Lecture 12

More Bellman-Ford, Floyd-Warshall,

and Dynamic Programming!

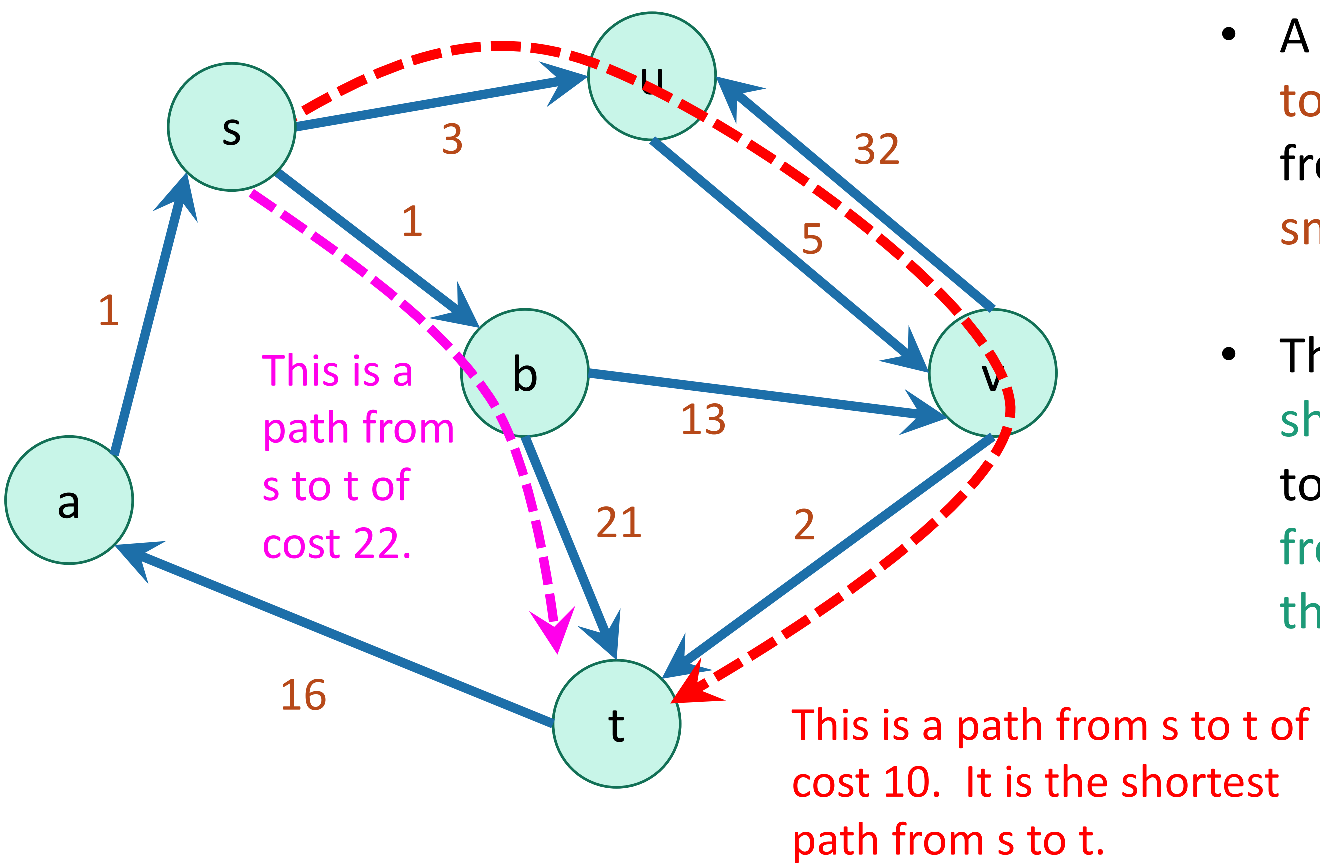
Announcements

- HW5 due Friday
- Midterms have been graded!
  - Available on Gradescope.
  - Mean/Median: 66 (it was a hard test!)
  - Max: 97
  - Std. Dev: 14

- Please look at the solutions and come to office hours if you have questions about your midterm!

Recall

- A weighted directed graph:

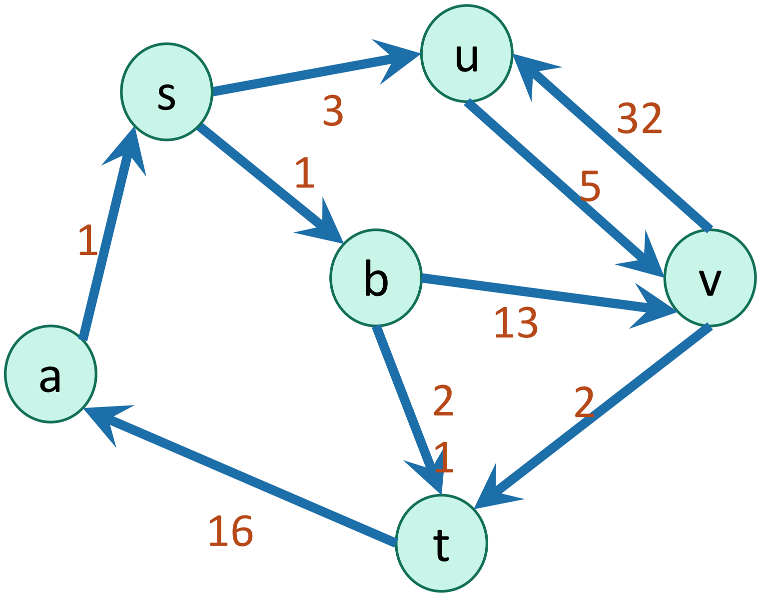


- Weights on edges represent costs.
- The cost of a path is the sum of the weights along that path.
- A shortest path from s to t is a directed path from s to t with the smallest cost.
- The single-source shortest path problem is to find the shortest path from s to v for all v in the graph.

Last time

- Dijkstra's algorithm!
- Bellman-Ford algorithm!
  - Both solve single-source shortest path in weighted graphs.

We didn't quite finish with the Bellman-Ford algorithm so let's do that now.



Bellman-Ford vs. Dijkstra

Bellman-Ford(G,s):

- d[v] = ∞ for all v in V
- d[s] = 0
- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.outNeighbors:
      - d[v] ← min(d[v] , d[u] + w(u,v))

Instead of picking u cleverly, just update for all of the u's.

Dijkstra(G,s):

- **While** there are **not-sure** nodes:
  - Pick the **not-sure** node u with the smallest estimate **d[u].**
  - **For** v in u.outNeighbors:
    - d[v] ← min(d[v] , d[u] + w(u,v))
  - Mark u as **sure**.

For pedagogical reasons

which we will see later today…

- We are actually going to change this to be dumber.
- Keep n arrays: $d^{(0)}$, $d^{(1)}$, …, $d^{(n-1)}$

Bellman-Ford*(G,s):

- $d^{(0)}[v] = \infty$ for all v in V
- $d^{(0)}[s] = 0$
- **For** i=0,…,n-2:
  - **For** u in V:
    - **For** v in u.outNeighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i)}[u] + w(u,v))$
- Then $dist(s,v) = d^{(n-1)}[v]$

Another way of writing this

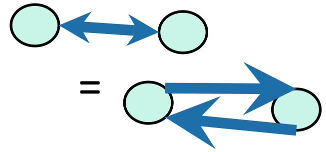
- We are actually going to change this to be dumber.
- Keep n arrays: $d^{(0)}$, $d^{(1)}$, …, $d^{(n-1)}$

Bellman-Ford*(G,s):

- $d^{(0)}[v] = \infty$ for all v in V
- $d^{(0)}[s] = 0$
- **For** i=0,…,n-2:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_{u \text{ in } v.\text{inNbrs}}\{d^{(i)}[u] + w(u,v)\})$
- Then $\text{dist}(s,v) = d^{(n-1)}[v]$

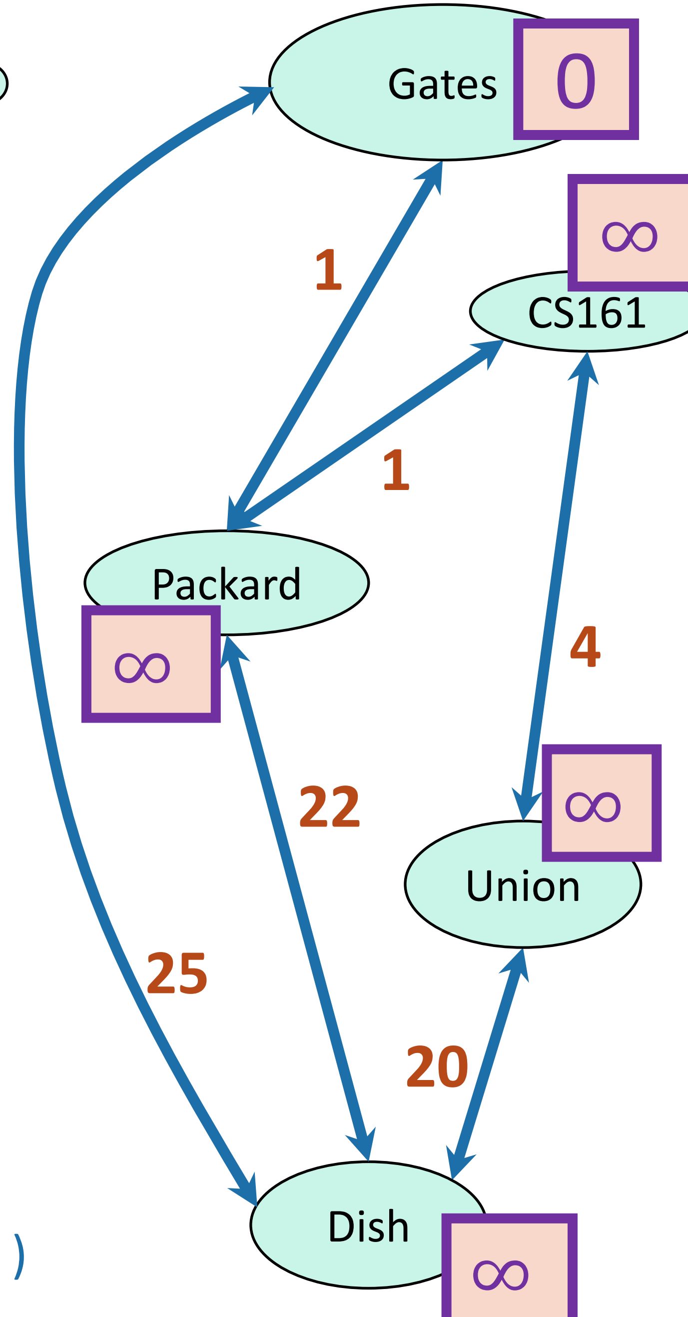
The for loop over u gets picked up in this min.

Bellman-Ford



How far is a node from Gates?

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$					
$d^{(2)}$					
$d^{(3)}$					
$d^{(4)}$					

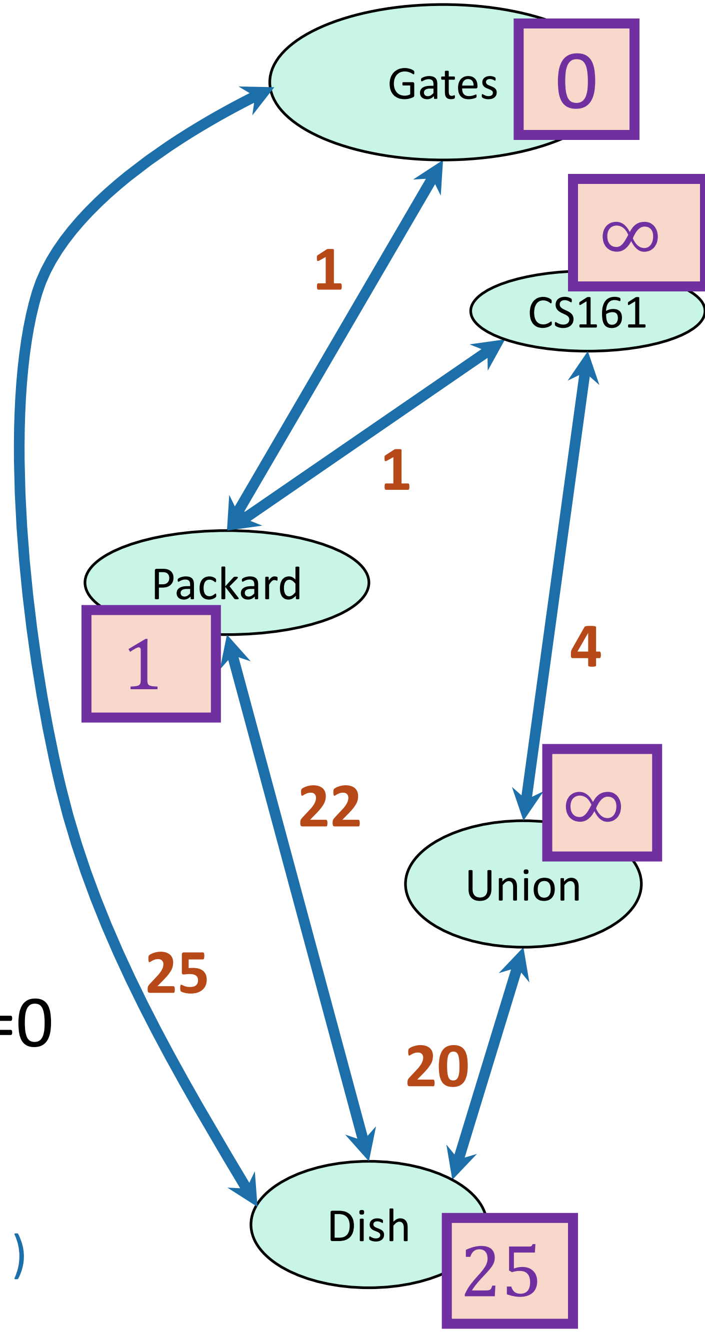


- **For** i=0,…,n-2:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_u \{d^{(i)}[u] + w(u,v)\})$

Bellman-Ford

How far is a node from Gates?

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$					
$d^{(3)}$					
$d^{(4)}$					



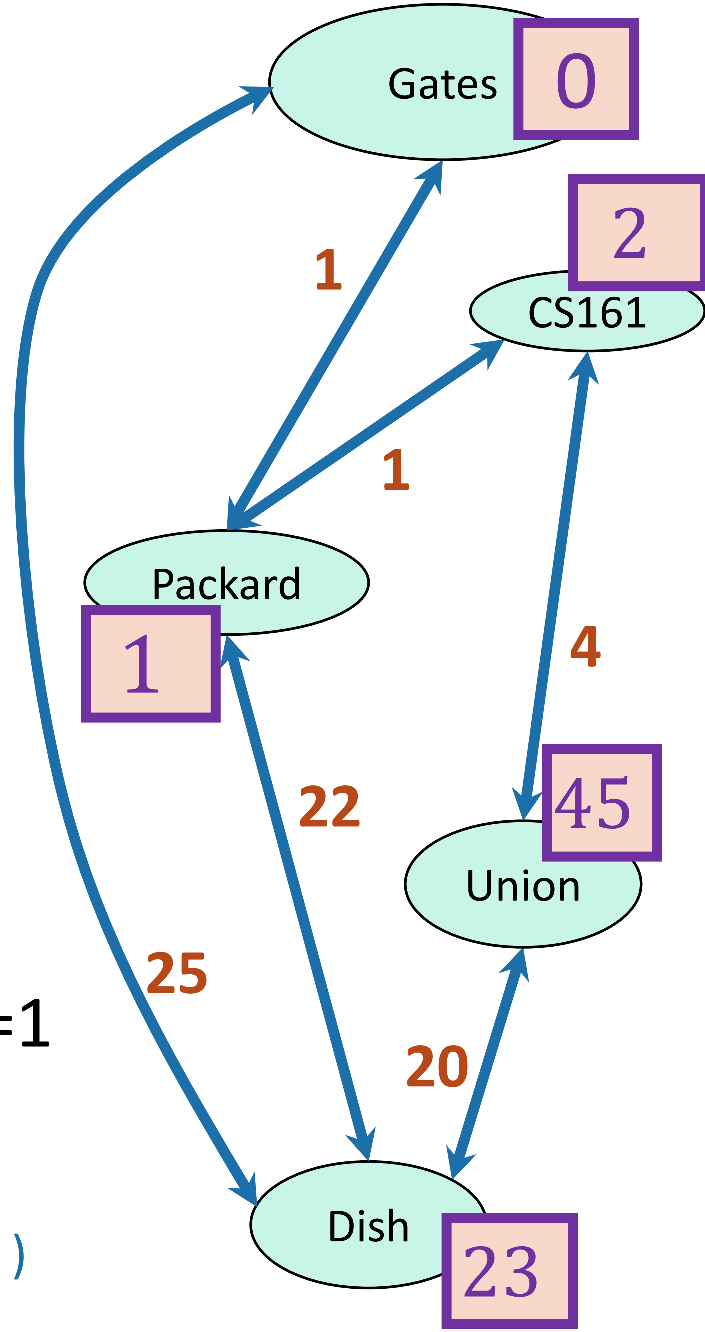
i=0

- **For** i=0,…,n-2:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_u \{d^{(i)}[u] + w(u,v)\})$

Bellman-Ford

How far is a node from Gates?

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$					
$d^{(4)}$					



i=1

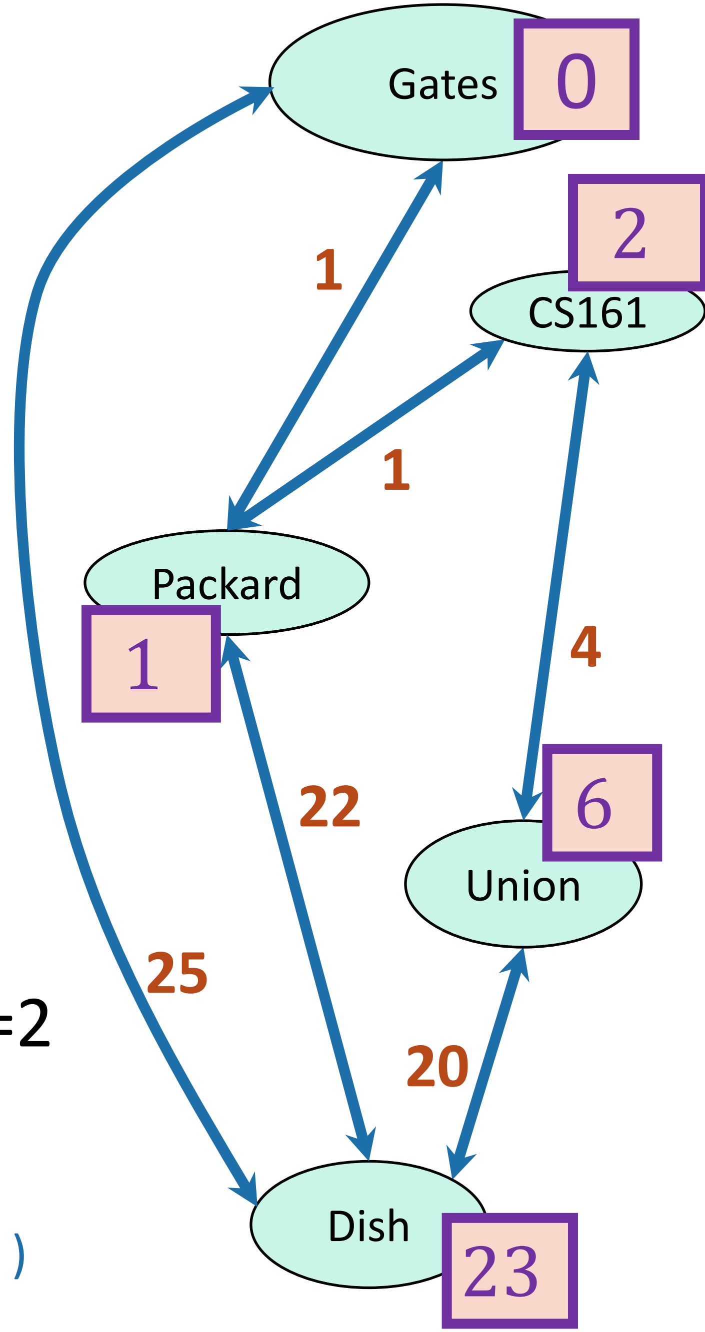
- **For** i=0,…,n-2:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_u \{d^{(i)}[u] + w(u,v)\})$

Bellman-Ford

How far is a node from Gates?

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$	0	1	2	**6**	23
$d^{(4)}$					

i=2

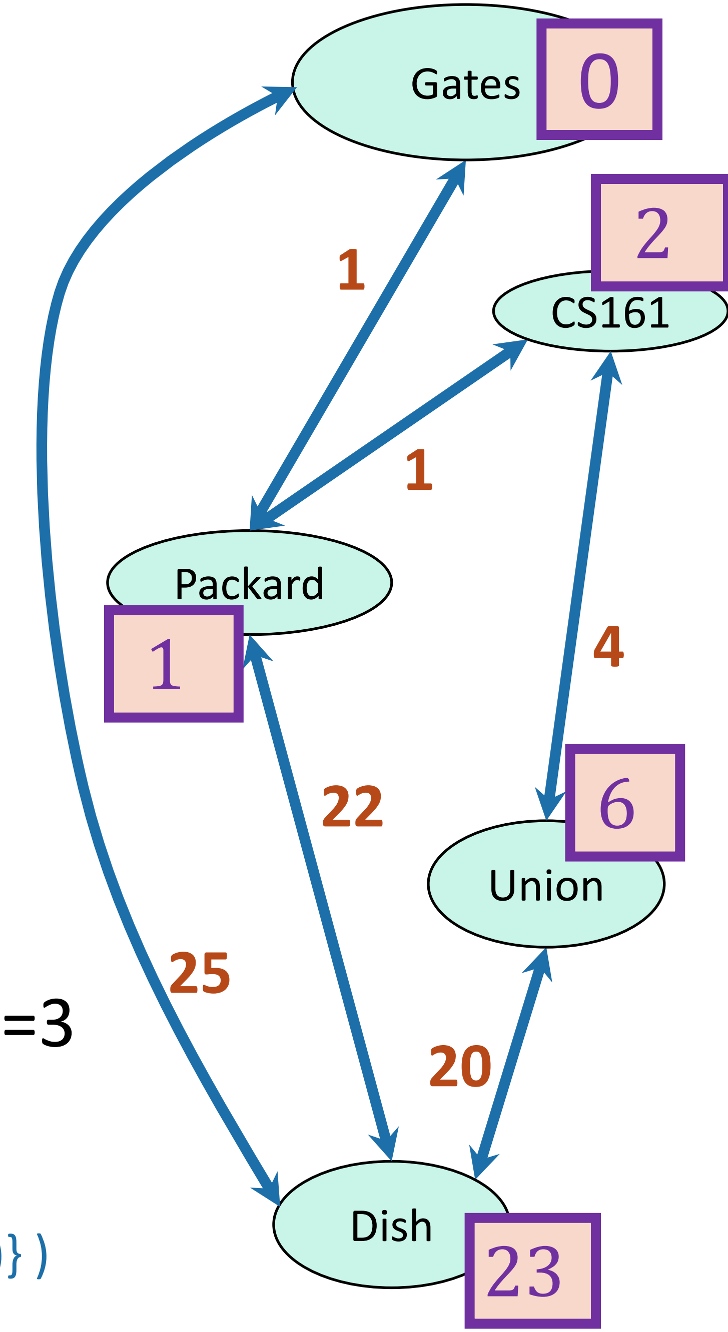


- **For** i=0,…,n-2:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_u \{d^{(i)}[u] + w(u,v)\})$

Bellman-Ford

How far is a node from Gates?

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$	0	1	2	**6**	23
$d^{(4)}$	0	1	2	6	23



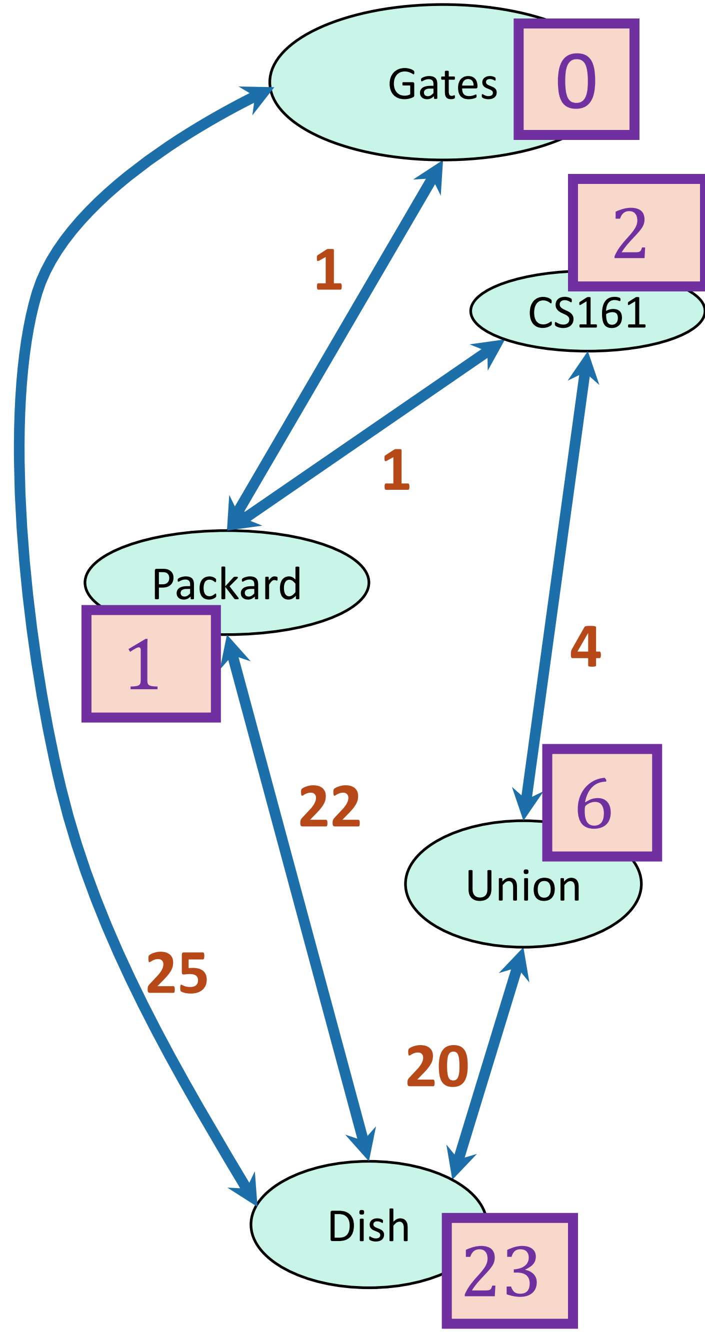
i=3

- **For** i=0,…,n-2:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(\, d^{(i)}[v]\, ,\, \min_u \{d^{(i)}[u] + w(u,v)\}\,)$

Interpretation of $d^{(i)}$

$d^{(i)}[v]$ is equal to the cost of the shortest path between s and v **with at most i edges**.

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$	0	1	2	**6**	23
$d^{(4)}$	0	1	2	6	23



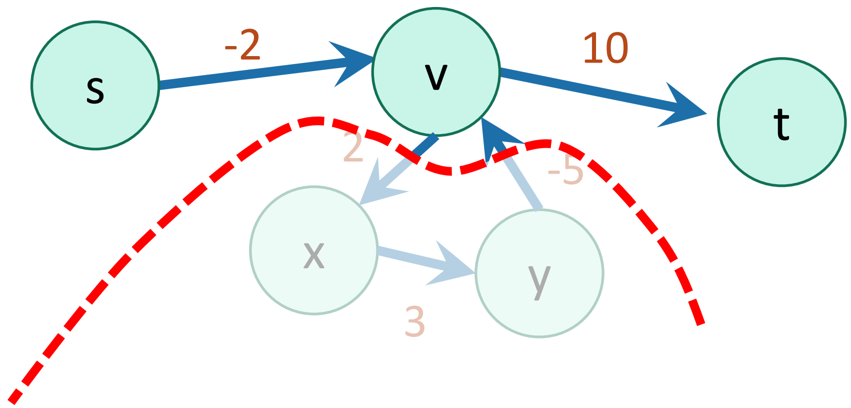
Why does Bellman-Ford work?

- Inductive hypothesis:
    - $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v **with at most i edges**.
- Conclusion:

Aside: simple paths

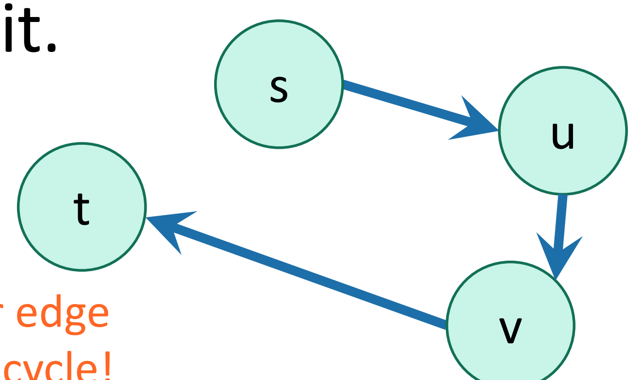
Assume there is no negative cycle.

- Then not only are there shortest paths, but actually there's always a simple shortest path.



This cycle isn't helping.
Just get rid of it.

- A simple path in a graph with n vertices has at most n-1 edges in it.



"Simple" means that the path has no cycles in it.

Can't add another edge without making a cycle!

Why does it work?

- Inductive hypothesis:
  - $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v **with at most i edges**.
- Conclusion(s):
  - $d^{(n-1)}[v]$ is equal to the cost of the shortest path between s and v **with at most n-1 edges**.
  - **If there are no negative cycles**, $d^{(n-1)}[v]$ is equal to the cost of the shortest path.

Notice that negative edge weights are fine.
Just not negative cycles.

Note on implementation

- Don't actually keep all n arrays around.
- Just keep two at a time: "last round" and "this round"

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$	0	1	2	**6**	23
$d^{(4)}$	0	1	2	6	23

Only need these two in order to compute $d^{(4)}$

This seems much slower than Dijkstra

- And it is:

Running time $O(mn)$

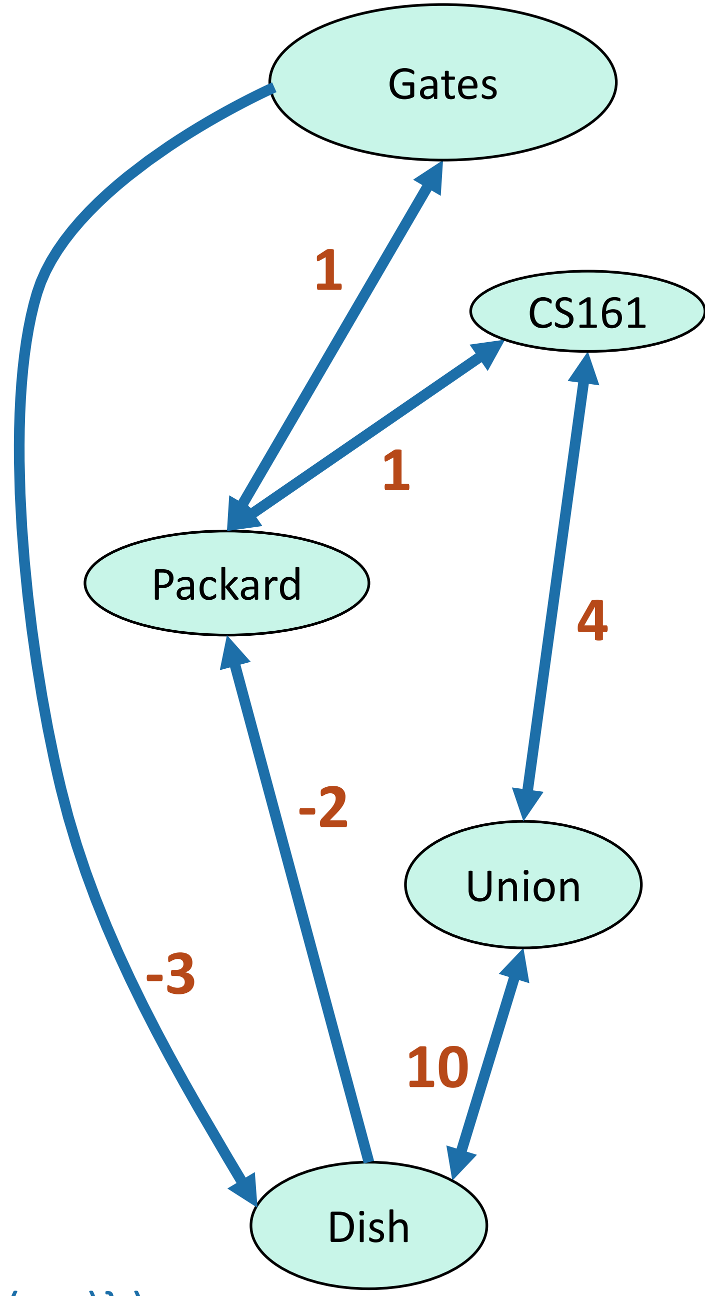
- However, it's also more flexible in a few ways.
  - Can handle negative edges
  - If we keep on doing these iterations, then changes in the network will propagate through.

- **For** $i=0,\ldots,n-2$:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(\, d^{(i)}[v]\, ,\, \min_{u \text{ in } v.nbrs}\{d^{(i)}[u] + w(u,v)\}\,)$
- Then $dist(s,v) = d^{(n-1)}[v]$

Negative cycles

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**-3**
$d^{(2)}$	0	**-5**	**2**	**7**	**-3**
$d^{(3)}$	**-4**	-5	**-4**	**6**	**-3**
$d^{(4)}$	-4	-5	-4	6	**-7**

This is not looking good!



- **For** i=0,…,n-2:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_{u \text{ in } v.\text{nbrs}}\{d^{(i)}[u] + w(u,v)\})$

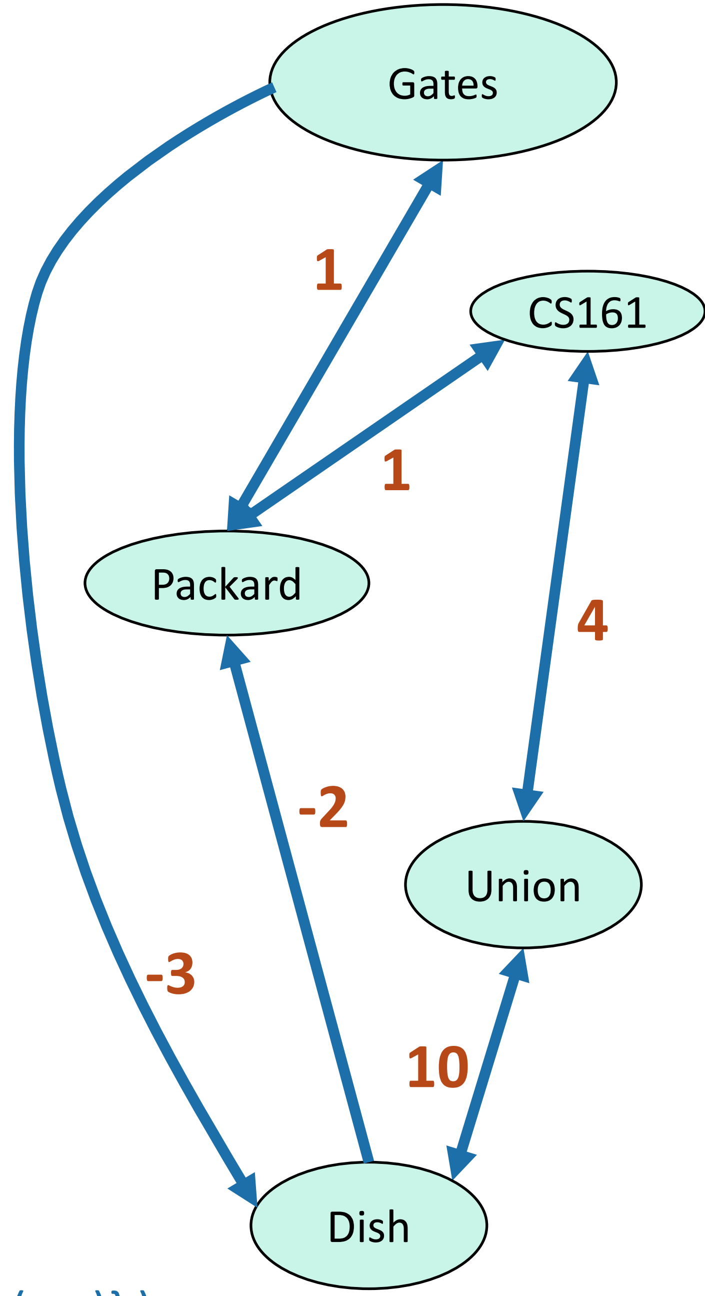
Negative edge weights

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**-3**
$d^{(2)}$	0	**-5**	**2**	**7**	**-3**
$d^{(3)}$	**-4**	-5	**-4**	**6**	**-3**
$d^{(4)}$	-4	-5	-4	6	**-7**

But **we can tell** that it's not looking good:

	Gates	Packard	CS161	Union	Dish
$d^{(5)}$	-4	**-9**	-4	**3**	-7

Some stuff changed!



- **For** i=0,…,n-2:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_{u \text{ in } v.nbrs}\{d^{(i)}[u] + w(u,v)\})$

Negative cycles in Bellman-Ford

- If there are no negative cycles:
  - Everything works as it should, and stabilizes.
- If there are negative cycles:
  - Not everything works as it should…
    - Note: it couldn’t possibly work, since shortest paths aren’t well-defined if there are negative cycles.
  - The d[v] values will keep changing.
- Solution:
  - Go one round more and see if things change.

Bellman-Ford algorithm

Bellman-Ford*(G,s):

- $d^{(0)}[v] = \infty$ for all v in V
- $d^{(0)}[s] = 0$
- **For** i=0,…,n-1:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(\, d^{(i)}[v] \,,\, \min_{u \text{ in v.inNeighbors}} \{d^{(i)}[u] + w(u,v)\} \,)$
- If $d^{(n-1)} \mathrel{!}= d^{(n)}$:
  - **Return** NEGATIVE CYCLE ☹
- Otherwise, $dist(s,v) = d^{(n-1)}[v]$

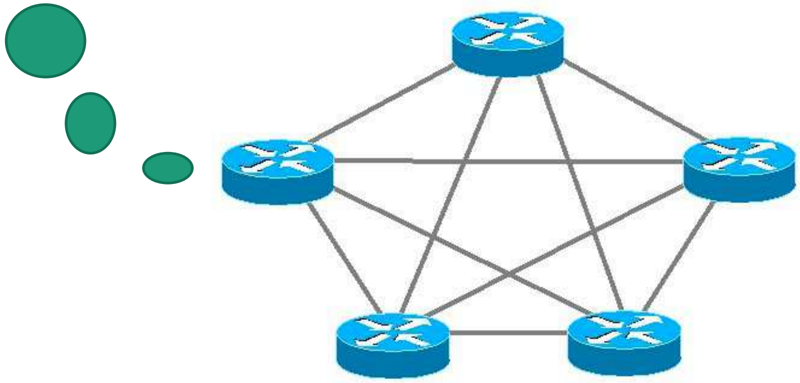
Running time: O(mn)

Bellman-Ford is also used in practice.

- eg, Routing Information Protocol (RIP) uses something like Bellman-Ford.
  - Older protocol, not used as much anymore.

- Each router keeps a **table** of distances to every other router.
- Periodically we do a Bellman-Ford update.
  - Aka, for an edge (u,v):
  - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i)}[u] + w(u,v))$
- This means that if there are changes in the network, this will propagate. (maybe slowly…)

Destination	Cost to get there	Send to whom?
172.16.1.0	34	172.16.1.1
10.20.40.1	10	192.168.1.2
10.155.120.1	9	10.13.50.0



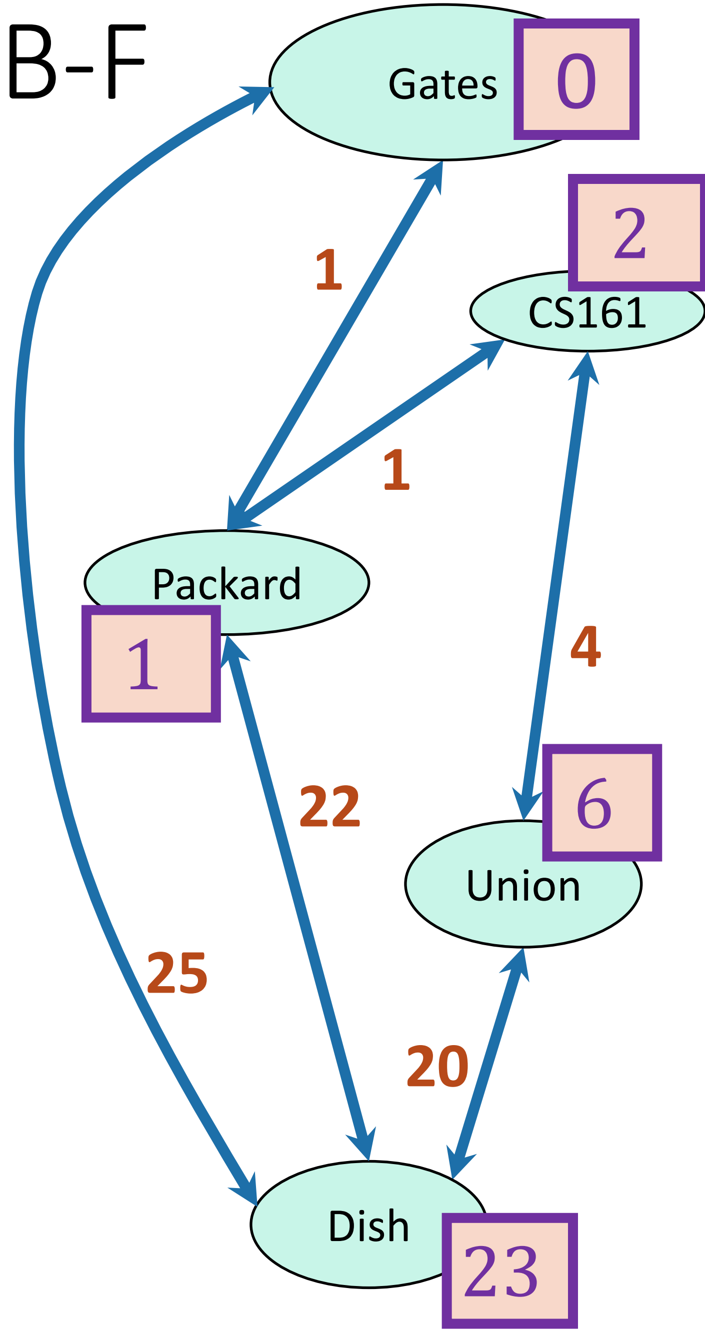
Recap: shortest paths

- BFS:
  - (+) O(n+m)
  - (-) only unweighted graphs
- Dijkstra's algorithm:
  - (+) weighted graphs
  - (+) O(nlog(n) + m) if you implement it right.
  - (-) no negative edge weights
  - (-) very "centralized" (need to keep track of all the vertices to know which to update).
- The Bellman-Ford algorithm:
  - (+) weighted graphs, even with negative weights
  - (+) can be done in a distributed fashion, every vertex using only information from its neighbors.
  - (-) O(nm)

Important thing about B-F
for the rest of this lecture

$d^{(i)}[v]$ is equal to the cost of the shortest path between s and v **with at most i edges**.

	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$	0	1	2	**6**	23
$d^{(4)}$	0	1	2	6	23



Bellman-Ford is an example of…

Dynamic Programming!

Today:

- Example of Dynamic programming:
  - Fibonacci numbers.
  - (And Bellman-Ford)
- What is dynamic programming, exactly?
  - And why is it called "dynamic programming"?
- Another example: Floyd-Warshall algorithm
  - An "all-pairs" shortest path algorithm

Pre-Lecture exercise:
How not to compute Fibonacci Numbers

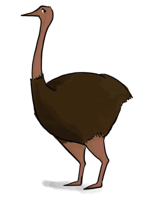
- Definition:
  - $F(n) = F(n-1) + F(n-2)$, with $F(0) = F(1) = 1$.
  - The first several are:

    1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,…

- Question:
  - Given n, what is $F(n)$?

Candidate algorithm

See CLRS Problem 4-4 for a walkthrough of how fast the Fibonacci numbers grow!

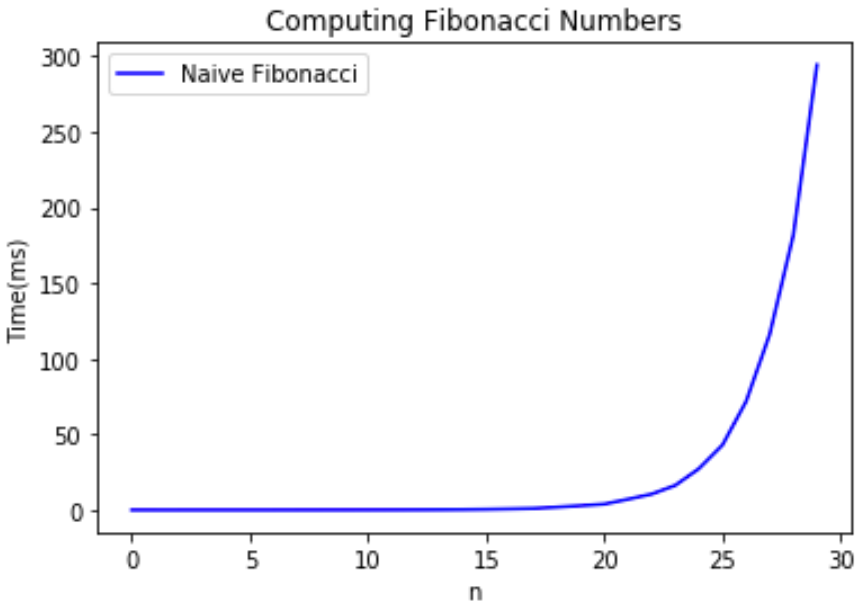


```
• def Fibonacci(n):
    • if n == 0 or n == 1:
        • return 1
    • return Fibonacci(n-1) + Fibonacci(n-2)
```

(Seems to work, according to the IPython notebook…)

Running time?

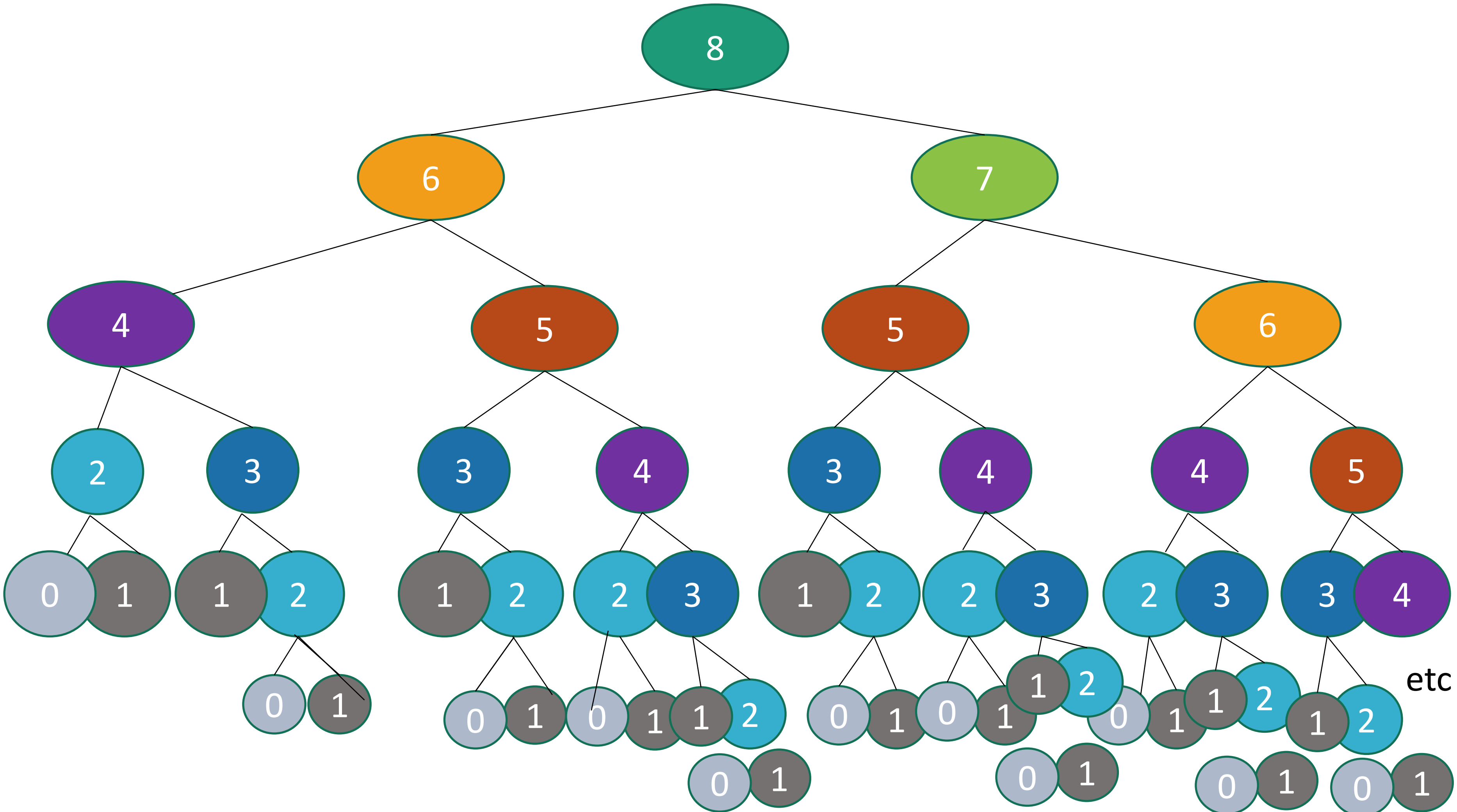
- T(n) = T(n-1) + T(n-2) + O(1)
- T(n) ≥ T(n-1) + T(n-2) for n ≥ 2
- So T(n) grows *at least* as fast as the Fibonacci numbers themselves…
- Fun fact, that's like ϕ^n where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.
- aka, **EXPONENTIALLY QUICKLY** ☹



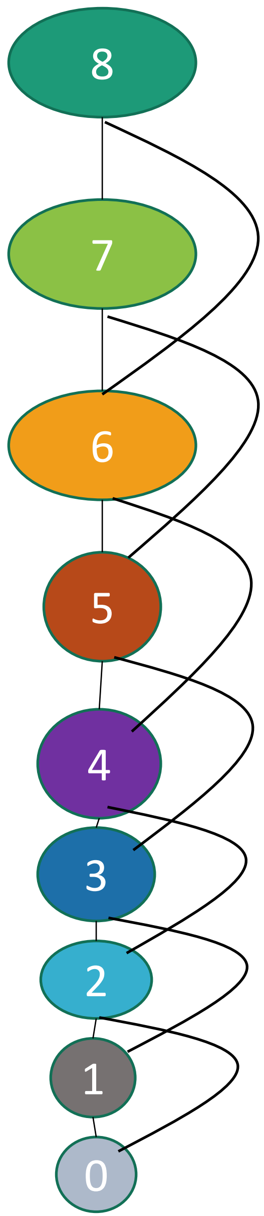
What’s going on?

Consider Fib(8)

That’s a lot of repeated computation!

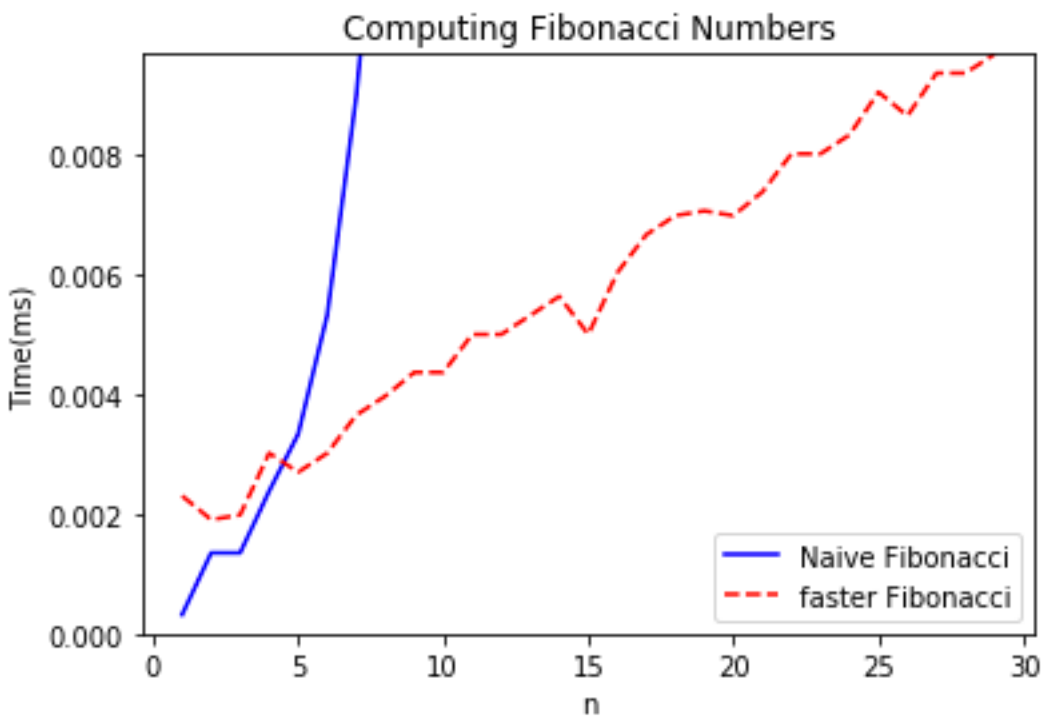


Maybe this would be better:




```
def fasterFibonacci(n):
    • F = [1, 1, None, None, …, None ]
        • \\ F has length n
    • for i = 2, …, n:
        • F[i] = F[i-1] + F[i-2]
    • return F[n]
```

Much better running time!



This was an example of…

Dynamic Programming!

What is *dynamic programming*?

- It is an algorithm design paradigm
  - like divide-and-conquer is an algorithm design paradigm.
- Usually it is for solving **optimization problems**
  - eg, ***shortest*** path
  - (Fibonacci numbers aren't an optimization problem, but they are a good example…)

Elements of dynamic programming

1. Optimal sub-structure:

- Big problems break up into sub-problems.
  - Fibonacci: $F(i)$ for $i \leq n$
  - Bellman-Ford: Shortest paths with at most i edges for $i \leq n$
- The solution to a problem can be expressed in terms of solutions to smaller sub-problems.
  - Fibonacci:

$$F(i+1) = F(i) + F(i-1)$$

  - Bellman-Ford:

$$d^{(i+1)}[v] \leftarrow \min\{ d^{(i)}[v],\ \min_u \{d^{(i)}[u] + \mathrm{weight}(u,v)\} \}$$

Shortest path with at most i edges from s to v

Shortest path with at most i edges from s to u.

Elements of dynamic programming

2. Overlapping sub-problems:

- The sub-problems overlap a lot.
  - Fibonacci:
    - Lots of different F[j] will use F[i].
  - Bellman-Ford:
    - Lots of different entries of $d^{(i+1)}$ will use $d^{(i)}[v]$.

  - This means that we can save time by solving a sub-problem just once and storing the answer.

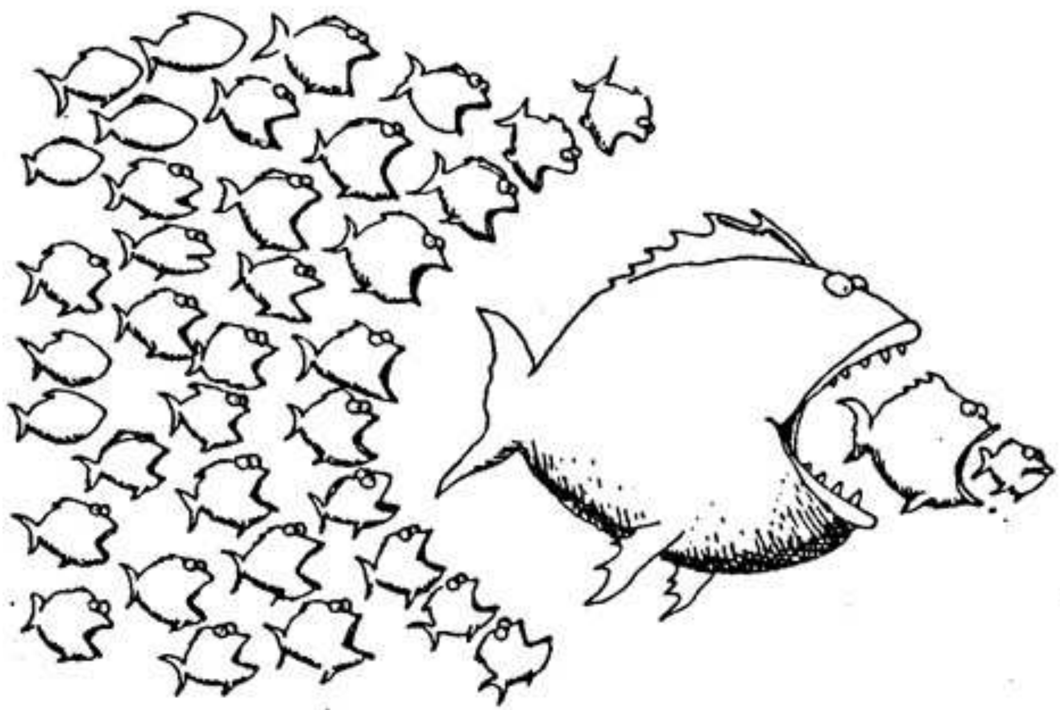
Elements of dynamic programming

- Optimal substructure.
  - Optimal solutions to sub-problems are sub-solutions to the optimal solution of the original problem.
- Overlapping subproblems.
  - The subproblems show up again and again

- Using these properties, we can design a ***dynamic programming*** algorithm:
  - Keep a table of solutions to the smaller problems.
  - Use the solutions in the table to solve bigger problems.
  - At the end we can use information we collected along the way to find the solution to the whole thing.

Two ways to think about and/or implement DP algorithms

- Top down
- Bottom up

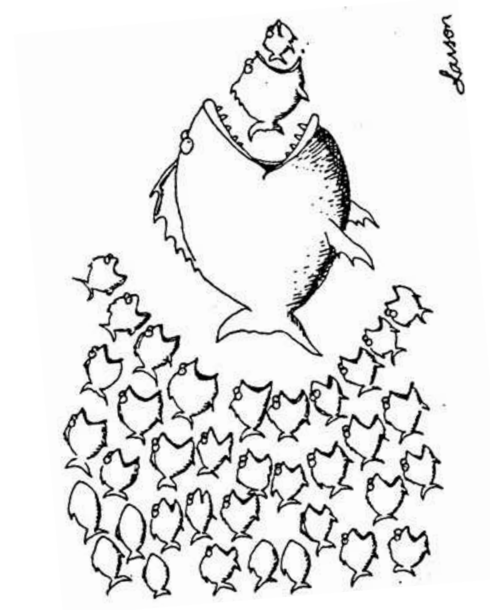


This picture isn't hugely relevant but I like it. Larson

Bottom up approach

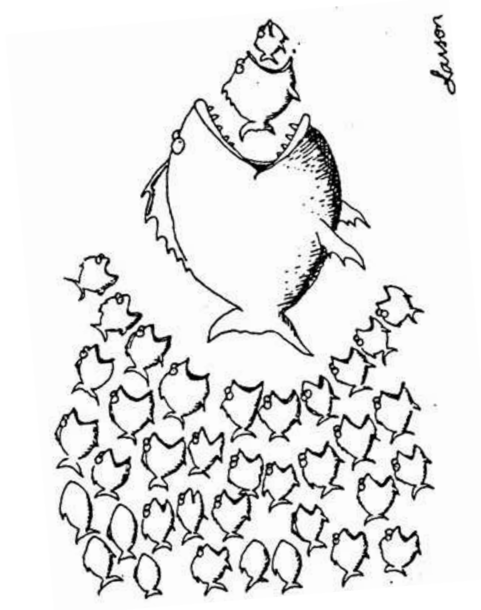
what we just saw.

- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
  - fill in F[2]
- …
- Then bigger problems
  - fill in F[n-1]
- Then finally solve the real problem.
  - fill in F[n]



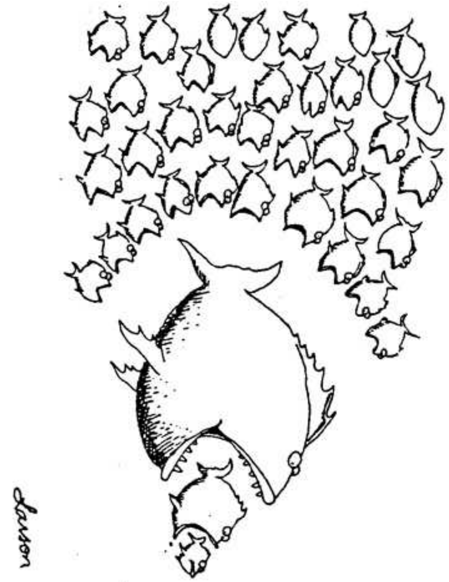
Bottom up approach

what we just saw.



- For Bellman-Ford:
- Solve the small problems first
  - fill in $d^{(0)}$
- Then bigger problems
  - fill in $d^{(1)}$
- …
- Then bigger problems
  - fill in $d^{(n-2)}$
- Then finally solve the real problem.
  - fill in $d^{(n-1)}$

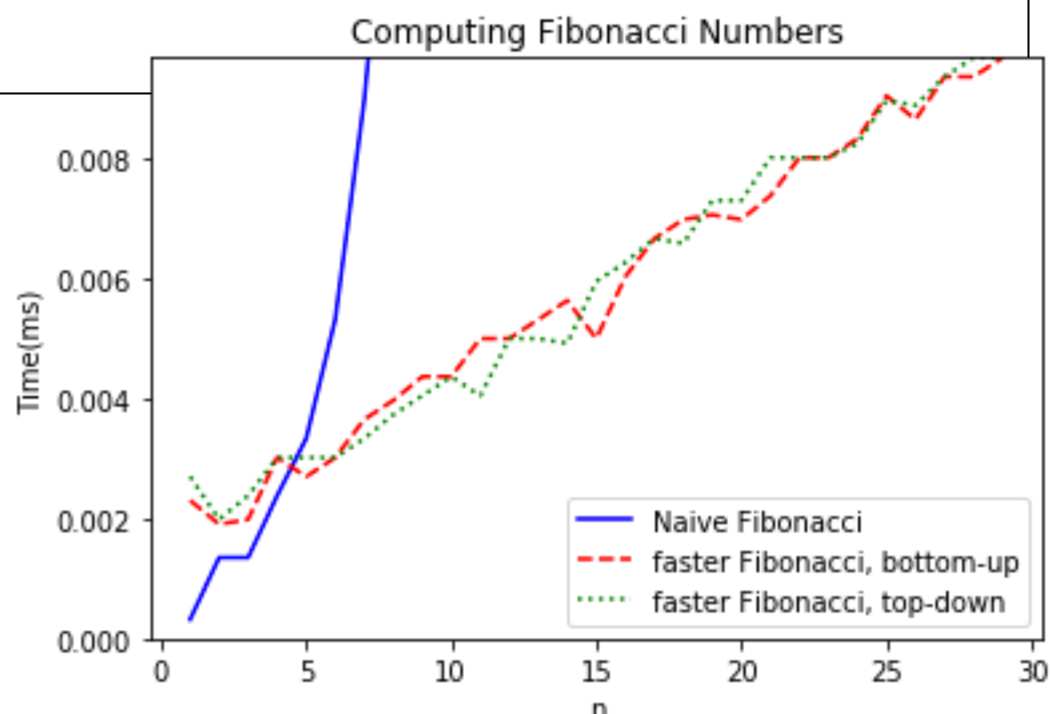
Top down approach



- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
      - etc..
- The difference from divide and conquer:
  - **Memo-ization**
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.

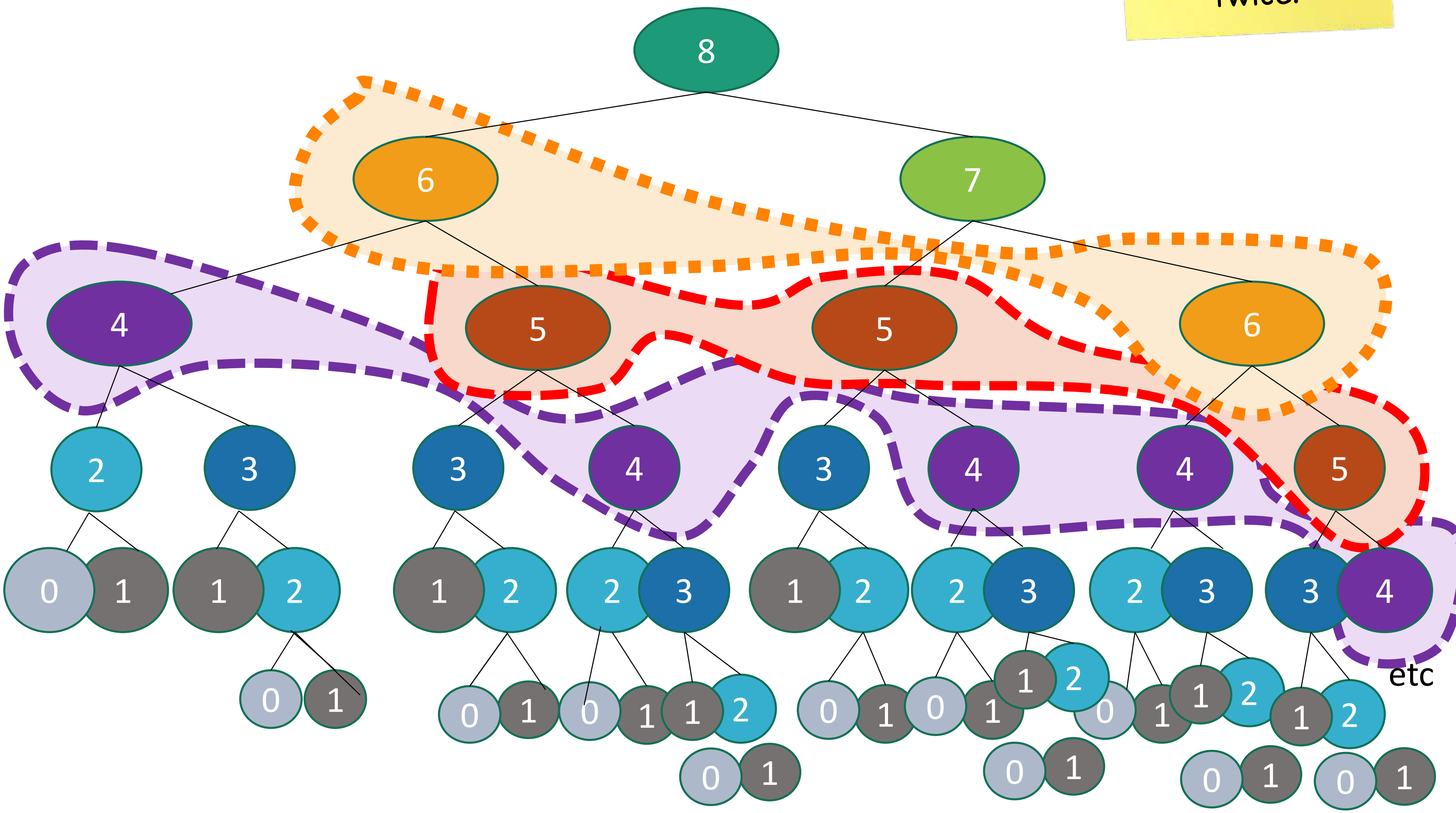
Example of top-down Fibonacci

```
• define a global list F = [1,1,None, None, …, None]
• def Fibonacci(n):
    • if F[n] != None:
        • return F[n]
    • else:
        • F[n] = Fibonacci(n-1) + Fibonacci(n-2)
    • return F[n]
```



Memo-ization visualization

Collapse repeated nodes and don't do the same work twice!

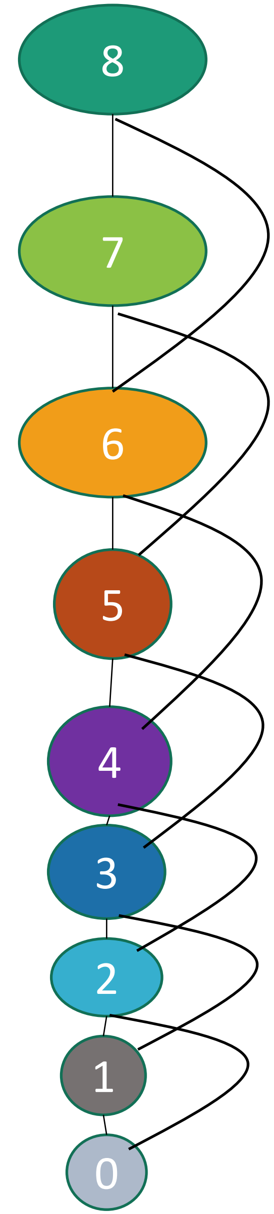


Memo-ization Visualization
ctd

Collapse repeated nodes and don't do the same work twice!

But otherwise treat it like the same old recursive algorithm.

```
• define a global list F = [1,1,None, None, …, None]
• def Fibonacci(n):
    • if F[n] != None:
        • return F[n]
    • else:
        • F[n] = Fibonacci(n-1) + Fibonacci(n-2)
    • return F[n]
```



What have we learned?

- ***Dynamic programming:***
  - Paradigm in algorithm design.
  - Uses **optimal substructure**
  - Uses **overlapping subproblems**
  - Can be implemented **bottom-up** or **top-down**.
  - It’s a fancy name for a pretty common-sense idea:

Why “*dynamic programming*” ?

- Programming refers to finding the optimal “program.”
  - as in, a shortest route is a *plan* aka a *program*.
- Dynamic refers to the fact that it’s multi-stage.
- But also it’s just a fancy-sounding name.



Manipulating computer code in an action movie?

Why “*dynamic programming*” ?

- Richard Bellman invented the name in the 1950’s.
- At the time, he was working for the RAND Corporation, which was basically working for the Air Force, and government projects needed flashy names to get funded.
- From Bellman’s autobiography:
    - “It’s impossible to use the word, dynamic, in the pejorative sense…I thought dynamic programming was a good name. It was something not even a Congressman could object to.”

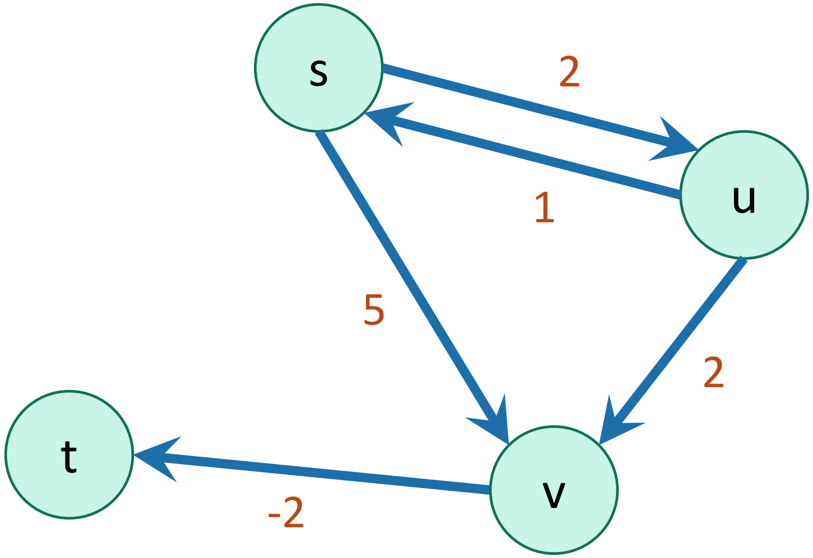
Floyd-Warshall Algorithm
Another example of DP

- This is an algorithm for **All-Pairs Shortest Paths** (APSP)
  - That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  - Not just from a special single source s.

Destination

Source

	s	**u**	**v**	**t**
s	0	2	4	2
u	1	0	2	0
v	∞	∞	0	-2
t	∞	∞	∞	0



Floyd-Warshall Algorithm

Another example of DP

- This is an algorithm for **All-Pairs Shortest Paths (**APSP)
  - That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  - Not just from a special single source s.
- Naïve solution (if we want to handle negative edge weights):
  - For all s in G:
    - Run Bellman-Ford on G starting at s.
  - Time $O(n \cdot nm) = O(n^2 m)$,
    - may be as bad as n^4 if $m=n^2$

Can we do better?

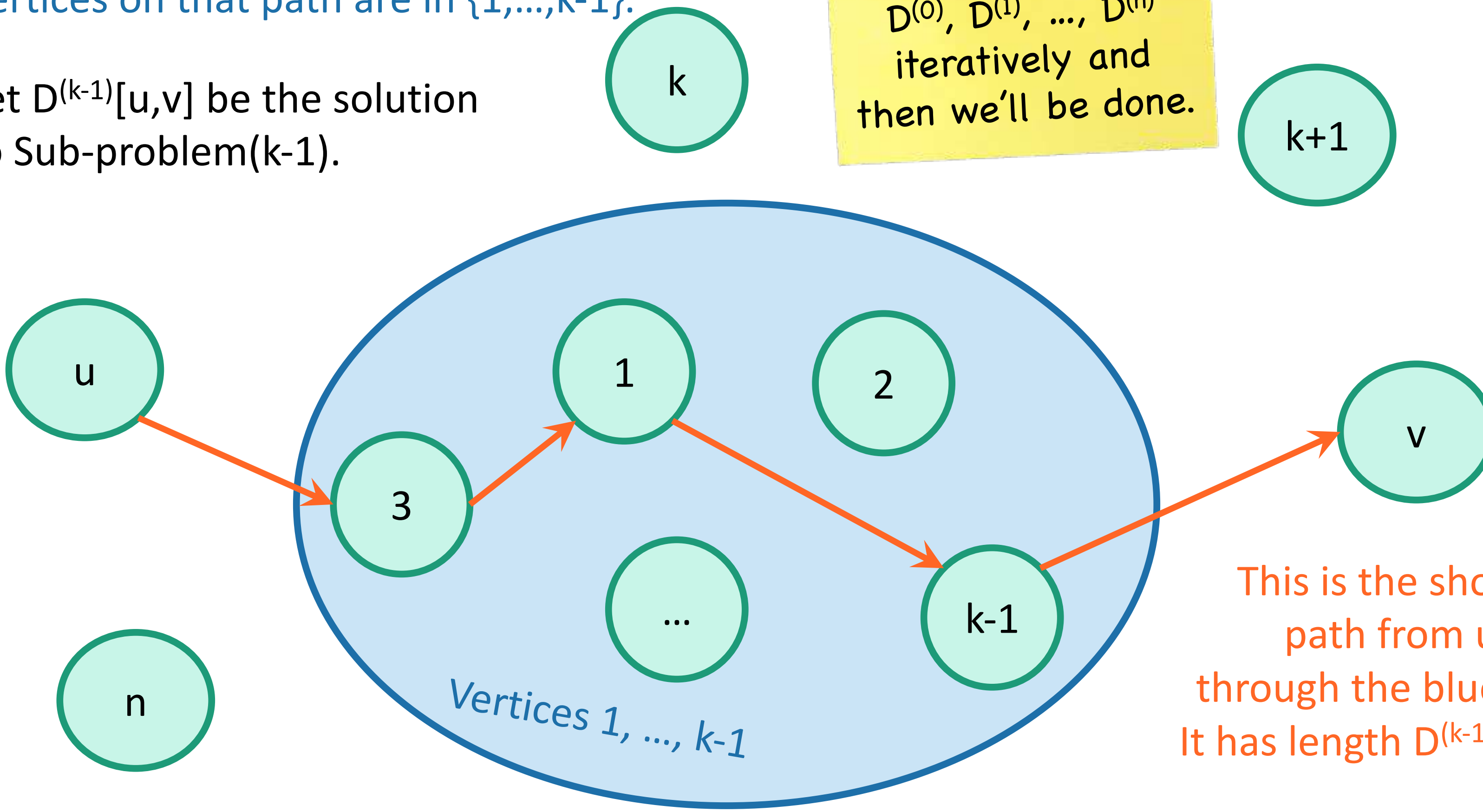
Optimal substructure

Label the vertices 1,2,…,n
(We omit some edges in the picture below).

Sub-problem(k-1):
For all pairs, u,v, find the cost of the shortest path from u to v, so that all the internal vertices on that path are in {1,…,k-1}.

Our DP algorithm will fill in the n-by-n arrays $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$ iteratively and then we'll be done.

Let $D^{(k-1)}[u,v]$ be the solution to Sub-problem(k-1).



This is the shortest path from u to v through the blue set. It has length $D^{(k-1)}[u,v]$

Optimal substructure

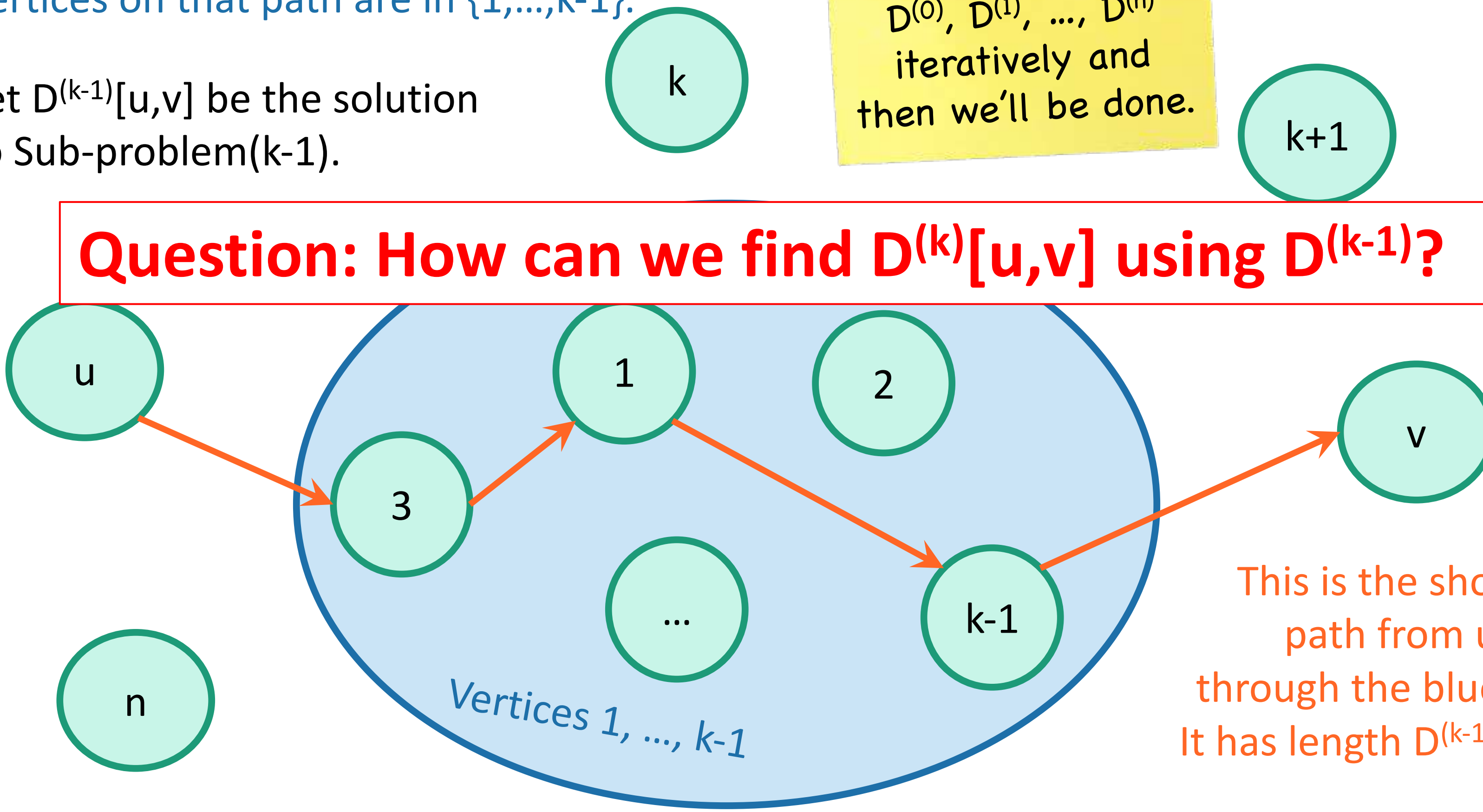
Label the vertices 1,2,…,n
(We omit some edges in the picture below).

Sub-problem(k-1):
For all pairs, u,v, find the cost of the shortest path from u to v, so that all the internal vertices on that path are in {1,…,k-1}.

Let $D^{(k-1)}[u,v]$ be the solution to Sub-problem(k-1).

Our DP algorithm will fill in the n-by-n arrays $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$ iteratively and then we'll be done.

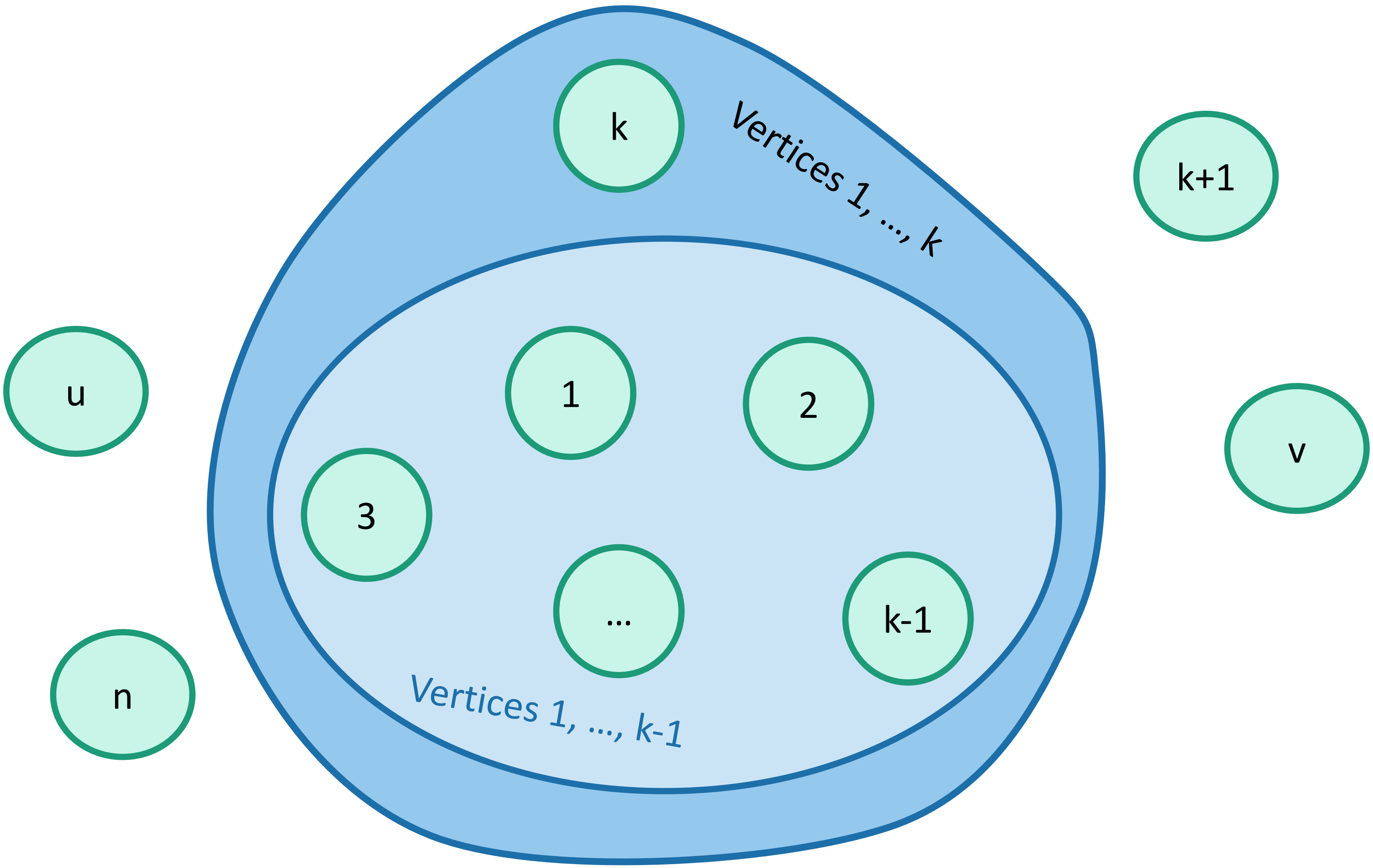
Question: How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?



This is the shortest path from u to v through the blue set. It has length $D^{(k-1)}[u,v]$

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

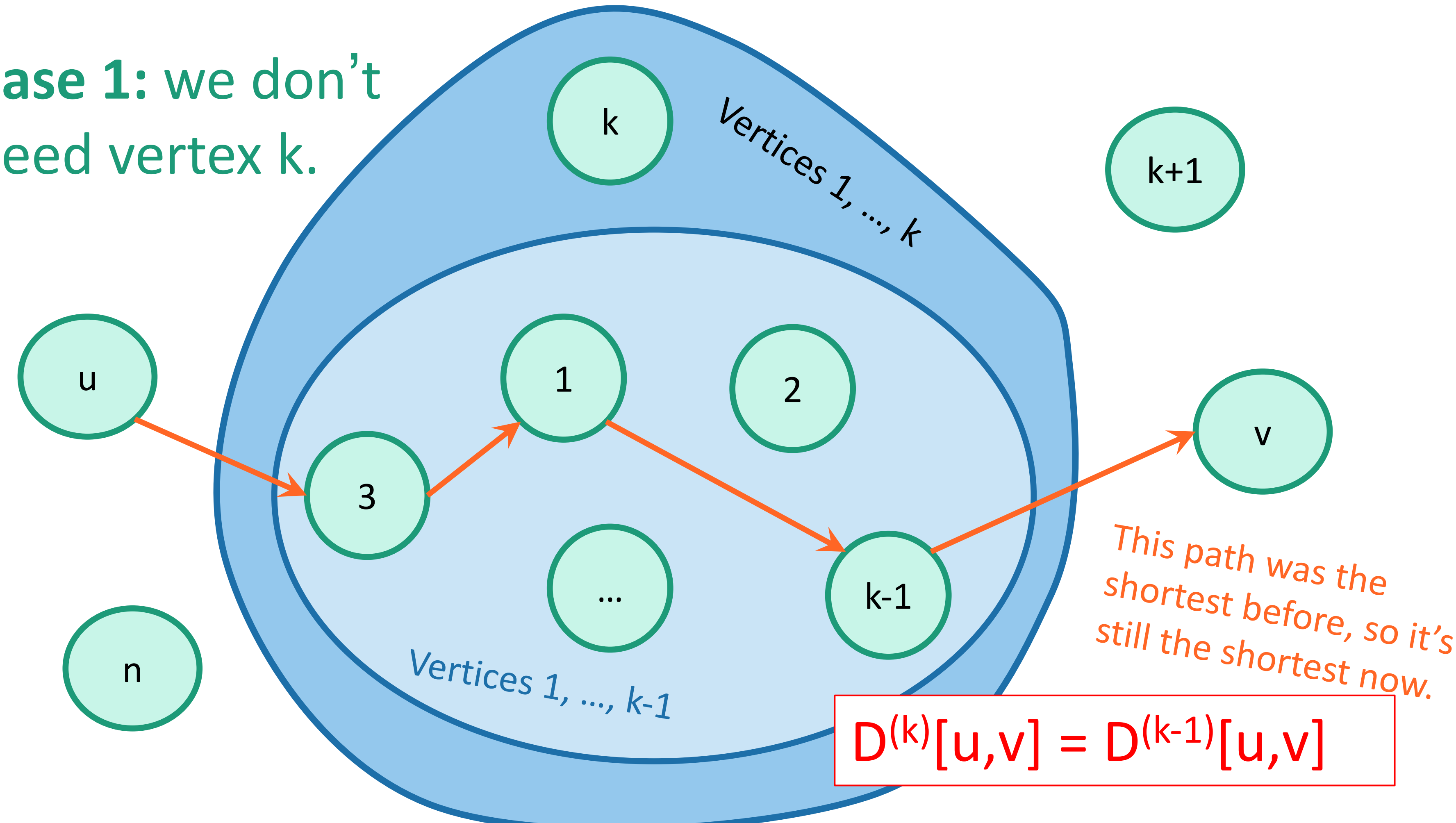
$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.



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

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

Case 1: we don't need vertex k.

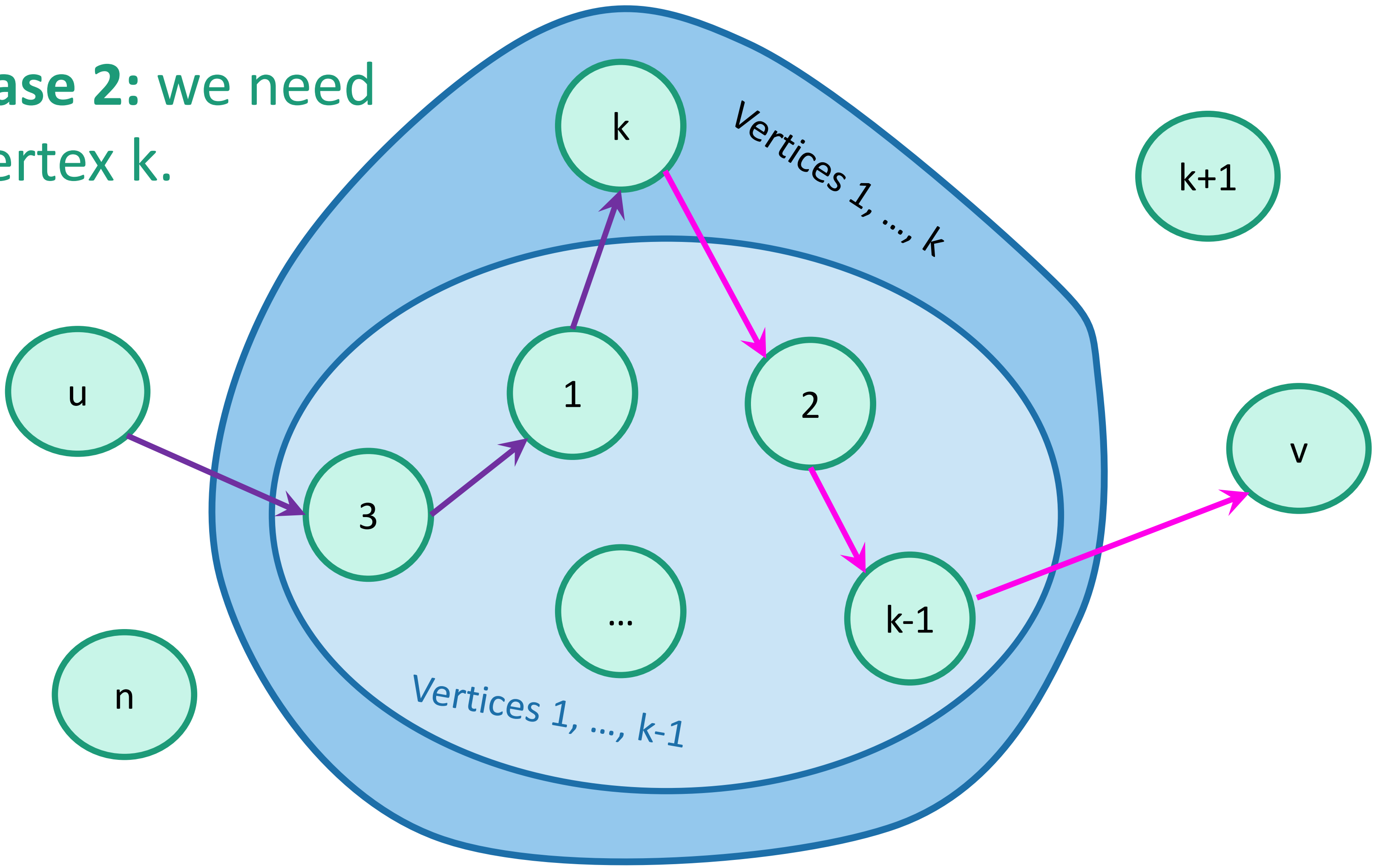


$$D^{(k)}[u,v] = D^{(k-1)}[u,v]$$

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

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

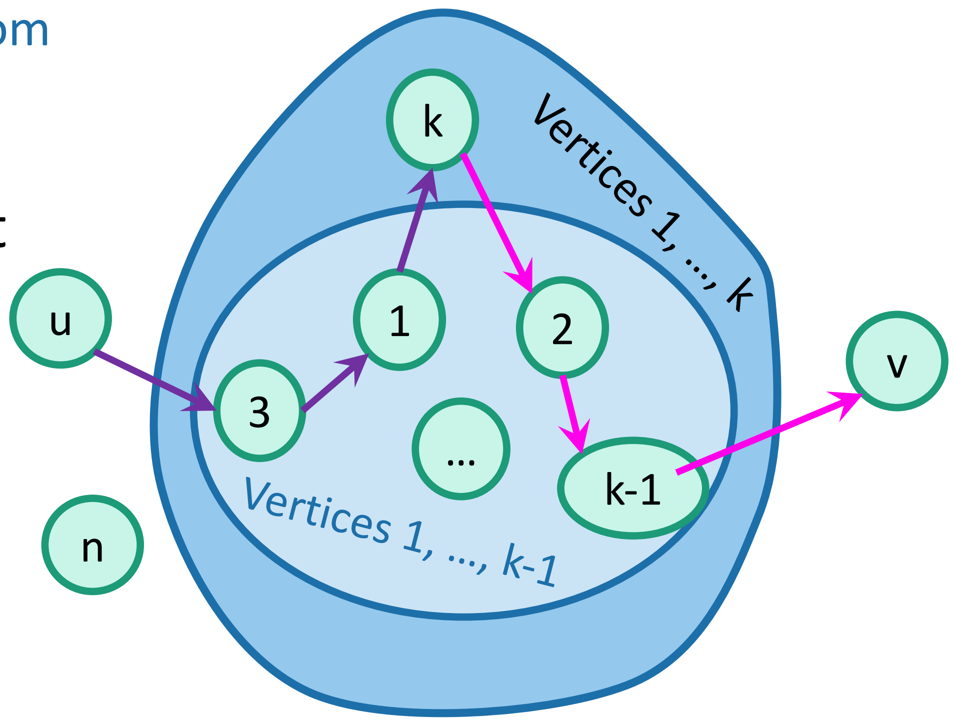
Case 2: we need vertex k.



Case 2 continued

- Suppose there are no negative cycles.
  - Then WLOG the shortest path from u to v through {1,…,k} is **simple**.
- If **that path** passes through k, it must look like this:
- **This path** is the shortest path from u to k through {1,…,k-1}.
  - sub-paths of shortest paths are shortest paths
- Same for **this path**.

Case 2: we need vertex k.



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

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

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

Case 1: Cost of shortest path through {1,…,k-1}

Case 2: Cost of shortest path from **u to k** and then from **k to v** through {1,…,k-1}

- Optimal substructure:
  - We can solve the big problem using smaller problems.
- Overlapping sub-problems:
  - $D^{(k-1)}[k,v]$ can be used to help compute $D^{(k)}[u,v]$ for lots of different u's.

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

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

Case 1: Cost of shortest path through {1,…,k-1}

Case 2: Cost of shortest path from **u to k** and then from **k to v** through {1,…,k-1}

- Using our ***Dynamic programming*** paradigm, this immediately gives us an algorithm!

Floyd-Warshall algorithm

- Initialize n-by-n arrays $D^{(k)}$ for k = 0,…,n
  - $D^{(k)}[u,u] = 0$ for all u, for all k
  - $D^{(k)}[u,v] = \infty$ for all $u \neq v$, for all k
  - $D^{(0)}[u,v] = \text{weight}(u,v)$ for all (u,v) in E.

The base case checks out: the only path through zero other vertices are edges directly from u to v.

- **For** k = 1, …, n:
  - **For** pairs u,v in V^2:
    - $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$
- **Return** $D^{(n)}$

This is a bottom-up ***Dynamic programming*** algorithm.

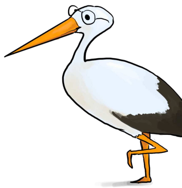
We've basically just shown

- Theorem:

  If there are no negative cycles in a weighted directed graph G, then the Floyd-Warshall algorithm, running on G, returns a matrix $D^{(n)}$ so that:

  $$D^{(n)}[u,v] = \text{distance between u and v in G.}$$

Work out the details of the proof! (Or see Lecture Notes for a few more details).



- Running time: $O(n^3)$
  - Better than running BF n times!
  - Not really better than running Dijkstra n times.
    - But it's simpler to implement and handles negative weights.
- Storage:
  - Need to store **two** n-by-n arrays, and the original graph.

  As with Bellman-Ford, we don't really need to store all n of the $D^{(k)}$.

What if there *are* negative cycles?

- Just like Bellman-Ford, Floyd-Warshall can detect negative cycles:
  - Negative cycle $\Leftrightarrow$ $\exists$ v s.t. there is a path from v to v that goes through all n vertices that has cost < 0.
  - Negative cycle $\Leftrightarrow$ $\exists$ v s.t. $D^{(n)}[v,v] < 0$.

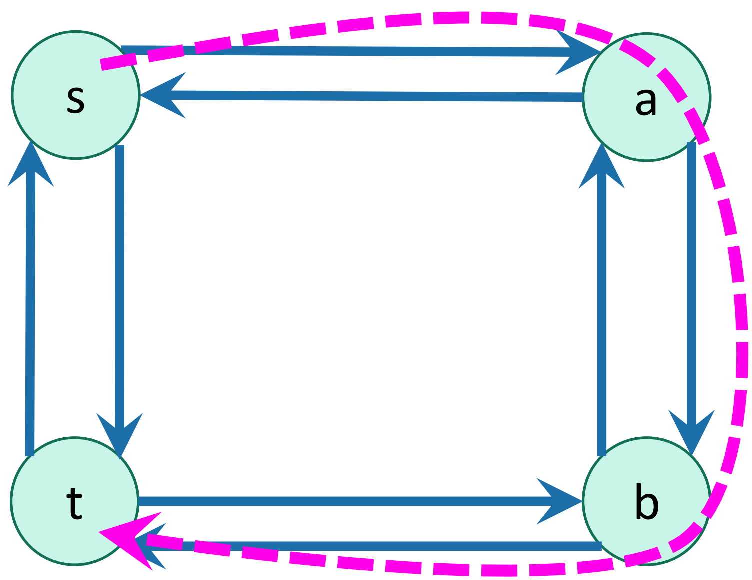
- Algorithm:
  - Run Floyd-Warshall as before.
  - If there is some v so that $D^{(n)}[v,v] < 0$:
    - **return** `negative cycle.`

What have we learned?

- The Floyd-Warshall algorithm is another example of *dynamic programming*.
- It computes All Pairs Shortest Paths in a directed weighted graph in time $O(n^3)$.

Another Example of DP?

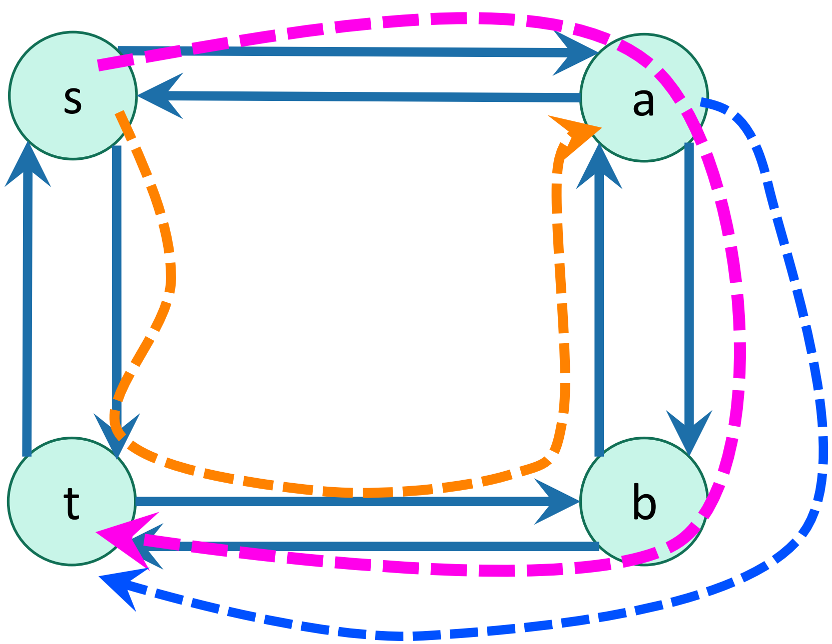
- Longest simple path (say all edge weights are 1):



What is the longest simple path from s to t?

This is an optimization problem…

- Can we use Dynamic Programming?
- Optimal Substructure?
  - Longest path from s to t = longest path from s to a + longest path from a to t?



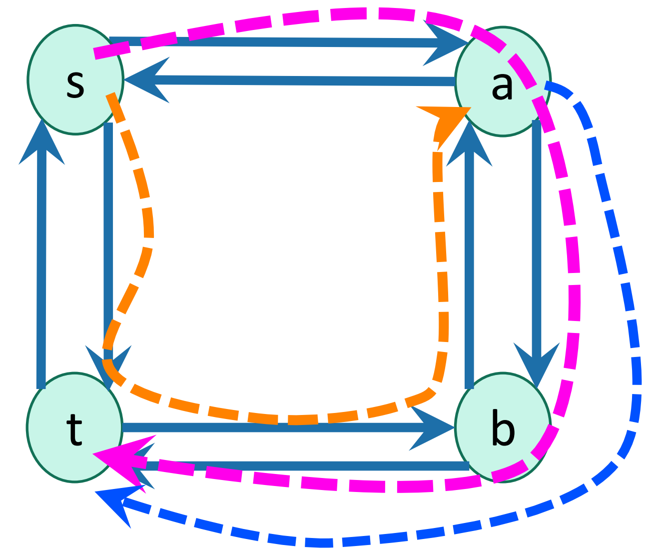
NOPE!

This doesn’t give optimal sub-structure

Optimal solutions to subproblems don’t give us an optimal solution to the big problem. (At least if we try to do it this way).

- The subproblems we came up with aren’t independent:
  - Once we’ve chosen the longest path from a to t
    - which uses b,
  - our longest path from s to a shouldn’t be allowed to use b
    - since b was already used.

- Actually, the longest simple path problem is NP-complete.
  - We don’t know of any polynomial-time algorithms for it, DP or otherwise!



Recap

- Two more shortest-path algorithms:
  - Bellman-Ford for single-source shortest path
  - Floyd-Warshall for all-pairs shortest path
- ***Dynamic programming!***
  - This is a fancy name for:
    - Break up an optimization problem into smaller problems
      - The optimal solutions to the sub-problems should be sub-solutions to the original problem.
    - Build the optimal solution iteratively by filling in a table of sub-solutions.
      - Take advantage of overlapping sub-problems!

Next time

- More examples of ***dynamic programming***!

We will stop bullets with our action-packed coding skills, and also maybe find longest common subsequences.

Before next time

- Pre-lecture exercise: finding optimal substructure