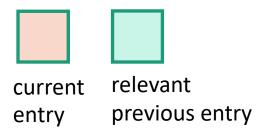


6





- 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 $w_i \leq x$:
 - K[x,j] = max{ K[x,j],
 K[x w_j, j-1] + v_j }
 - return K[W,n]













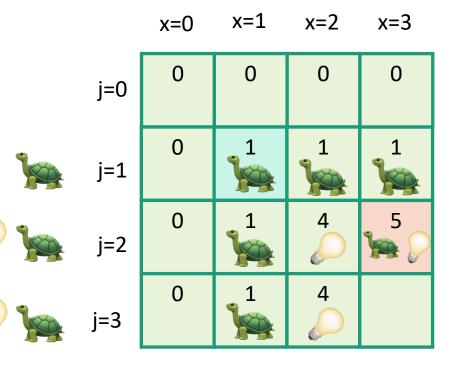




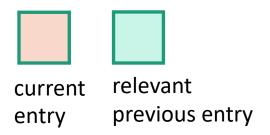


: 1 4

6



- 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 $w_i \le x$:
 - $K[x,j] = max\{K[x,j],$ $K[x - w_j, j-1] + v_j$
 - return K[W,n]





Weight: Value:







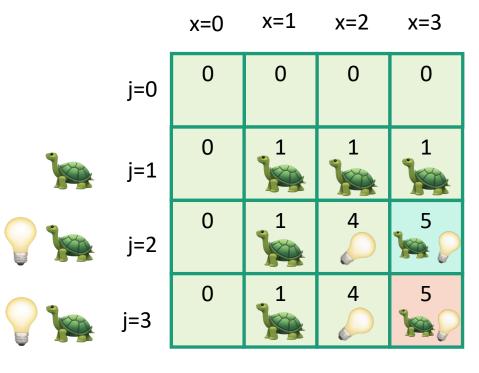
4



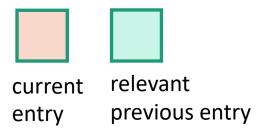


6





- 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 $w_i \le x$:
 - $K[x,j] = max\{ K[x,j],$ $K[x - w_j, j-1] + v_j$
 - return K[W,n]













4







6

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 $w_j \le x$:

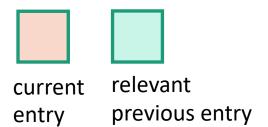
•	$K[x,j] = max\{ K[x,j],$	$K[x - w_{j}, j-1] + v_{j}$
---	--------------------------	-----------------------------

x=0 x=1 x=2 x=3 • return K[W,n] 0 0 0 0

j=1 0 1 1 1 1

j=2 0 1 4 5 0 0 1 4 6

j=3 0 1 4 6



Example

Item:



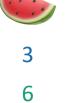




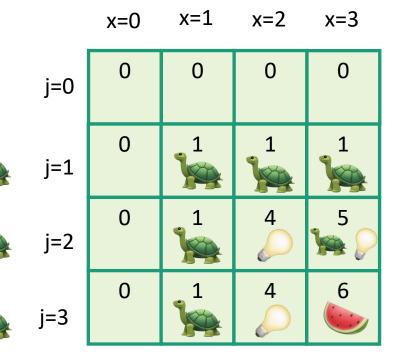


4



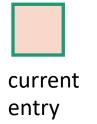


Capacity: 3



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

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





relevant previous entry



Weight: Value:







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.

 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:





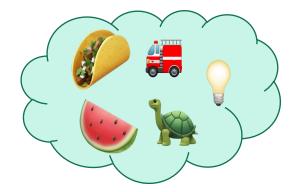


This one made sense for unbounded knapsack because it doesn't have any memory of what items have been used.

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!

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.