Dynamic Programming I

Summer 2018 • Lecture 07/26

A Few Notes

Midterm

Later today! Good luck!

Homework 4

Released after the midterm.

Course Overview

- Algorithmic Analysis
- Divide and Conquer
- Randomized Algorithms
- Tree Algorithms
- Graph Algorithms
- **Dynamic Programming**
- Greedy Algorithms
- Advanced Algorithms

Bellman-Ford

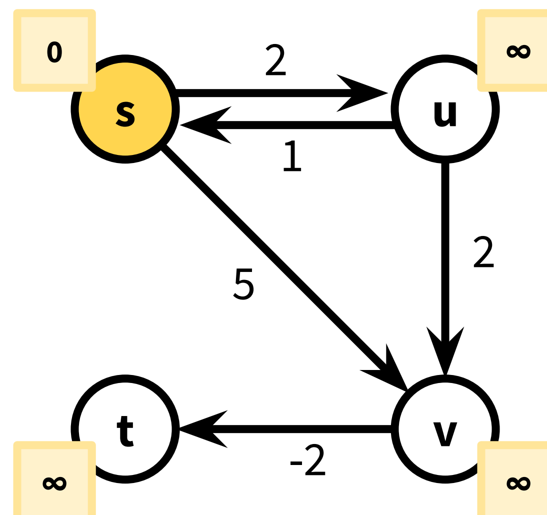
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

We know k = 0 i.e. shortest paths to each vertex with at most 0 edges in it.

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$				
$d^{(2)}$				
$d^{(3)}$				



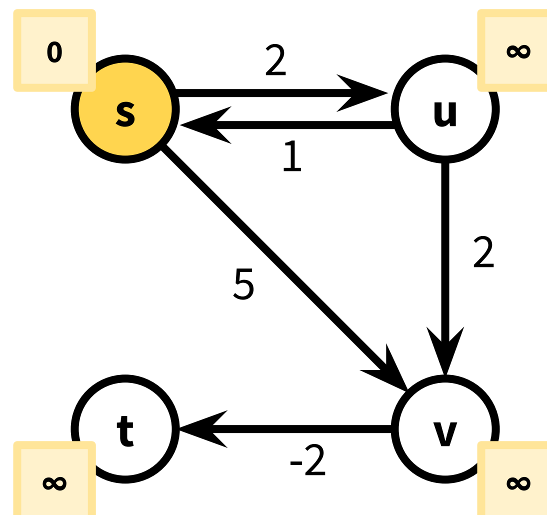
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

$d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$				
$d^{(2)}$				
$d^{(3)}$				

We will use table $d^{(0)}$ to fill in $d^{(1)}$. More generally, we will use table $d^{(k-1)}$ to fill in $d^{(k)}$.

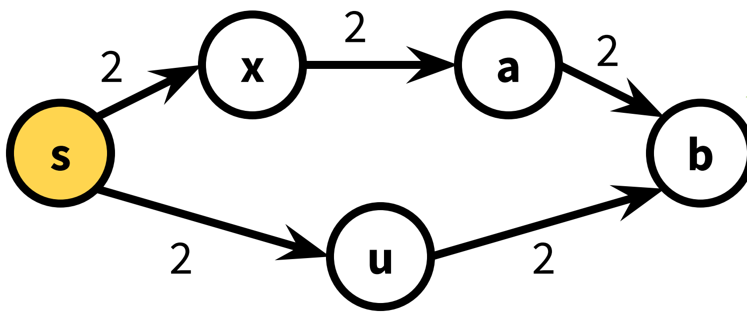


Bellman-Ford Algorithm

How do we use $d^{(k-1)}$ to fill in $d^{(k)}[b]$?

Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

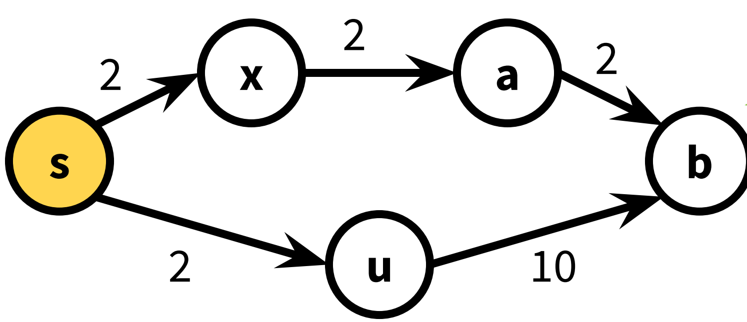
Case 1: the shortest path from s to b with at most k edges actually has at most k - 1 edges.



Suppose k = 3.

$d^{(k)}[b] = d^{(k-1)}[b]$ i.e. the shortest path of at most k - 1 edges is at least as short as any path of at most k edges.

Case 2: the shortest path from s to b with at most k edges really has k edges.



Suppose k = 3.

$d^{(k)}[b] = \min_a\{d^{(k-1)}[a] + w(a, b)\}$ i.e. the shortest path of at most k edges is shorter than any path of at most k - 1 edges.

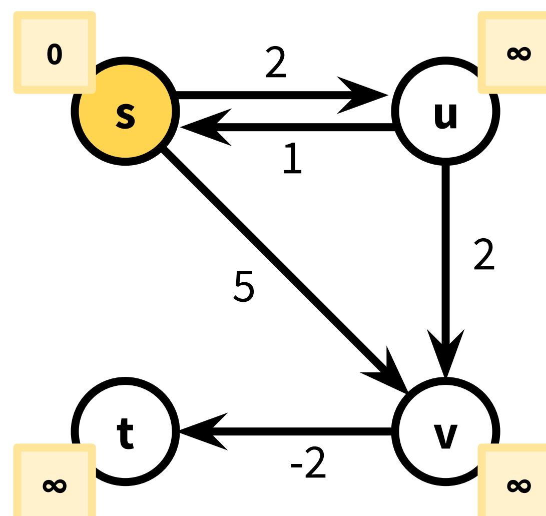
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

```
for k = 1 to |V|-1:
  for b in V:
    d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
```

	s	**u**	**v**	**t**
$\mathbf{d^{(0)}}$	**0**	**∞**	**∞**	**∞**
$\mathbf{d^{(1)}}$				
$\mathbf{d^{(2)}}$				
$\mathbf{d^{(3)}}$				

We will use table $d^{(0)}$ to fill in $d^{(1)}$. More generally, we will use table $d^{(k-1)}$ to fill in $d^{(k)}$.



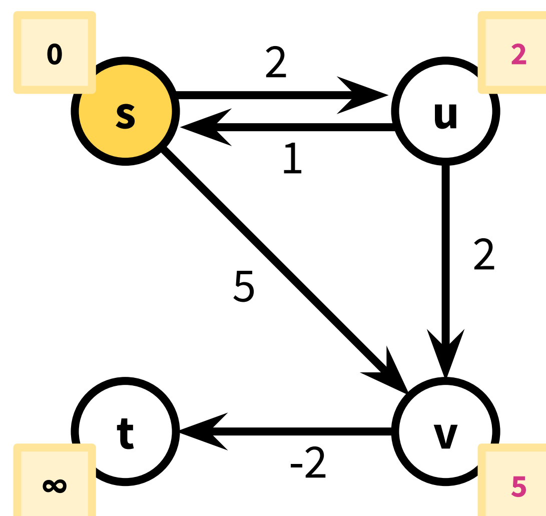
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

```
for k = 1 to |V|-1:
  for b in V:
    d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
```

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$				
$d^{(3)}$				

We will use table $d^{(0)}$ to fill in $d^{(1)}$. More generally, we will use table $d^{(k-1)}$ to fill in $d^{(k)}$.



Outline for Today

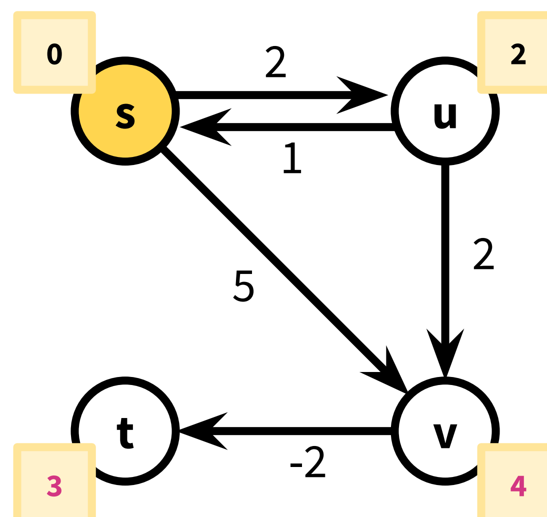
- Dynamic Programming
  - DP graph algorithms
    - Review Bellman Ford
    - Floyd Warshall

Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

```
for k = 1 to |V|-1:
  for b in V:
    d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
```

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$				

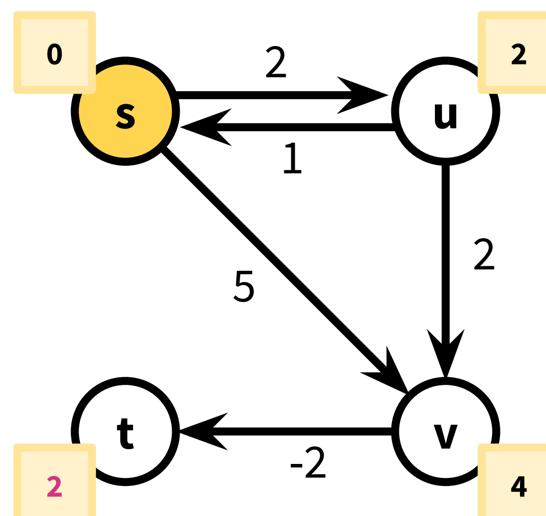


Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

```
for k = 1 to |V|-1:
  for b in V:
    d^(k)[b] = min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a,b)} }
```

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2

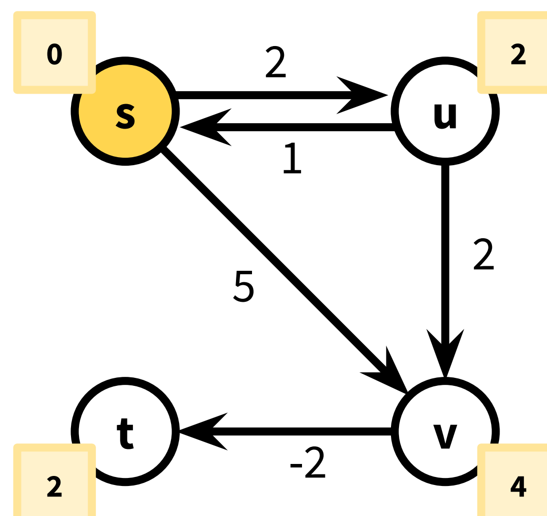


Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



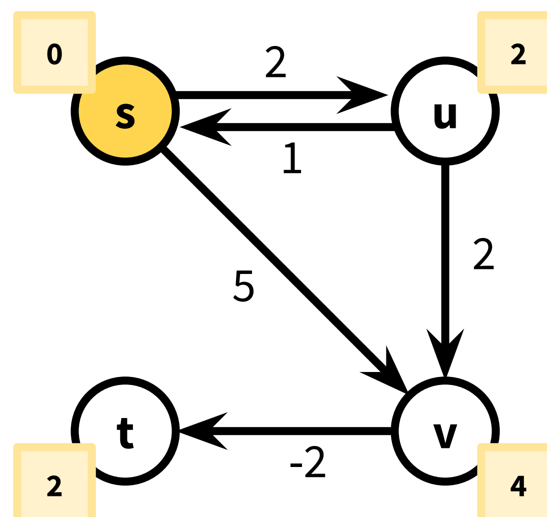
Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each $k = 0, 1, \ldots, |V|-1$.

Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

The shortest path from s to t with 1 edge has cost ∞ (no path exists).

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



Bellman-Ford Algorithm

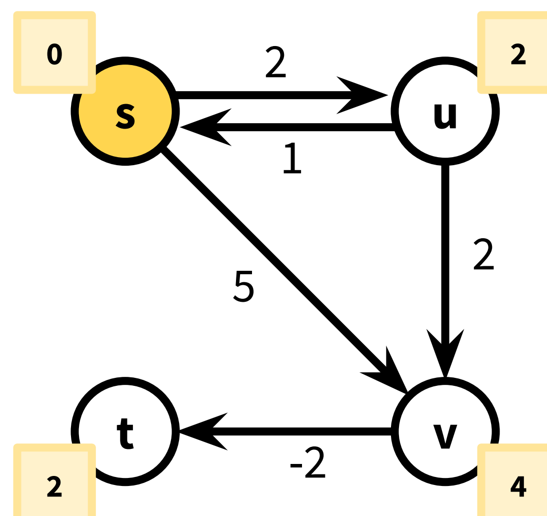
We maintain a list $d^{(k)}$ of length n for each $k = 0, 1, \ldots, |V|-1$.

Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

The shortest path from s to t with 1 edge has cost **∞** (no path exists).

The shortest path from s to t with 2 edges has cost **3** (s-v-t).

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



Bellman-Ford Algorithm

We maintain a list $d^{(k)}$ of length n for each k = 0, 1, …, |V|-1.

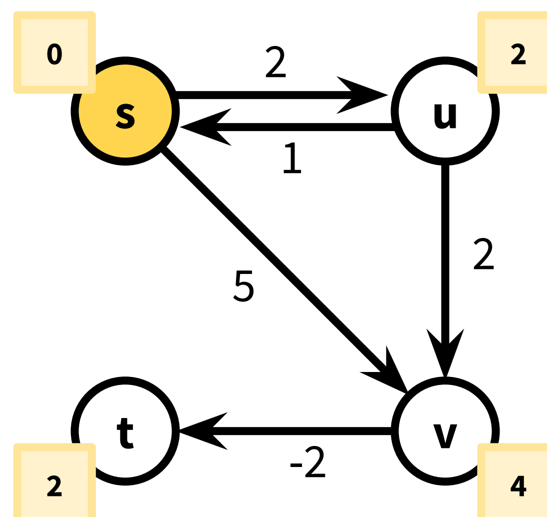
Recall $d^{(k)}[b]$ is the cost of the shortest path from s to b with at most k edges.

The shortest path from s to t with 1 edge has cost **∞** (no path exists).

The shortest path from s to t with 2 edges has cost **3** (s-v-t).

The shortest path from s to t with 3 edges has cost **2** (s-u-v-t).

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



Dynamic Programming

Bellman-Ford is an example of **dynamic programming**!

Dynamic programming is an algorithm design paradigm.

- Often it's used to solve optimization problems e.g. **shortest** path.

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

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.

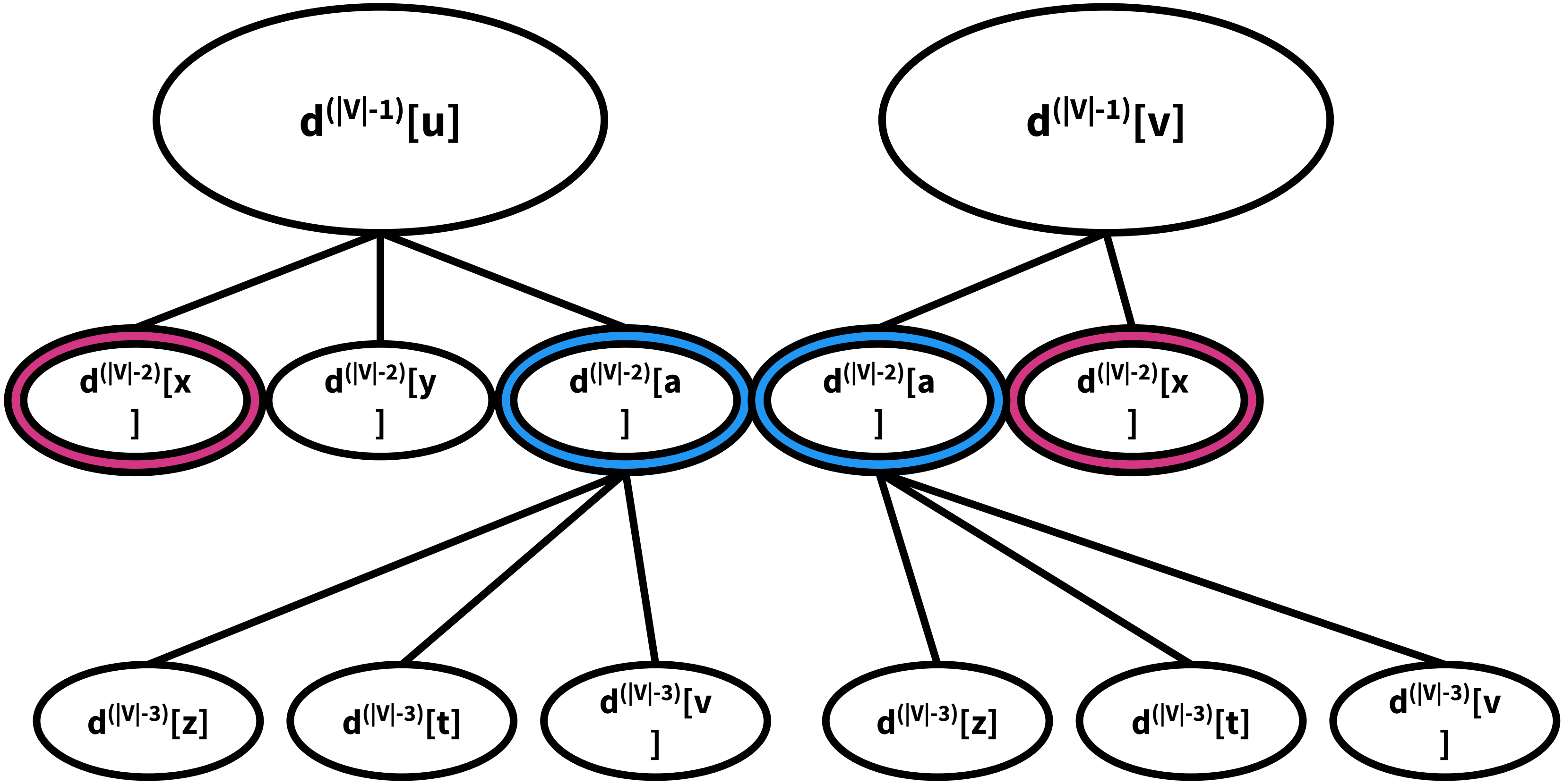
Top-Down BF Algorithm

```
def recursive_bellman_ford(G):
  d^(k) = [None] * |V| for k = 0 to |V|-1
  d^(0)[v] = ∞ for all v ≠ s
  d^(0)[s] = 0
  for b in V:
    recursive_bf_helper(G, b, |V|-1)

def recursive_bf_helper(G, b, k):
  A = {a such that (a, b) in E} ∪ {b}
  for a in A:
    if d^(k-1)[a] not None:
      d^(k-1)[a] = recursive_bf_helper(G, a, k-1)
  return min{d^(k-1)[b], min_a{d^(k-1)[a] + w(a, b)} }
```

Runtime: $O(|V||E|)$

Visualization of Top-Down



Floyd-Warshall

Floyd-Warshall Algorithm

Another example of a graph DP algorithm!

The algorithm solves the all-pairs shortest path (**APSP**) problem.

A naive solution

```
for s in V:
  run bellman_ford starting at s
```

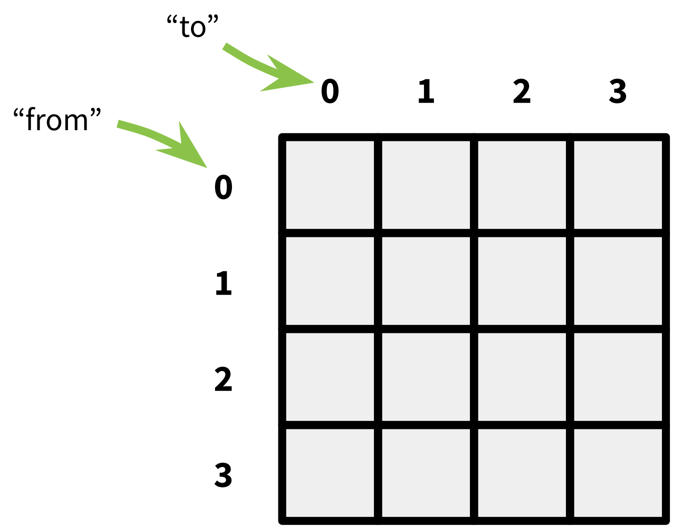
Runtime $O(|V|^2|E|)$

Can we do better?

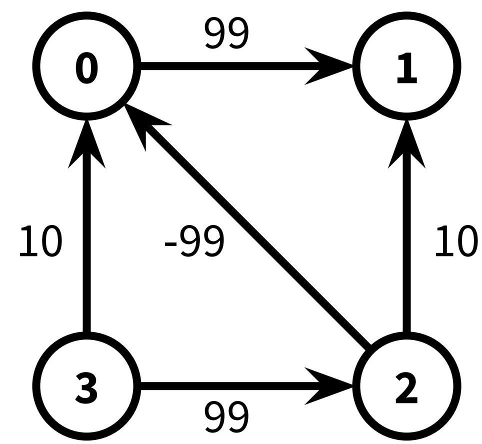
Floyd-Warshall Algorithm

We maintain an $|V| \times |V|$ matrix $D^{(k)}$ for each $k = 0, 1, \ldots, |V|$.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, k\text{-}1\}$.



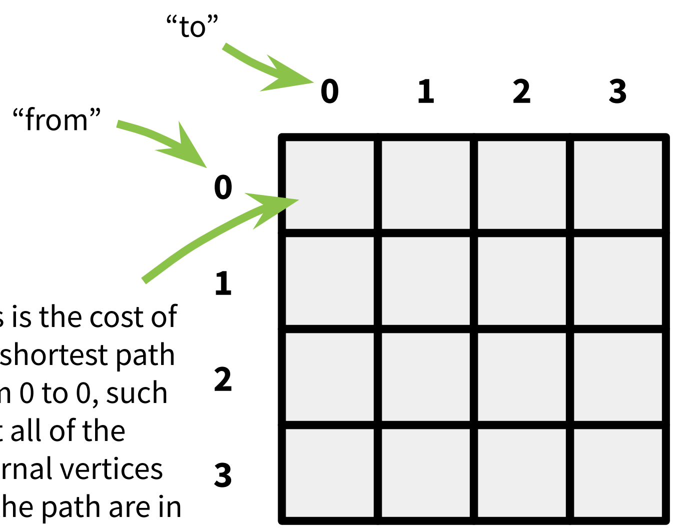
$D^{(0)}[u, v]$



Floyd-Warshall Algorithm

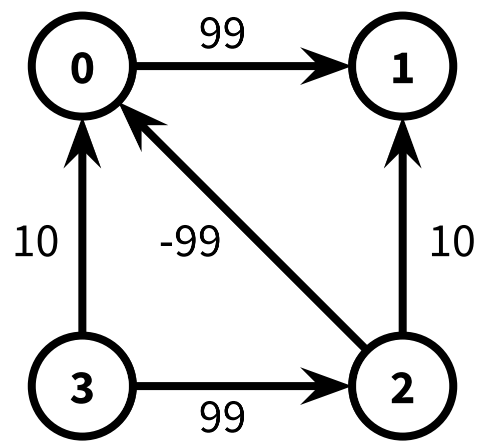
We maintain an |V|×|V| matrix $D^{(k)}$ for each k = 0, 1, …, |V|.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.



This is the cost of the shortest path from 0 to 0, such that all of the internal vertices on the path are in the set of vertices {0, …, -1} i.e. the cost of the shortest path from 0 to 0 that passes through no other vertices.

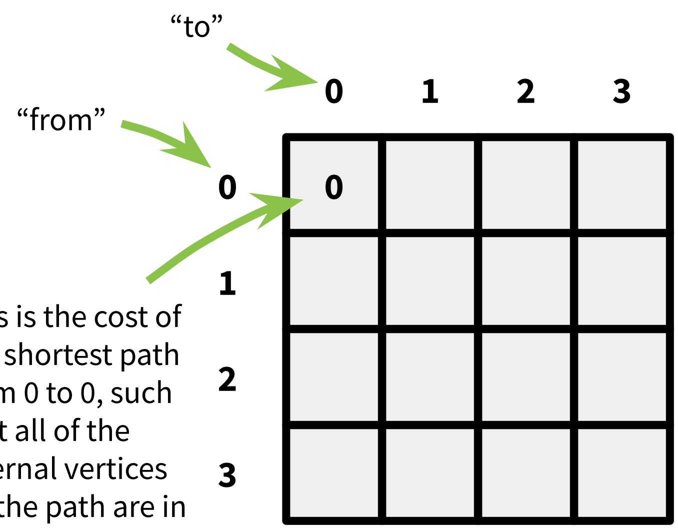
$D^{(0)}[u, v]$



Floyd-Warshall Algorithm

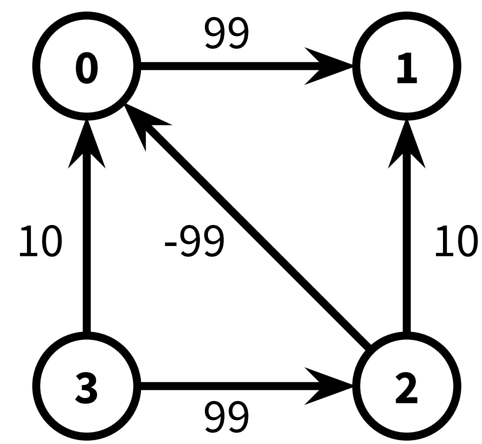
We maintain an $|V| \times |V|$ matrix $D^{(k)}$ for each $k = 0, 1, \ldots, |V|$.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, k-1\}$.



This is the cost of the shortest path from 0 to 0, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, -1\}$ i.e. the cost of the shortest path from 0 to 0 that passes through no other vertices.

$D^{(0)}[u, v]$



Floyd-Warshall Algorithm

We maintain an $|V| \times |V|$ matrix $D^{(k)}$ for each $k = 0, 1, \ldots, |V|$.

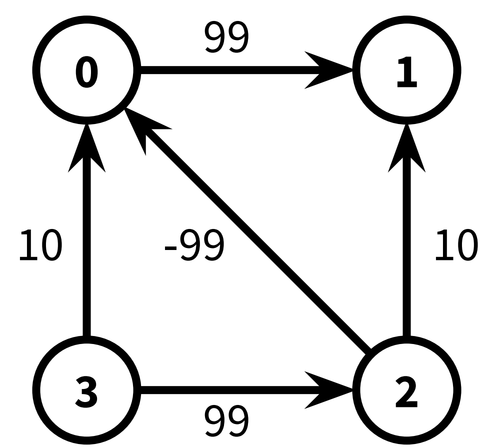
$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, k\text{-}1\}$.

“to”

“from”

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

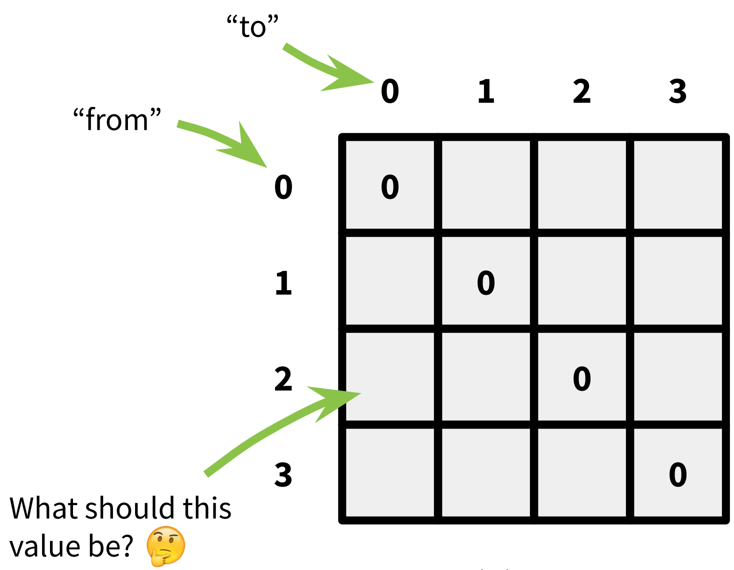
$\mathbf{D^{(0)}[u, v]}$



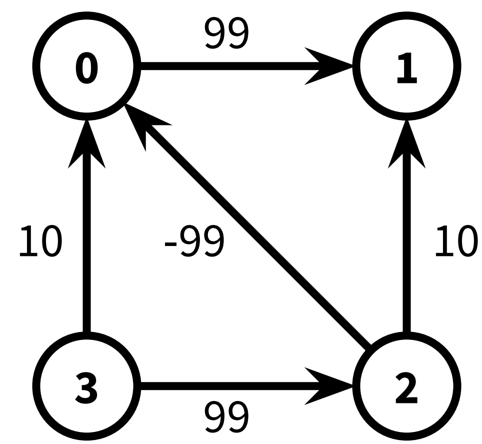
Floyd-Warshall Algorithm

We maintain an $|V| \times |V|$ matrix $D^{(k)}$ for each $k = 0, 1, \ldots, |V|$.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, k\text{-}1\}$.



$D^{(0)}[u, v]$



Floyd-Warshall Algorithm

We maintain an $|V| \times |V|$ matrix $D^{(k)}$ for each $k = 0, 1, \ldots, |V|$.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices $\{0, \ldots, k\text{-}1\}$.

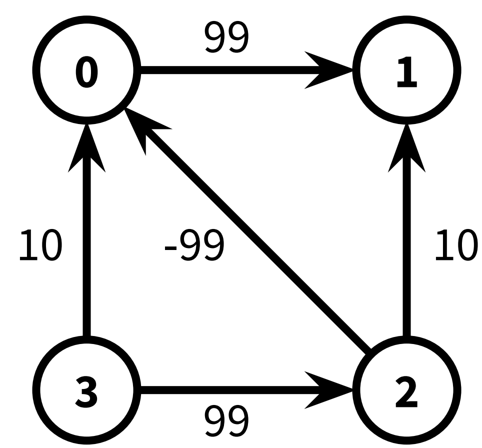
“to”

“from”

	0	**1**	**2**	**3**
0	**0**			
1		**0**		
2	**-99**		**0**	
3				**0**

$D^{(0)}[u, v]$

What should this value be? 🤔 -99, since the shortest path from 2 to 0, passing through no other vertices has weight -99.



Floyd-Warshall Algorithm

“to”

“from”

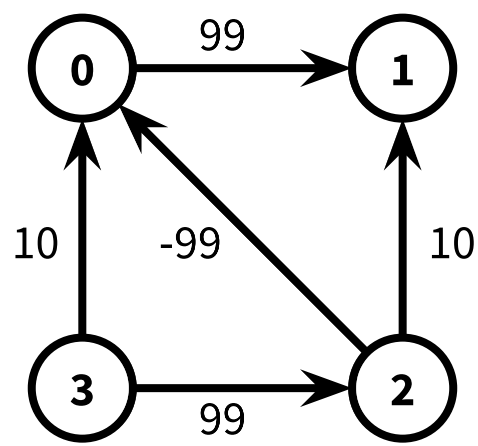
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

	0	1	2	3
0				
1				
2				
3				

$D^{(1)}[u, v]$

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.



Since k = 1, shortest paths are allowed to pass through vertices {0} now. So the we can compare the current cost to the cost of path 3-0-1. $D^{(0)}$ tells us the cost of 3-0 is **10** and the cost of 0-1 is **99**.

Floyd-Warshall Algorithm

“to”

“from”

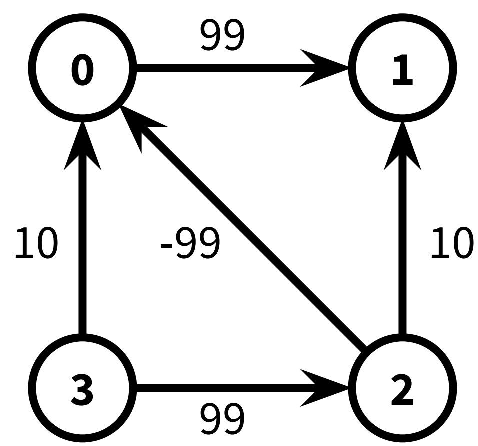
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

	0	1	2	3
0				
1				
2				
3		109		

$D^{(1)}[u, v]$

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.



Since k = 1, shortest paths are allowed to pass through vertices {0} now. So \ we can compare the current cost to the cost of path 3-0-1. $D^{(0)}$ tells us the cost of 3-0 is **10** and the cost of 0-1 is **99**. Since the sum of these values is less than **∞**, replace it.

Floyd-Warshall Algorithm

“to”

“from”

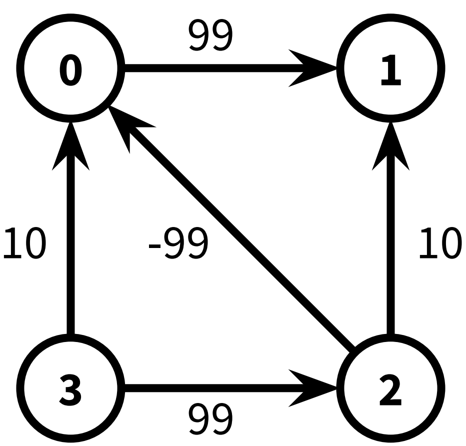
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

	0	1	2	3
0				
1				
2		0		
3		109		

$D^{(1)}[u, v]$

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.



Floyd-Warshall Algorithm

“to”

“from”

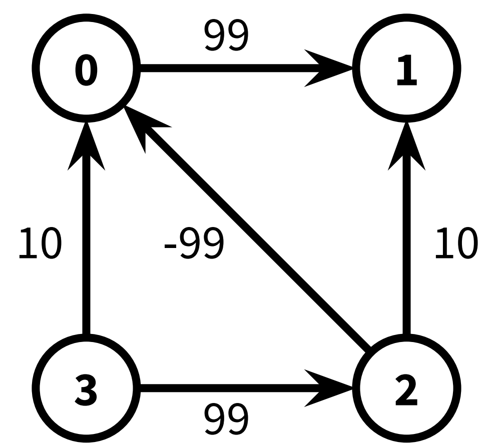
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

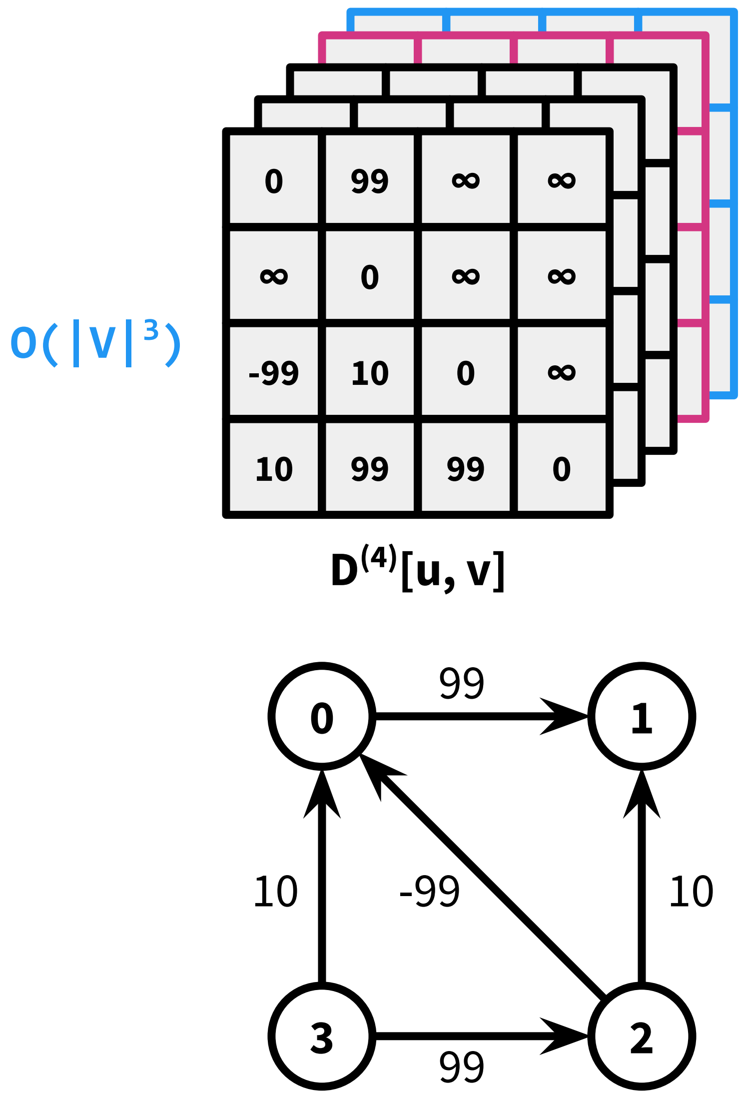
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	0	0	∞
3	10	109	99	0

$D^{(1)}[u, v]$

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.



Floyd–Warshall Algorithm

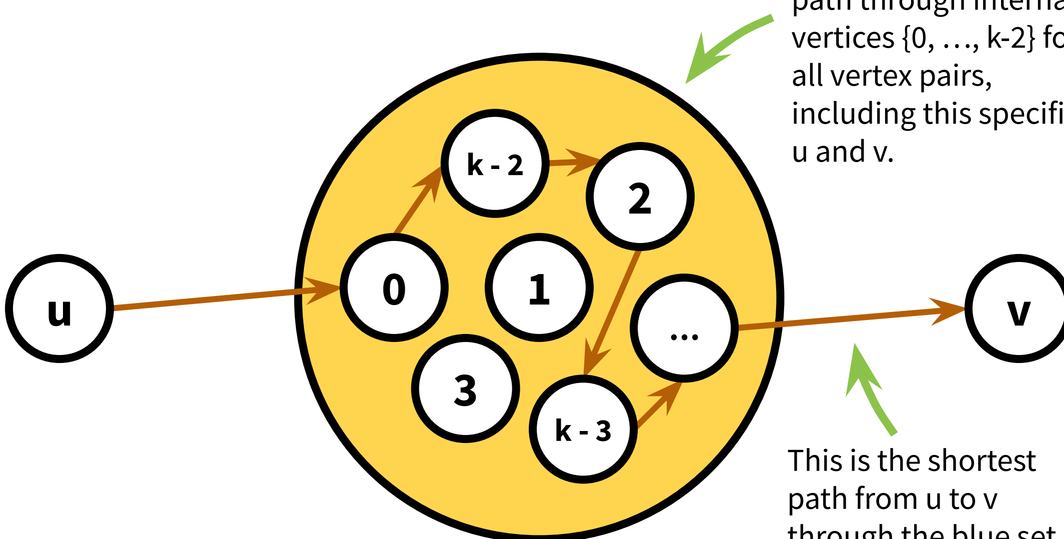


Floyd-Warshall Algorithm

We can represent it more graphically.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?



$D^{(k-1)}$ describes the cost of the shortest path through internal vertices {0, …, k-2} for all vertex pairs, including this specific u and v.

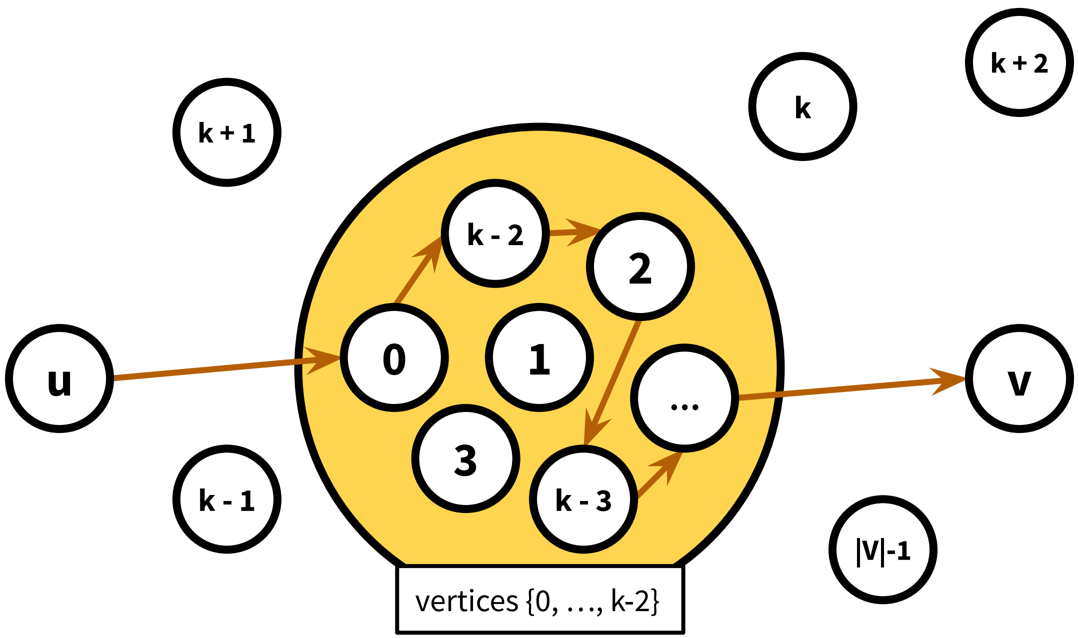
This is the shortest path from u to v through the blue set of vertices; it has weight $D^{(k-1)}[u, v]$.

Floyd-Warshall Algorithm

We can represent it more graphically.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

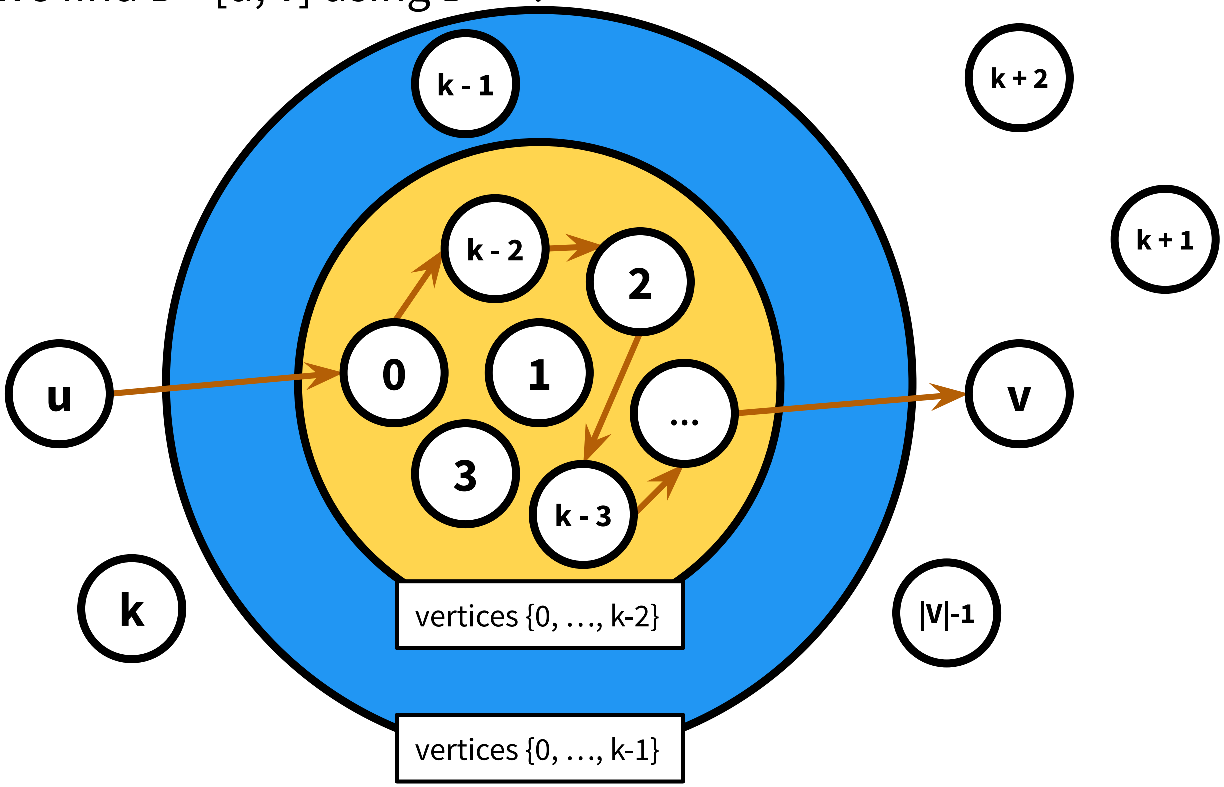


Floyd-Warshall Algorithm

We can represent it more graphically.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?



Floyd-Warshall Algorithm

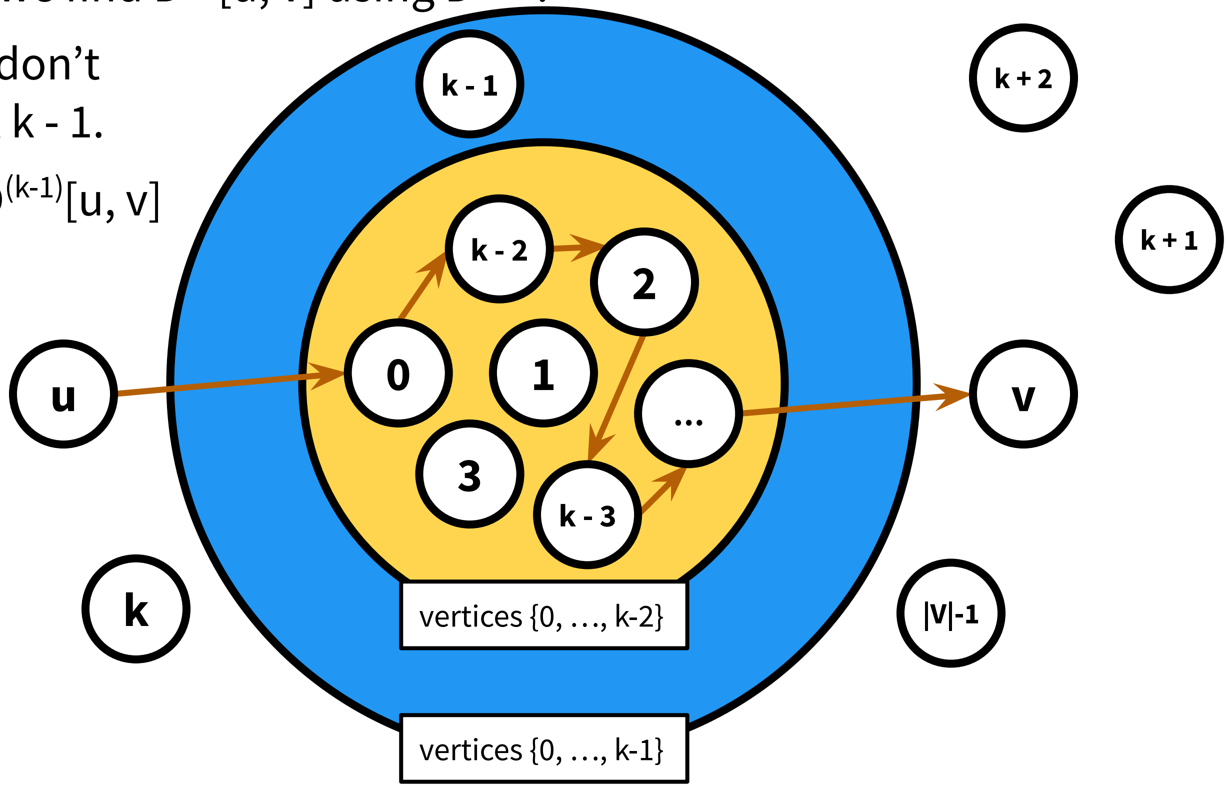
We can represent it more graphically.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

Case 1: we don't need vertex k - 1.

$D^{(k)}[u, v] = D^{(k-1)}[u, v]$



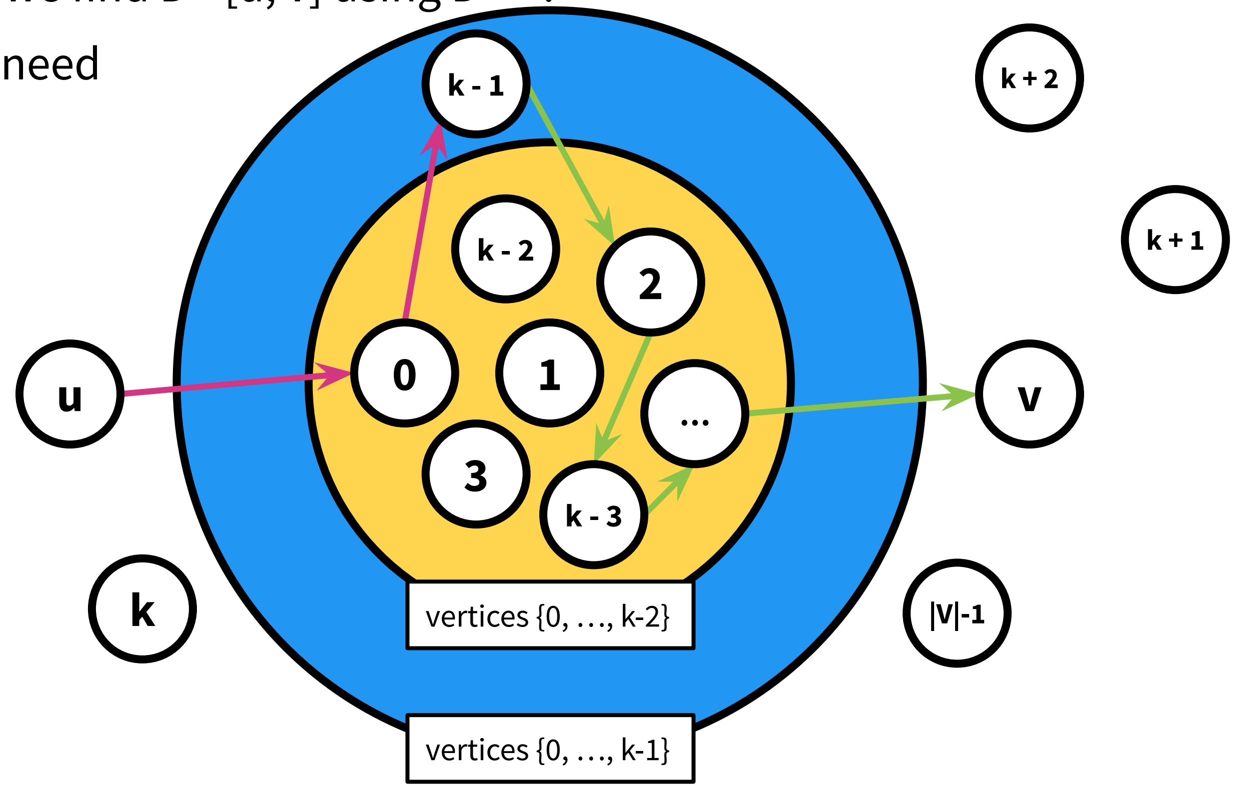
Floyd-Warshall Algorithm

We can represent it more graphically.

$D^{(k)}[u, v]$ is the cost of the shortest path from u to v, such that all of the internal vertices on the path are in the set of vertices {0, …, k-1}.

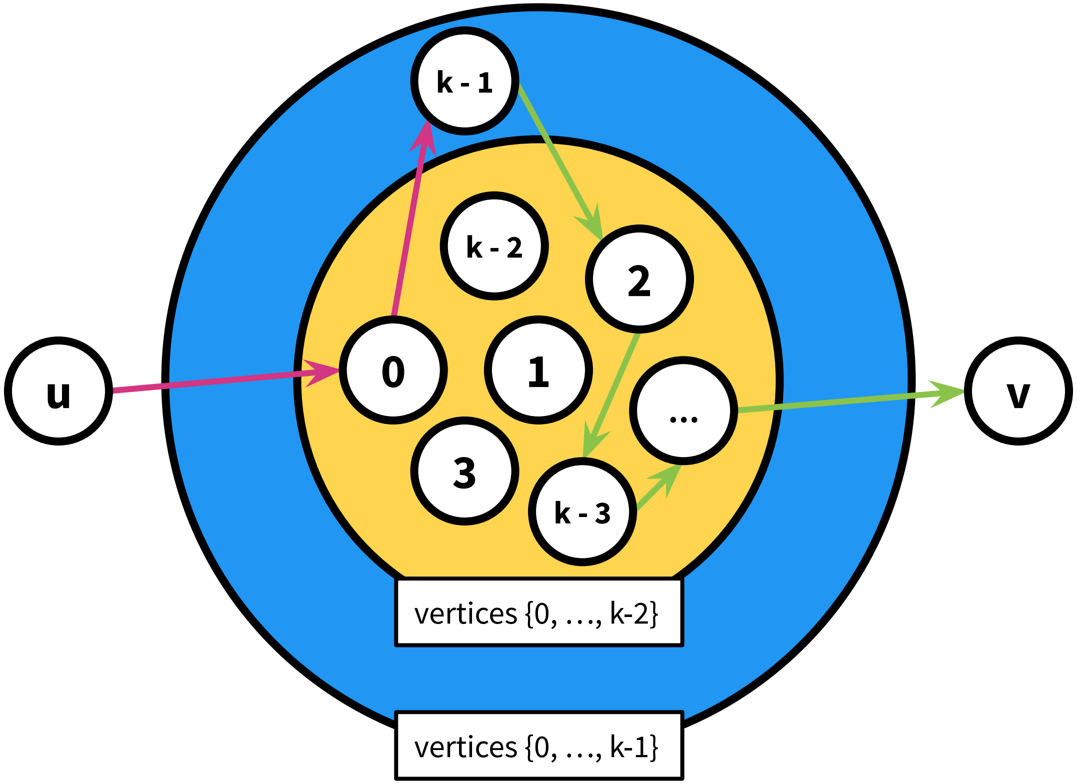
How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

Case 2: we need vertex k - 1.



Floyd-Warshall Algorithm

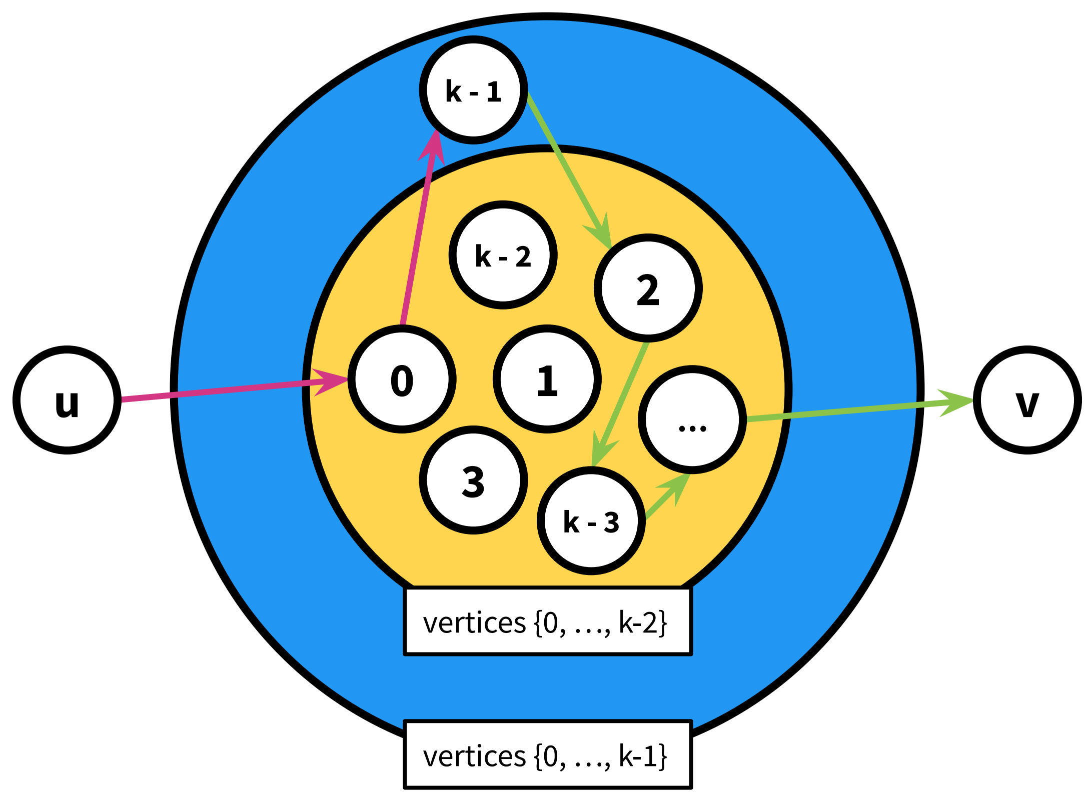
Case 2, cont.: we need vertex k - 1.



Floyd-Warshall Algorithm

Case 2, cont.: we need vertex k - 1.

If there are no negative cycles, then the shortest path from u to v through {0, …, k-1} is simple.

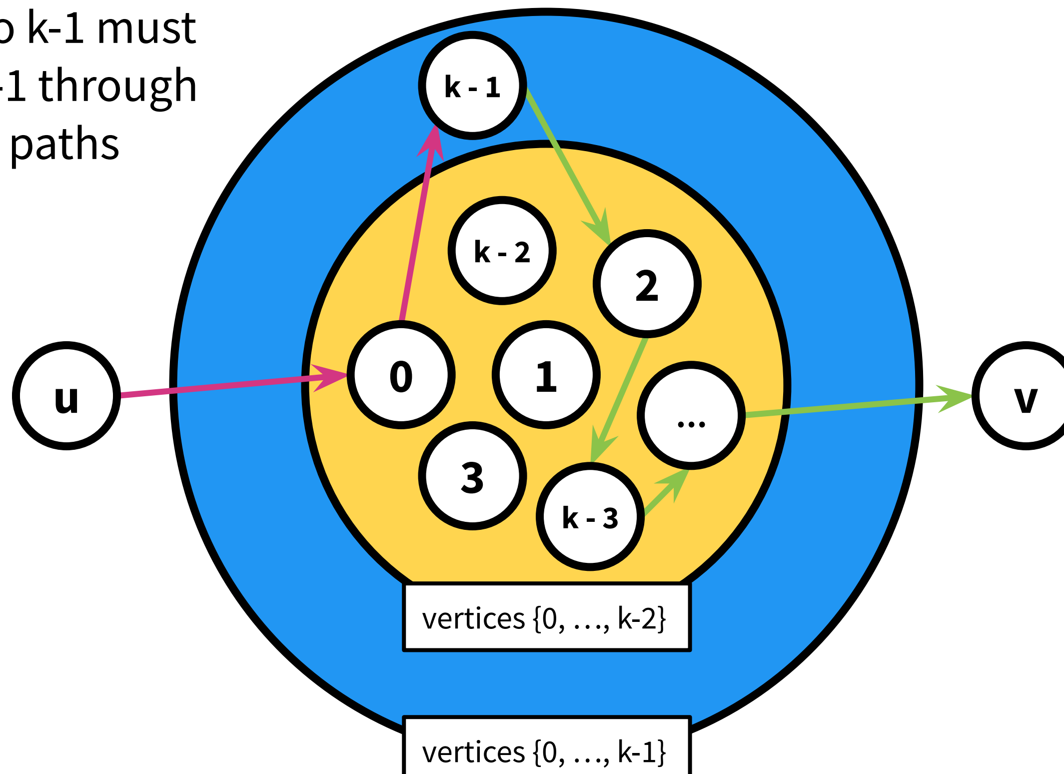


Floyd-Warshall Algorithm

Case 2, cont.: we need vertex k - 1.

If there are no negative cycles, then the shortest path from u to v through {0, …, k-1} is simple.

If the shortest path from u to v needs vertex k - 1, then **the subpath** from u to k-1 must be the shortest path from u to k-1 through {0, …, k-2} (subpaths of shortest paths are shortest paths).



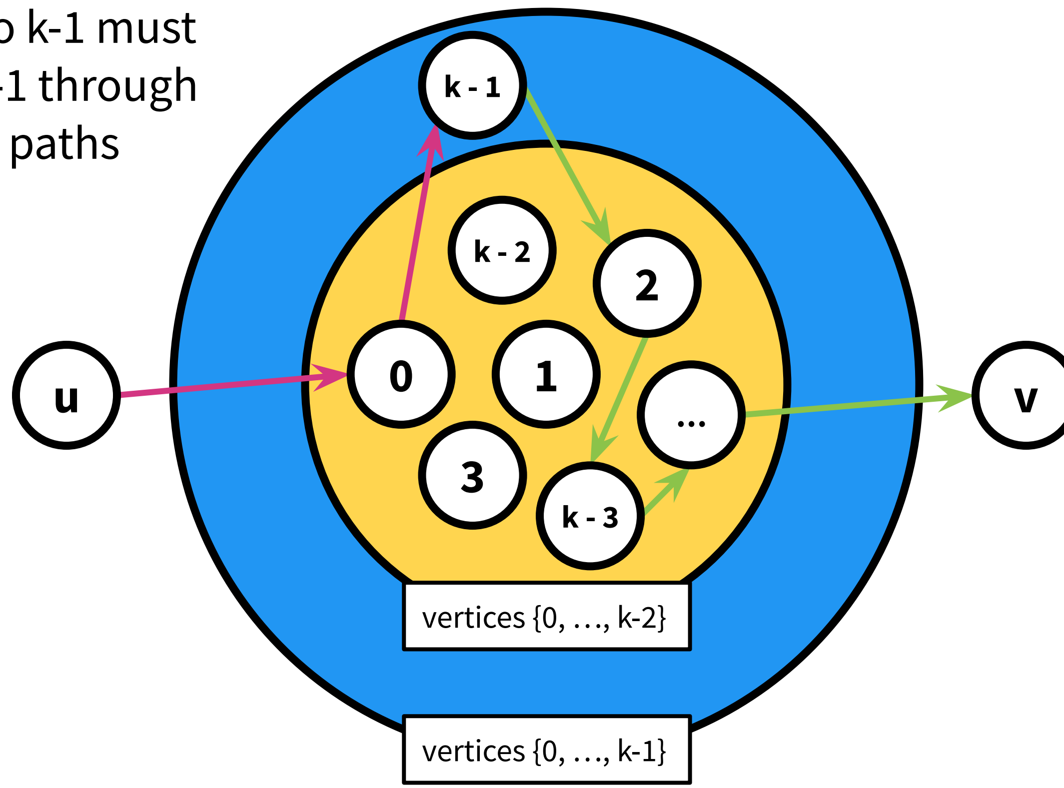
Floyd-Warshall Algorithm

Case 2, cont.: we need vertex k - 1.

If there are no negative cycles, then the shortest path from u to v through {0, …, k-1} is simple.

If the shortest path from u to v needs vertex k - 1, then **the subpath** from u to k-1 must be the shortest path from u to k-1 through {0, …, k-2} (subpaths of shortest paths are shortest paths).

This looks like our inductive hypothesis :)



Floyd-Warshall Algorithm

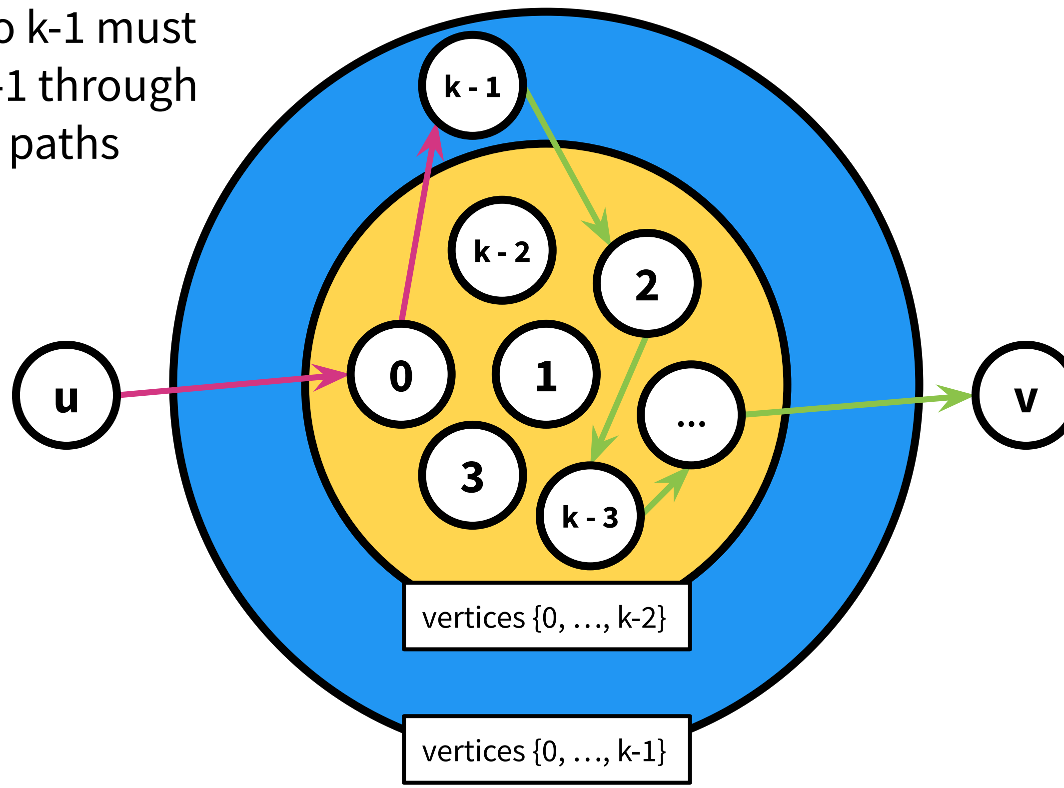
Case 2, cont.: we need vertex k - 1.

If there are no negative cycles, then the shortest path from u to v through {0, …, k-1} is simple.

This looks like our inductive hypothesis :)

If the shortest path from u to v needs vertex k - 1, then **the subpath** from u to k-1 must be the shortest path from u to k-1 through {0, …, k-2} (subpaths of shortest paths are shortest paths).

Same for **the path** from k-1 to v.



Floyd-Warshall Algorithm

Case 2, cont.: we need vertex k - 1.

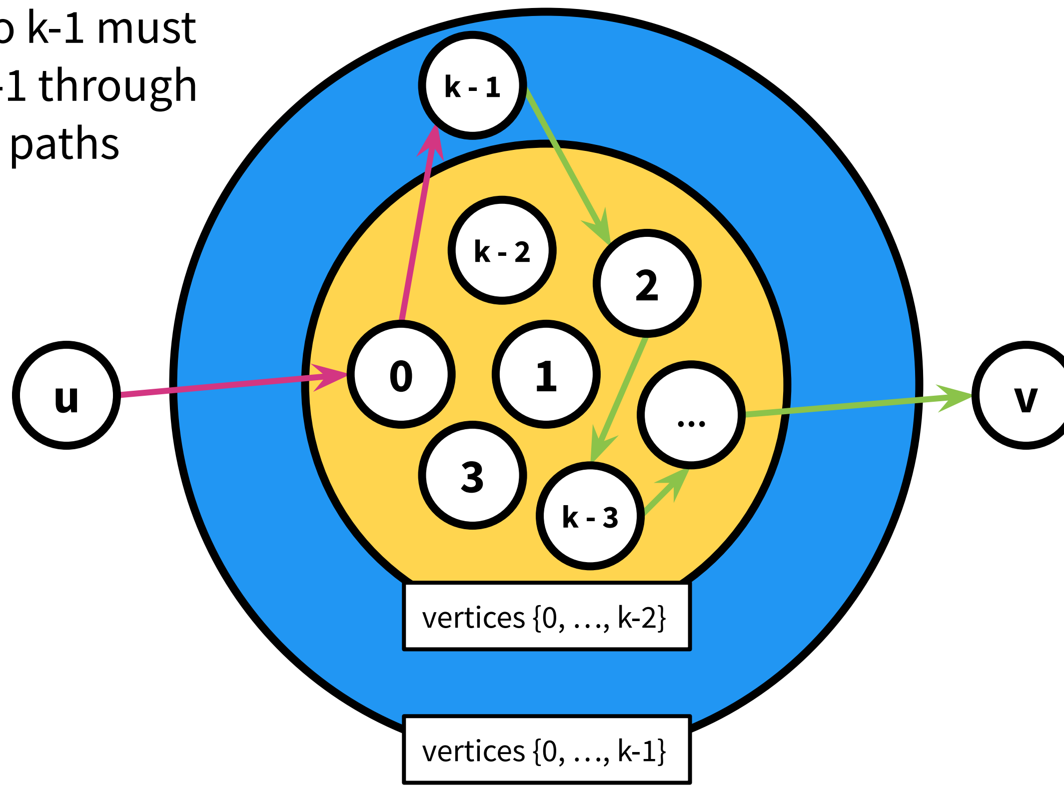
If there are no negative cycles, then the shortest path from u to v through {0, …, k-1} is simple.

This looks like our inductive hypothesis :)

If the shortest path from u to v needs vertex k - 1, then **the subpath** from u to k-1 must be the shortest path from u to k-1 through {0, …, k-2} (subpaths of shortest paths are shortest paths).

Same for **the path** from k-1 to v.

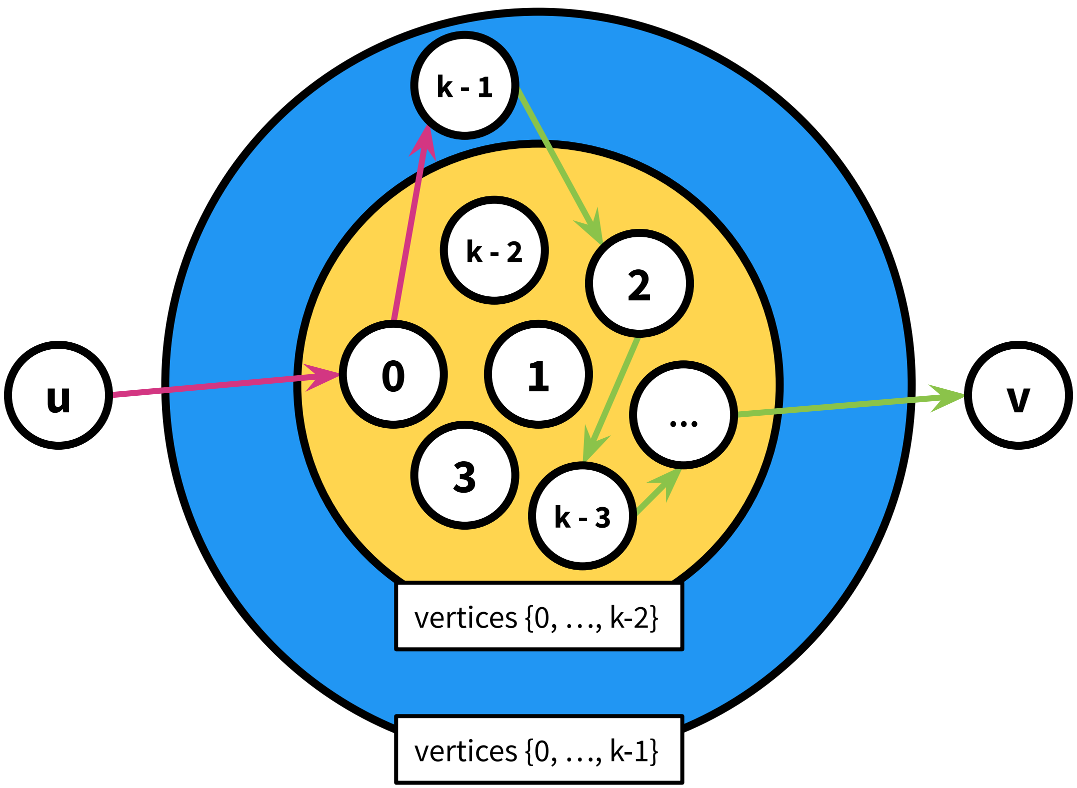
$D^{(k)}[u, v] =$
$D^{(k-1)}[u, k-1] + D^{(k-1)}[k-1, v]$



Floyd-Warshall Algorithm

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

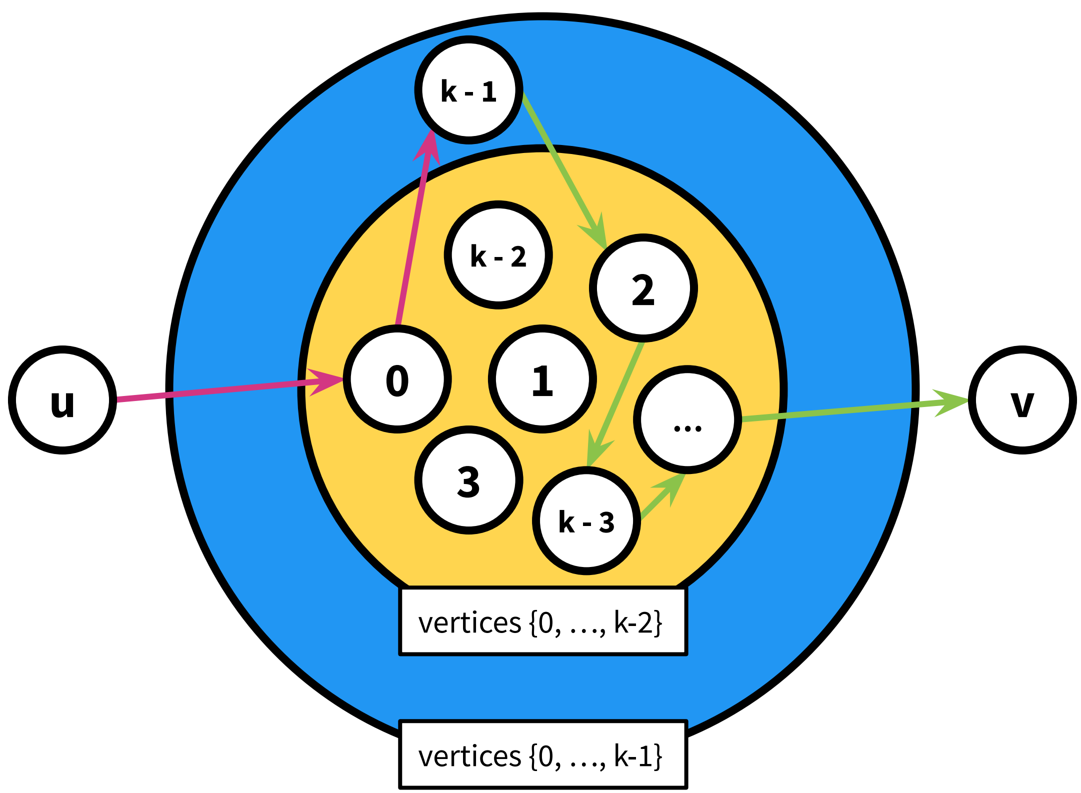
$D^{(k)}[u, v] = \min\{ \qquad , \qquad \}$



Floyd-Warshall Algorithm

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

$D^{(k)}[u, v] = \min\{\underbrace{D^{(k-1)}[u, v]}_{\text{Case 1}},\qquad\qquad\qquad\qquad\}$

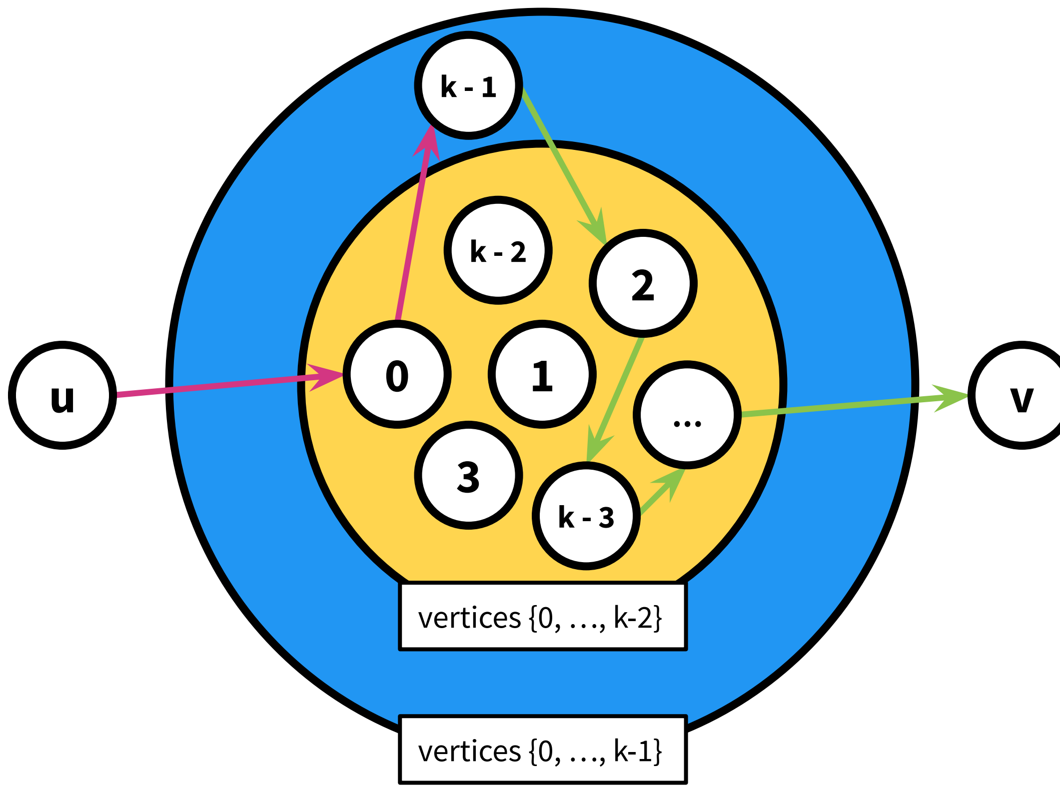


Floyd-Warshall Algorithm

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

$D^{(k)}[u, v] = \min\{D^{(k-1)}[u, v], D^{(k-1)}[u, k-1] + D^{(k-1)}[k-1, v]\}$

Case 1 **Case 2**



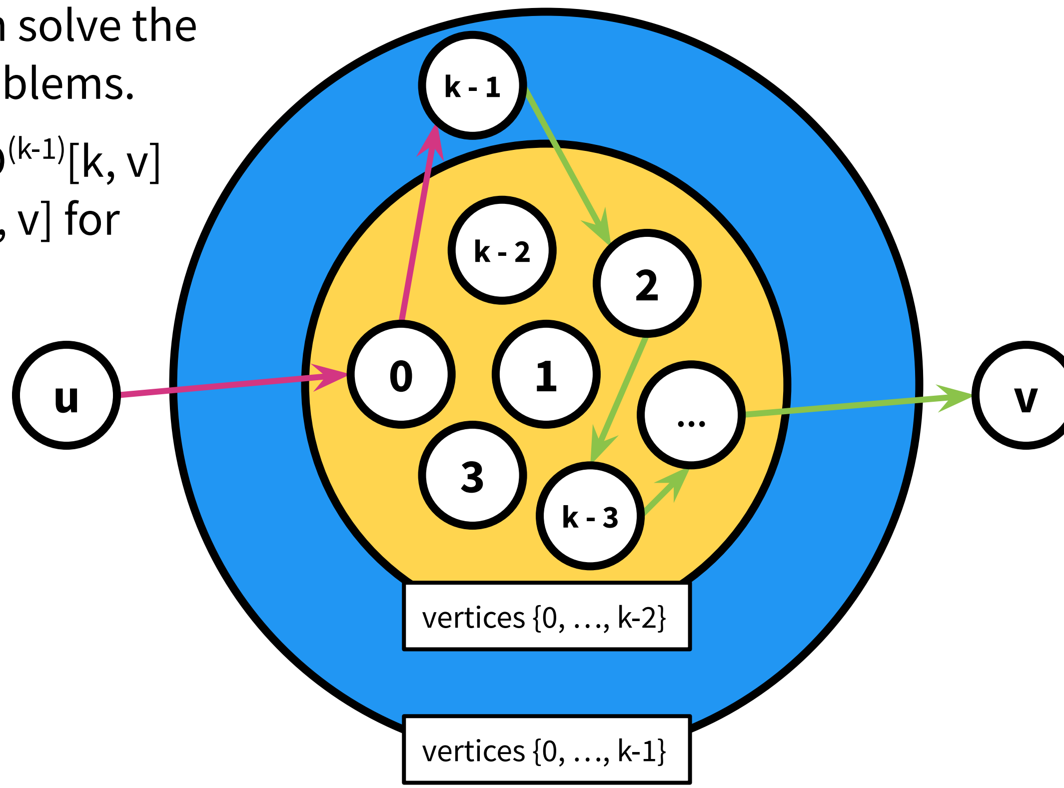
Floyd-Warshall Algorithm

How might we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

$$D^{(k)}[u, v] = \min\{\underbrace{D^{(k-1)}[u, v]}_{\text{Case 1}}, \underbrace{D^{(k-1)}[u, k-1] + D^{(k-1)}[k-1, v]}_{\text{Case 2}}\}$$

Optimal substructure We can solve the big problem using smaller problems.

Overlapping sub-problems $D^{(k-1)}[k, v]$ can be used to compute $D^{(k)}[u, v]$ for lots of different u's.



Floyd-Warshall Algorithm

Floyd-Warshall can detect negative cycles.

- If there's a negative cycle, then there's a path from v to v that has cost < 0.
- How do we check for this condition? 🤔

Floyd-Warshall Algorithm

Floyd-Warshall can detect negative cycles.

- If there's a negative cycle, then there's a path from v to v that has cost < 0.
- How do we check for this condition? 🤔 We can just check $D^{(|V|)}[v, v] < 0$ at the end of the algorithm.

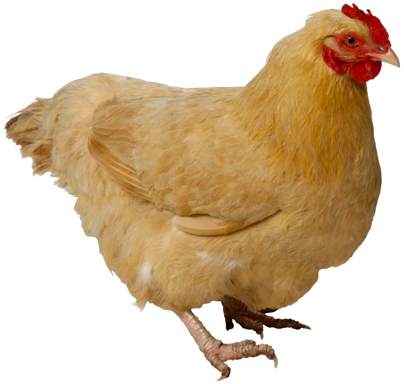
Graph Algorithms

	Dijkstra	Bellman-Ford	Floyd-Warshall
Problem	Single source shortest path	Single source shortest path	All pairs shortest path
Runtime	$O(\|E\|+\|V\|\log(\|V\|))$ **worst-case** with a fibonacci heap	$O(\|V\|\|E\|)$ **worst-case**	$O(\|V\|^3)$ **worst case**
Strengths	---	Works on graphs with negative edge-weights; also can detect negative cycles	Works on graphs with negative edge-weights; also can detect negative cycles
Weaknesses	Might not work on graphs with negative edge-weights	---	---

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.

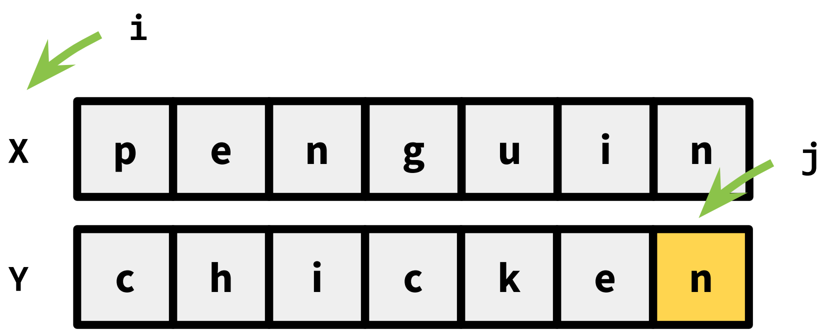
LCS

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

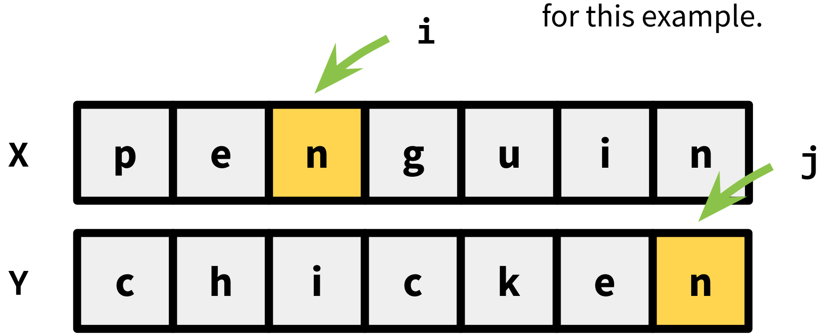
Task Find the LCS of two strings.

(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

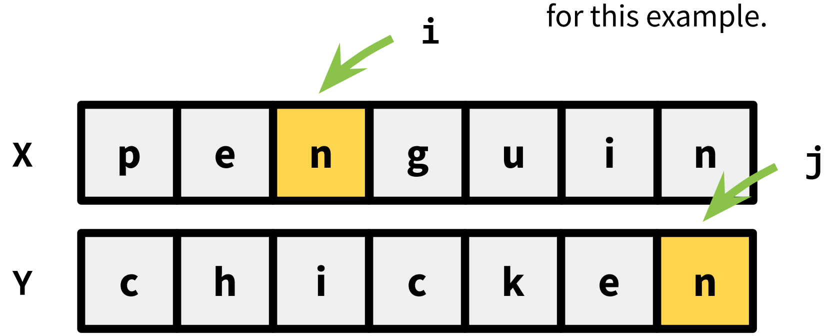
Task Find the LCS of two strings.

(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 `LCS(X[0:i+1], Y[0:j+1]) =`

LCS

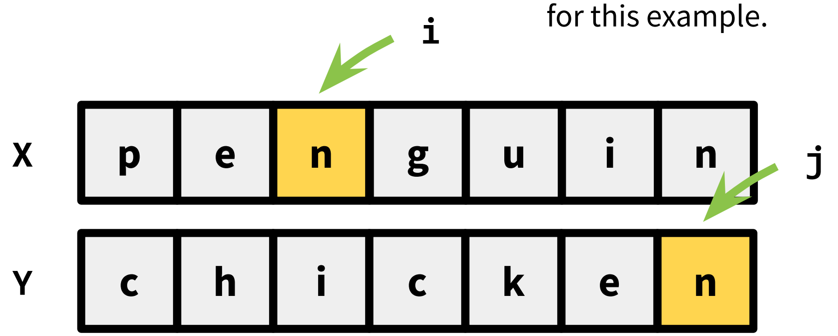
Task Find the LCS of two strings.

(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 `LCS(X[0:i+1], Y[0:j+1]) = LCS(X[0:i], Y[0:j])` 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.

i

X	p	e	**n**	g	u	i	n
Y	c	h	i	c	k	e	**n**

j

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

since `LCS(X[0:i+1], Y[0:j+1]) = LCS(X[0:i], Y[0:j])` followed by **n**

For this entire lecture, index ranges will be inclusive.

LCS

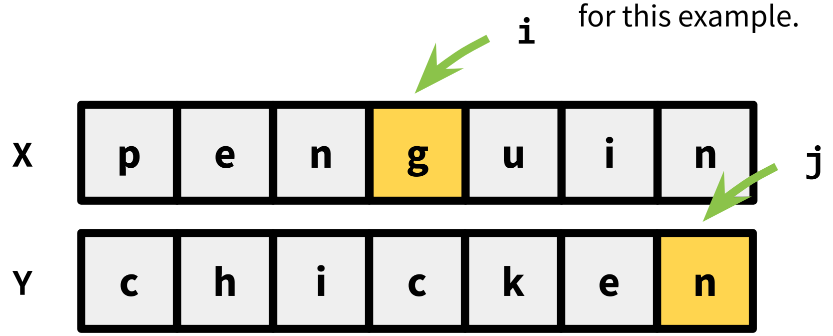
Task Find the LCS of two strings.

(2) Define a recursive formulation.

Consider two cases on the strings X and Y.

Case 2 `X[i] != 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

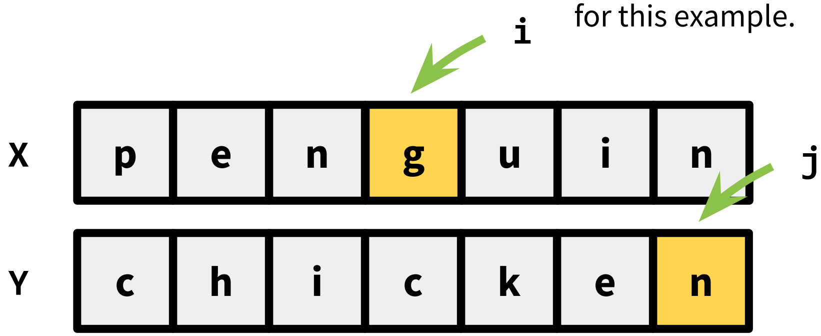
Task Find the LCS of two strings.

(2) Define a recursive formulation.

Consider two cases on the strings X and Y.

Case 2 `X[i] != 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+1], Y[0:j+1]) = LCS(X[0:i], Y[0:j+1]);` $\boxed{g}$ isn’t involved or

LCS

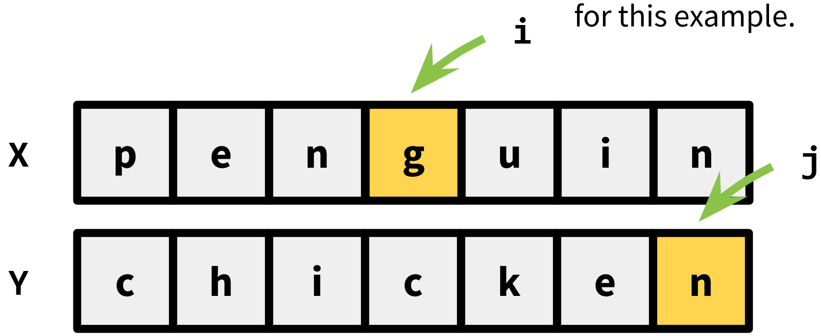
Task Find the LCS of two strings.

(2) Define a recursive formulation.

Consider two cases on the strings X and Y.

Case 2 `X[i] != 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+1], Y[0:j+1]) = LCS(X[0:i], Y[0:j+1]);` $\boxed{g}$ isn't involved or

`LCS(X[0:i+1], Y[0:j+1]) = LCS(X[0:i+1], Y[0:j]);` $\boxed{n}$ isn't 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.

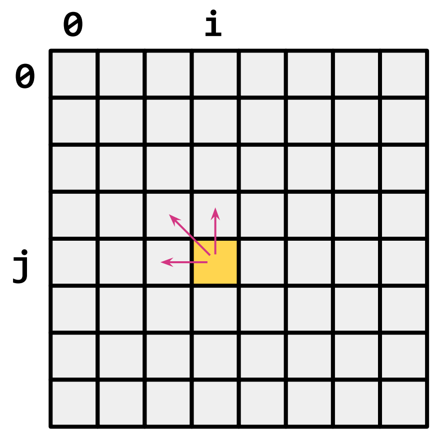
(4) If necessary, track additional information so that the algorithm from (3) can solve a related problem.

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)?



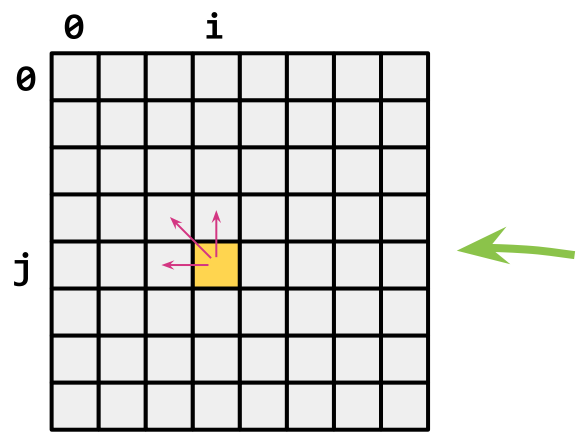
$$
T(i, j) = \begin{cases} 0 & \text{if } i \text{ or } j \text{ is -1} \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } i, j \geq 0 \\ \max\{T(i-1, j), \\ \quad T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } i, j \geq 0 \end{cases}
$$

LCS

Task Find the LCS of two strings.

(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).

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

LCS

```
def 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")`**.

		A	C	T	G
A					
C					
G					
G					
A					

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

LCS

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

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

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

LCS

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

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

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

LCS

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

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

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")`.

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

LCS

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

LCS

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

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

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

LCS

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

LCS

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

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

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

LCS G

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

LCS

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

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

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")`.

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

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

LCS G

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

LCS

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

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

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

LCS C G

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

LCS

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

		A	C	T	G
	0	0	0	0	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**

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

LCS

```
def 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!).