Dynamic Programming II

Outline for Today

Dynamic Programming

- More DP algorithms
  - Longest Common Subsequence
  - Knapsack (0/1 and Unbounded)
  - Maximal Independent Set (advanced topic)

Longest Common Subsequence

LCS

How similar are these two species?

DNA: ...CAGGACACATTA...

DNA: ...GATCAGAGATCA...

Similar, but definitely not the same species.

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. **eee** is a subsequence of **sequence**; so are **seen**, **sqnc**, and **quen**.

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. **eee** is a subsequence of **sequence**; so are **seen**, **sqnc**, and **quen**.

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

e.g. **que** is a common sequence of **sequence** and **queen**.

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. **eee** is a subsequence of **sequence**; so are **seen**, **sqnc**, and **quen**.

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

e.g. **que** is a common sequence of **sequence** and **queen**.

A **longest common subsequence** is the … longest common subsequence.

e.g. **quen** is the longest common subsequence of **sequence** and **queen**.

LCS

It's helpful to find LCS in bioinformatics, the unix command `diff`, merging in version control, plagiarism checking, 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("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.

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.

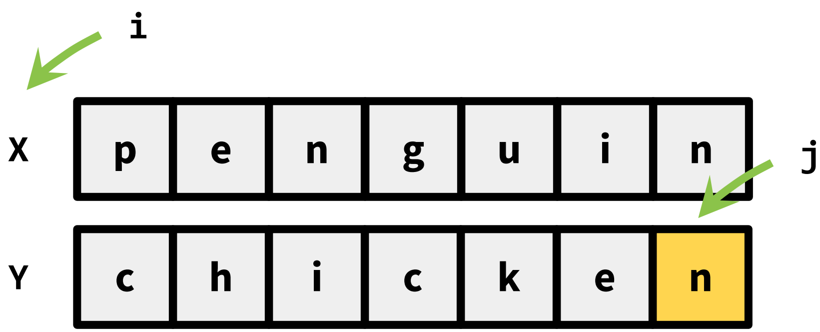
LCS

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



If i = -1 or j = -1, then `T(i, j) = 0`

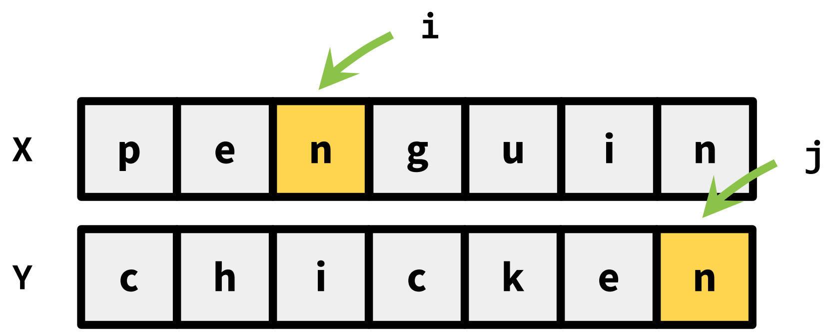
LCS

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)`

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

For this entire lecture, index ranges will be inclusive.

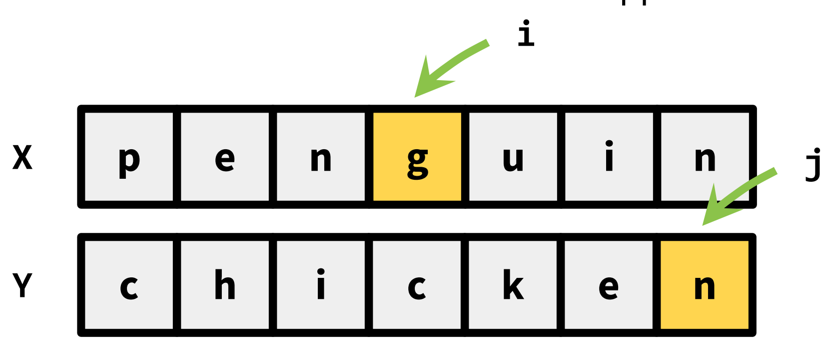
LCS

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] != 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 $\boxed{g}$ is not involved or

`LCS(X[0:i], Y[0:j]) = LCS(X[0:i], Y[0:j-1])` and $\boxed{n}$ 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.

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

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.

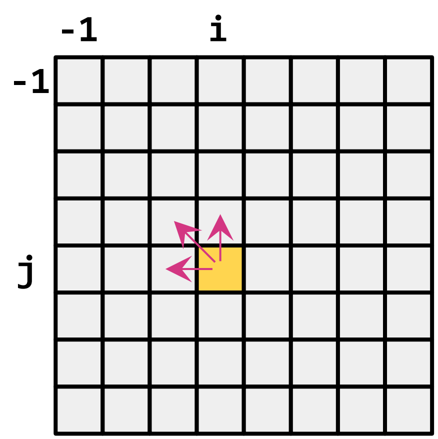
(4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

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)?



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).

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

LCS

```
algorithm lcs_helper(X, Y):
  T = {}
  for i = 0 to X.length-1:
    T[i, -1] = 0
  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
```

Index ranges are inclusive, so loop will end at the start of iteration i = X.length

Runtime: O(|X||Y|)

i.e. X.length

LCS

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

	-1	A	C	T	G
-1					
A					
C					
G					
G					
A					

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

LCS

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

	-1	A	C	T	G
-1	0	0	0	0	0
A	0				
C	0				
G	0				
G	0				
A	0				

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

LCS

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

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

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

LCS

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

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

The length of the LCS is 3!

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

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")`**.

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

LCS

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

LCS

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

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

LCS

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")`**.

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

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

LCS G

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

LCS

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

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

That **2** might have come from either of these **2**'s since **G** and **T** don't match; arbitrarily choose to go up.

LCS G

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

LCS

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

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

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

LCS G

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

LCS

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

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

That **2** must have come from this **1** since **C**'s match.

LCS **C G**

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

LCS

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

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

That **1** must have come from this **0** since **A**'s match.

LCS **A C G**

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

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

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. `T(i, j) = 1 + T(i-1, j-1) when X[i]=Y[j]`

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

e.g. Each element in the table is used for at least twice.

This means we're saving 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.

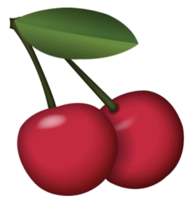
(4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Knapsack

Knapsack

Which items should I cram inside my knapsack?

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.

capacity: 10

Knapsack

	🍩	🍒	🌮	🎾	🐨
weight	6	2	4	3	11
value	20	8	14	13	35

capacity: 10

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

0/1 Knapsack

Suppose I only have one copy of each item.

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

🍒 🌮 🎾

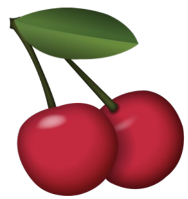
Total weight: 9

Total value: 35

Unbounded Knapsack

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

Then larger knapsacks

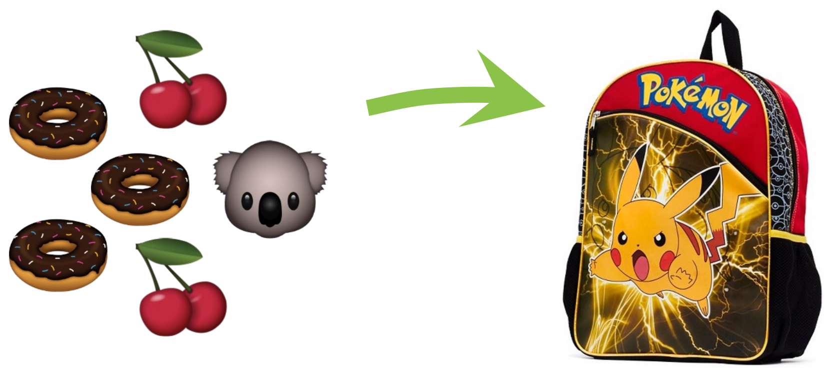
Then larger knapsacks

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

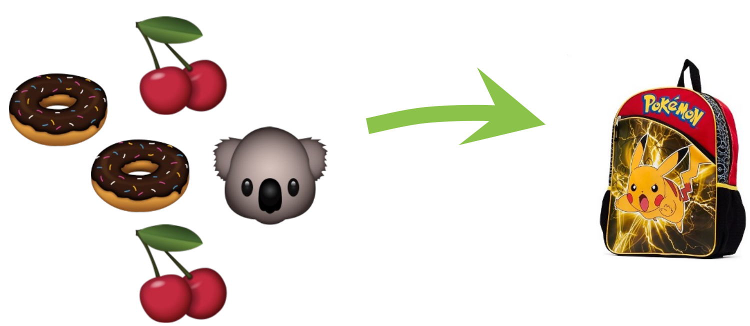
(1) Identify optimal substructure with overlapping subproblems.

If this is an optimal solution for capacity x



capacity x
value V

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



capacity x - w_i
value V - v_i

If there existed a more optimal solution, then adding a donut to that more optimal solution would improve the first solution.

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 **V[x]** be the optimal value for capacity x.

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

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

The optimal way to fill the smaller knapsack

The value of item i.

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

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.

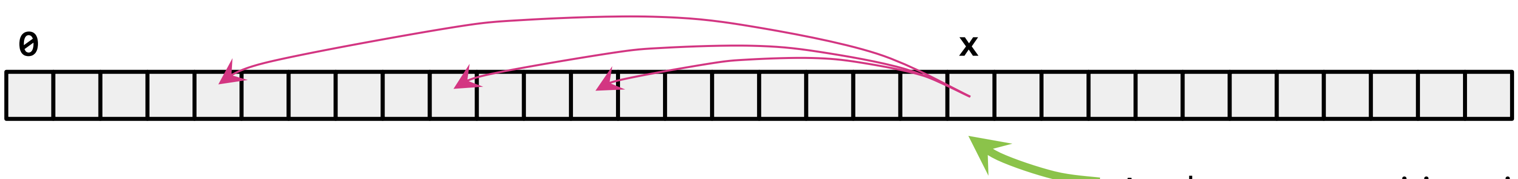
(4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

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)?



An element at position x in the table depends on elements at positions x - w_i for all i. So we want to fill out the values at these positions before x.

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

Unbounded Knapsack

```
algorithm unbounded_knapsack(capacity, weights, values):
  W = capacity
  n = weights.length //the number of items
  V[0] = 0
  for x = 1 to W:
    V[x] = 0
    for i = 0 to n-1:
      wᵢ = weights[i], vᵢ = values[i]
      if wᵢ ≤ x:
        V[x] = max{V[x], V[x-wᵢ] + vᵢ}
  return V[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. 👌

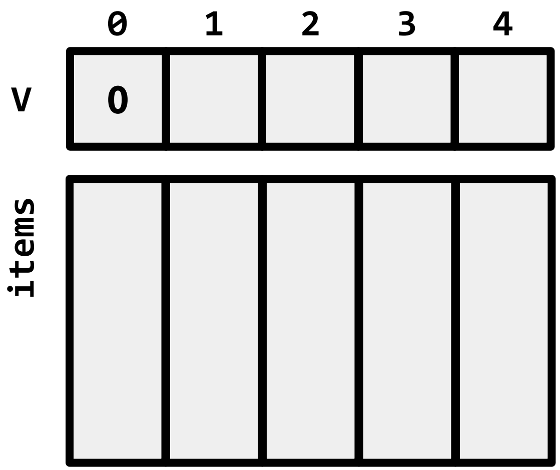
(4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

Unbounded Knapsack

```
algorithm unbounded_knapsack(capacity, weights, values):
  W = capacity
  n = weights.length  //the number of items
  V[0] = 0, items[0] = {}
  for x = 1 to W:
    V[x] = 0
    for i = 0 to n-1:
      wᵢ = weights[i], vᵢ = values[i]
      if wᵢ ≤ x:
        V[x] = max{V[x], V[x-wᵢ] + vᵢ}
        if V[x] updated: //keep track of the best option
          items[x] = items[x-wᵢ] ∪ {i}
  return items[W]
```

Runtime: $O(nW)$

Unbounded Knapsack



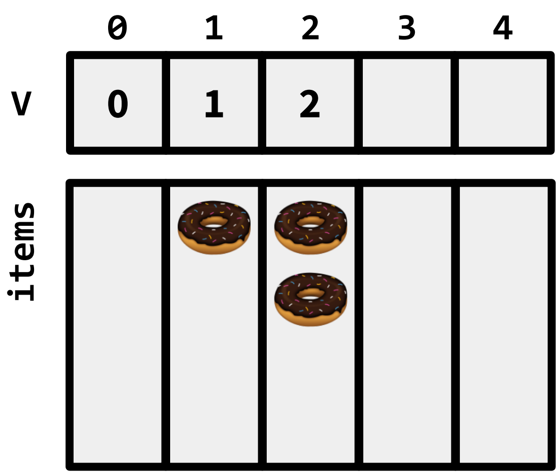
V[0] = 0

weight	1	2	3
value	1	4	6

capacity: 4

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

Unbounded Knapsack



weight	1	2	3	
value	1	4	6	capacity: 4

V[0] = 0

V[1] = max { V[1-1] + 1 } = max { **V[0] + 1** } = max { **1** } = 1

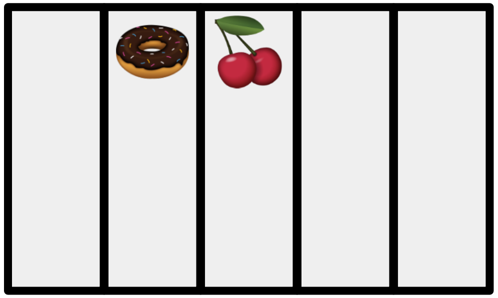
V[2] = max { V[2-1] + 1, V[2-2] + 4 } = max { **V[1] + 1**, V[0] + 4 } = max { **2**, 4 } = 4

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

Unbounded Knapsack

	0	1	2	3	4
V	0	1	4		

items



weight	1	2	3	
value	1	4	6	capacity: 4

V[0] = 0

V[1] = max { V[1-1] + 1 } = max { V[0] + 1 } = max { 1 } = 1

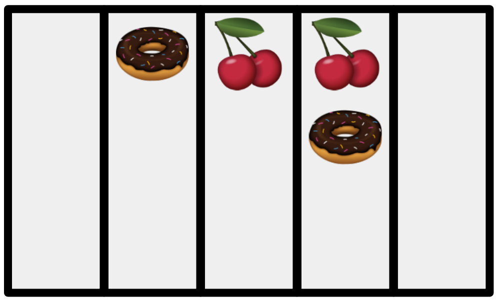
V[2] = max { V[2-1] + 1, V[2-2] + 4 } = max { V[1] + 1, **V[0] + 4** } = max { 2, **4** } = 4

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

Unbounded Knapsack

	0	1	2	3	4
V	0	1	4	5	

items



weight	1	2	3
value	1	4	6

capacity: 4

V[0] = 0

V[1] = max { V[1-1] + 1 } = max { V[0] + 1 } = max { 1 } = 1

V[2] = max { V[2-1] + 1, V[2-2] + 4 } = max { V[1] + 1, V[0] + 4 } = max { 2, 4 } = 4

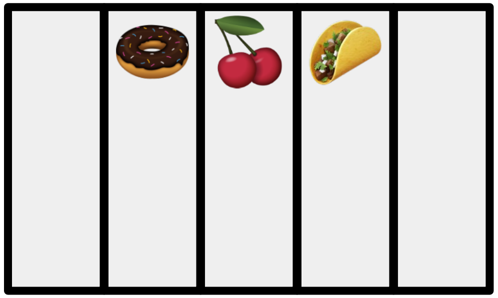
V[3] = max { V[3-1] + 1, V[3-2] + 4, V[3-3] + 6 }
= max { **V[2] + 1**, **V[1] + 4**, V[0] + 6 } = max { **5**, **5**, 6 } = 6

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

Unbounded Knapsack

	0	1	2	3	4
V	0	1	4	6	

items



weight	1	2	3
value	1	4	6

capacity: 4

V[0] = 0

V[1] = max { V[1-1] + 1 } = max { V[0] + 1 } = max { 1 } = 1

V[2] = max { V[2-1] + 1, V[2-2] + 4 } = max { V[1] + 1, V[0] + 4 } = max { 2, 4 } = 4

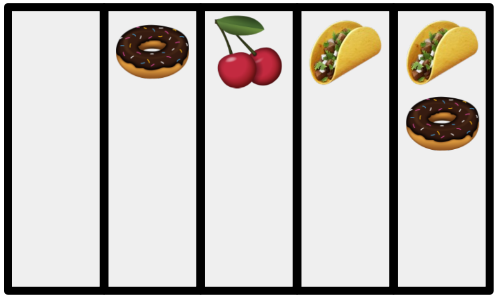
V[3] = max { V[3-1] + 1, V[3-2] + 4, V[3-3] + 6 }
= max { V[2] + 1, V[1] + 4, **V[0] + 6** } = max { 5, 5, **6** } = 6

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

Unbounded Knapsack

	0	1	2	3	4
V	0	1	4	6	7

items



weight	1	2	3
value	1	4	6

capacity: 4

V[0] = 0

V[1] = max { V[1-1] + 1 } = max { V[0] + 1 } = max { 1 } = 1

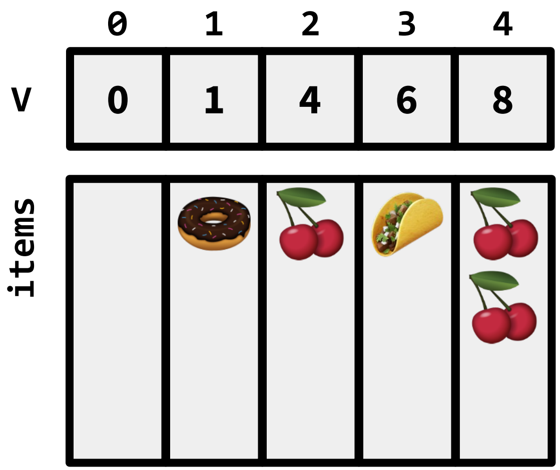
V[2] = max { V[2-1] + 1, V[2-2] + 4 } = max { V[1] + 1, V[0] + 4 } = max { 2, 4 } = 4

V[3] = max { V[3-1] + 1, V[3-2] + 4, V[3-3] + 6 }
= max { V[2] + 1, V[1] + 4, V[0] + 6 } = max { 5, 5, 6 } = 6

V[4] = max { V[4-1] + 1, V[4-2] + 4, V[4-3] + 6 }
= max { **V[3] + 1**, V[2] + 4, **V[1] + 6** } = max { **7**, 8, **7** } = 8

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

Unbounded Knapsack



weight	1	2	3	
value	1	4	6	capacity: 4

V[0] = 0

V[1] = max { V[1-1] + 1 } = max { V[0] + 1 } = max { 1 } = 1

V[2] = max { V[2-1] + 1, V[2-2] + 4 } = max { V[1] + 1, V[0] + 4 } = max { 2, 4 } = 4

V[3] = max { V[3-1] + 1, V[3-2] + 4, V[3-3] + 6 }
= max { V[2] + 1, V[1] + 4, V[0] + 6 } = max { 5, 5, 6 } = 6

V[4] = max { V[4-1] + 1, V[4-2] + 4, V[4-3] + 6 }
= max { V[3] + 1, **V[2] + 4**, V[1] + 6 } = max { 7, **8**, 7 } = 8

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

0/1 Knapsack

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

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 sub-problem needs information about which items have been used.

We have to guarantee that the sub-problem did not use Koko the koala.

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

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.

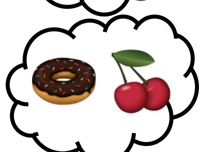
First solve the problem for small knapsacks

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

We need a two-dimensional table!

Then more items

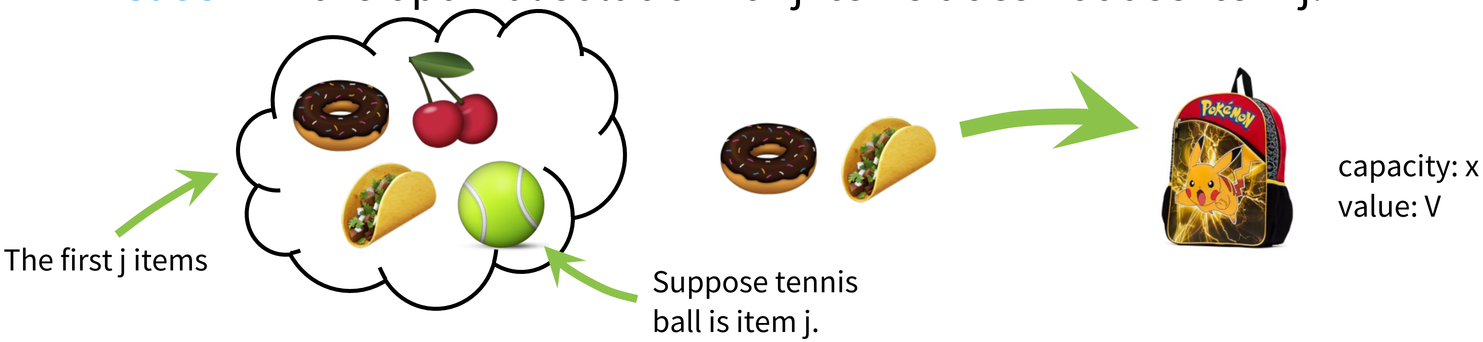
0/1 Knapsack

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



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

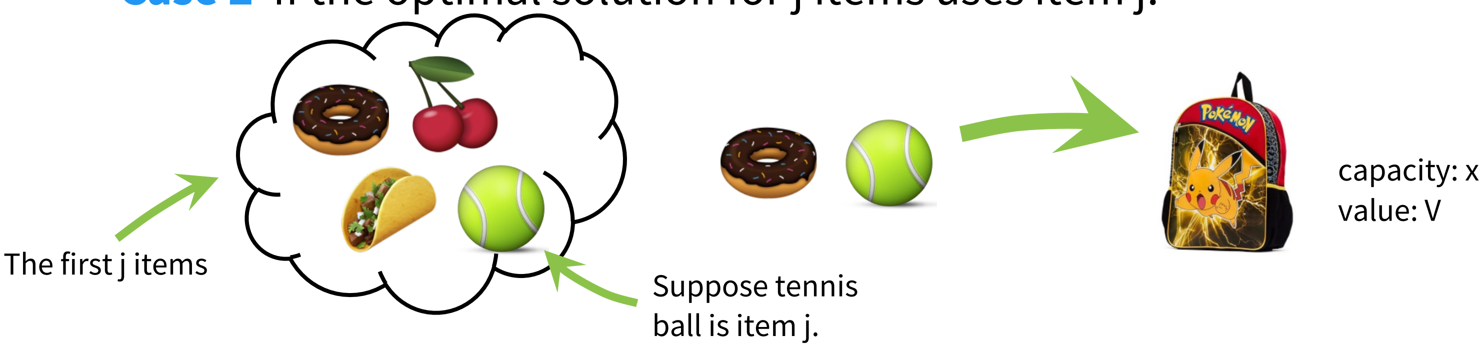
0/1 Knapsack

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 **V[x,j]** be the optimal value for capacity x with j items.

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

V[x,j-1] is for case 1: If optimal value for capacity x does not need item j, then we only need to fill capacity x with the previous j-1 items.

$V[x-w_j,j-1] + v_j$ is for case 2: If optimal value for capacity x indeed need item j, then we first need to fill capacity $x-w_j$ with the previous j-1 items, then put in item j.

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.

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
  V[x,0] = 0 for x = 0 to W
  V[0,i] = 0 for i = 0 to n
  for x = 1 to W:
    for j = 1 to n:
      V[x,j] = V[x,j-1]
      wⱼ = weights[j], vⱼ = values[j]
      if wⱼ ≤ x:
        V[x,j] = max{V[x,j], V[x-wⱼ] + vⱼ}
  return V[W,n]
```

Runtime: $O(nW)$

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

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

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

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

$x=1$, $j=1$, $w_j=1$, $v_j=1$ $V[1, 0] = 0$ $V[0,0] + 1 = 1$

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 (yellow) · relevant previous entry (green)

weight	1	2	3	
value	1	4	6	capacity: 3

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

$x=1,\ j=1,\ w_j=1,\ v_j=1$ $\quad V[1,0] = 0$ $\quad V[0,0] + 1 = 1$

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

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

$x=1$, $j=2$, $w_j=2$, $v_j=4$ $\quad V[1, 1] = 1$ $\quad x < w_j$

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

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

$x=1$, $j=3$, $w_j=3$, $v_j=6$ $V[1, 2] = 1$ $x < w_j$

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

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

$x=2$, $j=1$, $w_j=1$, $v_j=1$ $\quad$ $V[2, 0] = 0$ $\quad$ $V[1,0] + 1 = 1$

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

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

$x=2$, $j=1$, $w_j=1$, $v_j=1$ $\quad V[2, 0] = 0$ $\quad V[1,0] + 1 = 1$

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

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

$x=2,\ j=2,\ w_j=2,\ v_j=4$ $\quad V[2,1] = 1$ $\quad V[0,1] + 4 = 4$

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	
value	1	4	6	capacity: 3

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

$x=2,\ j=2,\ w_j=2,\ v_j=4$ $V[2,1] = 1$ $V[0,1] + 4 = 4$

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

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

$x=2$, $j=3$, $w_j=3$, $v_j=6$ $V[2,2] = 4$ $x < w_j$

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

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

$x=3$, $j=1$, $w_j=1$, $v_j=1$ $V[3,0] = 0$ $V[2,0] + 1 = 1$

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

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

$x=3$, $j=1$, $w_j=1$, $v_j=1$ $V[3,0] = 0$ $V[2,0] + 1 = 1$

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

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

$x=3$, $j=2$, $w_j=2$, $v_j=4$ $V[3,1] = 1$ $V[1,1] + 4 = 4$

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

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

$x=3,\ j=2,\ w_j=2,\ v_j=4$ $\quad V[3,1] = 1$ $\quad V[1,1] + 4 = 5$

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

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

$x=3,\ j=3,\ w_j=3,\ v_j=6$ $\quad V[3,2] = 5$ $\quad V[0,2] + 6 = 6$

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	6

current entry · relevant previous entry

weight	1	2	3
value	1	4	6

capacity: 3

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

$x=3,\ j=3,\ w_j=3,\ v_j=6$ $\quad V[3,2] = 5$ $\quad V[0,2] + 6 = 6$

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	6

weight	1	2	3	
value	1	4	6	capacity: 3

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

current entry

relevant previous entry

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

Independent Set

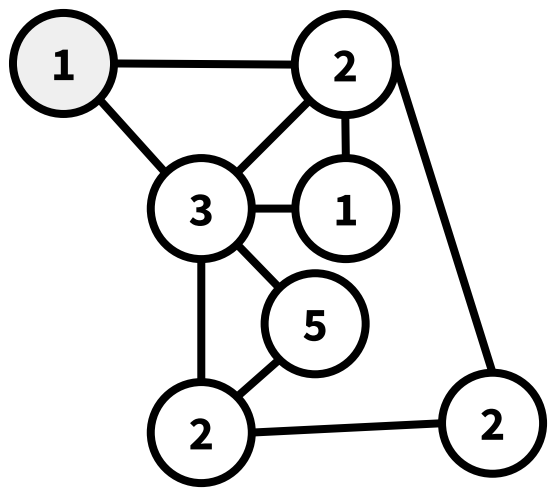
(Advanced Topic)

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.

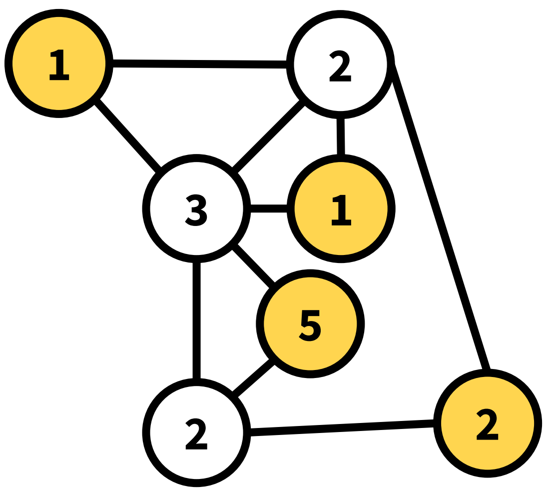


Maximal Independent Set

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.

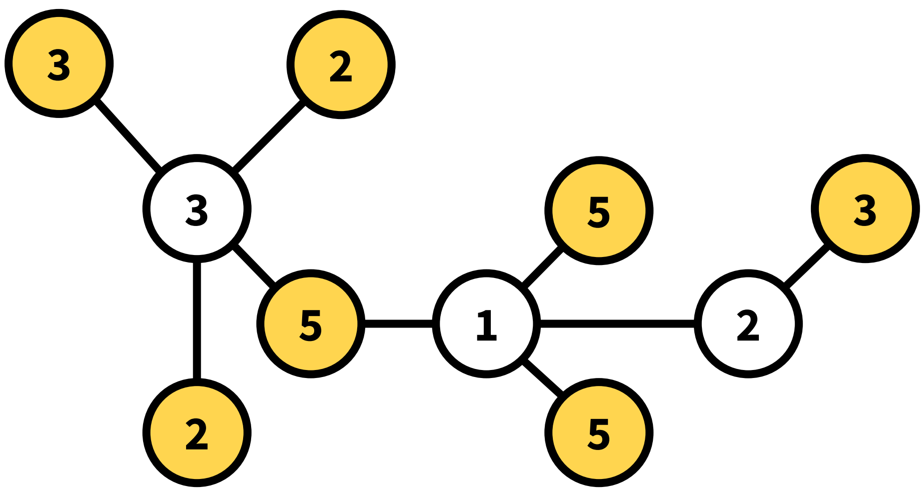
We'll learn what this means in a bit, but for now take it to mean we're unlikely to find an efficient algorithm.

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

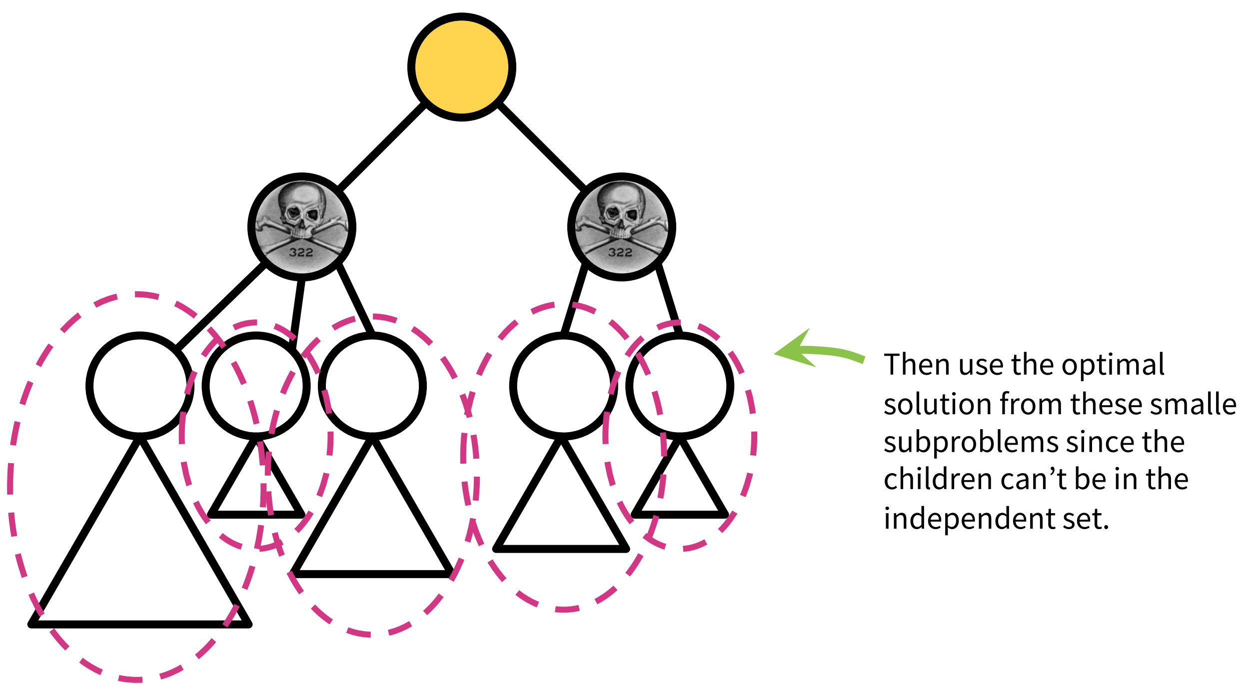
Task Find the maximal independent set in a tree.

(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.



Maximal Independent Set

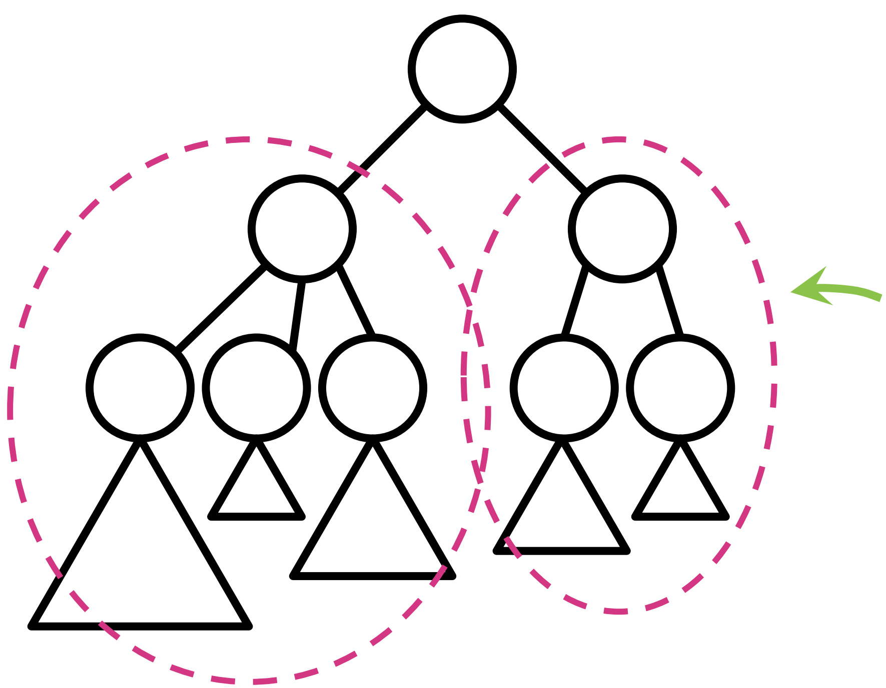
Task Find the maximal independent set in a tree.

(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.

$$\texttt{A[u]} = \max \begin{cases} \texttt{w(u)} + \Sigma_{\texttt{v}\in\texttt{u.grandchildren}} \texttt{A[v]} & \text{case 1} \\ \Sigma_{\texttt{v}\in\texttt{u.children}} \texttt{A[v]} & \text{case 2} \end{cases}$$

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:
```

$$w = \max \begin{cases} w(u) + \Sigma_{v \in u.grandchildren}\ \texttt{max\_independent\_set\_helper(v)} \\ \Sigma_{v \in u.children}\ \texttt{max\_independent\_set\_helper(v)} \end{cases}$$

```
    cache[root] = w
    return w
```

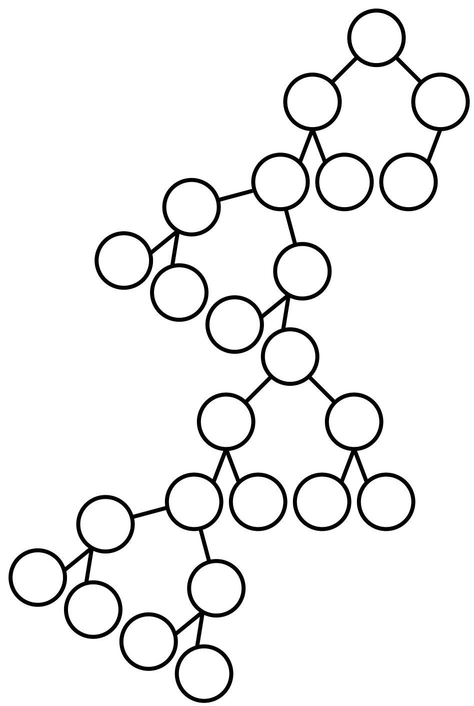
Runtime: O(|V|)

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**?

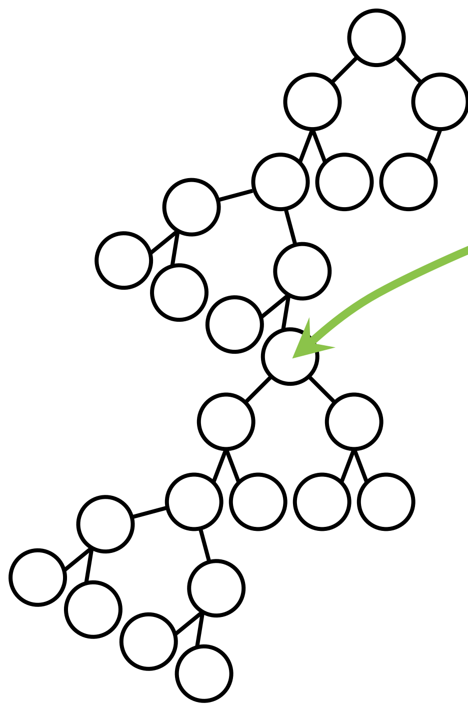


Maximal Independent Set

Task Find the maximal independent set in a tree.

(3) Use dynamic programming to solve the problem.

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



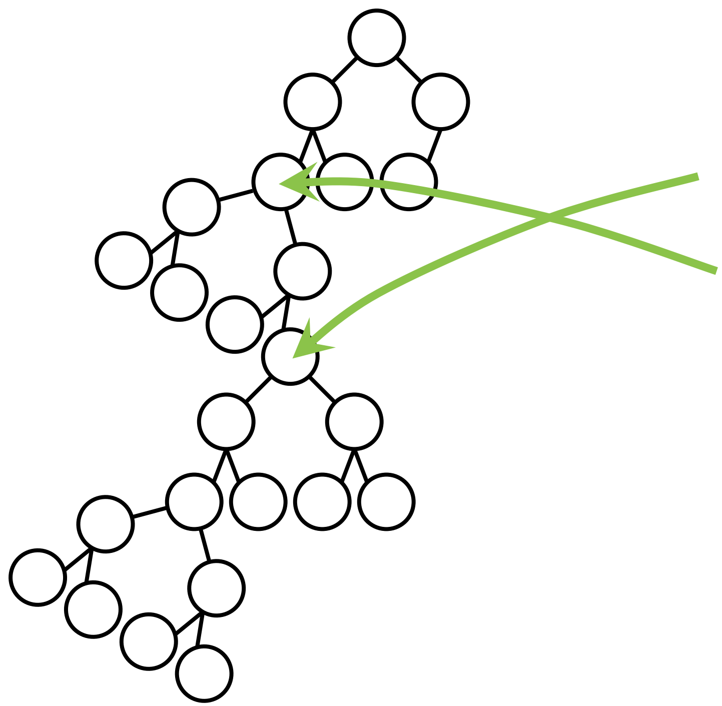
How often would we ask about the **subtree rooted here**?

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**?



How often would we ask about the **subtree rooted here**?

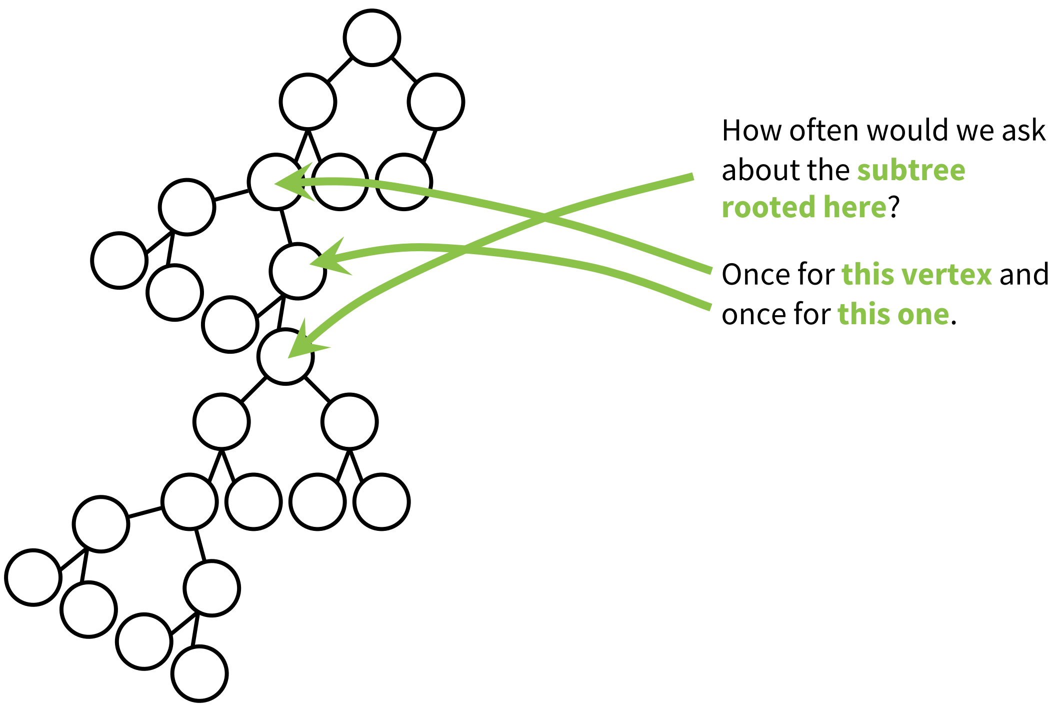
Once for **this vertex**

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**?

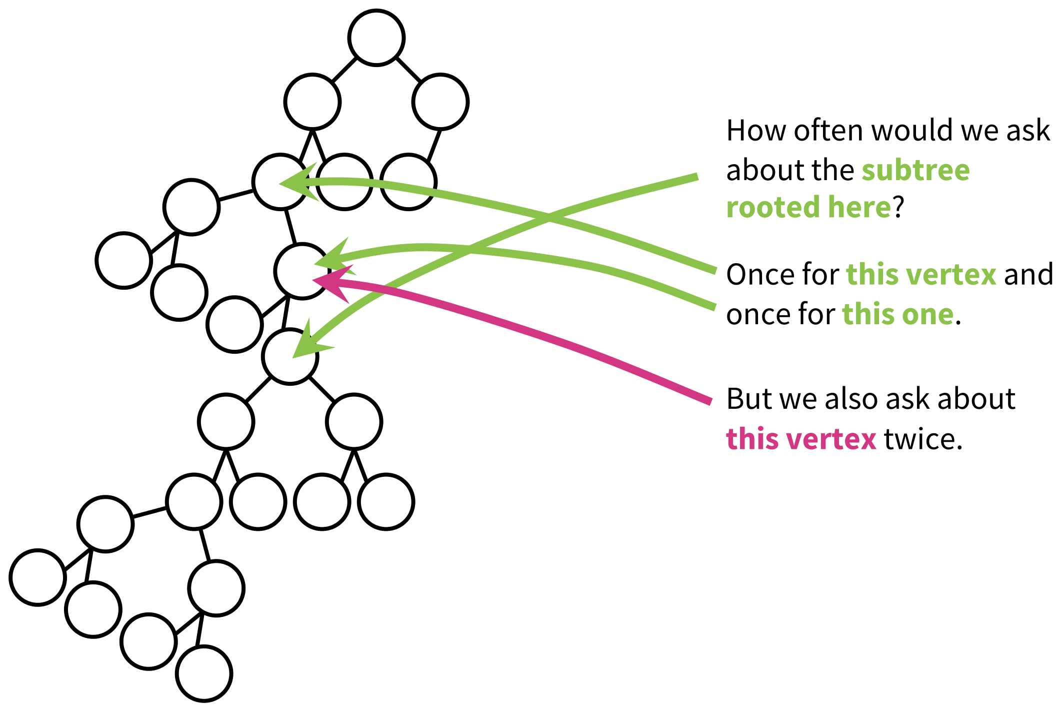


Maximal Independent Set

Task Find the maximal independent set in a tree.

(3) Use dynamic programming to solve the problem.

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

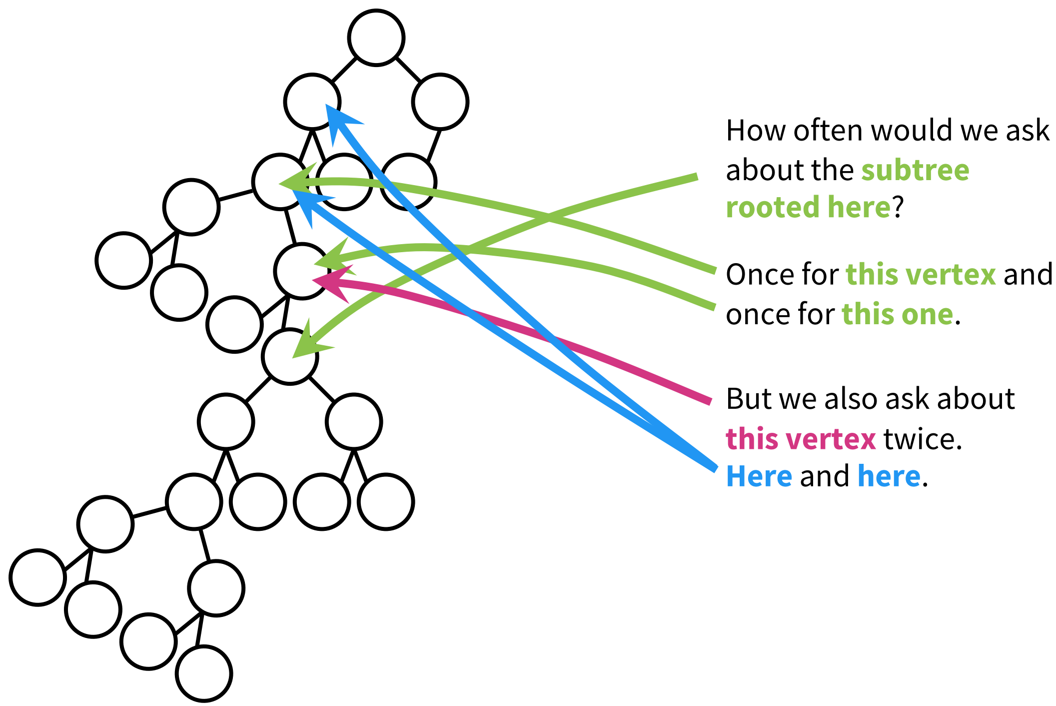


Maximal Independent Set

Task Find the maximal independent set in a tree.

(3) Use dynamic programming to solve the problem.

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

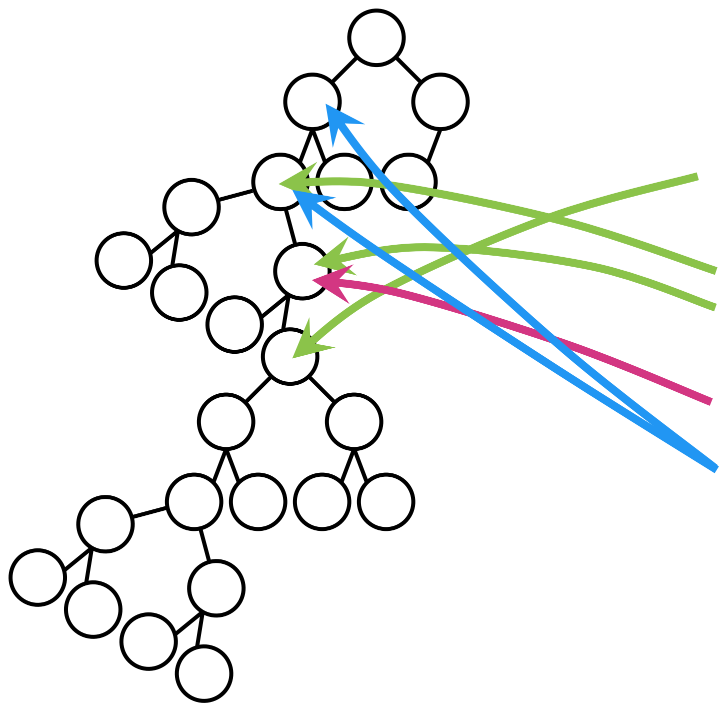


Maximal Independent Set

Task Find the maximal independent set in a tree.

(3) Use dynamic programming to solve the problem.

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



How often would we ask about the **subtree rooted here**?

Once for **this vertex** and once for **this one**.

But we also ask about **this vertex** twice. **Here** and **here**.

This would quickly blow up without DP to take advantage of overlapping subproblems!

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.

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.

Acknowledgement: Part of the materials are adapted from Virginia Williams and David Eng's lectures on algorithms. We appreciate their contributions.