Dynamic Programming II

Summer 2018 • Lecture 07/31

A Few Notes

Homework 4

Due Thursday at 5 p.m. on Gradescope.

Tutorial 6

RSVP at https://goo.gl/forms/PLJr5gknNWFgccww2

Outline for Today

- Dynamic Programming
  - More DP algorithms
    - 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.

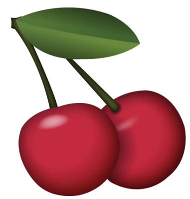
How is this different than divide and conquer? **Memoization**, which keeps track of the small problems you've already solved to prevent resolving the same problem more than once.

Knapsack

Knapsack

Which items should I cram inside my backpack?

We have n items with weights and values.

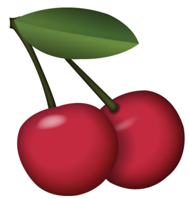
item:					
weight:	6	2	4	3	11
value:	20	8	14	13	35

Knapsack

Which items should I cram inside my backpack?

This is Koko the koala.

We have n items with weights and values.

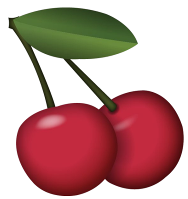
item:					
weight:	6	2	4	3	11
value:	20	8	14	13	35

Knapsack

Which items should I cram inside my backpack?

We have n items with weights and values.

This is Koko the koala.

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.

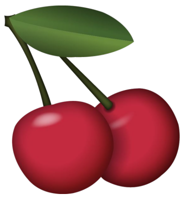
capacity: 10

Knapsack

This is Koko the koala.

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

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?

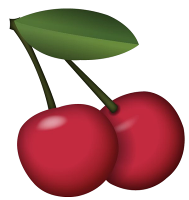
Total weight: 9

Total value: 35

Unbounded Knapsack

Some notation

This is Koko the koala.

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

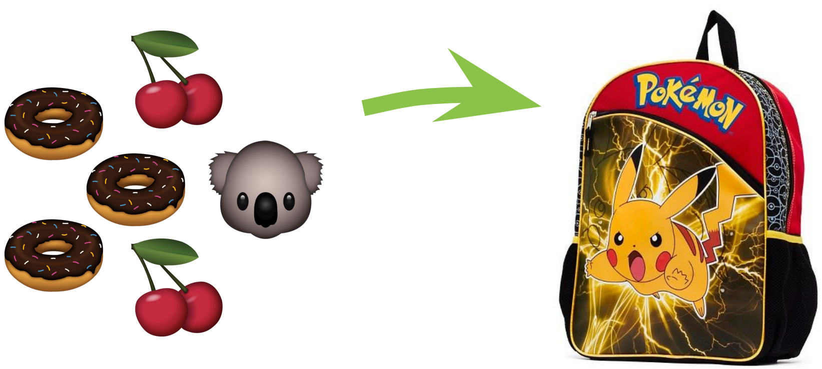
Then larger knapsacks

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



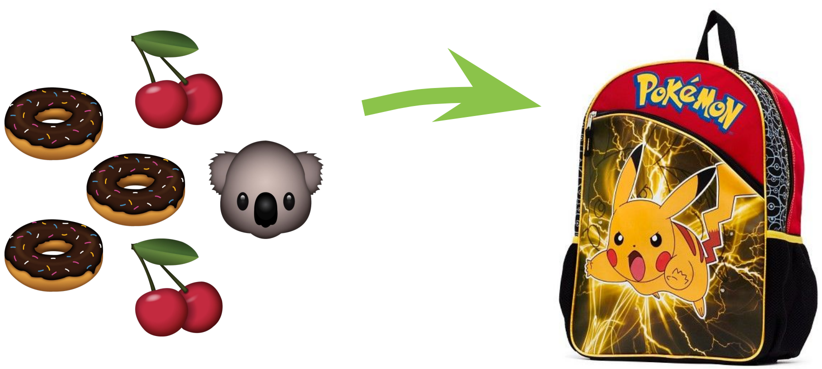
capacity x
value V

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

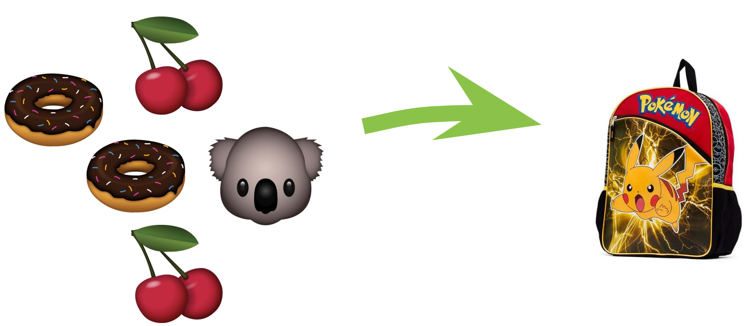
(1) Identify optimal substructure with overlapping subproblems.

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 = 



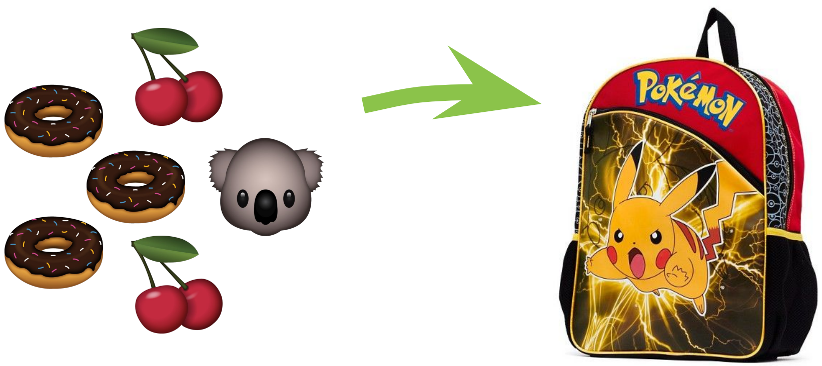
capacity x - w_i
value V - v_i

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

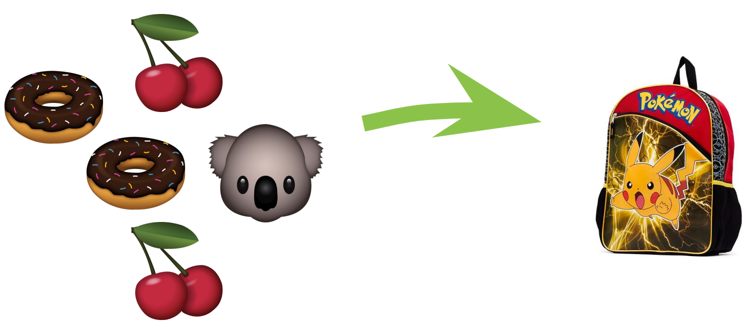
(1) Identify optimal substructure with overlapping subproblems.

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

K[x] = max$_i${  +  }

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

The optimal way to fill the smaller knapsack

The value of item i.

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

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.

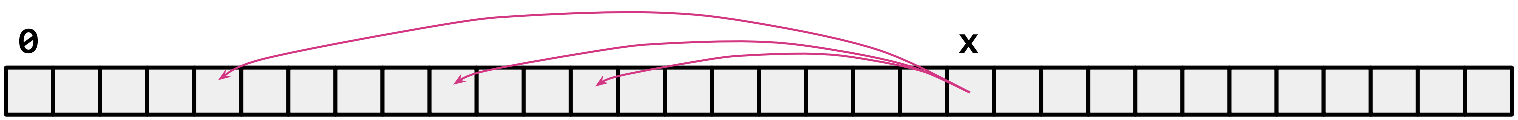
(4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

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)?



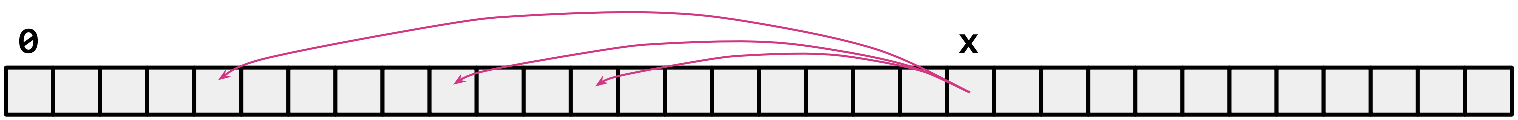
$$K[x] = \begin{cases} 0 & \text{if there are no } i \text{ where } w_i \leq x \\ \max_i\{K[x-w_i] + v_i\} & \text{otherwise} \end{cases}$$

Unbounded Knapsack

Task Find the items to put in an unbounded knapsack.

(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.

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

Unbounded Knapsack

```
def 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 = weights[i], v_i = values[i]
      if w_i ≤ 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.

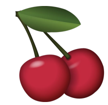
Unbounded Knapsack

```
def 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:
      w_i = weights[i], v_i = values[i]
      if w_i ≤ x:
        K[x] = max{K[x], K[x-w_i] + v_i}
        if K[x] updated:
          items[x] = items[x-w_i] ∪ {i}
  return items[W]
```

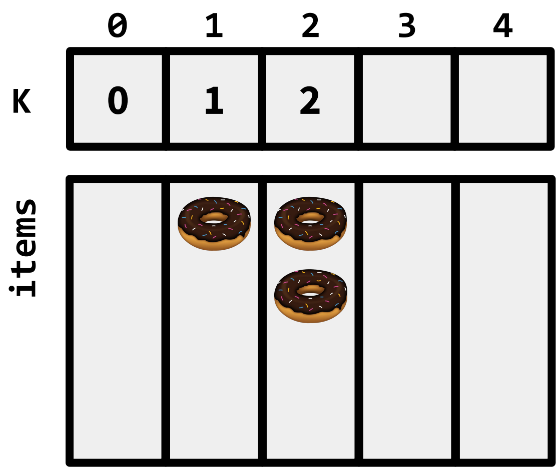
Runtime: $O(nW)$

Unbounded Knapsack

	0	1	2	3	4
K	0				
items					

				
weight	1	2	3	
value	1	4	6	capacity: 4

Unbounded Knapsack

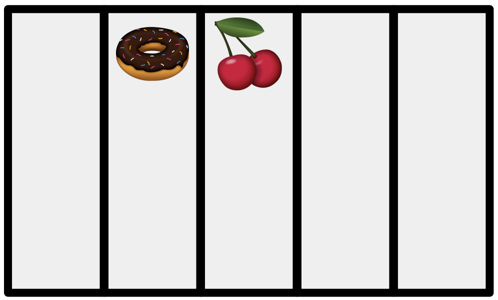


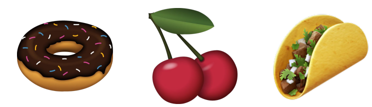
weight	1	2	3	
value	1	4	6	capacity: 4

Unbounded Knapsack

	0	1	2	3	4
K	0	1	4		

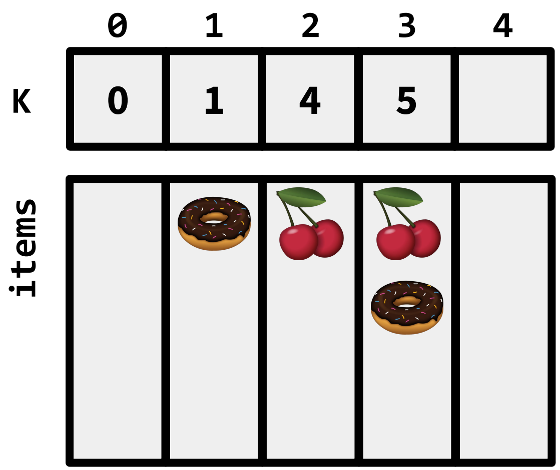
items

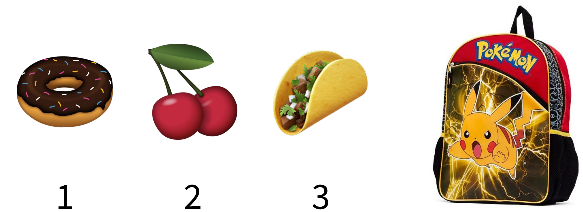




weight	1	2	3	
value	1	4	6	capacity: 4

Unbounded Knapsack

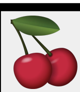


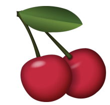


weight	1	2	3
value	1	4	6

capacity: 4

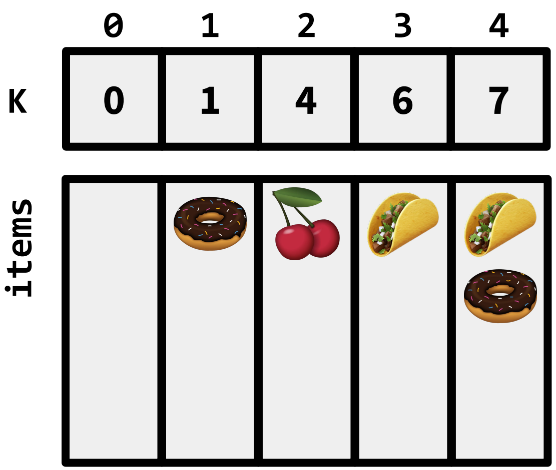
Unbounded Knapsack

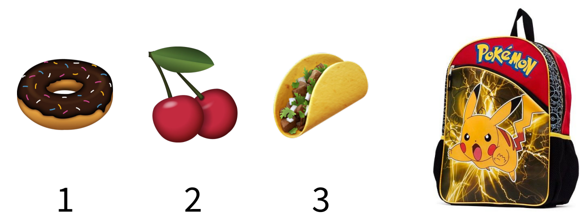
	0	1	2	3	4
K	0	1	4	6	
items					

			
weight	1	2	3
value	1	4	6

capacity: 4

Unbounded Knapsack

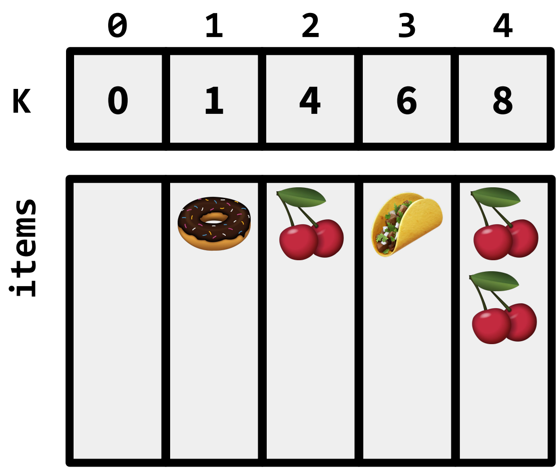


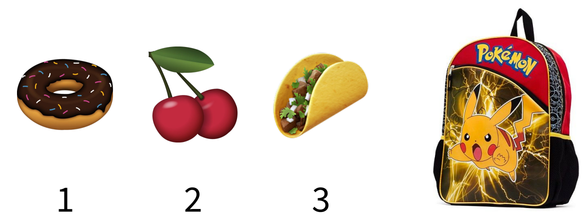


weight	1	2	3
value	1	4	6

capacity: 4

Unbounded Knapsack





weight	1	2	3	
value	1	4	6	capacity: 4

Knapsack

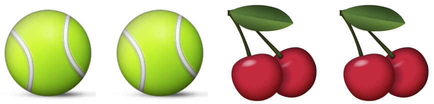
weight	6	2	4	3	11
value	20	8	14	13	35

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

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 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.

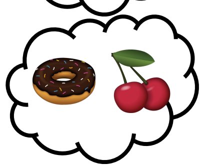
First solve the problem for small knapsacks

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.

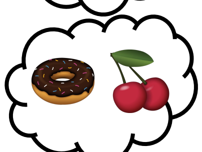
First solve the problem for small knapsacks

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.

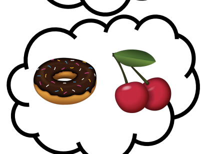
First solve the problem for small knapsacks

Then larger knapsacks

Then larger knapsacks

First solve the problem for few items

Then more items



Then more items

We need a two-dimensional table!

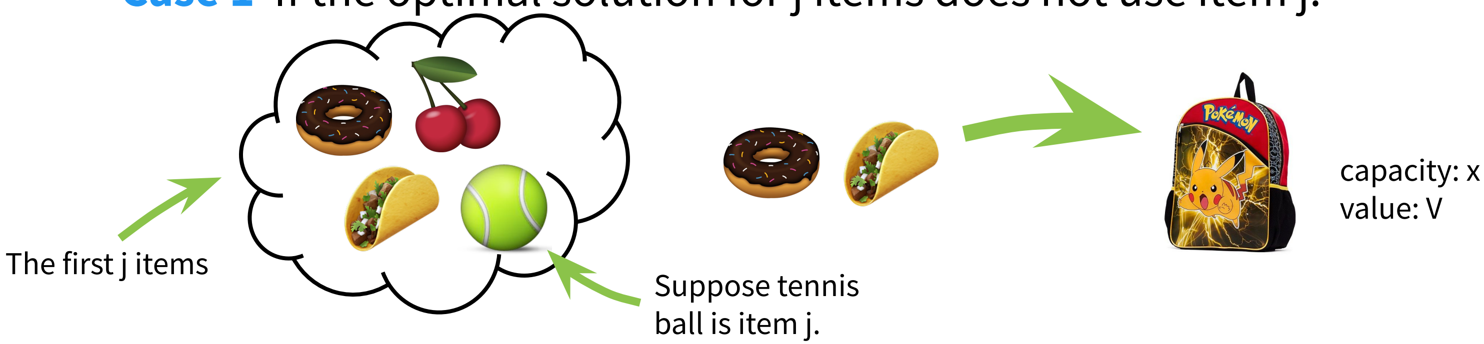
0/1 Knapsack

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

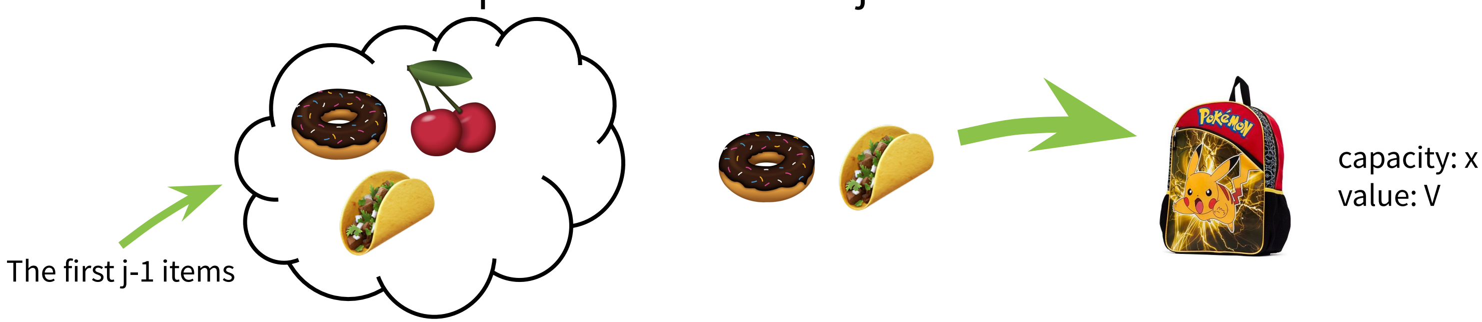
(1) Identify optimal substructure with overlapping subproblems.

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.



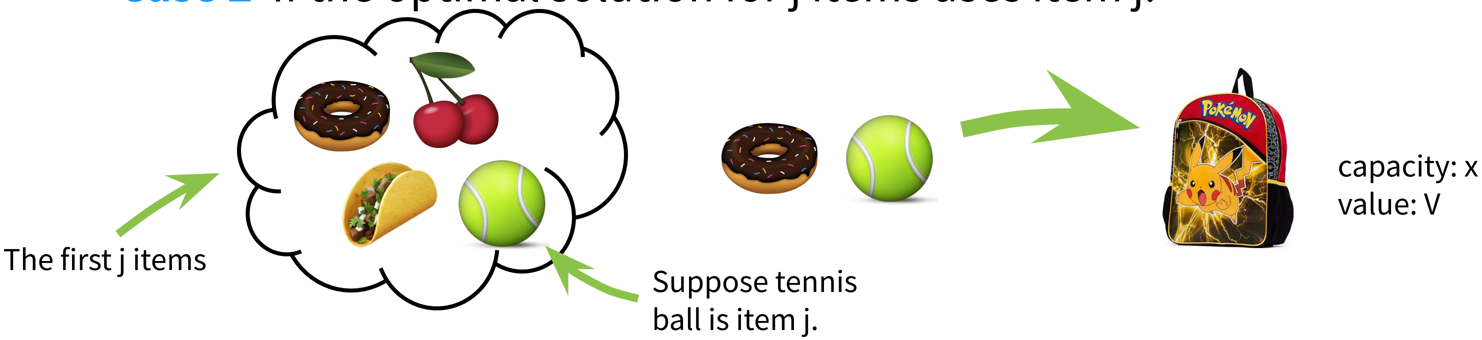
0/1 Knapsack

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

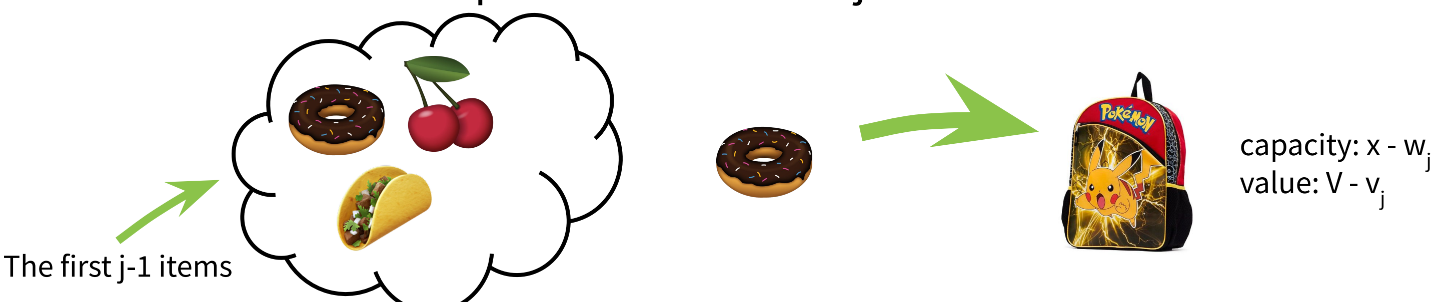
(1) Identify optimal substructure with overlapping subproblems.

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.

$$
\texttt{K[x,j]} = \begin{cases} \texttt{0} & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are 0} \\ \texttt{max\{K[x,j-1], K[x-w}_\texttt{j}\texttt{,j-1] + v}_\texttt{j}\texttt{\}} & \text{otherwise} \end{cases}
$$

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

```
def 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. 👌

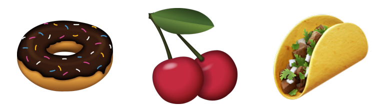
(4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

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

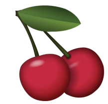
$$
K[x,j] = \begin{cases} 0 & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}
$$

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

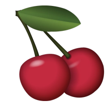
weight	1	2	3	
value	1	4	6	capacity: 3

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

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

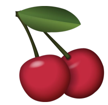
$$K[x,j] = \begin{cases} 0 & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}$$

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

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

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 (yellow) — relevant previous entry (green)

weight	1	2	3	
value	1	4	6	capacity: 3

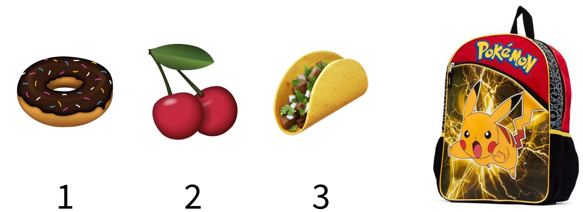
$$K[x,j] = \begin{cases} 0 & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}$$

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

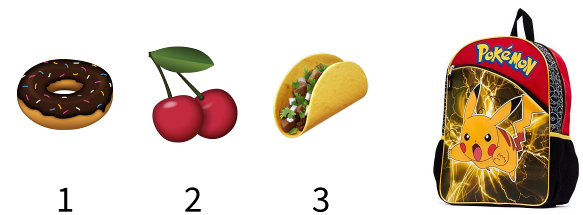
$$K[x,j] = \begin{cases} 0 & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}$$

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

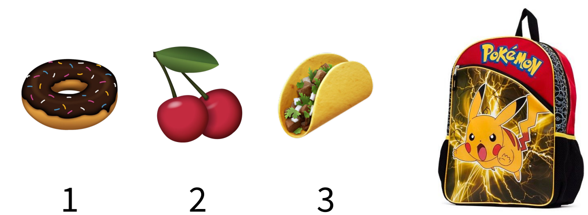
$$K[x,j] = \begin{cases} 0 & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}$$

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

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

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
value	1	4	6

capacity: 3

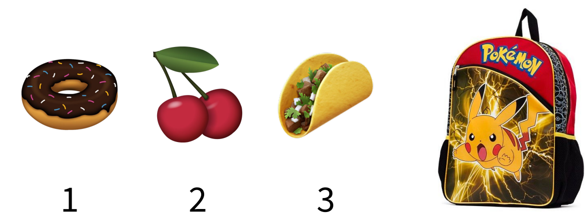
$$
K[x,j] = \begin{cases} 0 & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}
$$

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

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

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

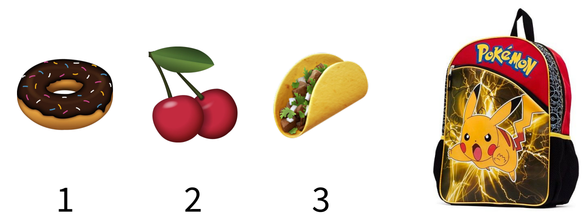
$$K[x,j] = \begin{cases} 0 & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}$$

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

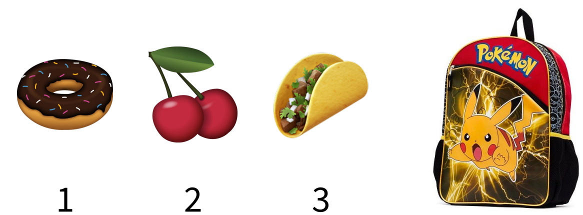
$$K[x,j] = \begin{cases} 0 & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}$$

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

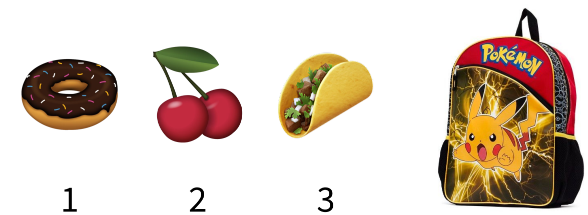
$$K[x,j] = \begin{cases} 0 & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}$$

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

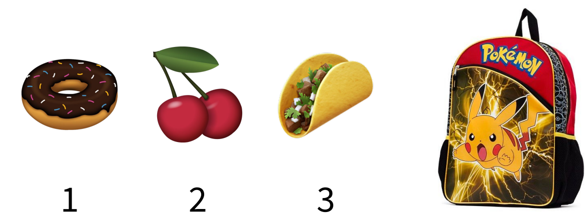
capacity: 3

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

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

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

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

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

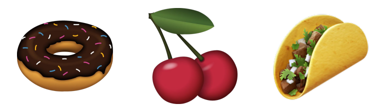
0/1 Knapsack

	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

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

current entry

relevant previous entry



weight	1	2	3	
value	1	4	6	capacity: 3

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

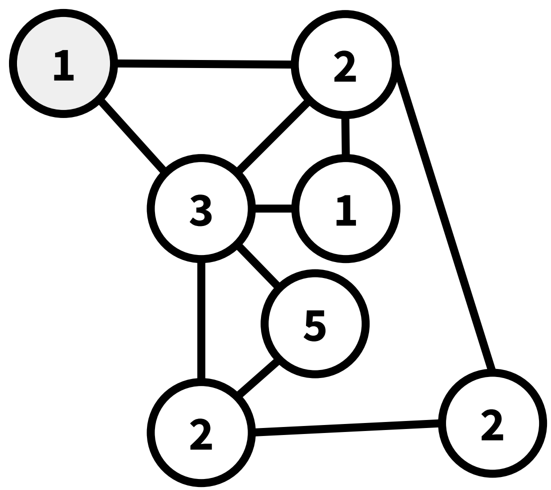
Independent Set

Maximum Independent Set

What is the maximum 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.

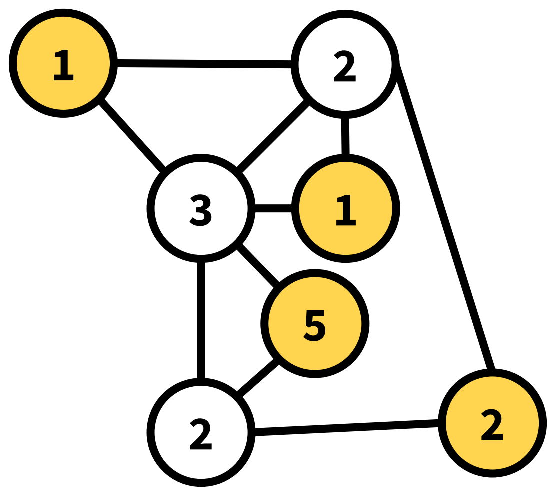
A maximum independent set has the largest weight.



Maximum Independent Set

What is the maximum 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 maximum independent set has the largest weight.

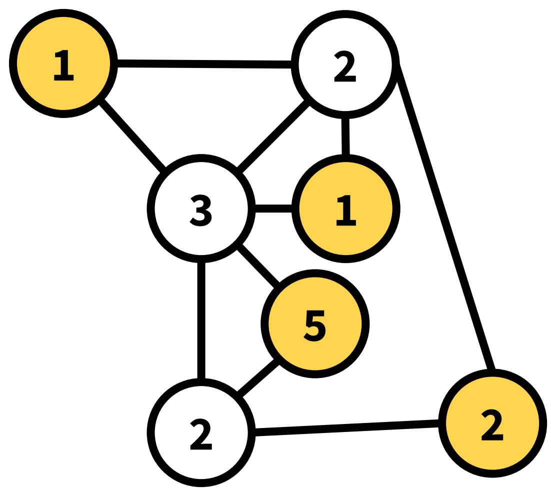


Maximum Independent Set

What is the maximum 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 maximum independent set has the largest weight.



This problem is NP-complete.

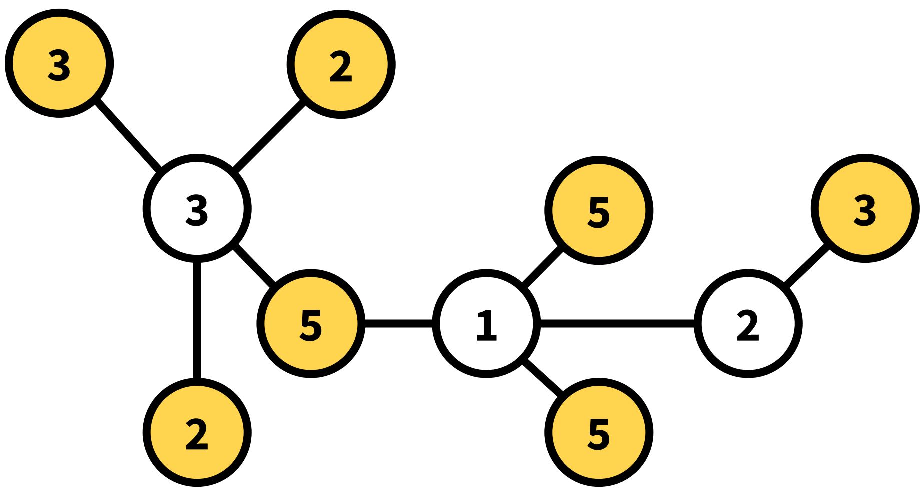
We'll learn what this means in a bit, but for now take it to mean we're unlikely to find an efficient algorithm.

Maximum Independent Set

What is the maximum 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 maximum independent set has the largest weight.



Maximum Independent Set

Task Find the maximum 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.

Maximum Independent Set

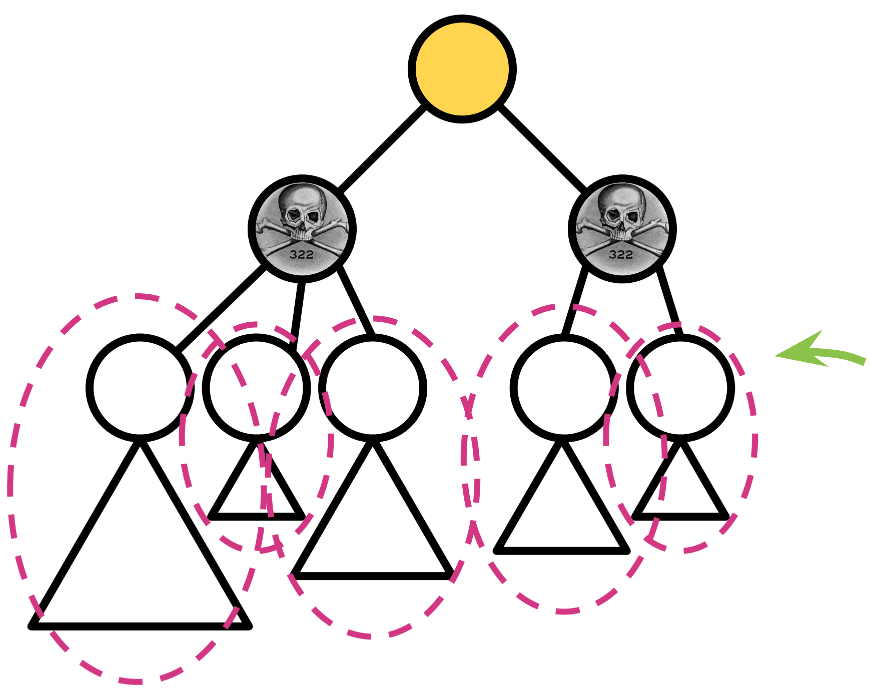
Task Find the maximum independent set in a tree.

(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 maximum independent set.



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

Maximum Independent Set

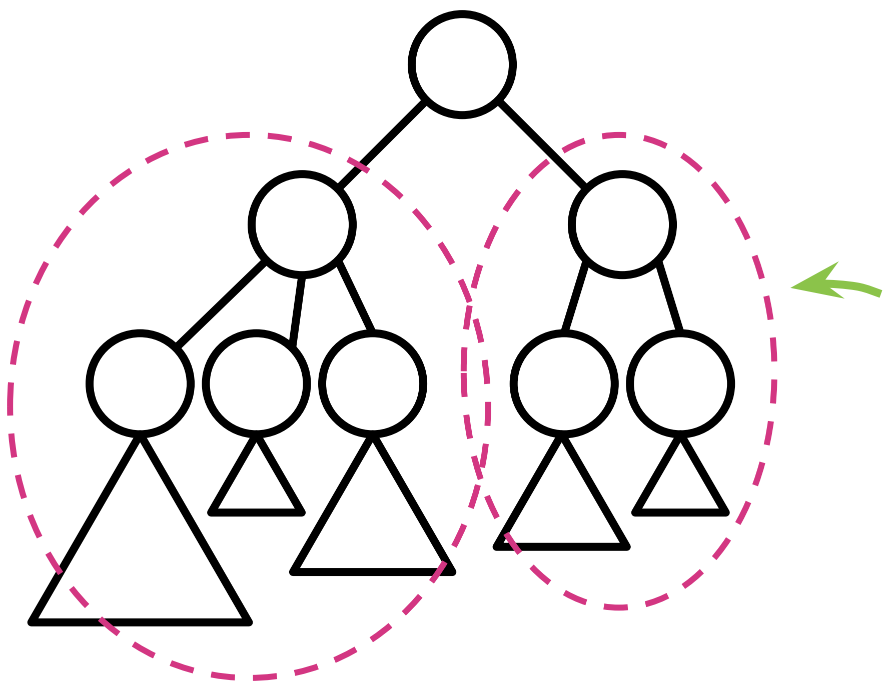
Task Find the maximum independent set in a tree.

(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 maximum independent set.



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

Maximum Independent Set

Task Find the maximum 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.

Maximum Independent Set

Task Find the maximum independent set in a tree.

(2) Define a recursive formulation.

Let **`A[u]`** be the weight of a maximum independent set in the tree rooted at u.

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

Maximum Independent Set

Task Find the maximum 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.

Maximum Independent Set

```
cache = {}
def max_independent_set_helper(root):
  if is_leaf(root):
    return root.weight
  else if root in cache:
    return cache[root]
  else:
    w = max { w(u) + Σ_{v∈u.grandchildren} max_independent_set_helper(v)
            { Σ_{v∈u.children} max_independent_set_helper(v)

    cache[root] = w
    return w
```

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

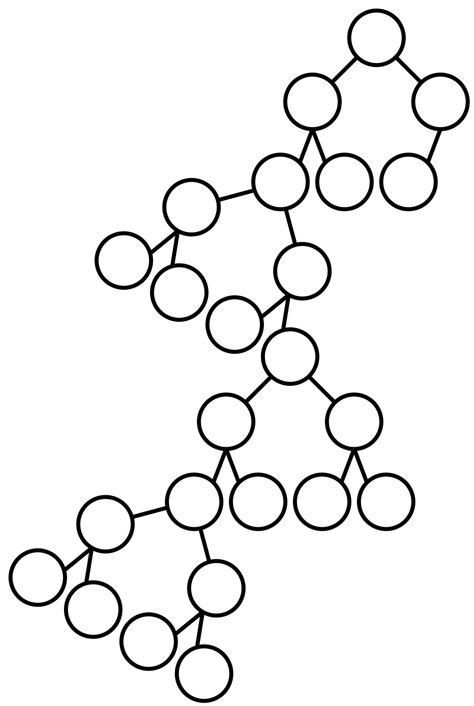
Runtime: $O(|V|)$

Maximum Independent Set

Task Find the maximum independent set in a tree.

(3) Use dynamic programming to solve the problem.

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

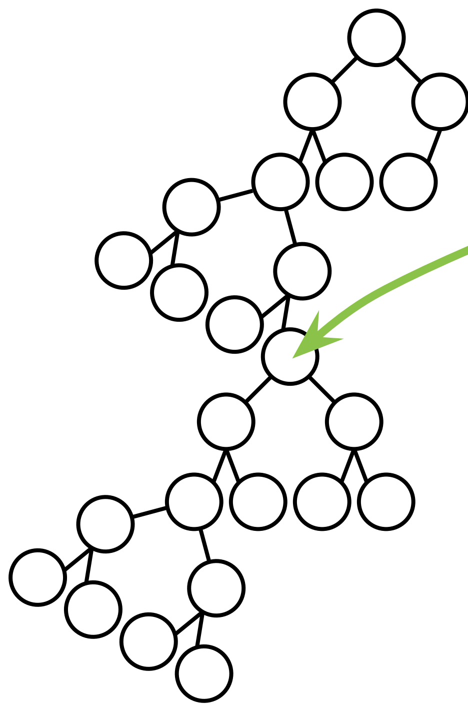


Maximum Independent Set

Task Find the maximum independent set in a tree.

(3) Use dynamic programming to solve the problem.

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



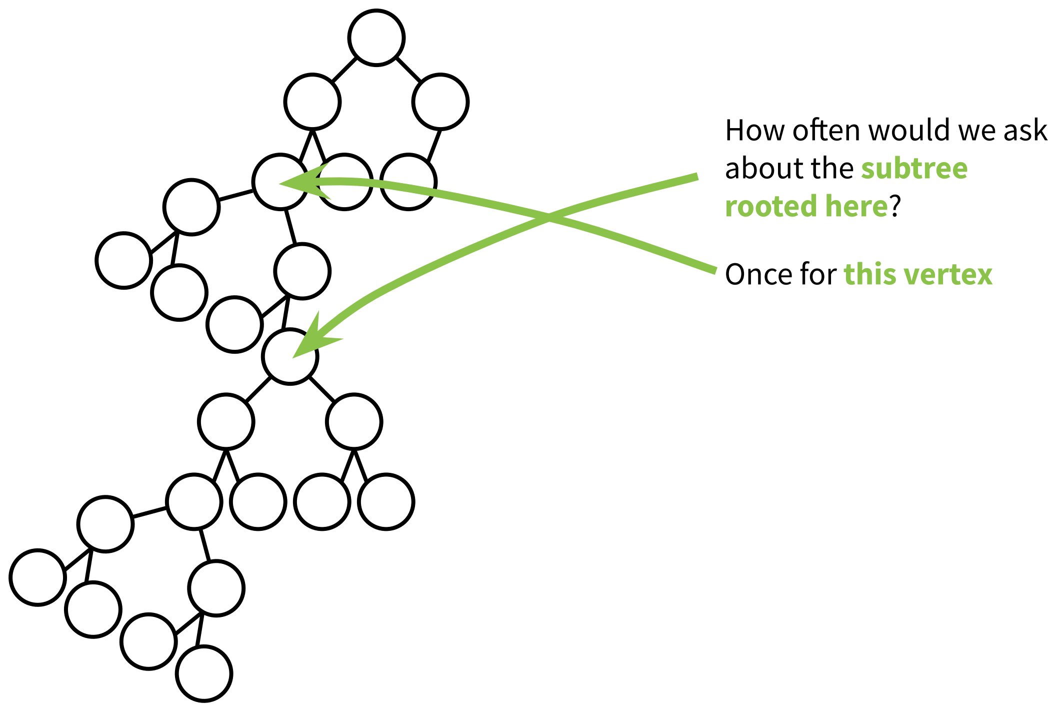
How often would we ask about the **subtree rooted here**?

Maximum Independent Set

Task Find the maximum independent set in a tree.

(3) Use dynamic programming to solve the problem.

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

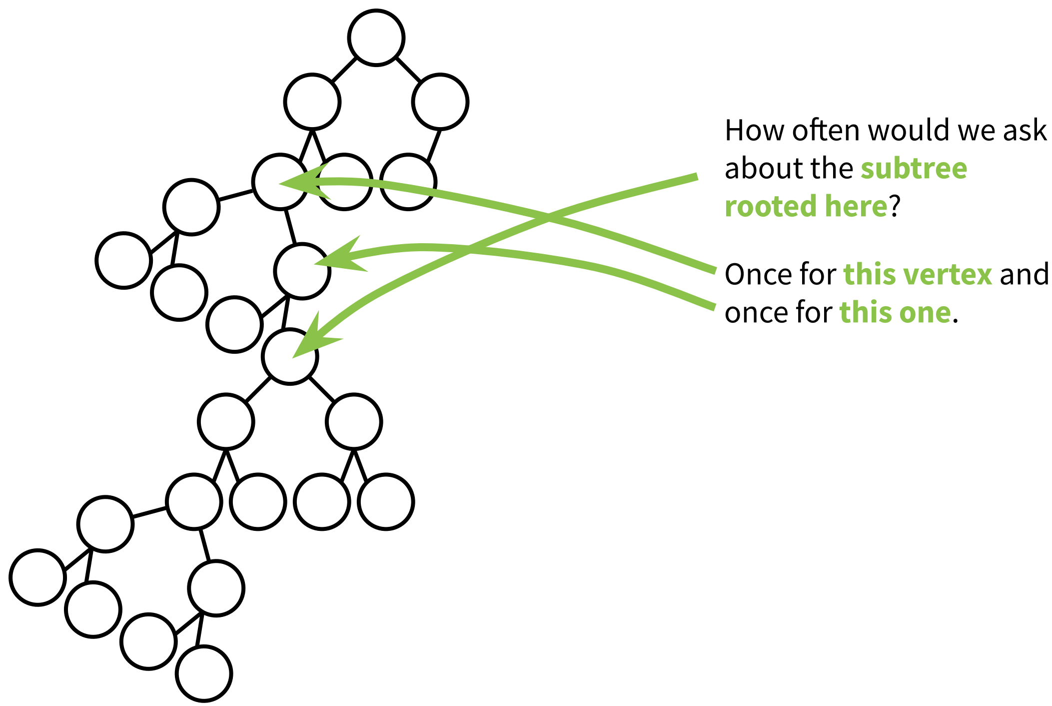


Maximum Independent Set

Task Find the maximum independent set in a tree.

(3) Use dynamic programming to solve the problem.

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

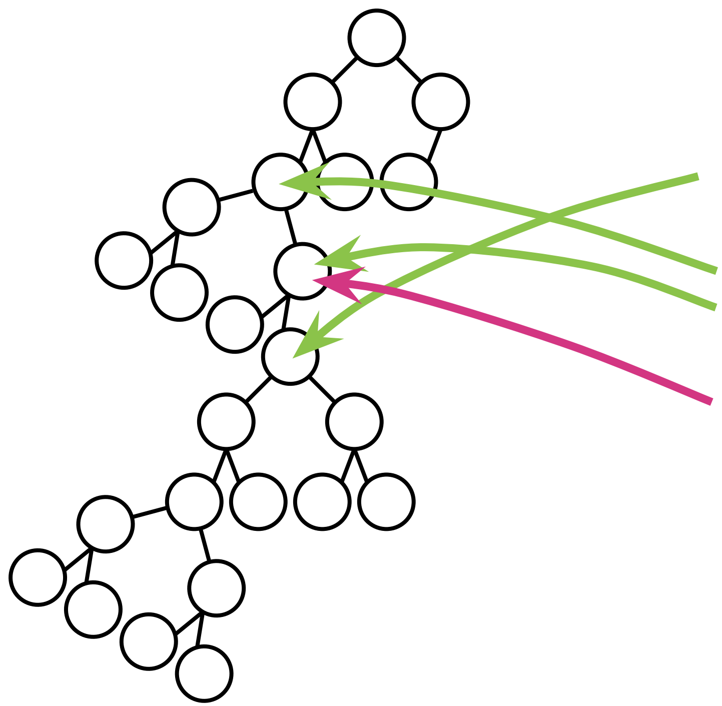


Maximum Independent Set

Task Find the maximum independent set in a tree.

(3) Use dynamic programming to solve the problem.

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

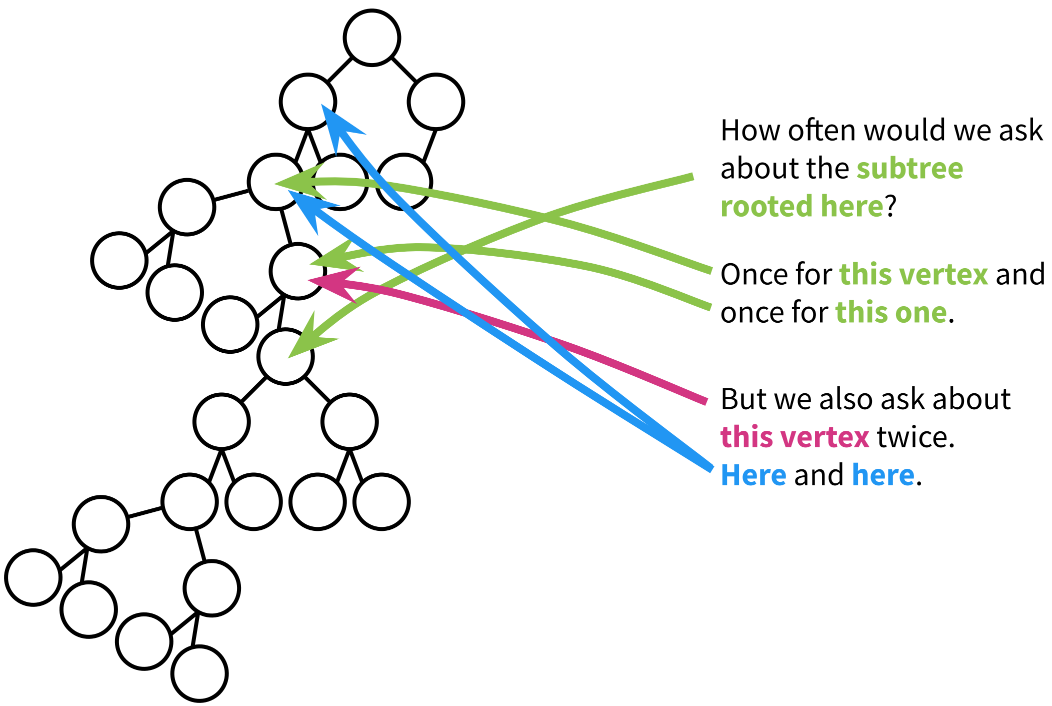


Maximum Independent Set

Task Find the maximum independent set in a tree.

(3) Use dynamic programming to solve the problem.

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

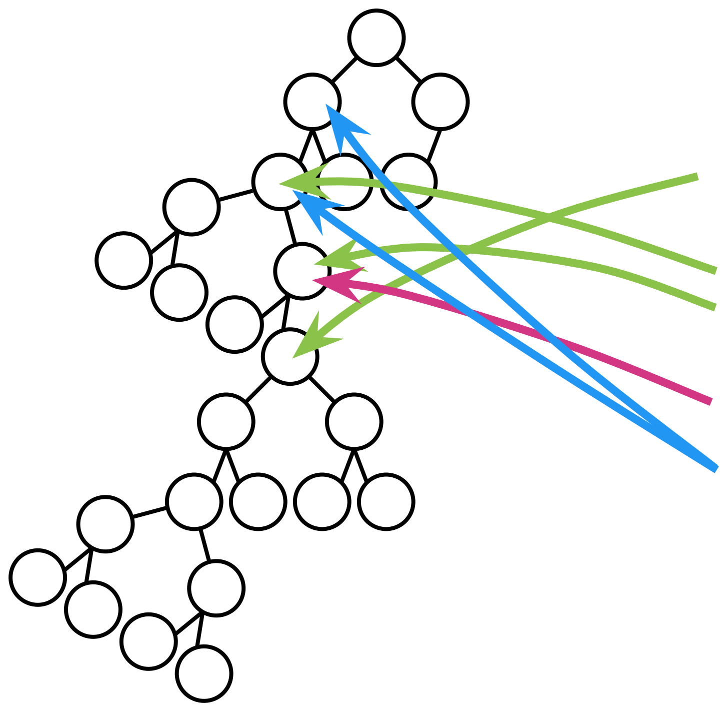


Maximum Independent Set

Task Find the maximum independent set in a tree.

(3) Use dynamic programming to solve the problem.

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



Maximum Independent Set

Task Find the maximum 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	$O(\lvert X\rvert\lvert Y\rvert)$ **worst-case**	$\lvert X\rvert$ and $\lvert Y\rvert$ 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.
Maximum Independent Set	$O(\lvert V\rvert)$ **worst-case**	$\lvert V\rvert$ is the number of vertices in the tree.	NP-complete for graphs, this alg. works for trees; top-down easier.