

Dynamic Programming II

Summer 2017 • Lecture 08/03

A Few Notes

Homework 5

Due tomorrow at 11:59 p.m. on Gradescope.

Homework 6

Released tomorrow.

Outline for Today

Dynamic Programming

- More DP algorithms

 - Longest Common Subsequence

 - Knapsack (0/1 and Unbounded)

 - Maximal Independent Set

Dynamic Programming

Elements of dynamic programming

Large problems break up into small problems.

e.g. shortest path with at most k edges.

Optimal substructure the optimal solution of a problem can be expressed in terms of optimal solutions of smaller sub-problems.

e.g. $d^{(k)}[b] = \min\{d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a,b)\}\}$

Overlapping sub-problems the sub-problems overlap a lot.

e.g. Lots of different entries of $d^{(k)}$ ask for $d^{(k-1)}[a]$.

This means we're save time by solving a sub-problem once and caching the answer.

Dynamic Programming

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems.
- (2) Define a recursive formulation.
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Dynamic Programming

Two approaches for DP: bottom-up and top-down.

Bottom-up iterates through problems by size and solves the small problems first (Bellman-Ford solves $d^{(0)}$ then $d^{(1)}$ then $d^{(2)}$, etc.)

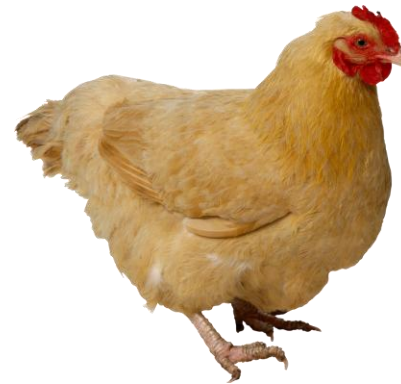
Top-down recurses to solve smaller problems, which recurse to solve even smaller problems.

How is this different than divide and conquer? **Memoization**, which keeps track of the small problems you've already solved to prevent resolving the same problem more than once.

Longest Common Subsequence

LCS

How similar are these two species?



DNA: ...CAGGACACATTA...

DNA: ...GATCAGAGATCA...

Similar, but definitely not the same species.

If only Wallace and Gromit knew about the LCS algorithm!

LCS

A **subsequence** is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.

e.g. **oae** is a subsequence of **soared**; so are **sore**, **sad**, and **srd**.

LCS

A **subsequence** is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.

e.g. **oae** is a subsequence of **soared**; so are **sore**, **sad**, and **srd**.

A common subsequence is a sequence that's a subsequence of two sequences.

e.g. **oae** is a common sequence of **soared** and **soaped**.

LCS

A **subsequence** is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.

e.g. **oae** is a subsequence of **soared**; so are **sore**, **sad**, and **srd**.

A common subsequence is a sequence that's a subsequence of two sequences.

e.g. **oae** is a common sequence of **soared** and **soaped**.

A longest common subsequence is the ... longest common subsequence.

e.g. **soaed** is the longest common subsequence of **soared** and **soaped**.

LCS

It's helpful to find LCS in bioinformatics, the unix command `diff`, merging in version control, etc.

LCS

Task Find the LCS of two strings.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems.
- (2) Define a recursive formulation.
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

LCS

Task Find the LCS of two strings.

(1) Identify optimal substructure with overlapping subproblems.

It seems helpful to know the LCS of prefixes of two strings.

e.g. if we wanted to know the `lcs("penguin", "chicken")`, it seems helpful to know

LCS

Task Find the LCS of two strings.

(1) Identify optimal substructure with overlapping subproblems.

It seems helpful to know the LCS of prefixes of two strings.

e.g. if we wanted to know the `lcs("penguin", "chicken")`, it seems helpful to know

```
lcs("pengui", "chicke")  
lcs("pengui", "chicken")  
lcs("penguin", "chicke")
```

LCS

Task Find the LCS of two strings.

(1) Identify optimal substructure with overlapping subproblems.

It seems helpful to know the LCS of prefixes of two strings.

e.g. if we wanted to know the `lcs("penguin", "chicken")`, it seems helpful to know

```
lcs("pengui", "chicke")  
lcs("pengui", "chicken")  
lcs("penguin", "chicke")
```

These subproblems overlap a lot!

LCS

Task Find the LCS of two strings.

(1) Identify optimal substructure with overlapping subproblems.

Also, it seems simpler to solve for the length of the LCS, and reconstruct the LCS itself after that in (4).

LCS

Task Find the LCS of two strings.

(1) Identify optimal substructure with overlapping subproblems.

Also, it seems simpler to solve for the length of the LCS, and reconstruct the LCS itself after that in (4).

Let $T(i, j)$ be the length of the LCS between the prefix from 0 and i (inclusive) of one string and the prefix from 0 and j (inclusive) of the other string.

LCS

Task Find the LCS of two strings.

(1) Identify optimal substructure with overlapping subproblems.

Also, it seems simpler to solve for the length of the LCS, and reconstruct the LCS itself after that in (4).

Let $T(i, j)$ be the length of the LCS between the prefix from 0 and i (inclusive) of one string and the prefix from 0 and j (inclusive) of the other string.

e.g. $T(2, 6)$ for strings “penguin” and “chicken” is 2.

0	1	2	3	4	5	6
p	e	n	g	u	i	n
c	h	i	c	k	e	n

LCS

Task Find the LCS of two strings.


(1) Identify optimal substructure with overlapping subproblems.

Also, it seems simpler to solve for the length of the LCS, and reconstruct the LCS itself after that in (4).

Let $T(i, j)$ be the length of the LCS between the prefix from 0 and i (inclusive) of one string and the prefix from 0 and j (inclusive) of the other string.

e.g. $T(2, 6)$ for strings “penguin” and “chicken” is 2.

“T” stands for “Table”, but other than that, this name has no special meaning.



0	1	2	3	4	5	6
p	e	n	g	u	i	n
c	h	i	c	k	e	n

LCS

Task Find the LCS of two strings.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems. 🙌
- (2) Define a recursive formulation.
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

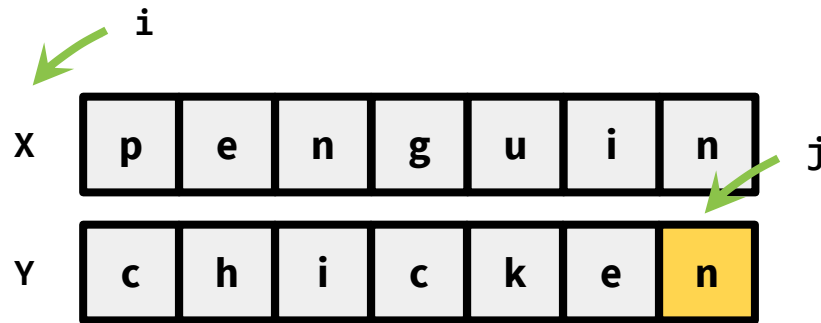
LCS

Task Find the LCS of two strings.

(2) Define a recursive formulation.

Consider two cases on the strings X and Y.

Base case (Case 0) i or j is -1




Then $T(i, j) = 0$

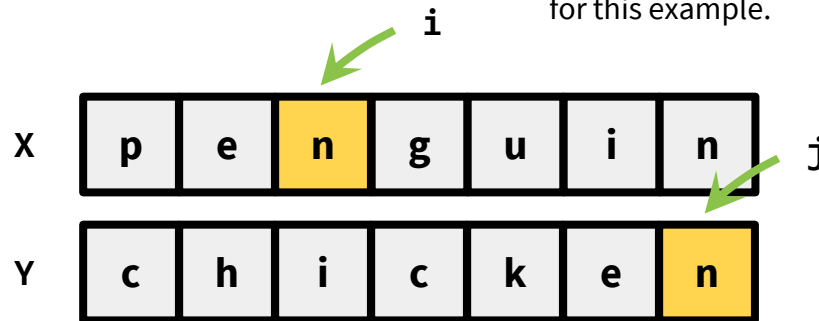
LCS

Task Find the LCS of two strings.

(2) Define a recursive formulation.

Consider two cases on the strings X and Y.

Case 1 $X[i] = Y[j]$  Suppose $i = 2$ and $j = 6$ for this example.




Then $T(i, j) = 1 + T(i-1, j-1)$

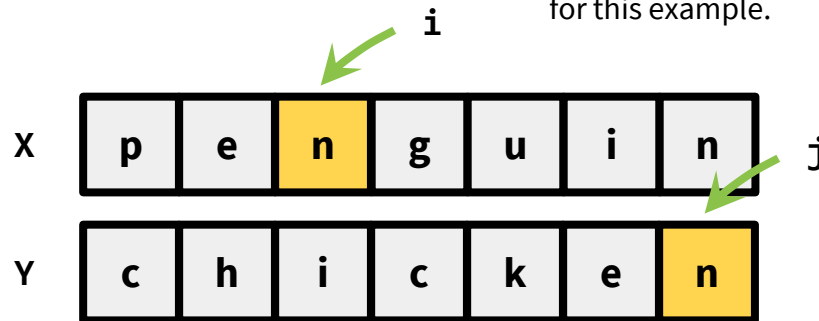
LCS

Task Find the LCS of two strings.

(2) Define a recursive formulation.

Consider two cases on the strings X and Y.

Case 1 $X[i] = Y[j]$  Suppose $i = 2$ and $j = 6$ for this example.



Then $T(i, j) = 1 + T(i-1, j-1)$


since $\text{LCS}(X[0:i], Y[0:j]) =$

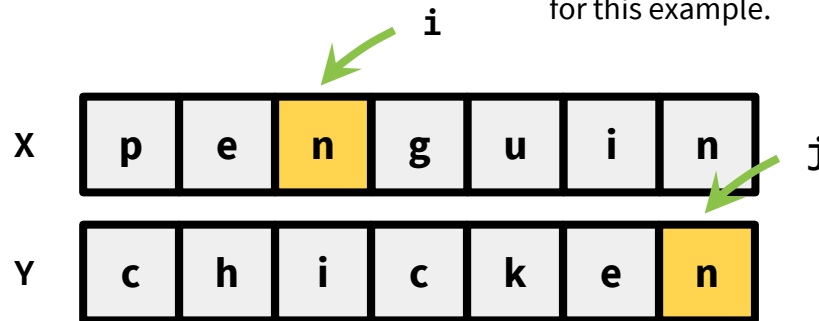
LCS

Task Find the LCS of two strings.

(2) Define a recursive formulation.

Consider two cases on the strings X and Y.

Case 1 $X[i] = Y[j]$  Suppose $i = 2$ and $j = 6$ for this example.



Then $T(i, j) = 1 + T(i-1, j-1)$


since $\text{LCS}(X[0:i], Y[0:j]) = \text{LCS}(X[0:i-1], Y[0:j-1])$ followed by n

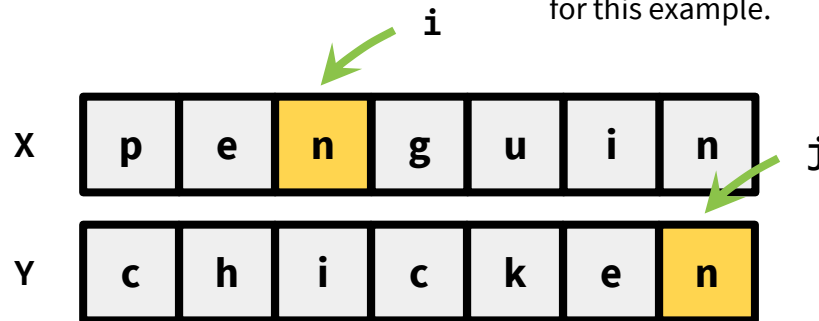
LCS

Task Find the LCS of two strings.

(2) Define a recursive formulation.


Consider two cases on the strings X and Y.

Case 1 $X[i] = Y[j]$  Suppose $i = 2$ and $j = 6$ for this example.



Then $T(i, j) = 1 + T(i-1, j-1)$

since $LCS(X[0:i], Y[0:j]) = LCS(X[0:i-1], Y[0:j-1])$ followed by n

 For this entire lecture, index ranges will be inclusive.

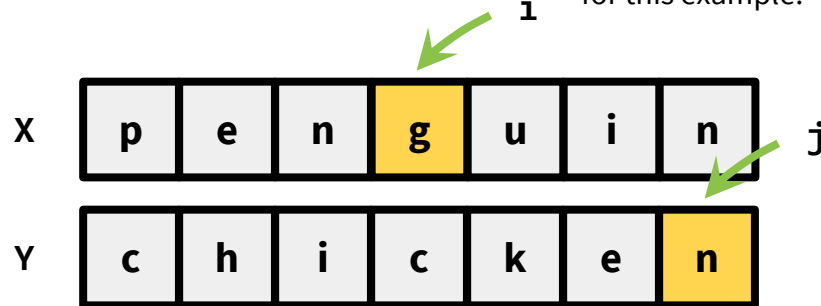
LCS

Task Find the LCS of two strings.

(2) Define a recursive formulation.

Consider two cases on the strings X and Y.

Case 2 $X[i] \neq Y[j]$  Suppose $i = 3$ and $j = 6$ for this example.



Then $T(i, j) = \max\{T(i-1, j), T(i, j-1)\}$

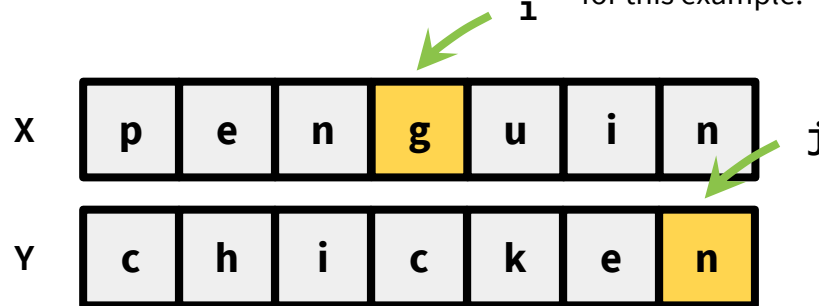
LCS

Task Find the LCS of two strings.

(2) Define a recursive formulation.

Consider two cases on the strings X and Y.

Case 2 $X[i] \neq Y[j]$  Suppose $i = 3$ and $j = 6$ for this example.



Then $T(i, j) = \max\{T(i-1, j), T(i, j-1)\}$

since either

$LCS(X[0:i], Y[0:j]) = LCS(X[0:i-1], Y[0:j])$ and **g** is not involved or

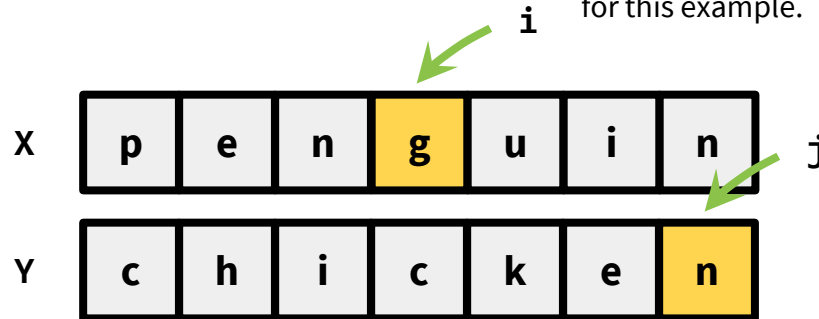
LCS

Task Find the LCS of two strings.

(2) Define a recursive formulation.

Consider two cases on the strings X and Y.

Case 2 $X[i] \neq Y[j]$  Suppose $i = 3$ and $j = 6$ for this example.



Then $T(i, j) = \max\{T(i-1, j), T(i, j-1)\}$

since either

$LCS(X[0:i], Y[0:j]) = LCS(X[0:i-1], Y[0:j])$ and **g** is not involved or

$LCS(X[0:i], Y[0:j]) = LCS(X[0:i], Y[0:j-1])$ and **n** is not involved

LCS

Task Find the LCS of two strings.

(2) Define a recursive formulation.

So, we get three cases in our recursive definition.

$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

Task Find the LCS of two strings.

Steps of dynamic programming

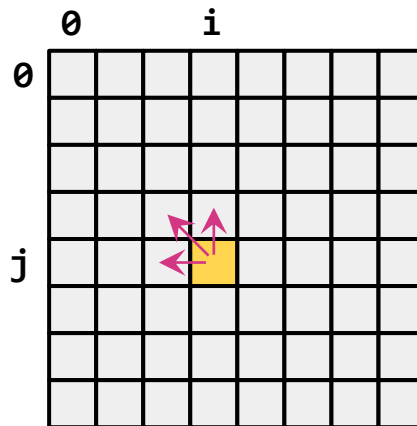
- (1) Identify optimal substructure with overlapping subproblems. 🙌
- (2) Define a recursive formulation. 🙌
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

LCS

Task Find the LCS of two strings.

(3) Use dynamic programming to solve the problem.

In what order do we need to fill our table according to the formulation from (2)?



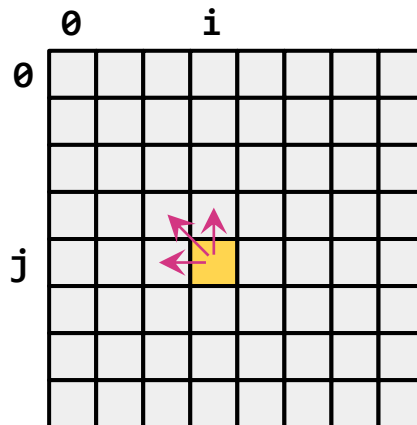
$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

Task Find the LCS of two strings.

(3) Use dynamic programming to solve the problem.


In what order do we need to fill our table according to the formulation from (2)?




← An element at position (i, j) in the table depends on elements at positions $(i-1, j)$, $(i, j-1)$, and $(i-1, j-1)$. So we want to fill out the values at these positions before (i, j) .

$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

```
algorithm lcs_helper(X, Y):  
    T = {}  
    for i = 0 to X.length-1:  Index ranges are inclusive, so loop will  
        T[i, -1] = 0 end at the start of iteration i = X.length  
    for j = 0 to Y.length-1:  
        T[-1, j] = 0  
    for i = 0 to X.length-1:  
        for j = 0 to Y.length-1:  
            if X[i] = Y[j]:  
                T[i, j] = 1 + T[i-1, j-1]  
            else:  
                T[i, j] = max{T[i, j-1], T[i-1, j]}  
    return T
```

Runtime: $O(|X| |Y|)$

 i.e. X.length

LCS

For example, consider `lcs_helper("ACGGA", "ACTG")`.

	A	C	T	G
A				
C				
G				
G				
A				

$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

For example, consider `lcs_helper("ACGGA", "ACTG")`.

	A	C	T	G
	0	0	0	0
A	0			
C	0			
G	0			
G	0			
A	0			

$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

For example, consider `lcs_helper("ACGGA", "ACTG")`.

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

$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

For example, consider `lcs_helper("ACGGA", "ACTG")`.

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

← The length of the LCS is 3!

$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

Task Find the LCS of two strings.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems. 🙌
- (2) Define a recursive formulation. 🙌
- (3) Use dynamic programming to solve the problem. 🙌
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

LCS

For example, consider `lcs_helper("ACGGA", "ACTG")`.

LCS

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

$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

For example, consider `lcs_helper("ACGGA", "ACTG")`.

LCS

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

That **3** must have come from this **3** since **A** and **G** don't match.

$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

For example, consider `lcs_helper("ACGGA", "ACTG")`.

LCS **G**

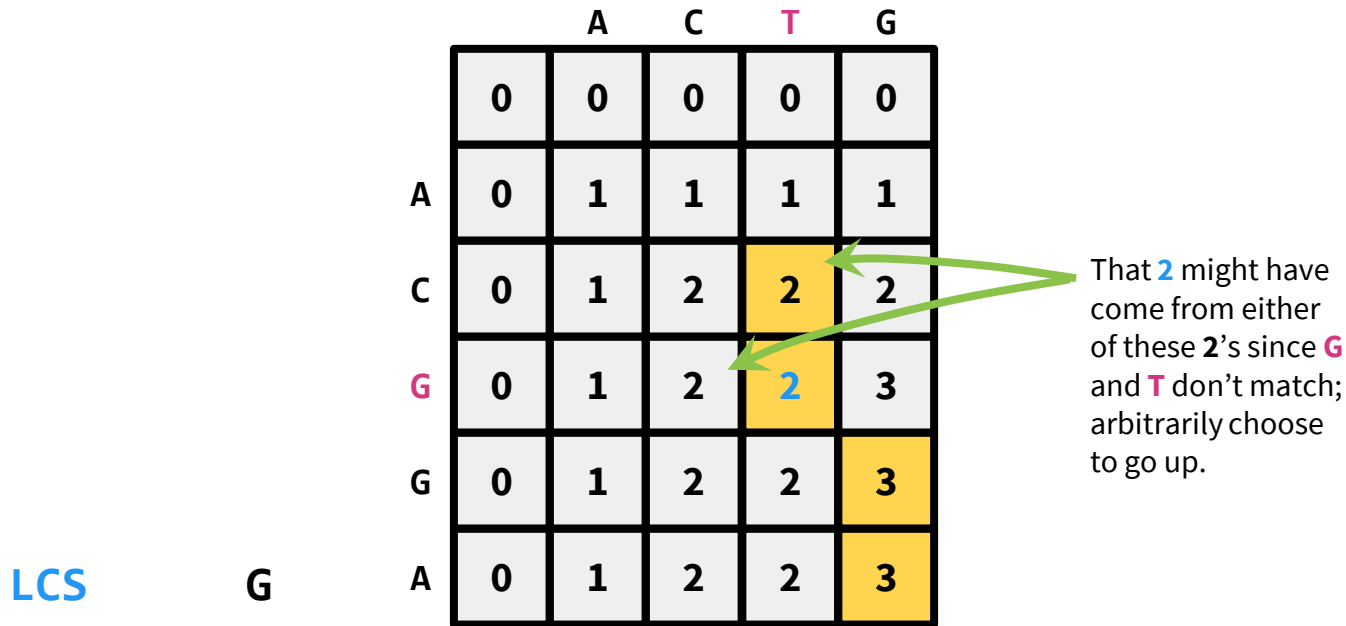
		A	C	T	G
		0	0	0	0
A		0	1	1	1
C		0	1	2	2
G		0	1	2	3
G		0	1	2	3
A		0	1	2	3

That **3** must have come from this **2** since **G**'s match.

$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

For example, consider `lcs_helper("ACGGA", "ACTG")`.



$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

For example, consider `lcs_helper("ACGGA", "ACTG")`.

LCS **G**

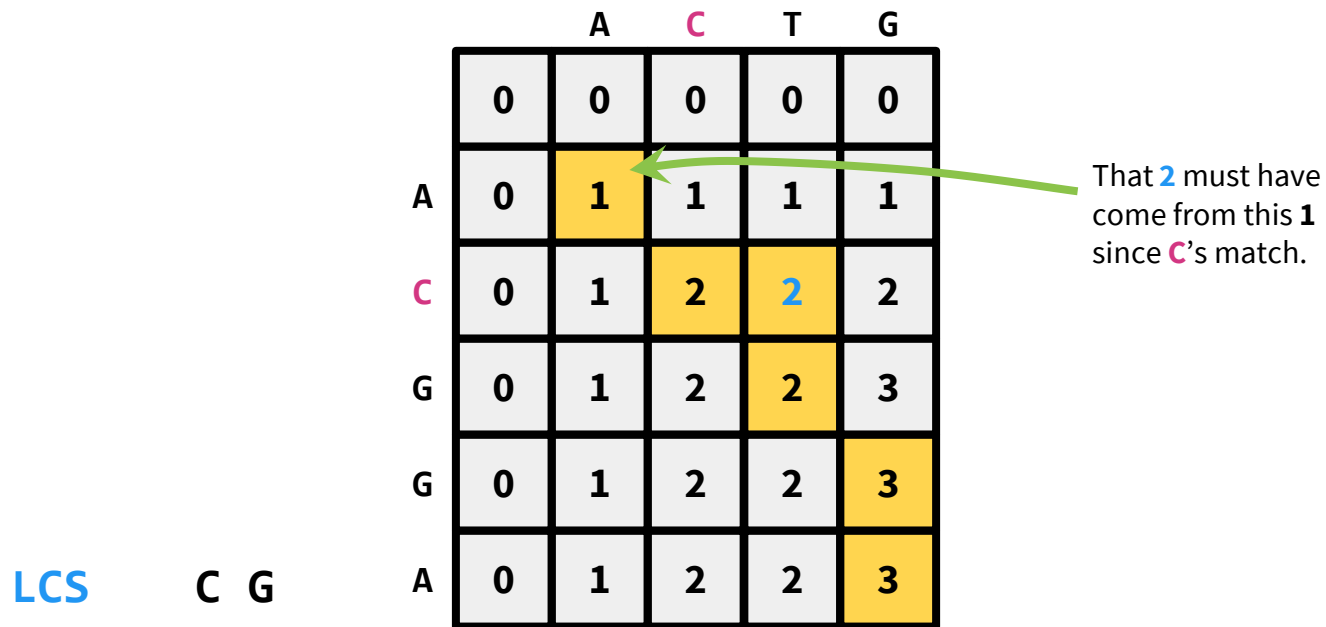
		A	C	T	G
		0	0	0	0
A		0	1	1	1
C		0	1	2	2
G		0	1	2	3
G		0	1	2	3
A		0	1	2	3

That 2 must have come from this 2 since C and T don't match.

$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

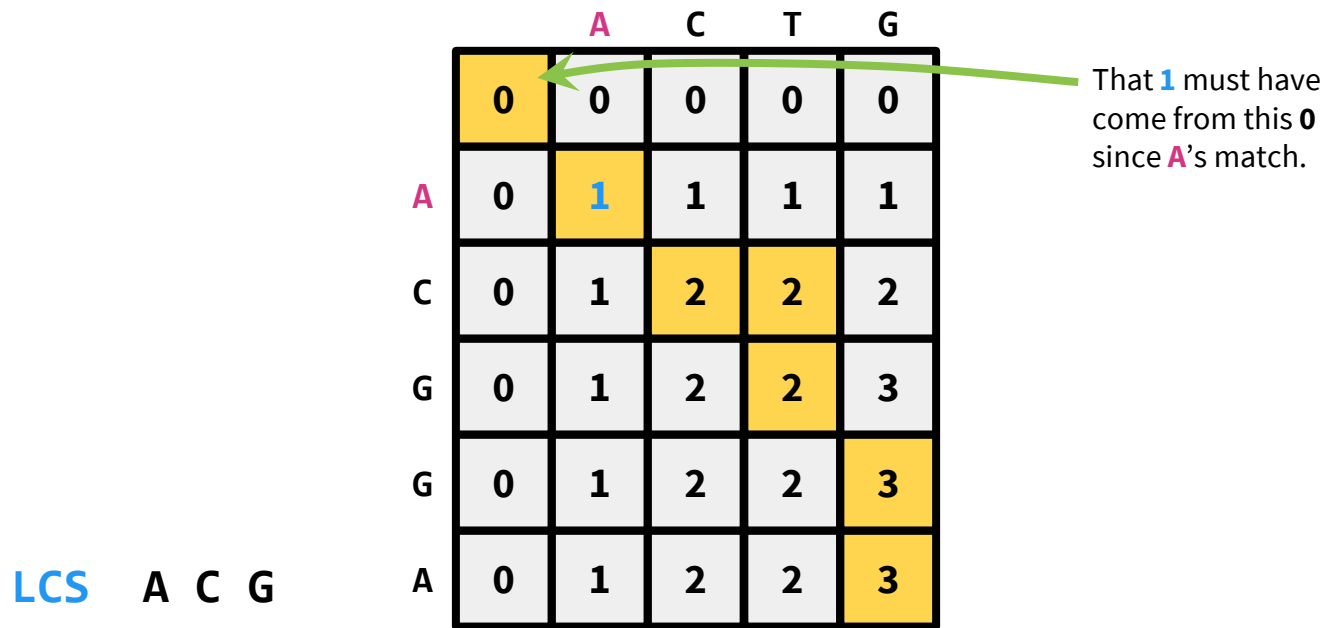
For example, consider `lcs_helper("ACGGA", "ACTG")`.



$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

For example, consider `lcs_helper("ACGGA", "ACTG")`.



$$T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is } -1 \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}$$

LCS

```
algorithm lcs(X, Y):  
  T = lcs_helper(X, Y)  
  lcs = backtrack(T)  
  return lcs
```

Must be only $O(|X|+|Y|)$
since step up and left in a
 $|X|$ by $|Y|$ table.

Runtime: $O(|X| |Y|)$

It's possible to do better
than this by a log factor
(think about it!).

Knapsack

Knapsack

Which items should I cram inside my backpack?

We have n items with weights and values.

item:					
weight:	6	2	4	3	11
value:	20	8	14	13	35

Knapsack

Which items should I cram inside my backpack?

We have n items with weights and values.

item:					
weight:	6	2	4	3	11
value:	20	8	14	13	35

This is Koko
the koala.



Knapsack

Which items should I cram inside my backpack?

We have n items with weights and values.

item:					
weight:	6	2	4	3	11
value:	20	8	14	13	35

This is Koko
the koala.



And we have a knapsack that can only carry so much weight.



capacity: 10

Knapsack

Which items should I cram inside my backpack?

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 that can only carry so much weight.

This is called a “knapsack”;
thank goodness since
“Knapsack algorithm”
sounds cooler than
“Backpack algorithm.”



capacity: 10

Knapsack

Unbounded Knapsack

Suppose I have infinite copies of all items.

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



Total weight: 10

Total value: 42

weight

6

2

4

3

11

value

20

8

14

13

35



capacity: 10

0/1 Knapsack

Suppose I only have one copy of each item.

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



Total weight: 9

Total value: 35

Unbounded Knapsack

Some notation

item:



...



weight:

w_1

w_2

w_3

w_n

value:

v_1

v_2

v_3

v_n



capacity: W

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems.
- (2) Define a recursive formulation.
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

(1) Identify optimal substructure with overlapping subproblems.

The problem statement restricts us from reducing the number of items.

By process of elimination, we reason that we must solve the problem for smaller knapsacks.



First solve the
problem for small
knapsacks

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

(1) Identify optimal substructure with overlapping subproblems.

The problem statement restricts us from reducing the number of items.

By process of elimination, we reason that we must solve the problem for smaller knapsacks.



First solve the
problem for small
knapsacks



Then larger
knapsacks

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

(1) Identify optimal substructure with overlapping subproblems.

The problem statement restricts us from reducing the number of items.

By process of elimination, we reason that we must solve the problem for smaller knapsacks.



First solve the
problem for small
knapsacks



Then larger
knapsacks



Then larger
knapsacks

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

(1) Identify optimal substructure with overlapping subproblems.

If this is an optimal solution for capacity x



Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

(1) Identify optimal substructure with overlapping subproblems.

If this is an optimal solution for capacity x



Then this must be an optimal solution for capacity $x - w_i$ for item $i =$ 



Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

(1) Identify optimal substructure with overlapping subproblems.

If this is an optimal solution for capacity x



Then this must be an optimal solution for capacity $x - w_i$ for item $i =$ 



Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems. 🙌
- (2) Define a recursive formulation.
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

(2) Define a recursive formulation.

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

$$K[x] = \max_i \{ \text{[Pokeball icon]} + \text{[Donut icon]} \}$$

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

The optimal way to fill the smaller knapsack

The value of item i .

$$K[x] = \begin{cases} 0 & \text{if there are no } i \text{ where } w_i \leq x \\ \max_i \{ K[x - w_i] + v_i \} & \text{otherwise} \end{cases}$$

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

Steps of dynamic programming

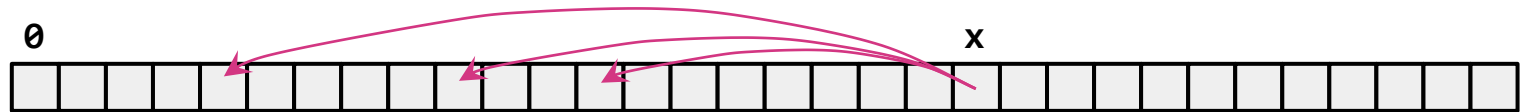
- (1) Identify optimal substructure with overlapping subproblems. 🤞
- (2) Define a recursive formulation. 🤞
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

(3) Use dynamic programming to solve the problem.

In what order do we need to fill our table according to the formulation from (2)?



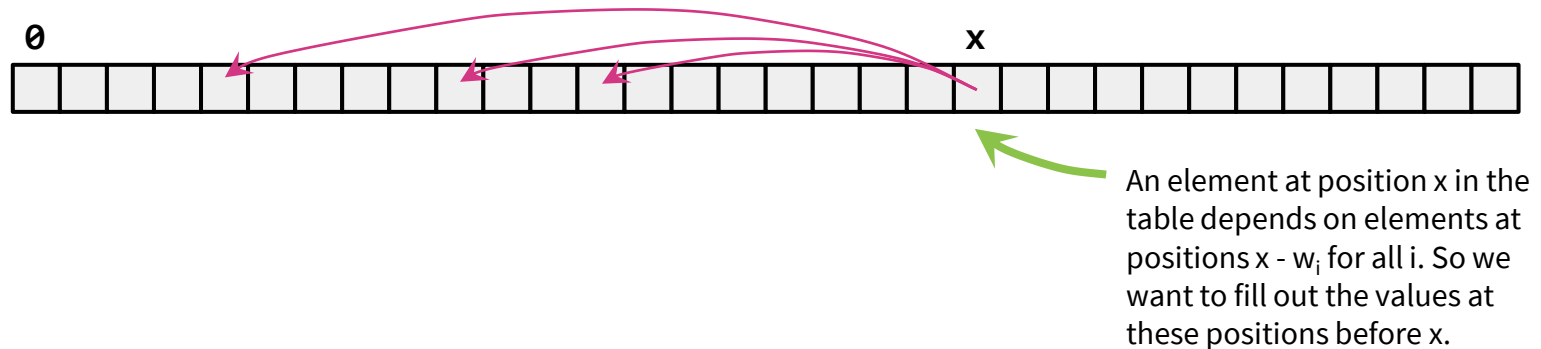
$$K[x] = \begin{cases} \emptyset & \text{if there are no } i \text{ where } w_i \leq x \\ \max_i \{K[x - w_i] + v_i\} & \text{otherwise} \end{cases}$$

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

(3) Use dynamic programming to solve the problem.

In what order do we need to fill our table according to the formulation from (2)?



$$K[x] = \begin{cases} \emptyset & \text{if there are no } i \text{ where } w_i \leq x \\ \max_i \{K[x - w_i] + v_i\} & \text{otherwise} \end{cases}$$

Unbounded Knapsack

```
algorithm unbounded_knapsack(capacity, weights, values):  
    W = capacity  
    n = weights.length  
    K[0] = 0  
    for x = 1 to W:  
        K[x] = 0  
        for i = 0 to n-1:  
             $w_i = \text{weights}[i]$ ,  $v_i = \text{values}[i]$   
            if  $w_i \leq x$ :  
                 $K[x] = \max\{K[x], K[x-w_i] + v_i\}$   
    return K[W]
```

Runtime: $O(nW)$

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems. 🙌
- (2) Define a recursive formulation. 🙌
- (3) Use dynamic programming to solve the problem. 🙌
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Unbounded Knapsack

```
algorithm unbounded_knapsack(capacity, weights, values):  
    W = capacity  
    n = weights.length  
    K[0] = 0, items[0] = {}  
    for x = 1 to W:  
        K[x] = 0  
        for i = 0 to n-1:  
            wi = weights[i], vi = values[i]  
            if wi ≤ x:  
                K[x] = max{K[x], K[x-wi] + vi}  
                if K[x] updated:  
                    items[x] = items[x-wi] ∪ {i}  
    return items[W]
```

Runtime: $O(nW)$

Unbounded Knapsack

	0	1	2	3	4
K	0				
items					



weight

1

2

3

value




1

4

6

capacity: 4

Unbounded Knapsack

	0	1	2	3	4
K	0	1	2		
items			 		



weight

1

2

3

value



1

4

6





capacity: 4

Unbounded Knapsack

	0	1	2	3	4
K	0	1	4		
items					

				
weight	1	2	3	
value	1	4	6	
capacity: 4				

Unbounded Knapsack

	0	1	2	3	4
K	0	1	4	5	
items				 	



weight

1

2

3

value




1

4

6





capacity: 4

Unbounded Knapsack

	0	1	2	3	4
K	0	1	4	6	
items					





				
weight	1	2	3	
value	1	4	6	
capacity: 4				

Unbounded Knapsack

	0	1	2	3	4
K	0	1	4	6	7
items					

				
weight	1	2	3	
value	1	4	6	
				capacity: 4

Unbounded Knapsack

	0	1	2	3	4
K	0	1	4	6	8
items					

				
weight	1	2	3	
value	1	4	6	
				capacity: 4

Knapsack

Unbounded Knapsack 🤞

Suppose I have infinite copies of all items.

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



Total weight: 10

Total value: 42

weight

value



6

20



2

8



4

14



3

13



11

35



capacity: 10

0/1 Knapsack

Suppose I only have one copy of each item.

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



Total weight: 9

Total value: 35

0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems.
- (2) Define a recursive formulation.
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

(1) Identify optimal substructure with overlapping subproblems.

Can we use the same optimal substructure as unbounded knapsack?



First solve the
problem for small
knapsacks



Then larger
knapsacks



Then larger
knapsacks

0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

(1) Identify optimal substructure with overlapping subproblems.

Can we use the same optimal substructure as unbounded knapsack?

No, the subproblem needs information about which items have been used.



0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

(1) Identify optimal substructure with overlapping subproblems.

Can we use the same optimal substructure as unbounded knapsack?

No, the subproblem needs information about which items have been used.



0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

(1) Identify optimal substructure with overlapping subproblems.

We reason that we must solve the problem for a smaller number of items and for smaller knapsacks.

First solve the
problem for few
items



First solve the
problem for small
knapsacks



Then larger
knapsacks



Then larger
knapsacks



0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

(1) Identify optimal substructure with overlapping subproblems.

We reason that we must solve the problem for a smaller number of items and for smaller knapsacks.



First solve the
problem for small
knapsacks



Then larger
knapsacks



Then larger
knapsacks

First solve the
problem for few
items



Then more items



0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

(1) Identify optimal substructure with overlapping subproblems.

We reason that we must solve the problem for a smaller number of items and for smaller knapsacks.



First solve the
problem for small
knapsacks



Then larger
knapsacks



Then larger
knapsacks

First solve the
problem for few
items



Then more items



Then more items



0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

(1) Identify optimal substructure with overlapping subproblems.

We reason that we must solve the problem for a smaller number of items and for smaller knapsacks.



First solve the
problem for small
knapsacks



Then larger
knapsacks



Then larger
knapsacks

First solve the
problem for few
items



Then more items



Then more items



We need a two-dimensional table!

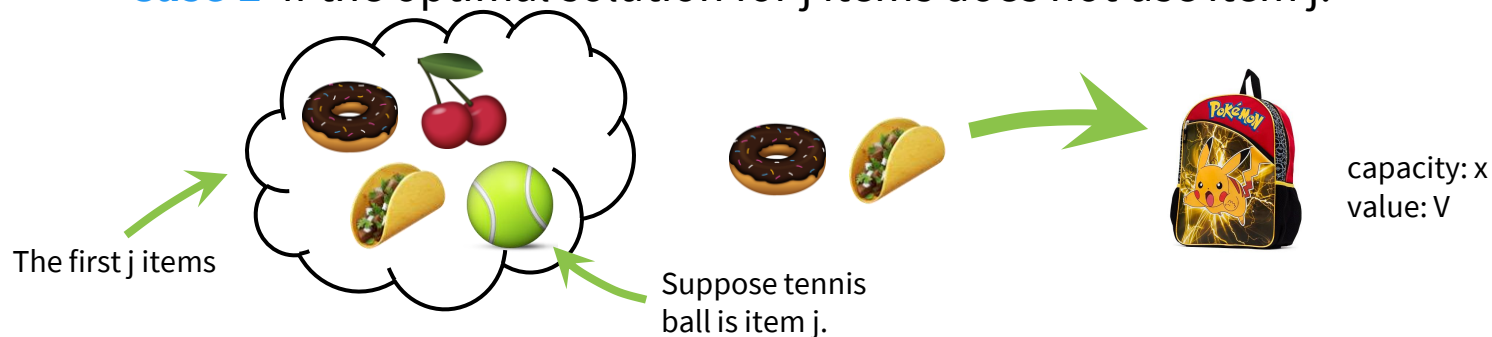
0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

(1) Identify optimal substructure with overlapping subproblems.

Handle items in a similar way to how we handled vertices in Floyd-Warshall; restrict the set of items to be used to a specific set 0 to $j-1$.

Case 1 If the optimal solution for j items does not use item j .



Then this is an optimal solution for $j - 1$ items.



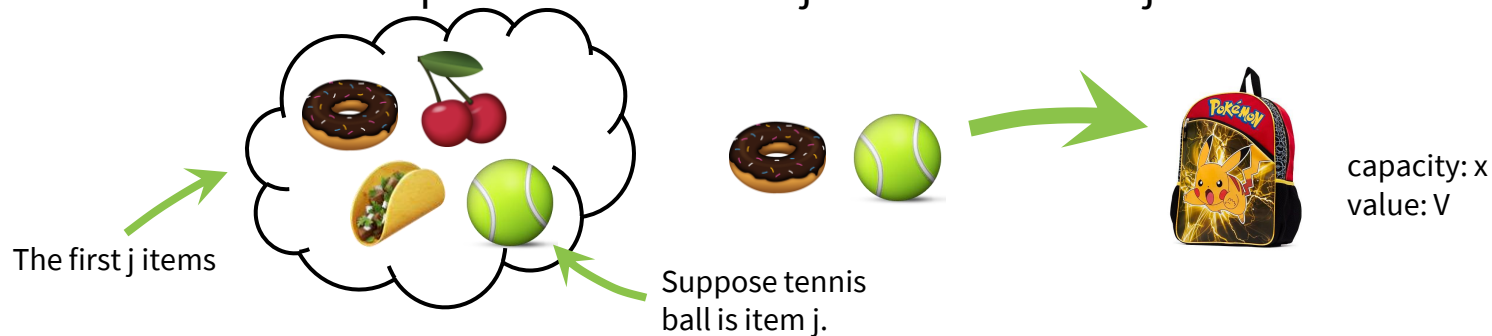
0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

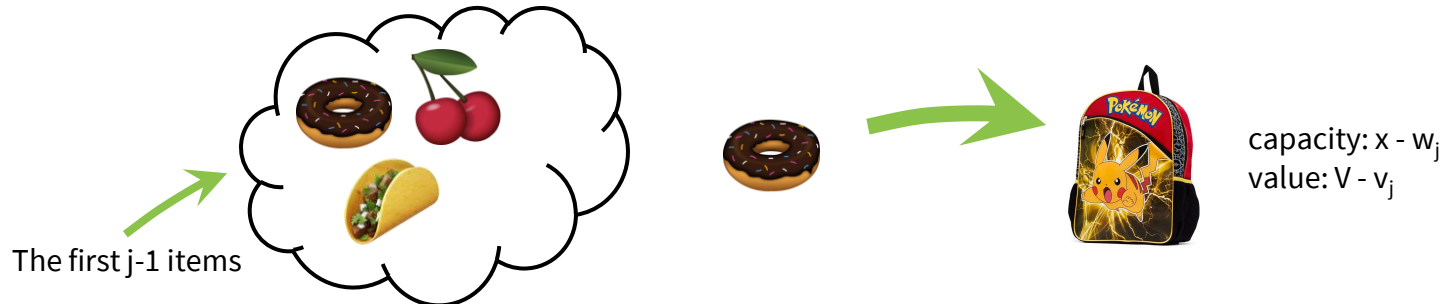
(1) Identify optimal substructure with overlapping subproblems.

Handle items in a similar way to how we handled vertices in Floyd-Warshall; restrict the set of items to be used to a specific set 0 to $j-1$.

Case 2 If the optimal solution for j items uses item j .



Then this is an optimal solution for $j - 1$ items.



0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems. 🙌
- (2) Define a recursive formulation.
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

0/1 Knapsack

Task Find the items to put in an unbounded knapsack.

(2) Define a recursive formulation.

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

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems. 🙌
- (2) Define a recursive formulation. 🙌
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

0/1 Knapsack

```
algorithm zero_one_knapsack(capacity, weights, values):  
    W = capacity  
    n = weights.length  
    K[x,0] = 0 for x = 0 to W  
    K[0,i] = 0 for i = 0 to n  
    for x = 1 to W:  
        for j = 1 to n:  
            K[x,j] = K[x,j-1]  
            wj = weights[j], vj = values[j]  
            if wj ≤ x:  
                K[x,j] = max{K[x,j], K[x-wj] + vj}  
    return K[W,n]
```

Runtime: $O(nW)$

0/1 Knapsack

Task Find the items to put in a 0/1 knapsack.


Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems. 🙌
- (2) Define a recursive formulation. 🙌
- (3) Use dynamic programming to solve the problem. 🙌
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

0/1 Knapsack

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

 current entry
  relevant previous entry


				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack


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



 current entry
  relevant previous entry

				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack



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

 current entry
  relevant previous entry

				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack




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



 current entry
  relevant previous entry

				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack




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

 current entry
  relevant previous entry


				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack

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

 current entry
  relevant previous entry

				
weight	1	2	3	
value	1	4	6	
	capacity: 3			

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack






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

 current entry
  relevant previous entry



				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack

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

 current entry
  relevant previous entry


				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack







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

 current entry
  relevant previous entry


				
weight	1	2	3	capacity: 3
value	1	4	6	

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack







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

 current entry
  relevant previous entry



				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack








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



 current entry
  relevant previous entry

				
weight	1	2	3	
value	1	4	6	
	capacity: 3			

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack









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

 current entry
  relevant previous entry



				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack









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

 current entry
  relevant previous entry


				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack










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

 current entry
  relevant previous entry

				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack










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

 current entry
  relevant previous entry



				
weight	1	2	3	
value	1	4	6	
				capacity: 3

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack










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

 current entry
  relevant previous entry


				
weight	1	2	3	
value	1	4	6	
				capacity: 3


$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

0/1 Knapsack

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

 current entry
  relevant previous entry

				
weight	1	2	3	capacity: 3
value	1	4	6	

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

$$K[x, j] = \begin{cases} 0 & \text{if } x \text{ or } j \text{ are } 0 \\ \max\{K[x, j-1], K[x-w_j, j-1] + v_j\} & \text{otherwise} \end{cases}$$

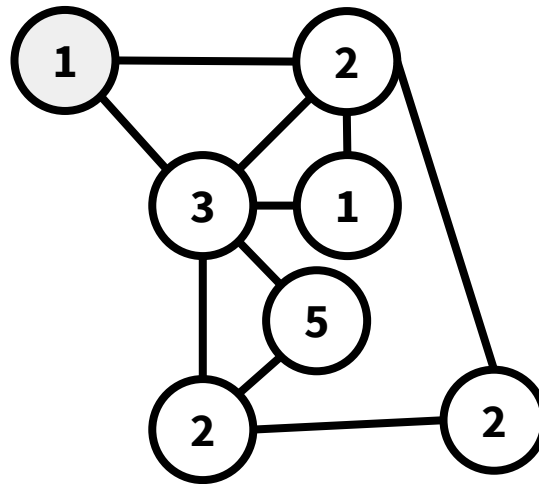
Independent Set

Maximal Independent Set

What is the maximal independent set in a graph?

An independent set describes a set of weighted vertices where no pair of vertices in the set shares an edge.

A maximal independent set has the largest weight.

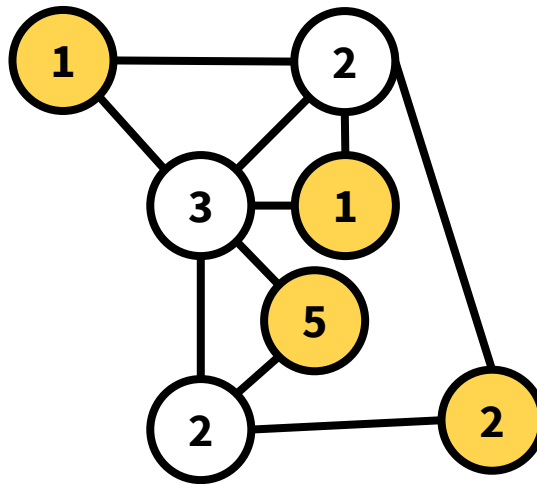


Maximal Independent Set

What is the maximal independent set in a graph?

An independent set describes a set of weighted vertices where no pair of vertices in the set shares an edge.

A maximal independent set has the largest weight.

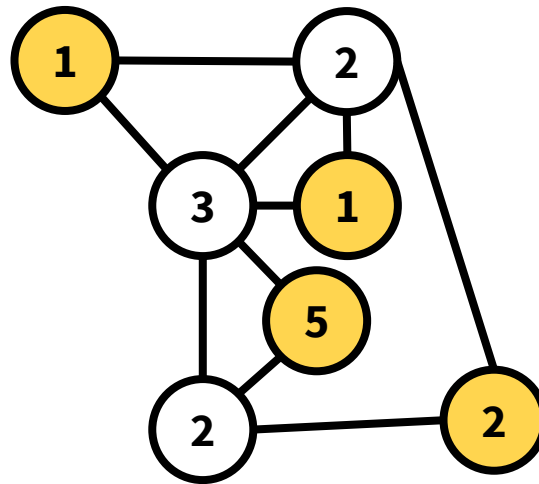


Maximal Independent Set

What is the maximal independent set in a graph?

An independent set describes a set of weighted vertices where no pair of vertices in the set shares an edge.

A maximal independent set has the largest weight.



This problem is NP-complete. ←

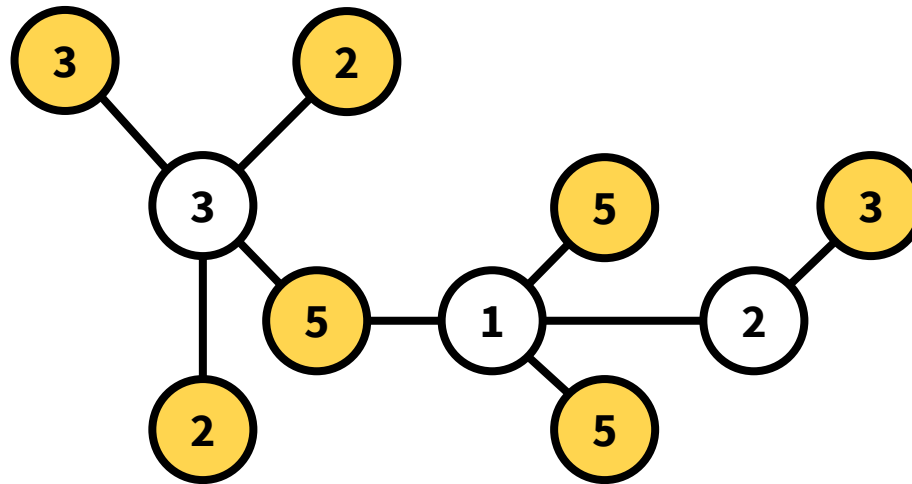
We'll learn what this means in a bit, but for now take it to mean we're unlikely to find an efficient algorithm.

Maximal Independent Set

What is the maximal independent set **in a tree**?

An independent set describes a set of weighted vertices where no pair of vertices in the set shares an edge.

A maximal independent set has the largest weight.



Maximal Independent Set

Task Find the maximal independent set in a tree.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems.
- (2) Define a recursive formulation.
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Maximal Independent Set

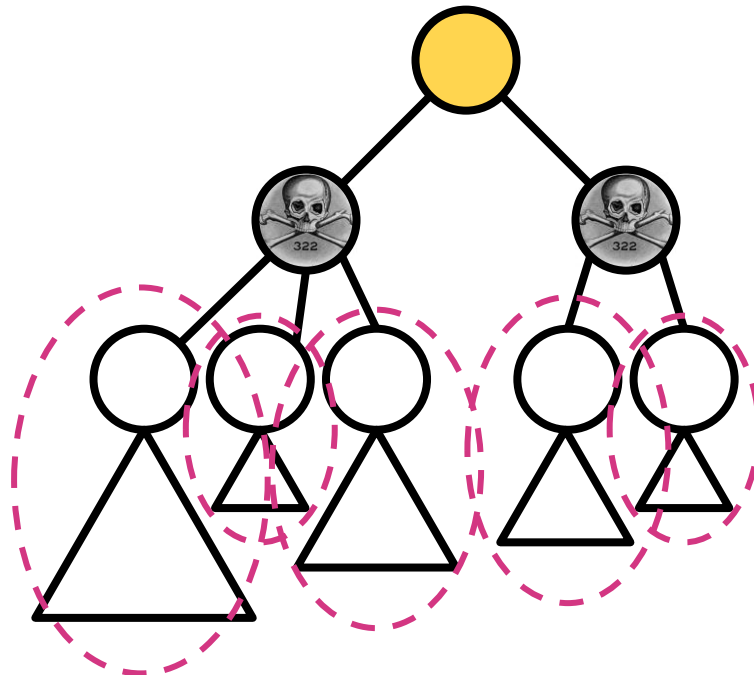
Task Find the maximal independent set in a tree.

(1) Identify optimal substructure with overlapping subproblems.

Subtree are a natural candidate.

Consider two cases:

Case 1 The root of this tree is in a maximal independent set.



Then use the optimal solution from these smaller subproblems since the children can't be in the independent set.

Maximal Independent Set

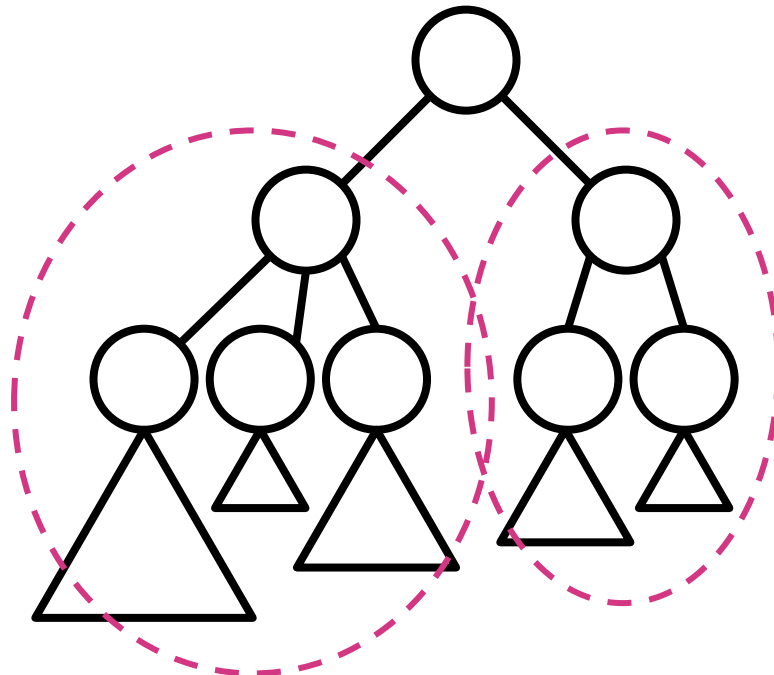
Task Find the maximal independent set in a tree.

(1) Identify optimal substructure with overlapping subproblems.

Subtree are a natural candidate.

Consider two cases:

Case 2 The root of this tree is not in a maximal independent set.



Then use the optimal solution from these smaller subproblems since the children can be in the independent set.

Maximal Independent Set

Task Find the maximal independent set in a tree.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems. 🙌
- (2) Define a recursive formulation.
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Maximal Independent Set

Task Find the maximal independent set in a tree.

(2) Define a recursive formulation.

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

$$A[u] = \max \left\{ \begin{array}{ll} w(u) + \sum_{v \in u.\text{grandchildren}} A[v] & \text{case 1} \\ \sum_{v \in u.\text{children}} A[v] & \text{case 2} \end{array} \right\}$$

Maximal Independent Set

Task Find the maximal independent set in a tree.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems. 🤔
- (2) Define a recursive formulation. 🤔
- (3) Use dynamic programming to solve the problem.
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Maximal Independent Set

```
cache = {}  
algorithm max_independent_set_helper(root):  
    if is_leaf(root):  
        return root.weight  
    else if root in cache:  
        return cache[root]  
    else:  
         $w = \max \begin{cases} w(u) + \sum_{v \in u.\text{grandchildren}} \text{max\_independent\_set\_helper}(v) \\ \sum_{v \in u.\text{children}} \text{max\_independent\_set\_helper}(v) \end{cases}$   
  
    cache[root] = w  
    return w
```

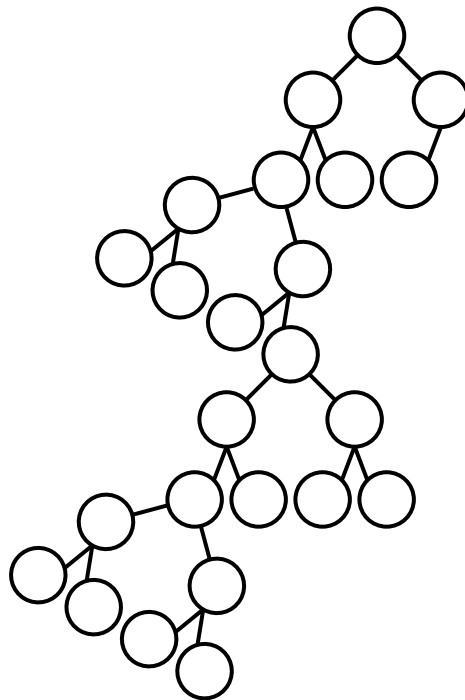
Runtime: $O(|V|)$

Maximal Independent Set

Task Find the maximal independent set in a tree.

(3) Use dynamic programming to solve the problem.

What's the point of **cache**?

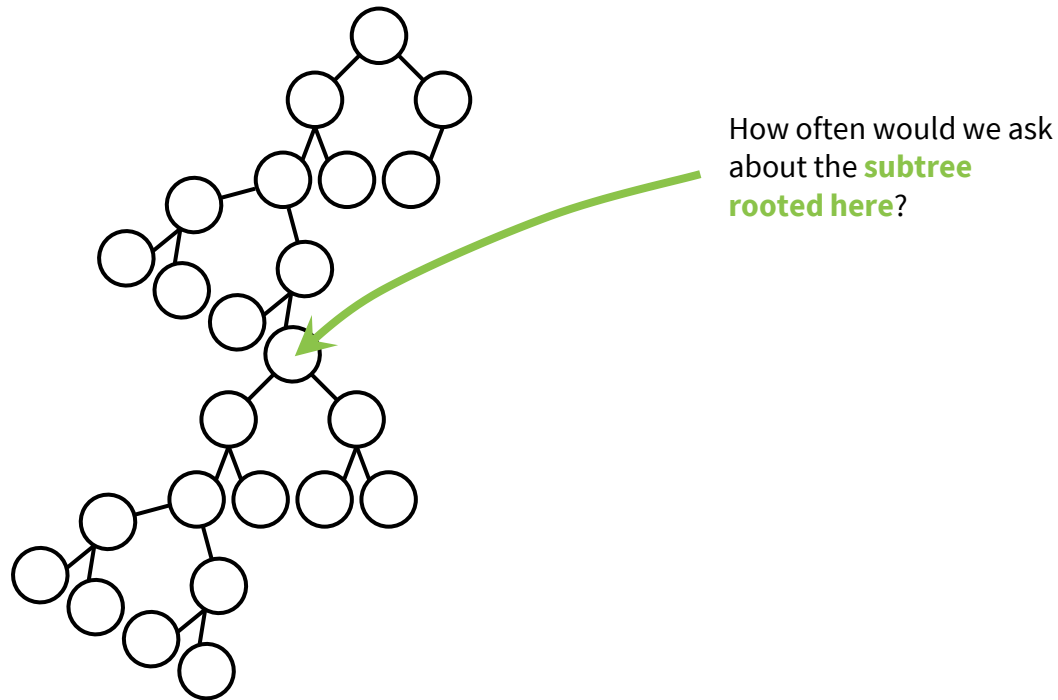


Maximal Independent Set

Task Find the maximal independent set in a tree.

(3) Use dynamic programming to solve the problem.

What's the point of **cache**?

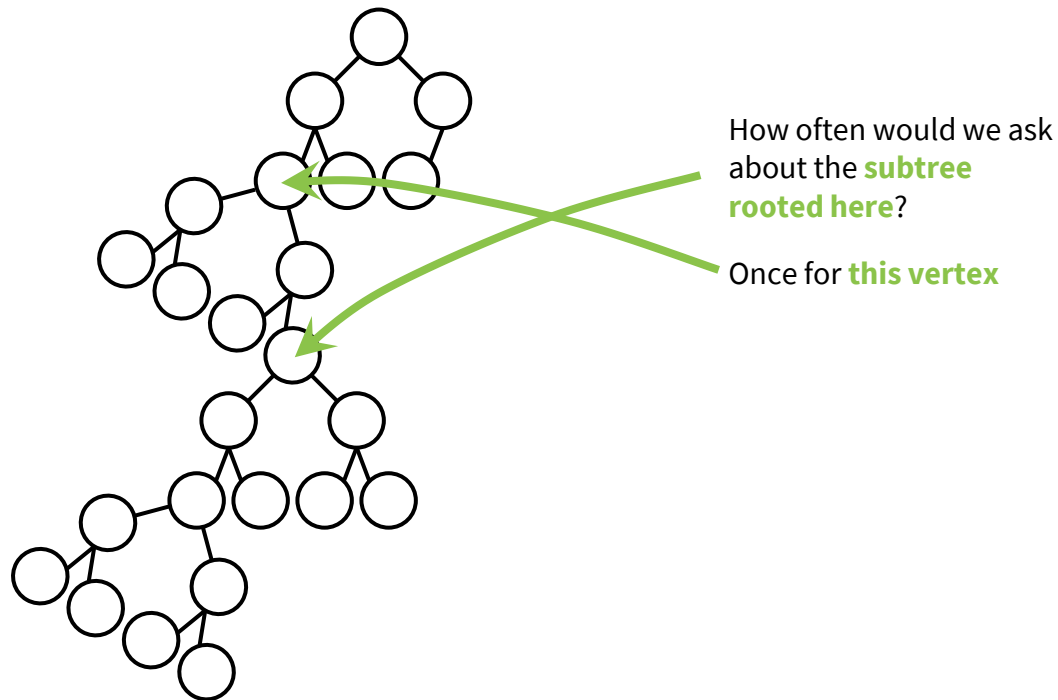


Maximal Independent Set

Task Find the maximal independent set in a tree.

(3) Use dynamic programming to solve the problem.

What's the point of **cache**?

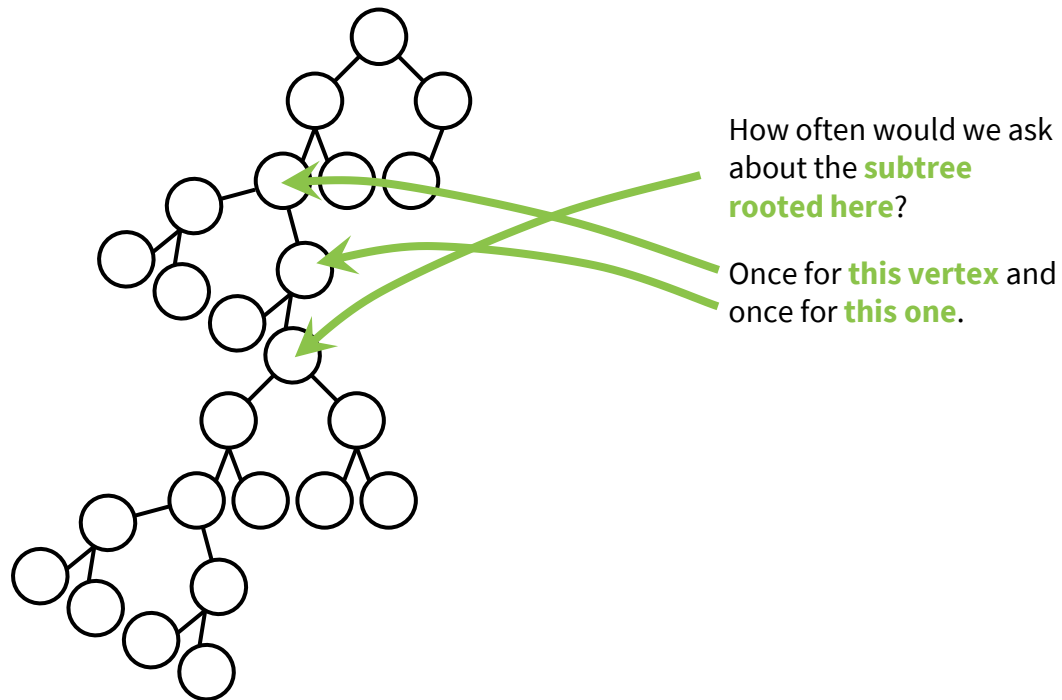


Maximal Independent Set

Task Find the maximal independent set in a tree.

(3) Use dynamic programming to solve the problem.

What's the point of **cache**?

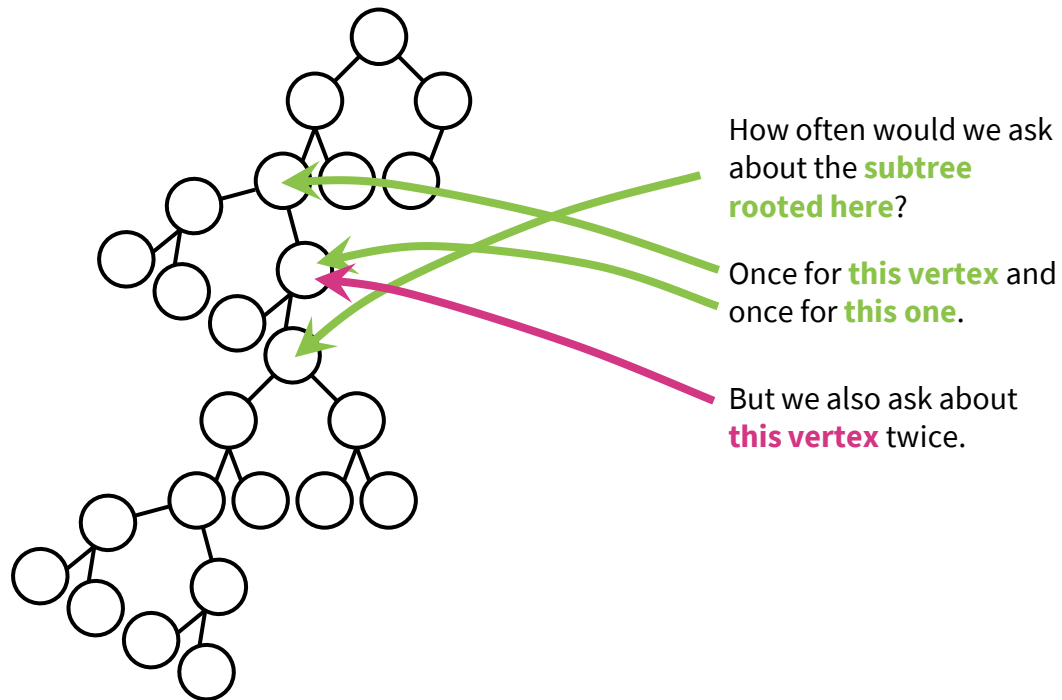


Maximal Independent Set

Task Find the maximal independent set in a tree.

(3) Use dynamic programming to solve the problem.

What's the point of **cache**?

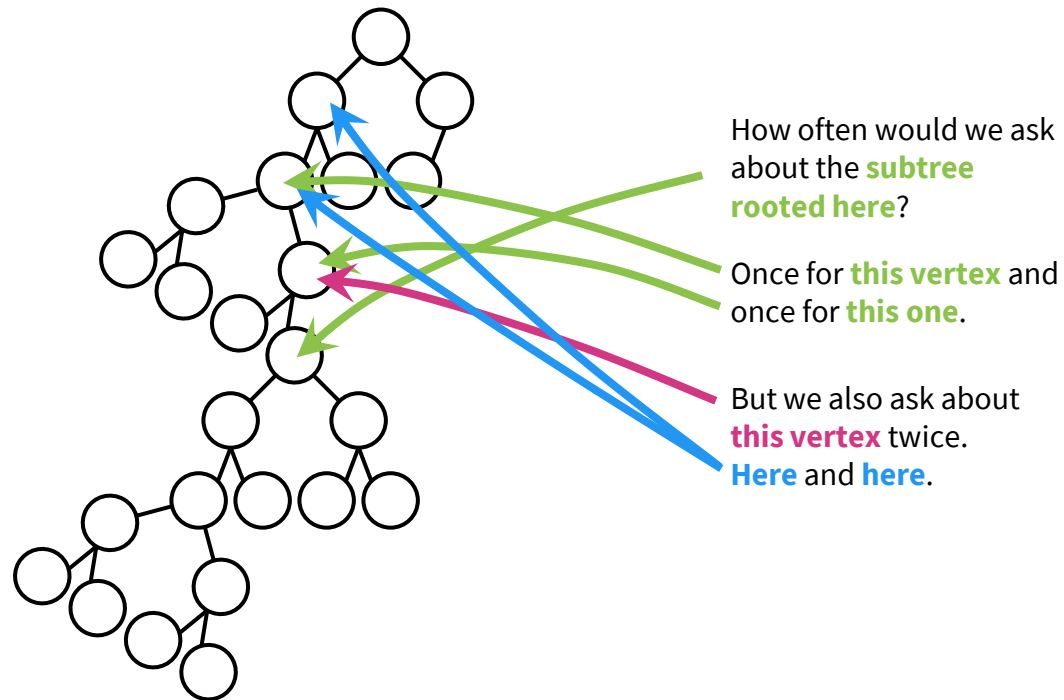


Maximal Independent Set

Task Find the maximal independent set in a tree.

(3) Use dynamic programming to solve the problem.

What's the point of **cache**?

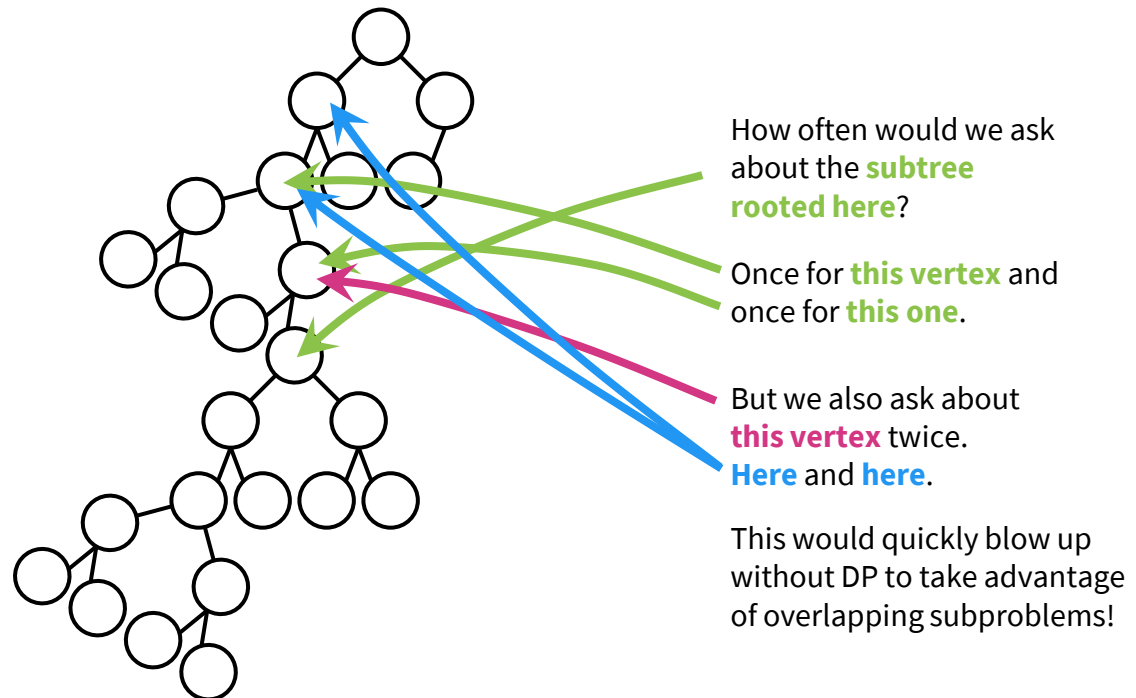


Maximal Independent Set

Task Find the maximal independent set in a tree.

(3) Use dynamic programming to solve the problem.

What's the point of **cache**?



Maximal Independent Set

Task Find the maximal independent set in a tree.

Steps of dynamic programming

- (1) Identify optimal substructure with overlapping subproblems. 🙌
- (2) Define a recursive formulation. 🙌
- (3) Use dynamic programming to solve the problem. 🙌
- (4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Conclusion

	Runtime	Variables	Notes
LCS	$O(X Y)$ worst-case	$ X $ and $ Y $ are the lengths of the strings being compared.	Finding the actual LCS is possible from first solving the length of the LCS.
Unbounded Knapsack	$O(nW)$ worst-case	n is the number of items and W is the knapsack capacity.	Ditto
0/1 Knapsack	$O(nW)$ worst-case	Ditto	Same as unbounded knapsack except can only use 1 of each item, so it requires a 2D table.
Maximal Independent Set	$O(V)$ worst-case	$ V $ is the number of vertices in the tree.	NP-complete for graphs, this alg. works for trees; top-down easier.