Lecture 12

Bellman-Ford, Floyd-Warshall,

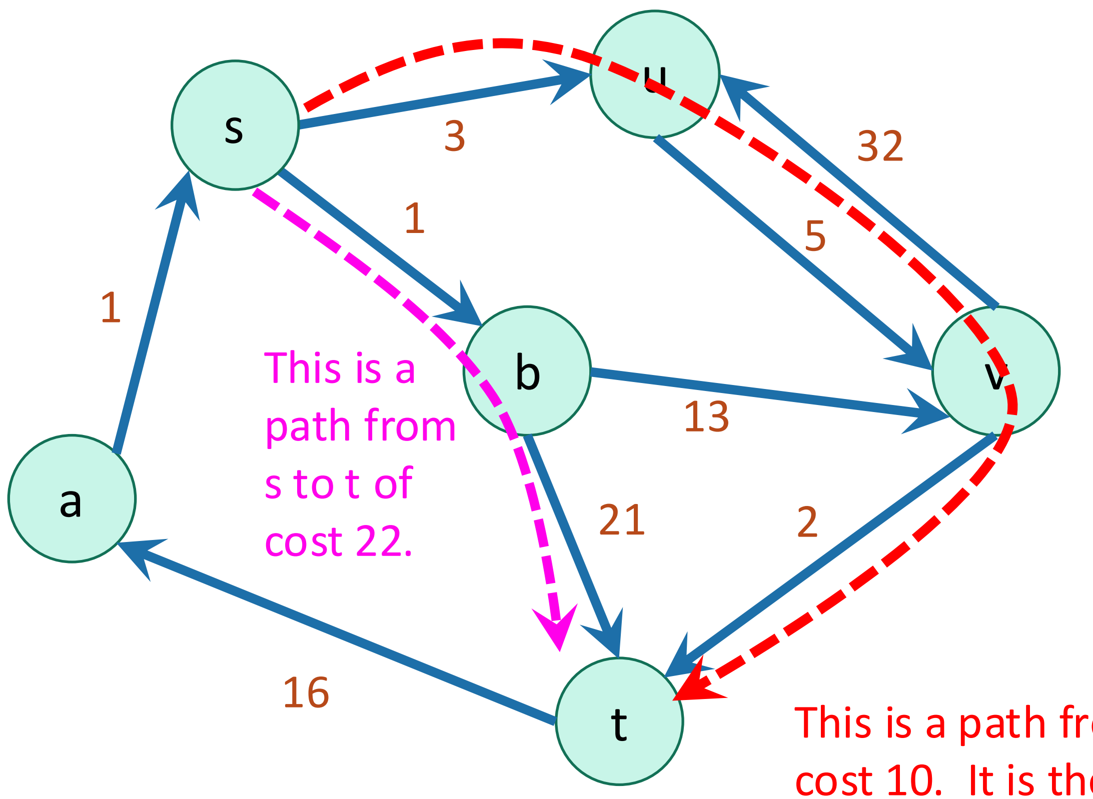
and Dynamic Programming!

Today

- Bellman-Ford Algorithm
- Bellman-Ford is a special case of ***Dynamic Programming!***
- What is dynamic programming?
  - Warm-up example: Fibonacci numbers
- Another example:
  - Floyd-Warshall Algorithm

Recall

- A weighted directed graph:



This is a path from s to t of cost 22.

This is a path from s to t of cost 10. It is the shortest path from s to t.

- 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!
  - Solves the single-source shortest path problem in weighted graphs with non-negative edge weights.

Dijkstra pseudocode:

(Note: the presentation in the textbook is slightly different)

```
• Pick the not-sure node u with the smallest estimate d[u].
• Update all u's neighbors v:
    • d[v] = min( d[v] , d[u] + edgeWeight(u,v))
• Mark u as sure.
• Repeat
```

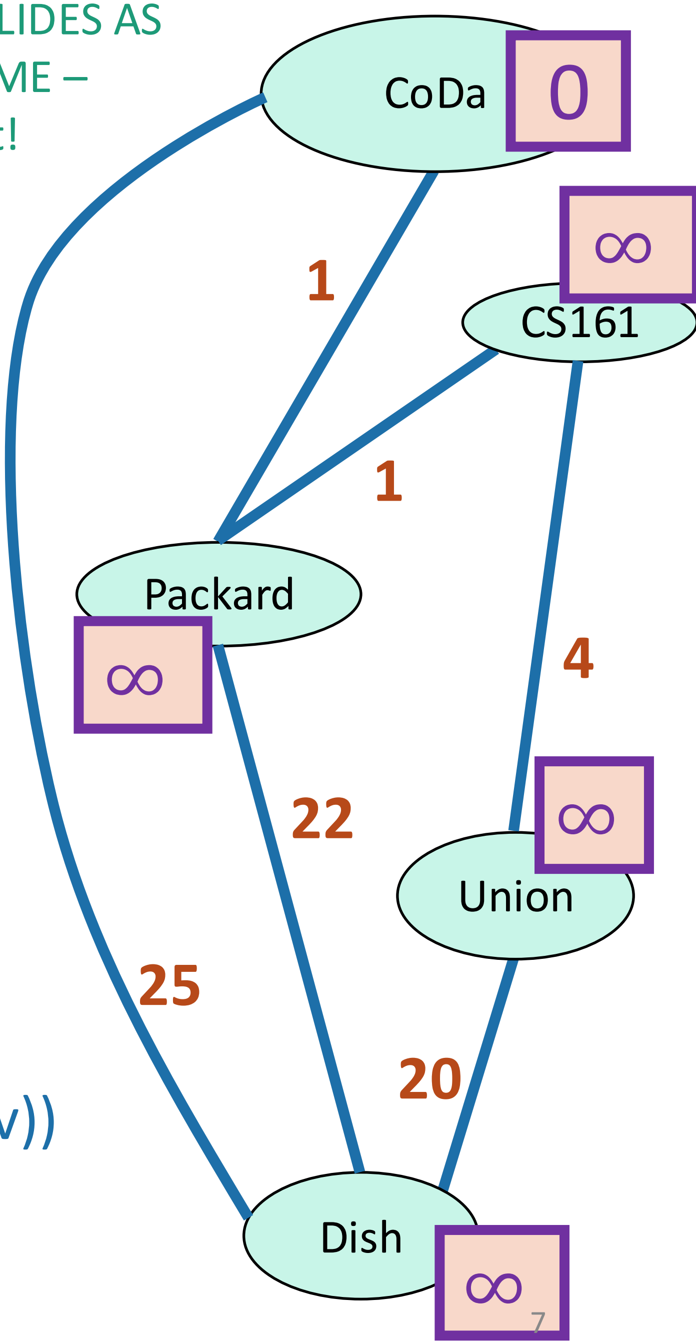
Dijkstra by example

SAME SLIDES AS LAST TIME – real fast!

How far is a node from CoDa?

⚪	I'm not sure yet
🟢	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
🟠	Current node u

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

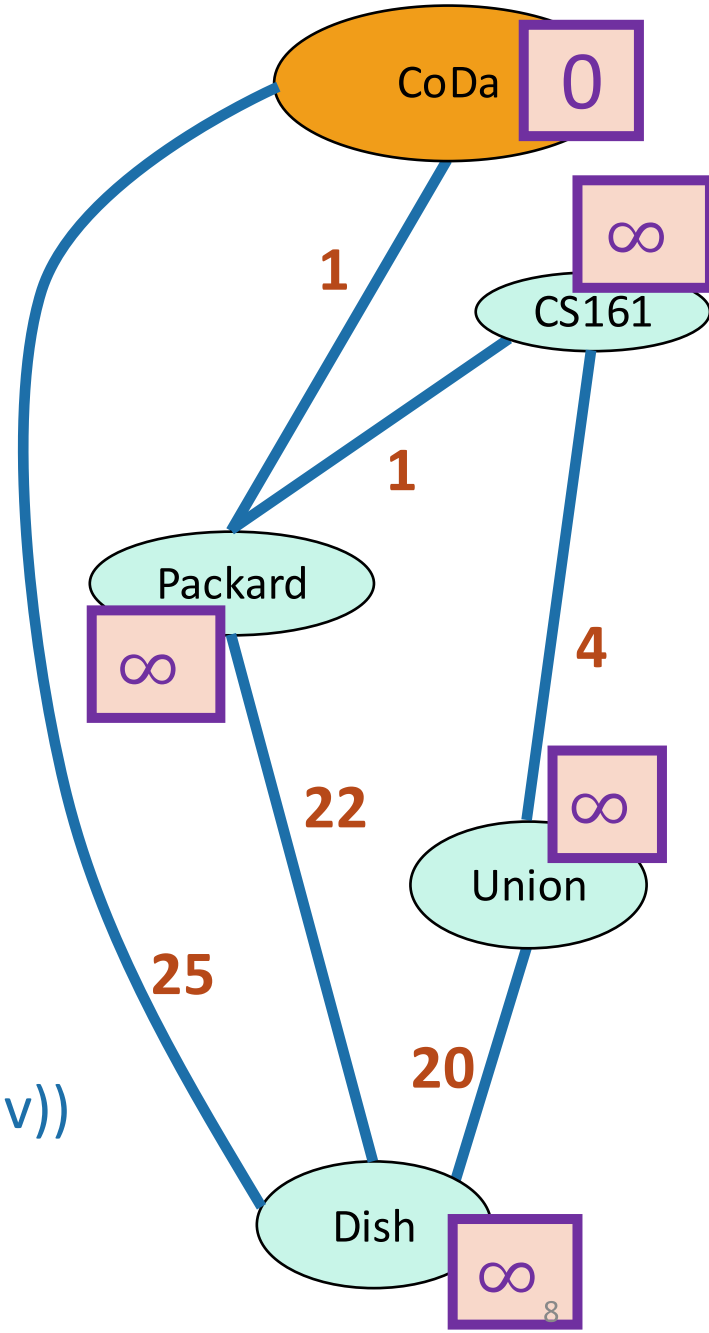


Dijkstra by example

How far is a node from CoDa?

●	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
●	Current node u

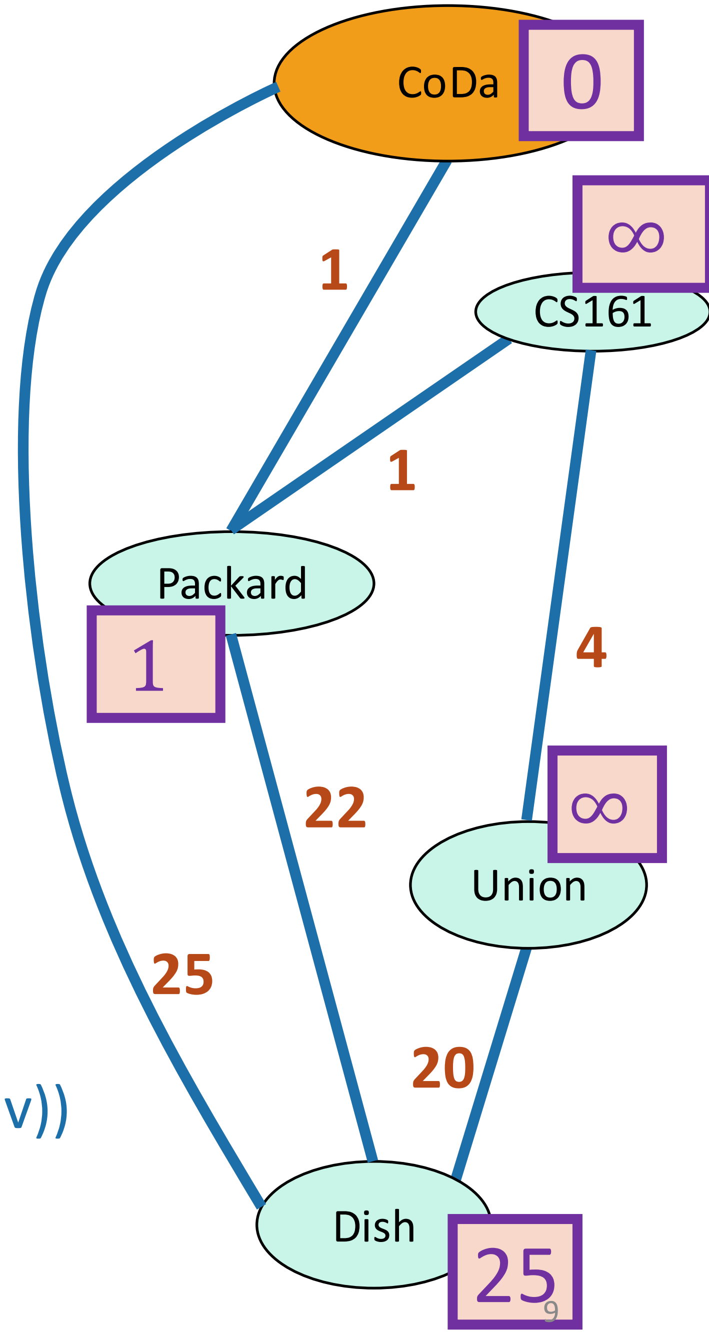
- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat



Dijkstra by example

How far is a node from CoDa?

◯	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
●	Current node u

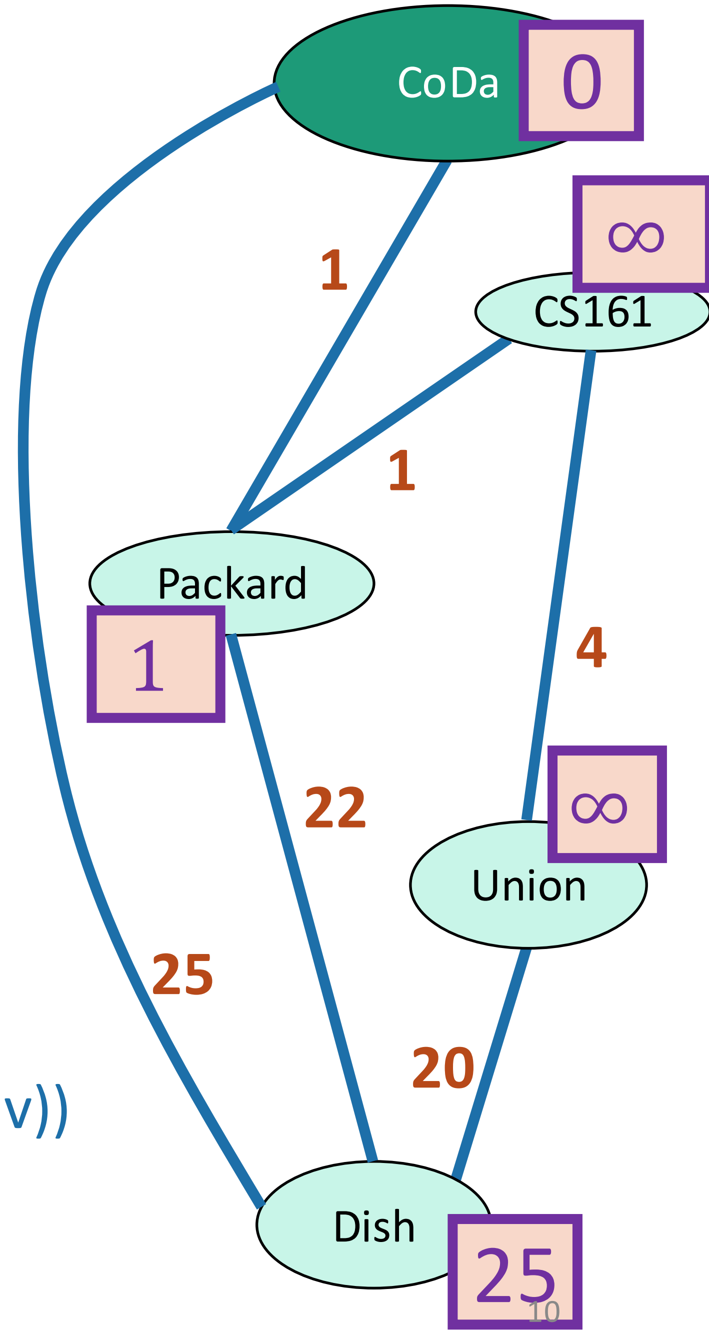


- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

Dijkstra by example

How far is a node from CoDa?

◯	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
●	Current node u

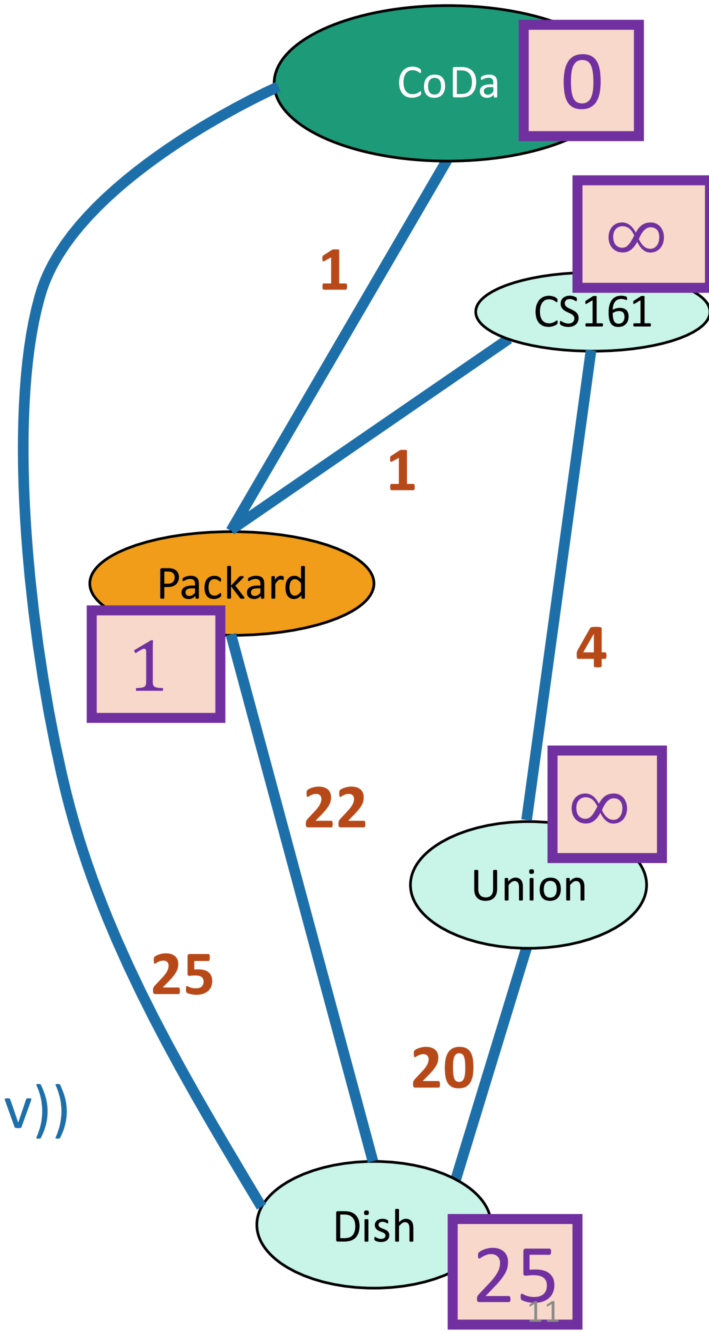


- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

Dijkstra by example

How far is a node from CoDa?

●	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
●	Current node u



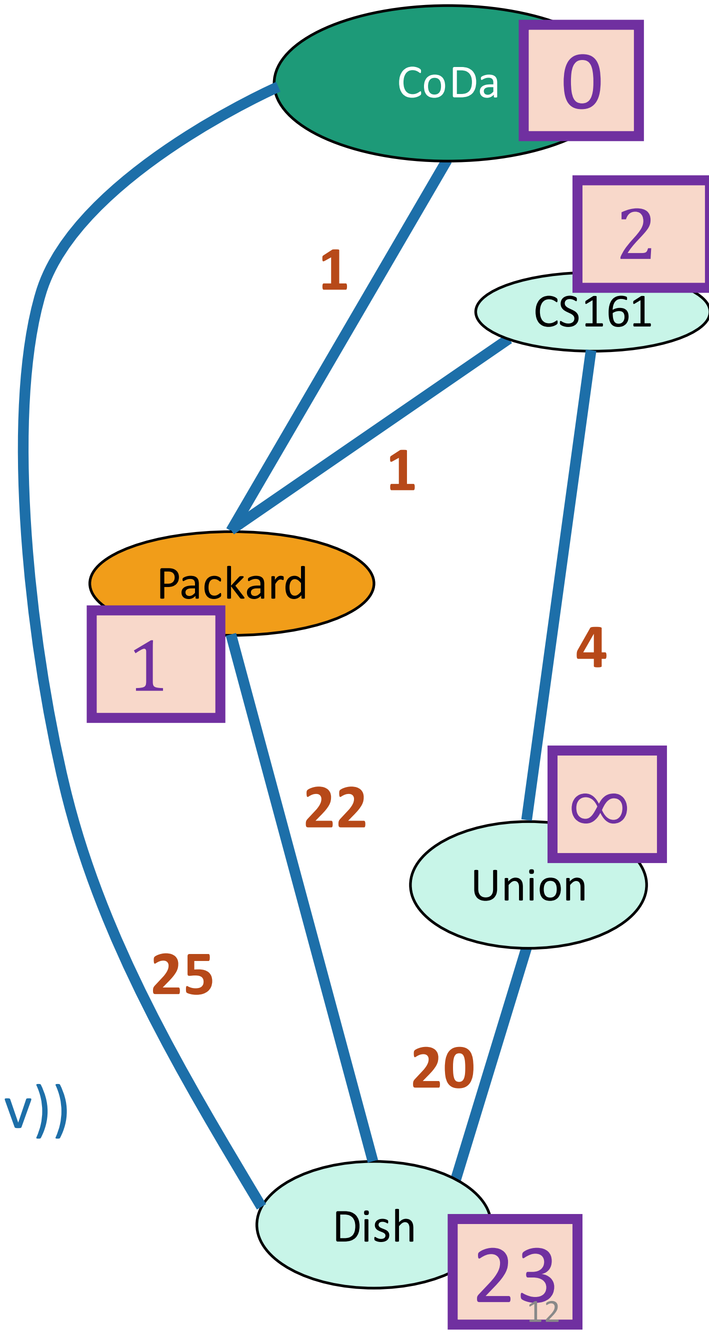
- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

Dijkstra by example

How far is a node from CoDa?

⚪	I'm not sure yet
🟢	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
🟠	Current node u

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

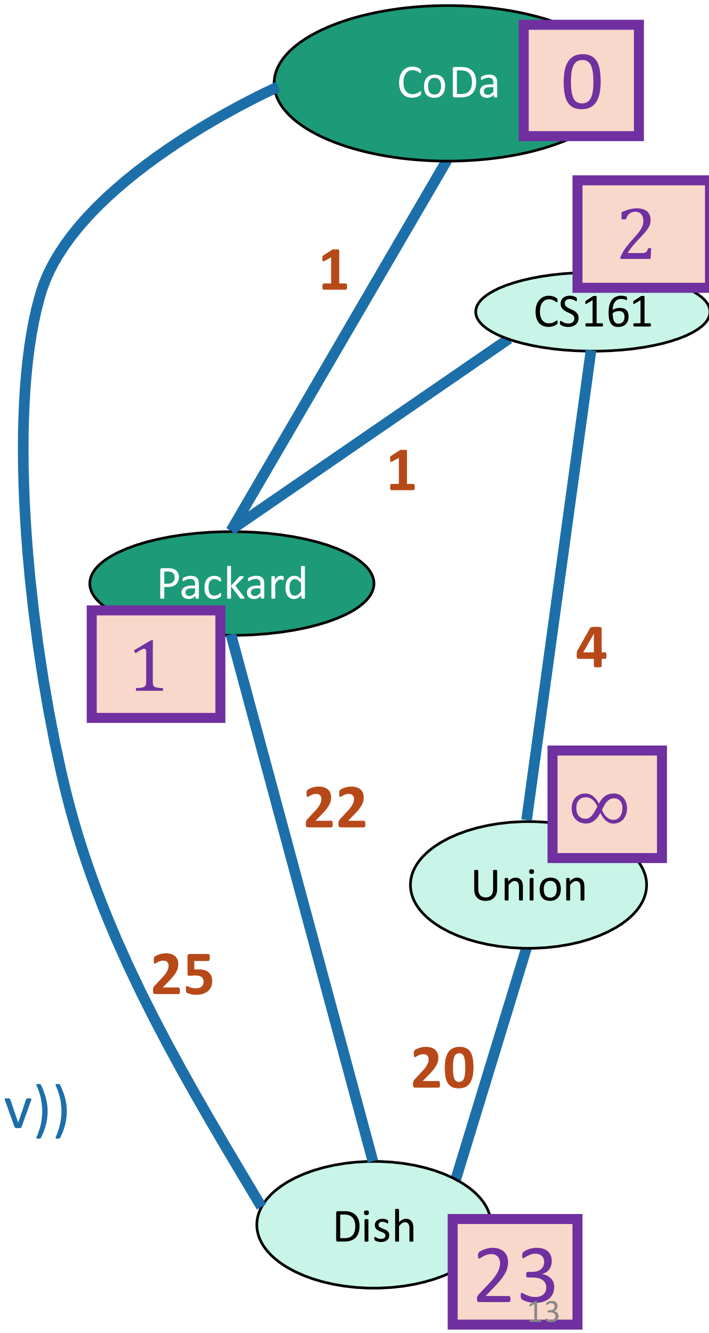


Dijkstra by example

How far is a node from CoDa?

◯	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
●	Current node u

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

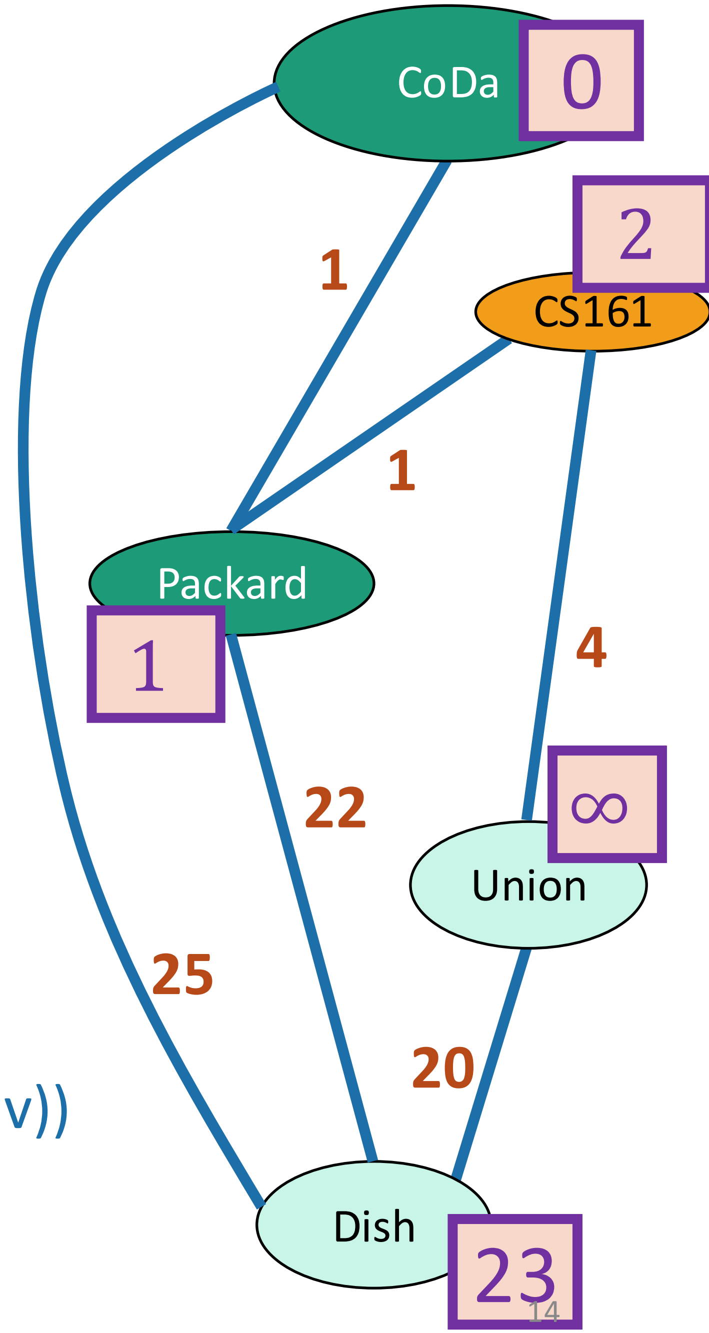


Dijkstra by example

How far is a node from CoDa?

○	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
●	Current node u

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

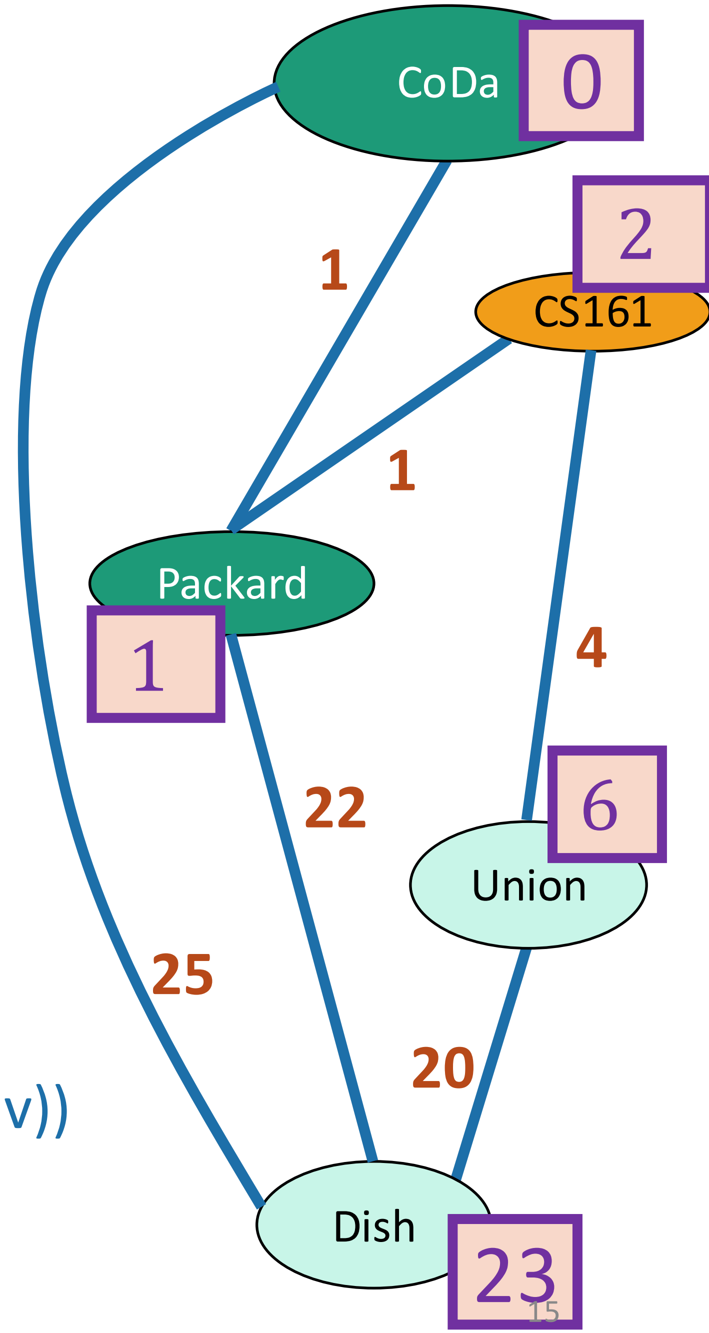


Dijkstra by example

How far is a node from CoDa?

○	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
●	Current node u

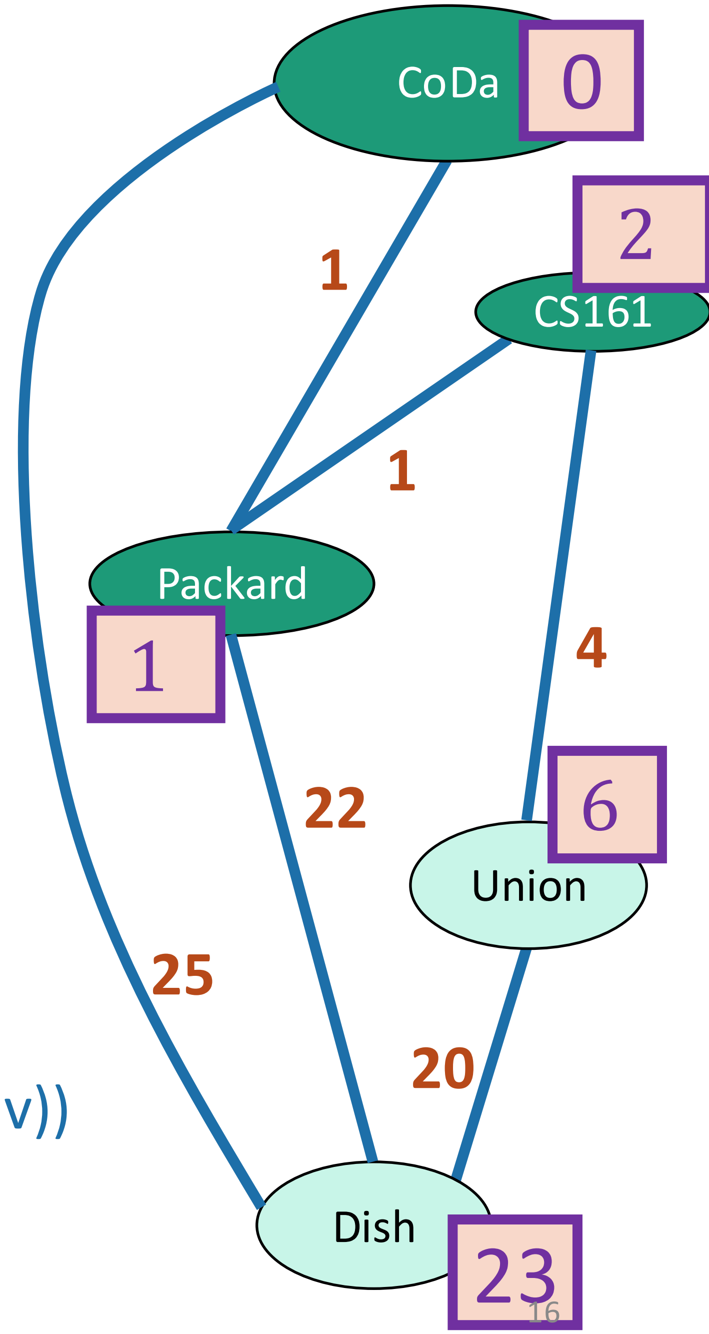
- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat



Dijkstra by example

How far is a node from CoDa?

◯	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
●	Current node u



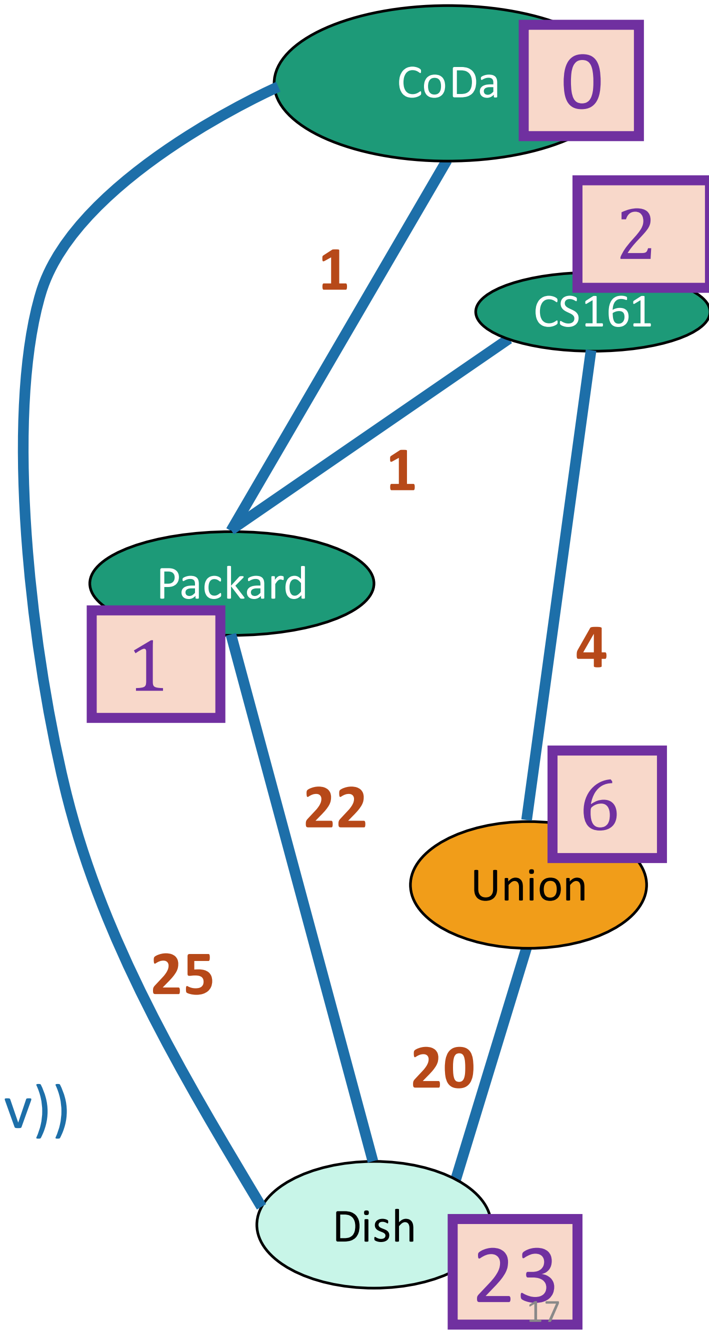
- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

Dijkstra by example

How far is a node from CoDa?

◯	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
●	Current node u

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

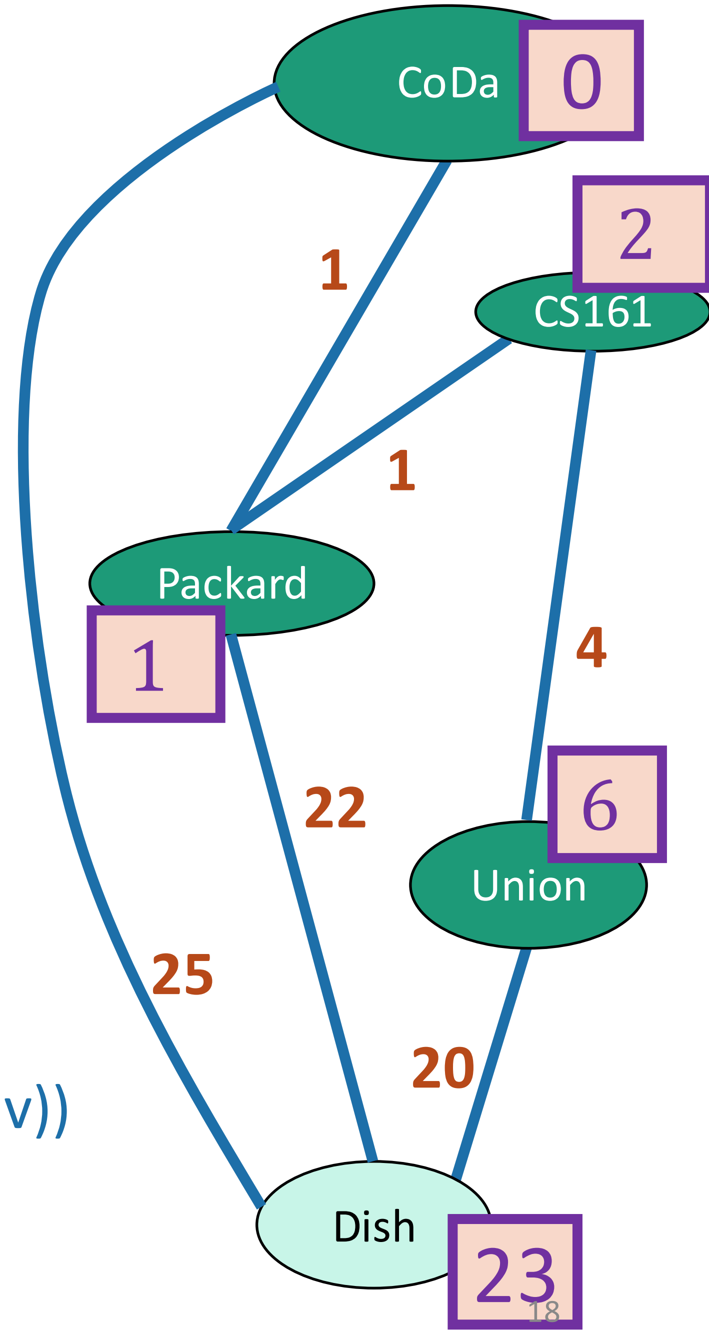


Dijkstra by example

How far is a node from CoDa?

⚪	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
🟠	Current node u

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

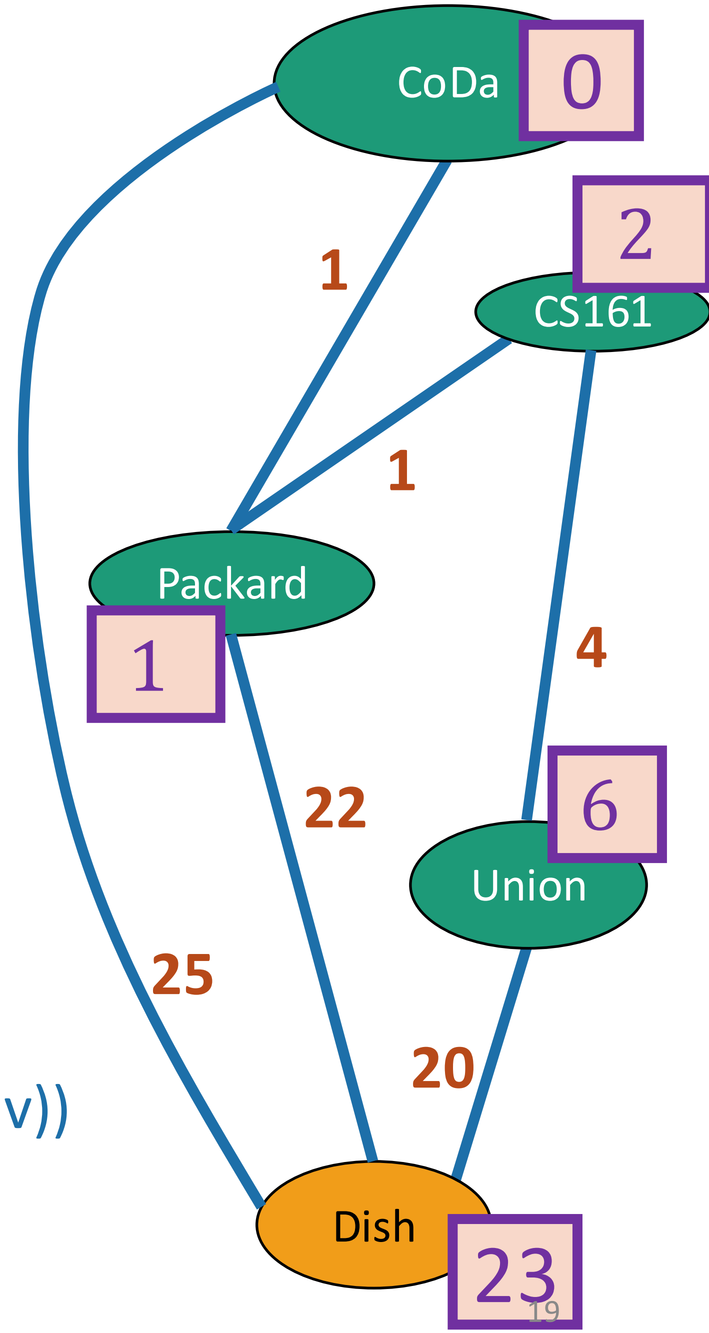


Dijkstra by example

How far is a node from CoDa?

⚪	I'm not sure yet
🟢	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
🟠	Current node u

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

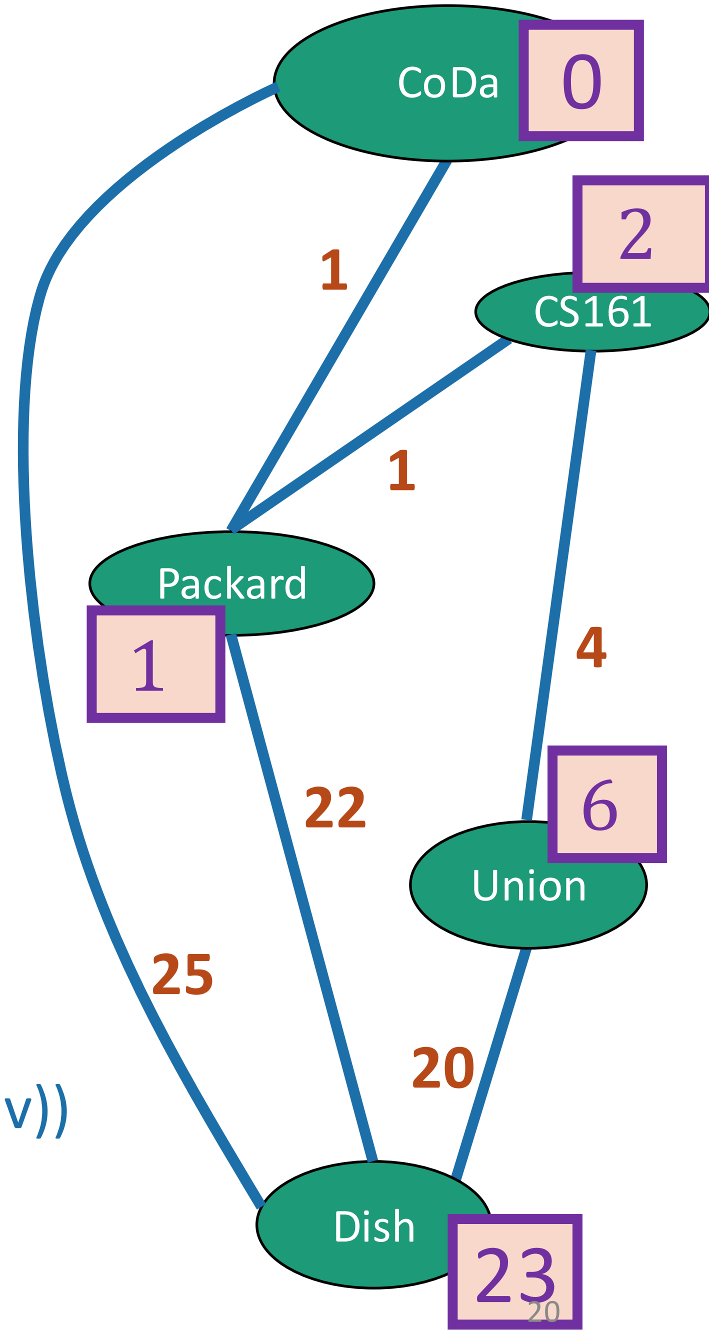


Dijkstra by example

How far is a node from CoDa?

○	I'm not sure yet
●	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
●	Current node u

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat

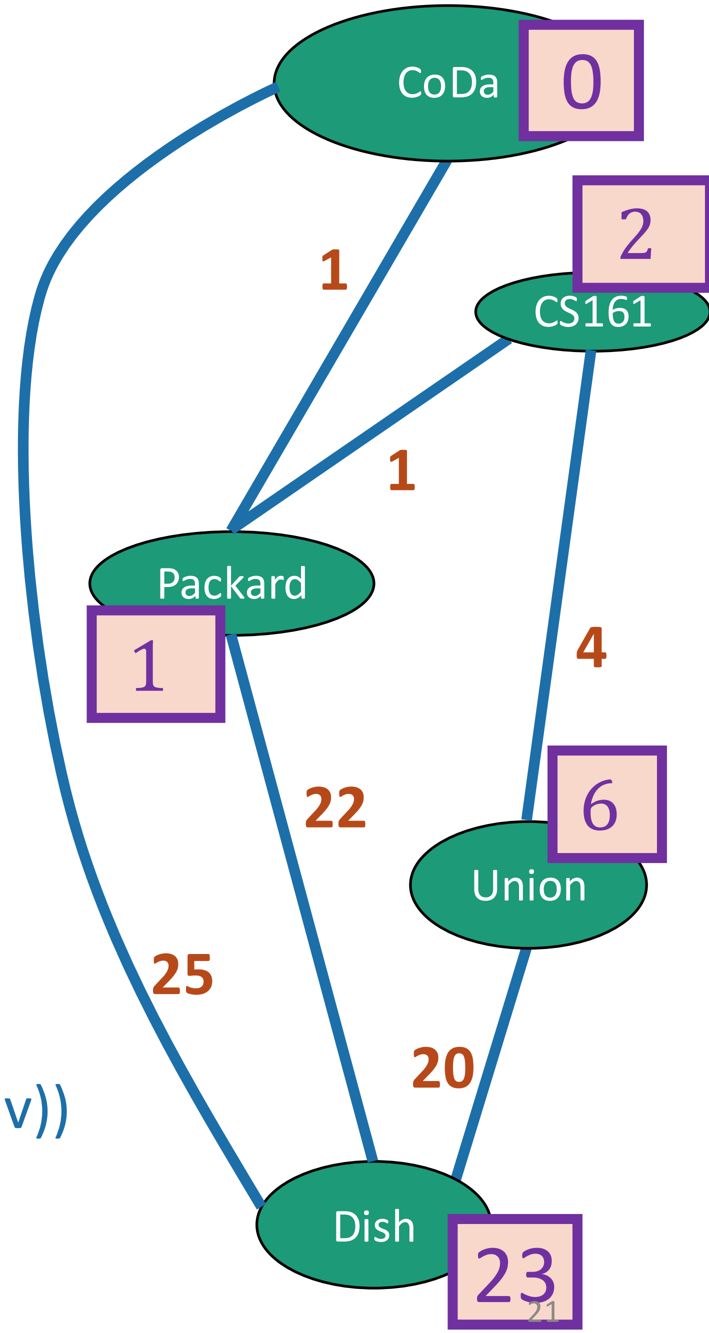


Dijkstra by example

How far is a node from CoDa?

⚪	I'm not sure yet
🟢	I'm sure
$\boxed{x}$	**x = d[v]** is my best **over-estimate** for dist(CoDa,v).
🟠	Current node u

- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat
- After all nodes are **sure**, say that d(CoDa, v) = d[v] for all v



How do we actually implement Dijkstra?

- We need a data structure that holds (key, value) pairs $(v, d[v])$
- That data structure needs to support:
  - FindMin
  - RemoveMin
  - UpdateKey
- We can use a RBTree for $O(\log n)$ time for each of these!
  - $O((n+m)\log n)$ time for Dijkstra
- We can use a *Fibonacci heap* for slightly better amortized performance.
  - $O(n\log n + m)$ time for Dijkstra

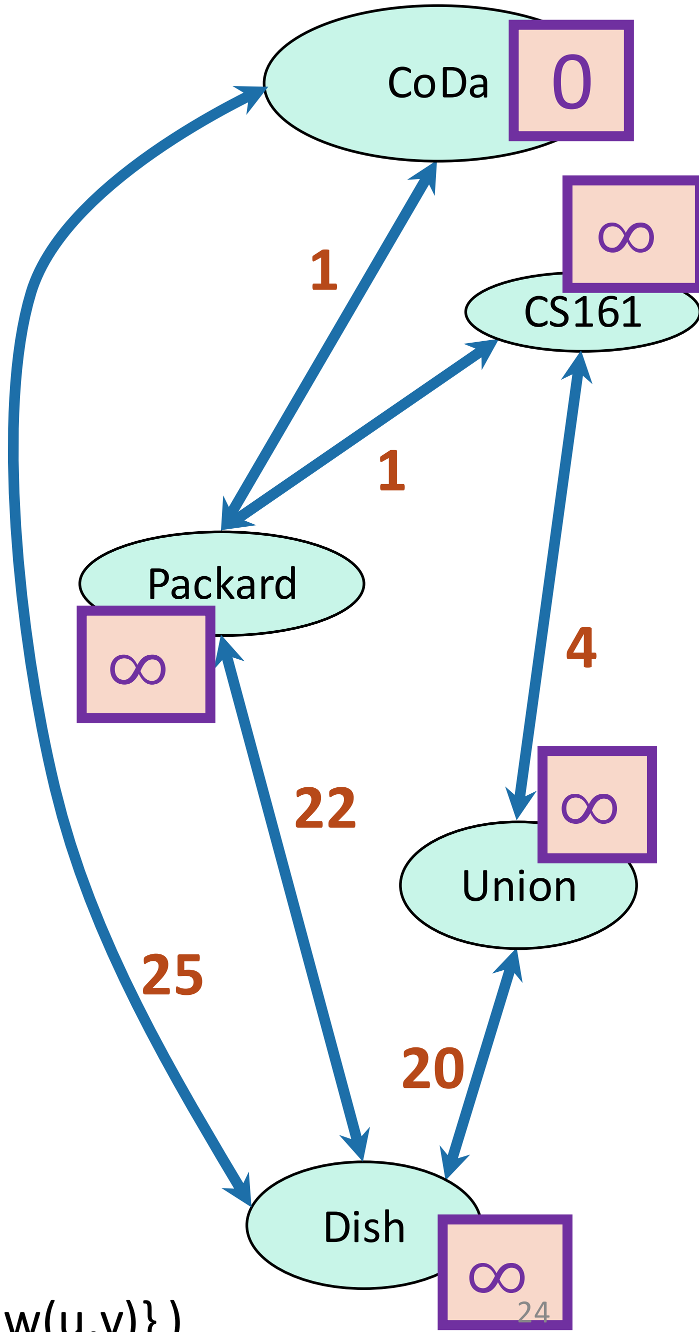
Let’s see another way to do it!

- Bellman-Ford Algorithm!

Bellman-Ford

How far is a node from CoDa?

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

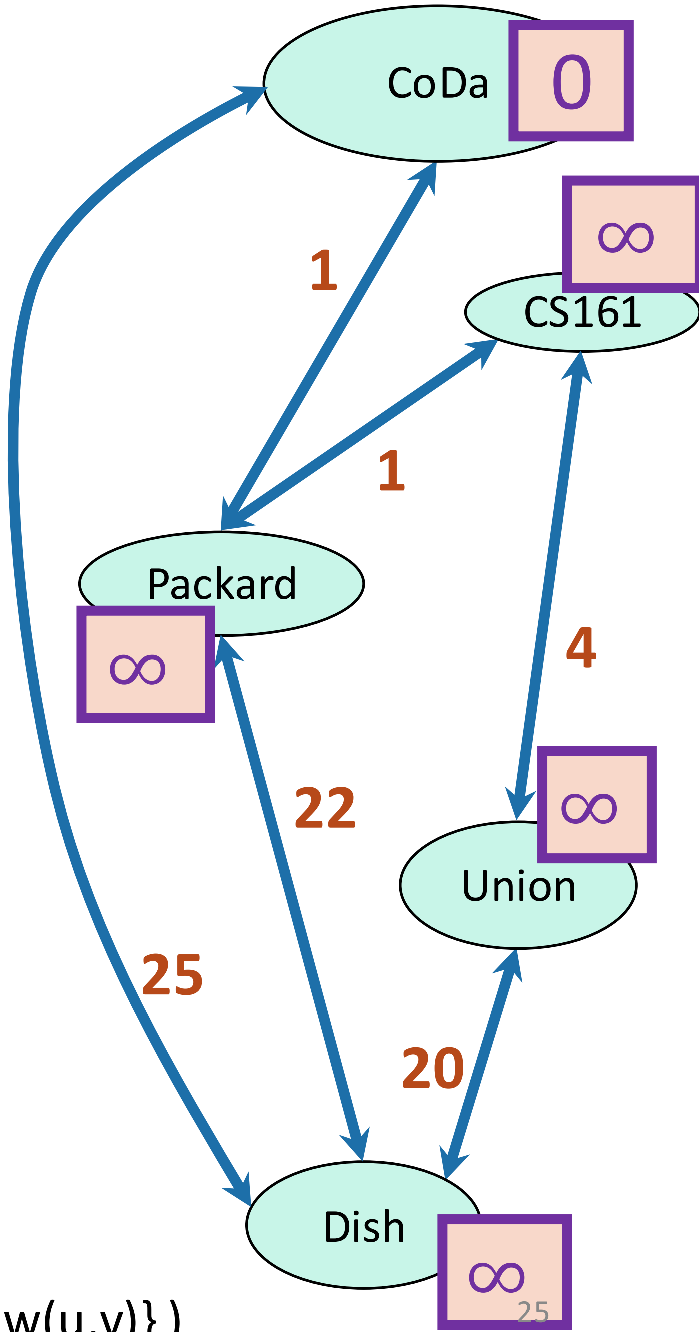


- **For** i=0,…,n-2:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_{u \text{ in } v.\text{inNbrs}}\{d^{(i)}[u] + w(u,v)\})$

Bellman-Ford

How far is a node from CoDa?

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

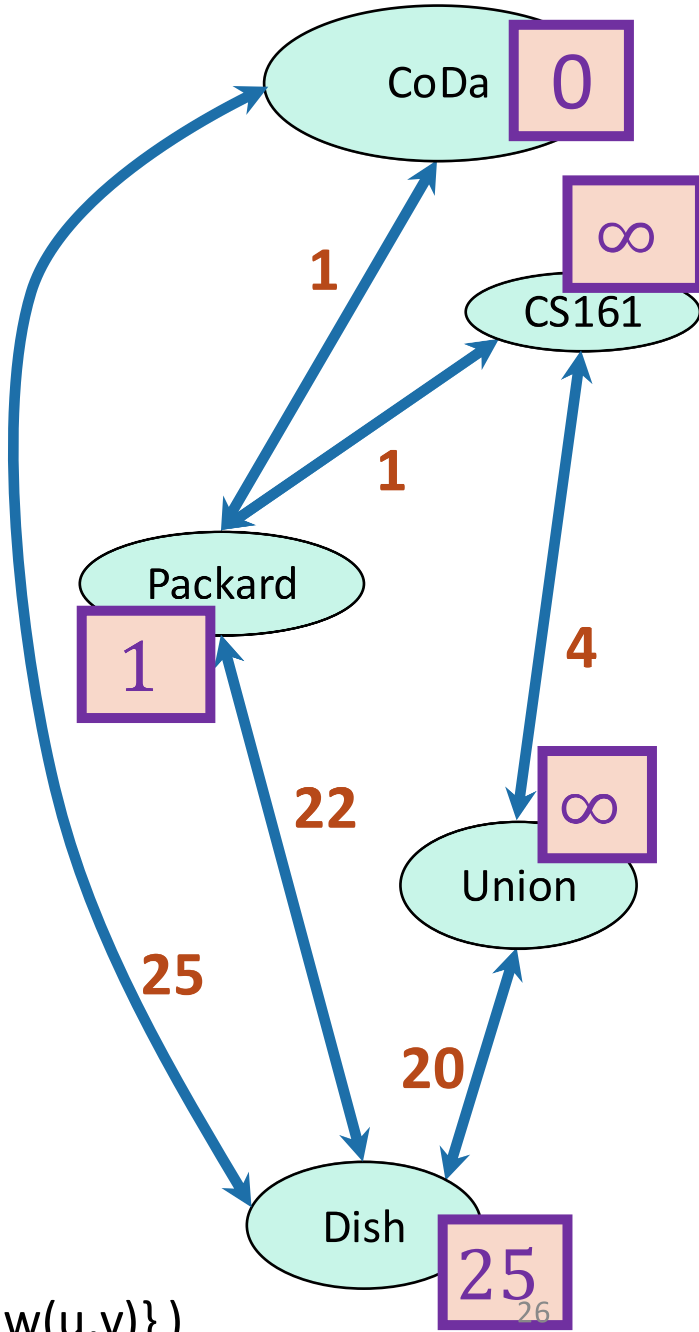


```
• For i=0,…,n-2:
    • For v in V:
        • $d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , \min_{u \text{ in } v.\text{inNbrs}}\{d^{(i)}[u] + w(u,v)\} )$
```

Bellman-Ford

How far is a node from CoDa?

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

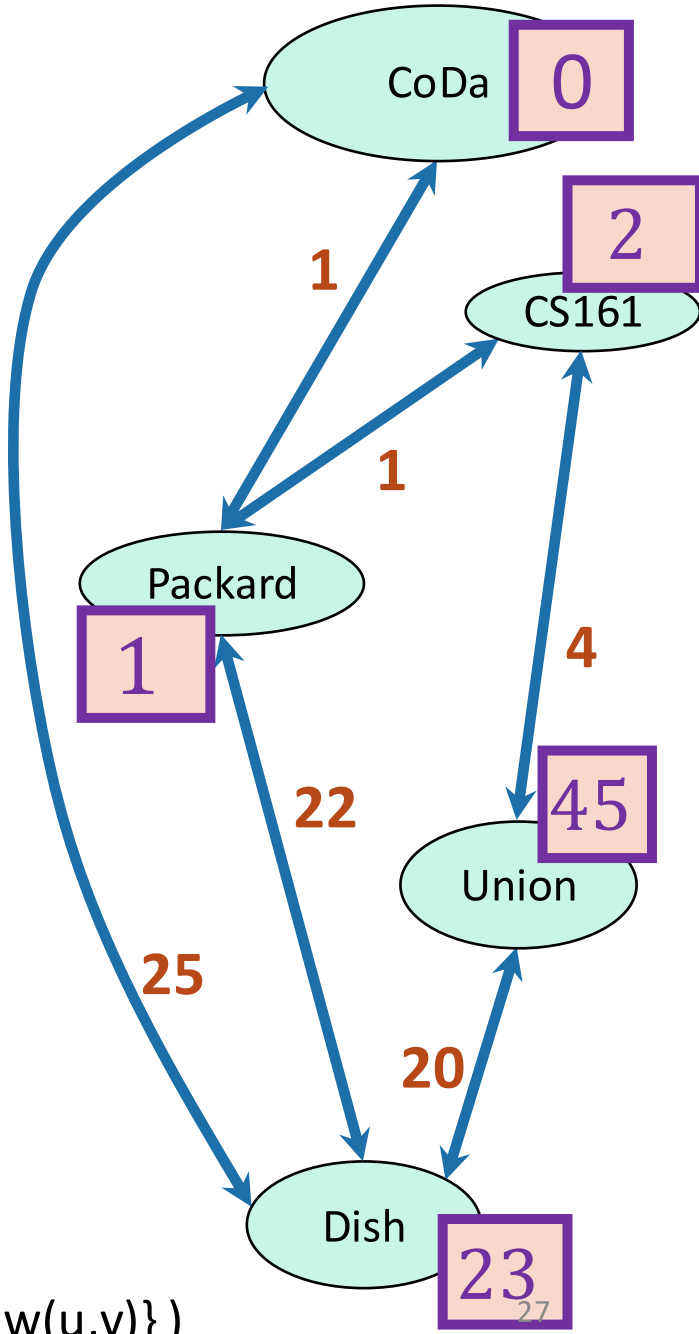


```
• For i=0,…,n-2:
    • For v in V:
        • $d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , \min_{u \text{ in } v.\text{inNbrs}}\{d^{(i)}[u] + w(u,v)\} )$
```

Bellman-Ford

How far is a node from CoDa?

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

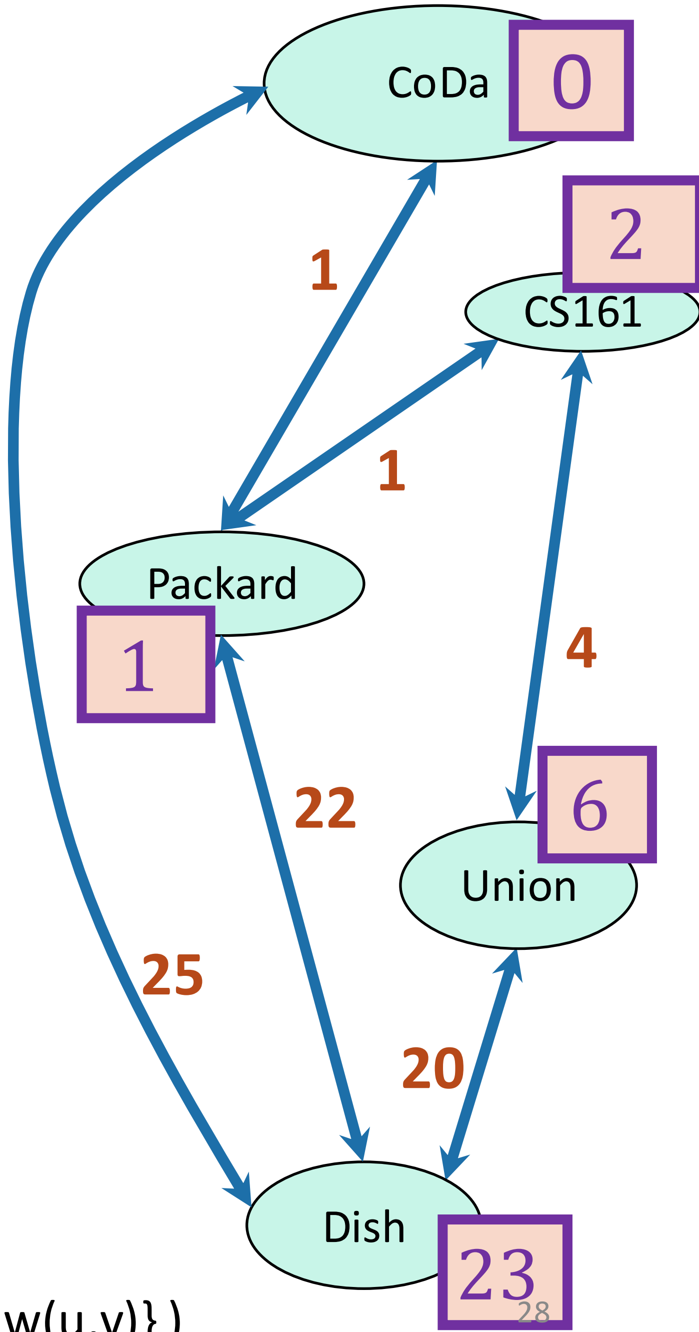


```
• For i=0,…,n-2:
    • For v in V:
        • $d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , \min_{u \text{ in } v.\text{inNbrs}}\{d^{(i)}[u] + w(u,v)\} )$
```

Bellman-Ford

How far is a node from CoDa?

	CoDa	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
$d^{(4)}$					



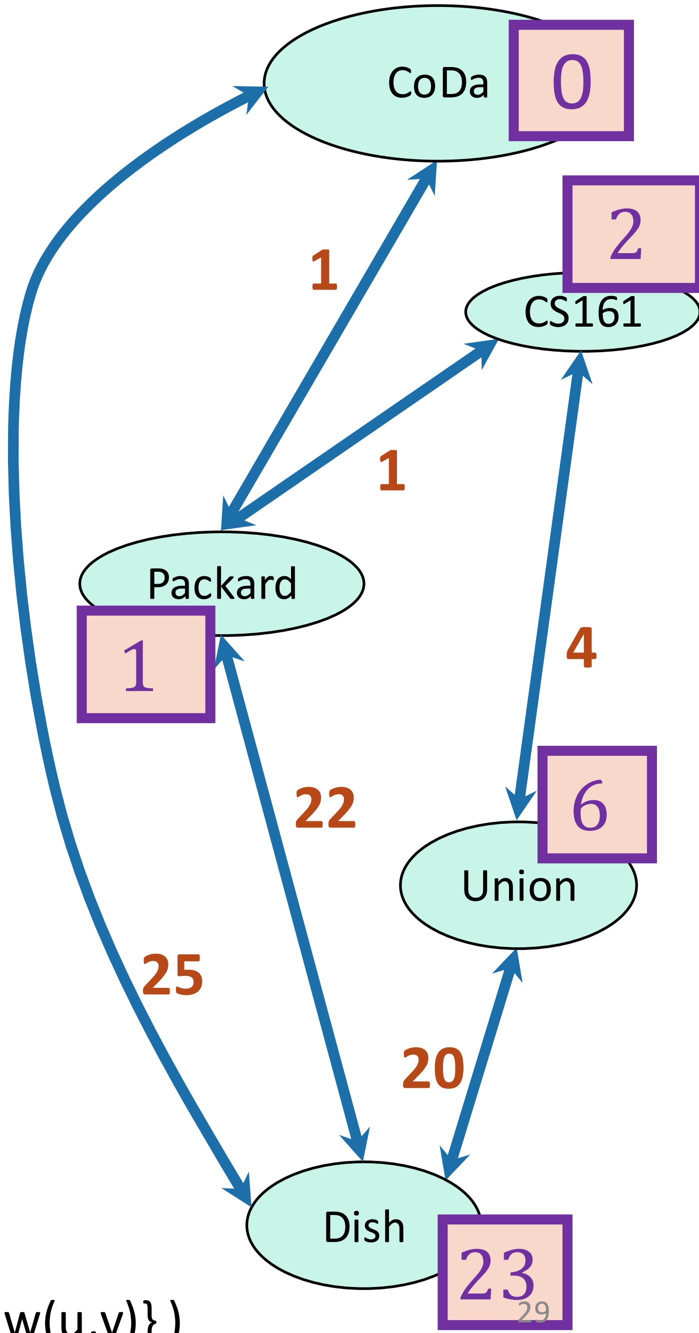
```
• For i=0,…,n-2:
    • For v in V:
        • $d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , \min_{u \text{ in } v.\text{inNbrs}}\{d^{(i)}[u] + w(u,v)\} )$
```

Bellman-Ford

How far is a node from CoDa?

	CoDa	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

These are the final distances!



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

G = (V,E) is a graph with n vertices and m edges.

Bellman-Ford* algorithm

Bellman-Ford*(G,s):

```
• Initialize arrays $d^{(0)},\ldots,d^{(n-1)}$ of length n
• $d^{(0)}[v] = \infty$ for all v in V
• $d^{(0)}[s] = 0$
• For i=0,…,n-2:
    • For v in V:
        • $d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , \min_{u \text{ in } v.\text{inNbrs}}\{d^{(i)}[u] + w(u,v)\} )$
• Now, $\text{dist}(s,v) = d^{(n-1)}[v]$ for all v in V.
    • (Assuming no negative cycles)
```

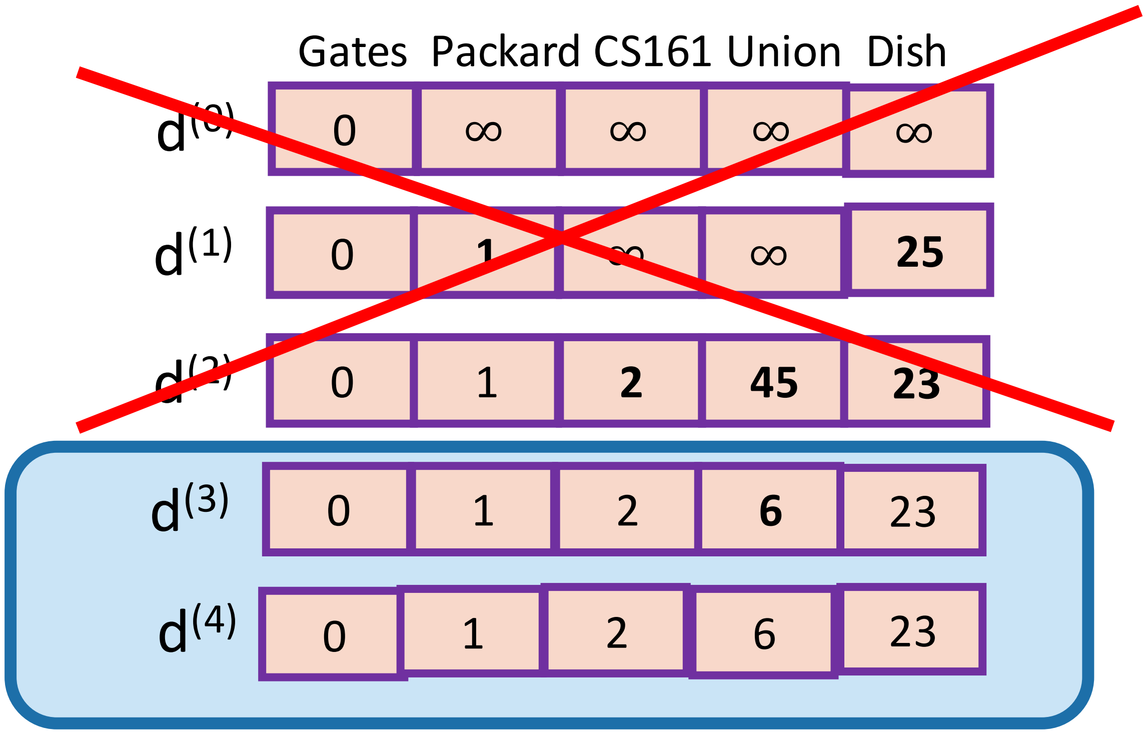
Here, Dijkstra picked a special vertex u and updated u's neighbors – Bellman-Ford will update *all* the vertices.

*Slightly different than some versions of Bellman-Ford…but this way is pedagogically convenient for today's lecture.

Note on implementation

to save on space

- 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 $d^{(4)}$

Bellman-Ford Algorithm

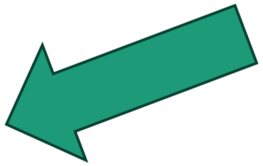
- Does it work?

- Is it fast?

Bellman-Ford Algorithm

- Does it work?

- Is it fast?



$O(nm)$

So… not that fast?

Compare to Dijkstra (with an RBTree):

$O((n+m)\log n)$

Technically, as written the running time would be $O(n(n+m))$… can you see how to modify the algorithm to solve the single-source-shortest-path problem in time $O(nm)$?

Siggi the Studious Stork

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

	CoDa	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 Algorithm

- Does it work? **Yes!**

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

- Is it fast?

$O(nm)$

So… not that fast?

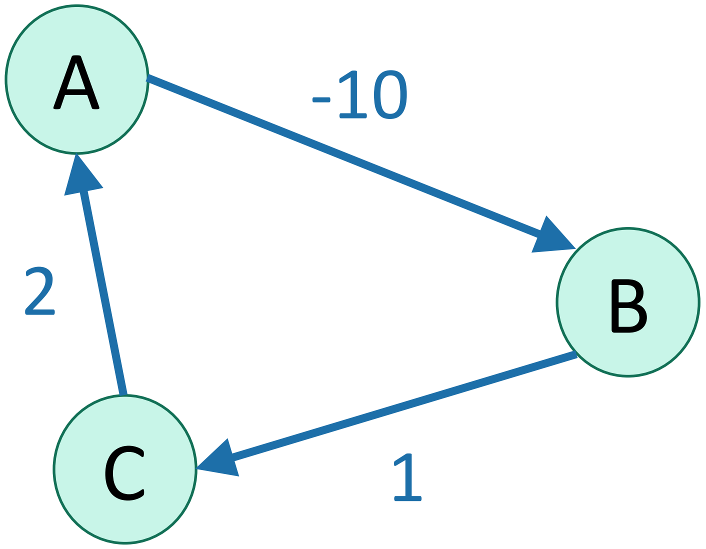
Compare to Dijkstra (with an RBTree):

$O((n+m)\log n)$

```
• For i=0,…,n-2:
    • For u in V:
        • For v in u.neighbors:
            • $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
```

Aside: Negative Cycles

- A **negative cycle** is a cycle whose edge weights sum to a negative number.
- Shortest paths aren't defined when there are negative cycles!



The shortest path from A to B has cost…negative infinity?

Bellman-Ford Algorithm

	CoDa	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

- Does it work? **Yes!**

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

Corollary: If there are no negative cycles, $d^{(n-1)}[v]$ is the cost of the shortest path between s and v.

- Is it fast?

$O(nm)$

So… not really?

Compare to Dijkstra (with an RBTree):
$O((n+m)\log n)$

Some proof required to get from the Lemma to the Corollary!

- If there are no negative cycles, there is a *shortest* path from s to v with at most $n-1$ edges (why?)

```
• For i=0,…,n-2:
    • For u in V:
        • For v in u.neighbors:
            • $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v], d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$
```

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

Detecting Negative Cycles

- Bellman-Ford can be adapted to **detect** negative cycles.

- If there are no negative cycles, after n iterations, the distance estimates should stop changing.

- If there is a negative cycle, then the distances will keep updating (down to $-\infty$)

- So run one more iteration of BF (to compute $d^{(n)}$ as well as $d^{(n-1)}$), and see if they are the same or not.

	CoDa	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
$d^{(5)}$	0	1	2	6	23

No negative cycles, and these are the distances!

Summary

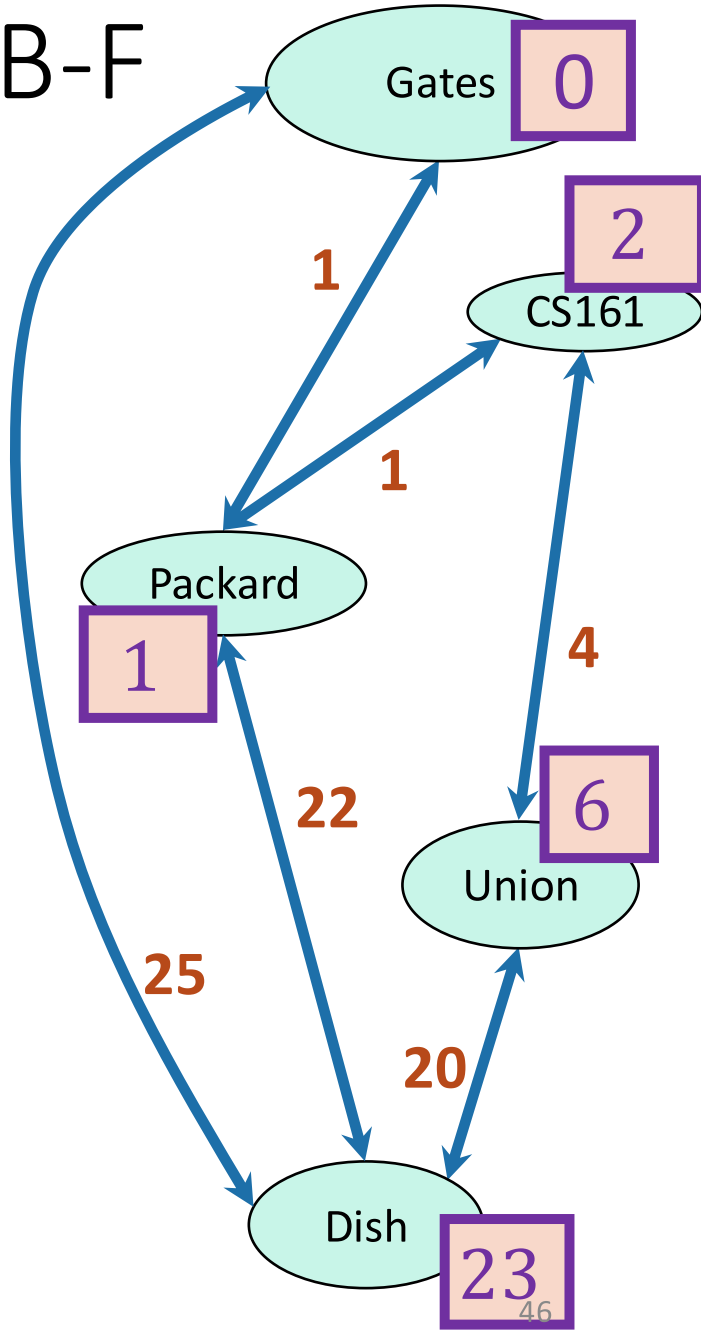
Bellman-Ford vs. Dijkstra

- Both solve **single-source shortest path**
- Dijkstra:
  - Find the u with the smallest d[u]
  - Update u's neighbors: d[v] = min(d[v], d[u] + w(u,v))
  - Runs in time $O\big((n+m)\log n\big)$ with a RBTree.
    - Or $O(n\log n + m)$ amortized time with a Fibonacci heap
- Bellman-Ford:
  - Don't bother finding the u with the smallest d[u]
    - Everyone updates!
  - Slower -- $O(nm)$ -- but more flexible:
    - Can handle negative edge weights (as long as there aren't negative cycles)
    - Can do updates in a decentralized way.

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

For all vertices v, $d^{(i)}[v]$ 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

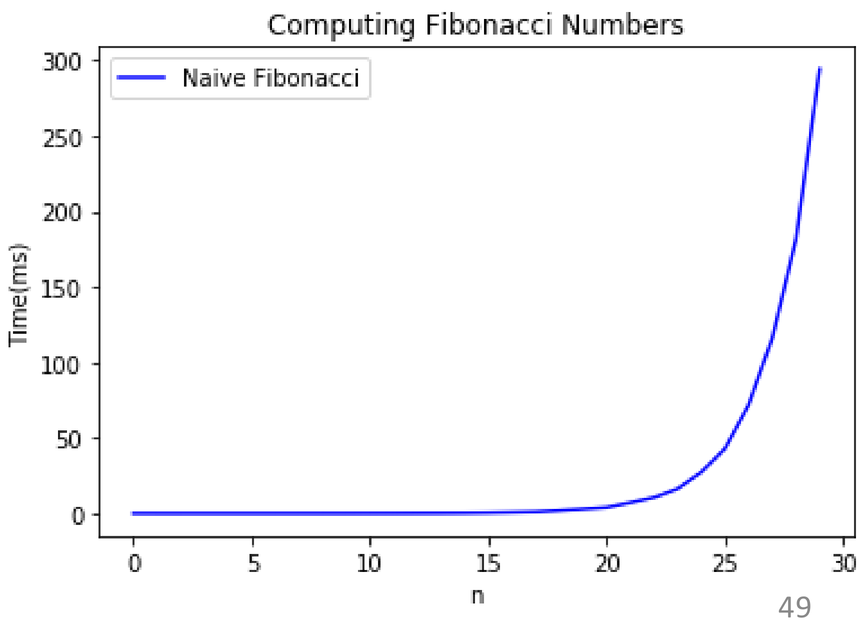
- Definition:
  - F(n) = F(n-1) + F(n-2), with F(1) = F(2) = 1.
  - The first several are:
    - 1
    - 1
    - 2
    - 3
    - 5
    - 8
    - 13, 21, 34, 55, 89, 144,…
- Question:
  - Given n, what is F(n)?

Candidate algorithm

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

Running time?

- $T(n) = T(n-1) + T(n-2) + O(1)$
- $T(n) \geq T(n-1) + T(n-2)$ for $n \geq 2$
- So $T(n)$ grows *at least* as fast as the Fibonacci numbers themselves…
- This is **EXPONENTIALLY QUICKLY!**



See IPython notebook for lecture 12

Slide skipped in class

Why do the Fibonacci numbers grow exponentially quickly?

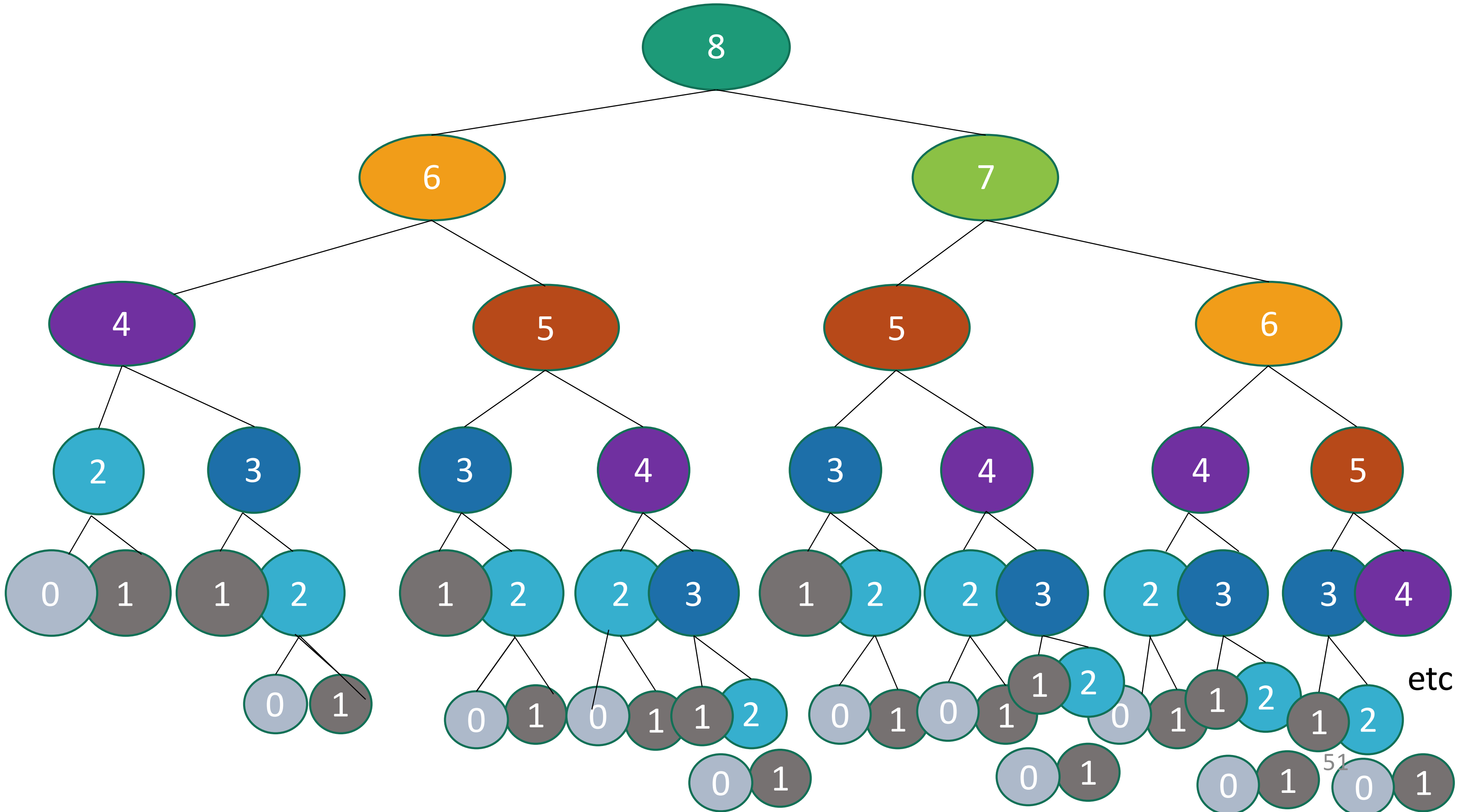
- $T(n) = T(n-1) + T(n-2)$
- $\qquad \geq 2T(n-2)$
- Try unrolling this:
- $T(n) \geq 2T(n-2)$
- $\qquad \geq 4T(n-4)$
- $\qquad \geq 8T(n-6)$
- … $\qquad \geq 2^j T(n-2j)$ for any $j < n/2$
- … $\qquad \geq 2^{n/2} T(1)$ by plugging in $j = \frac{n-1}{2}$
- So $\boldsymbol{T(n) \geq 2^{n/2}}$, which is REALLY BIG!!!

To be really precise, we could use induction! Also here we are assuming n is odd for convenience.

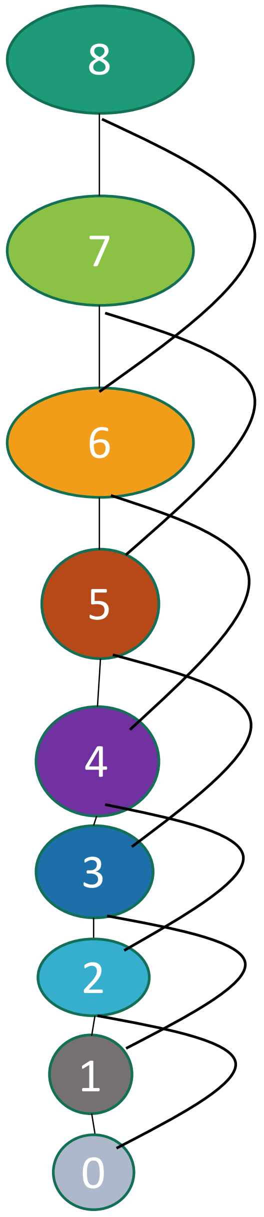
What's going on?

Consider Fib(8)

That's a lot of repeated computation!

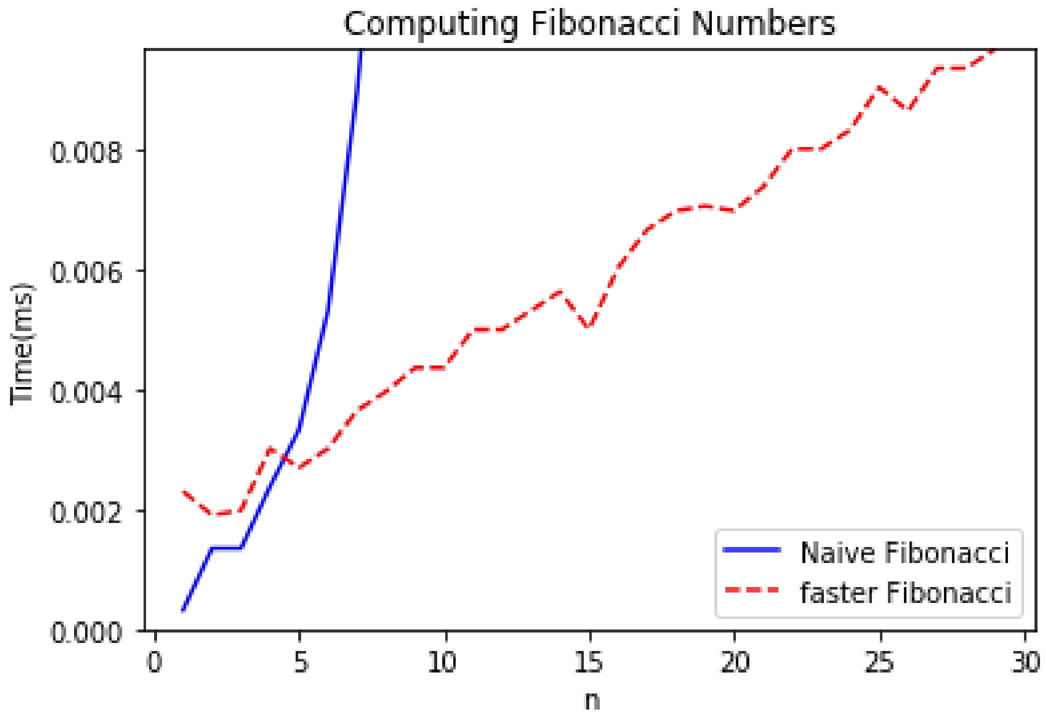


Maybe this would be better:



```
def fasterFibonacci(n):
    • F = [0, 1, None, None, …, None ]
        • \\ F has length n + 1
    • 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.
- Useful for solving **optimization problems**
    - eg, ***shortest*** path
    - (Fibonacci numbers aren’t an optimization problem, but they are a good example of DP anyway…)
- Also useful for counting problems
    - You’ll see some examples on HW6

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 $i < n$
- The optimal solution to a problem can be expressed in terms of optimal solutions to smaller sub-problems.
  - Fibonacci:

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

  - Bellman-Ford:

$$d^{(i+1)}[v] \leftarrow \min\{ \underbrace{d^{(i)}[v]}_{\text{Shortest path with at most i edges from s to v}},\ \min_u \{\underbrace{d^{(i)}[u]}_{\text{Shortest path with at most i edges from s to u.}} + \text{weight}(u,v)\} \}$$

*The word "optimal" makes sense in the context of optimization problems like shortest path, and is why this is called "Optimal Sub-structure."

Elements of dynamic programming

2. Overlapping sub-problems:

- The sub-problems overlap.
  - Fibonacci:
    - Both F[i+1] and F[i+2] directly use F[i].
    - And lots of different F[i+x] indirectly use F[i].
  - Bellman-Ford:
    - Many different entries of $d^{(i+1)}$ will directly use $d^{(i)}[v]$.
    - And lots of different entries of $d^{(i+x)}$ will indirectly 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 can be used to find 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.

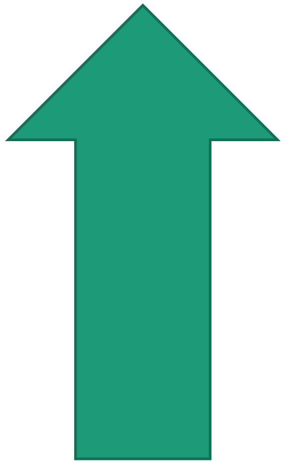
Two ways to think about and/or implement DP algorithms

- Top down

- Bottom up

Bottom up approach

what we just saw.

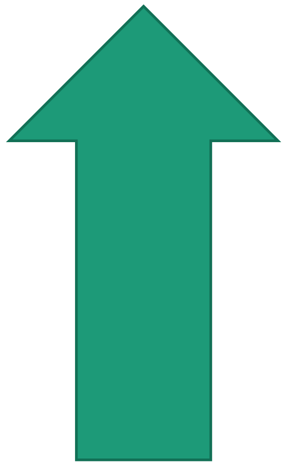


- For Fibonacci:
- Solve the small problems first
    - fill in F[0],F[1]
- Then bigger problems
    - fill in F[2]
- …
- Then bigger problems
    - fill in F[n-1]
- Then finally solve the real problem.
    - fill in F[n]

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

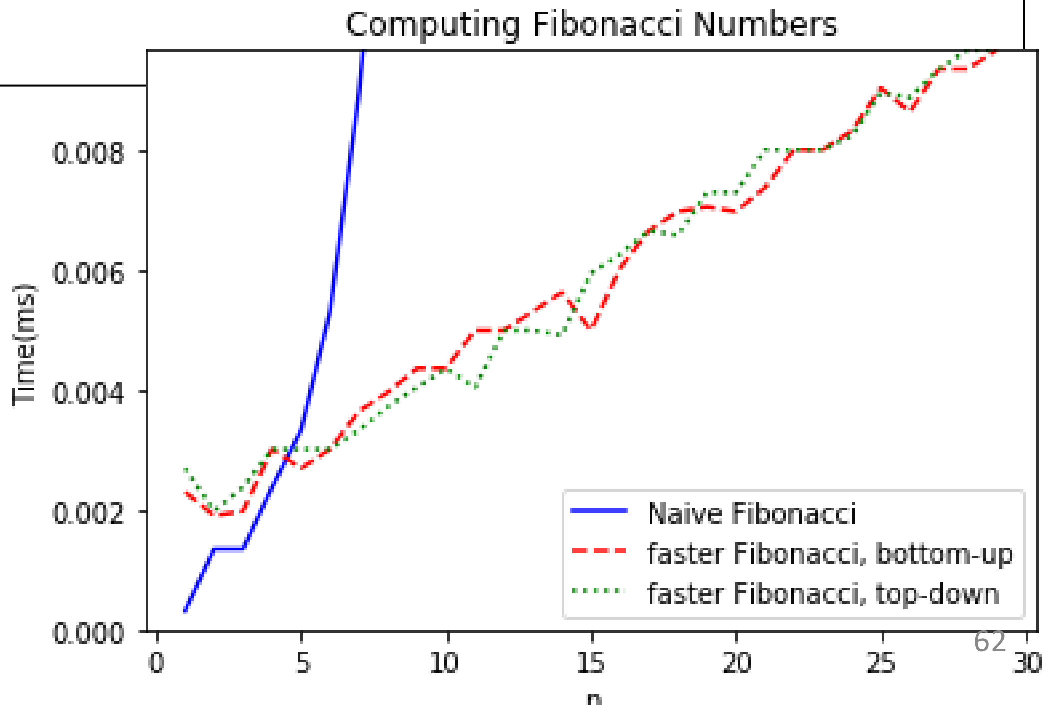
- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
      - etc..

- The difference from divide and conquer:
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.
  - Aka, "**memo-ization**"

Example of top-down Fibonacci

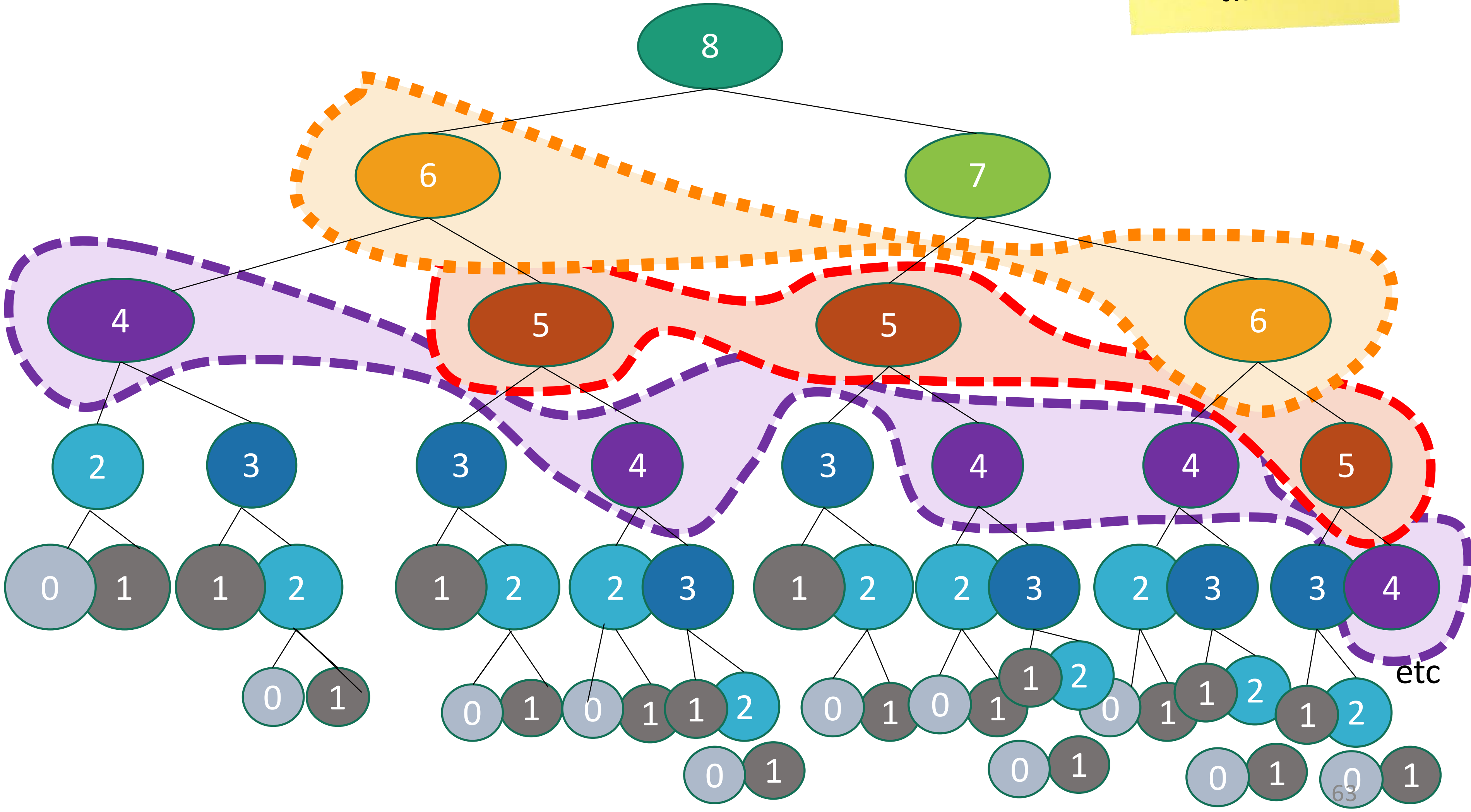
```
• define a global list F = [0,1,None, None, …, None]
• def Fibonacci(n):
    • if F[n] != None:
        • return F[n]
    • else:
        • F[n] = Fibonacci(n-1) + Fibonacci(n-2)
    • return F[n]
```

Memo-ization:
Keeps track (in F) of the stuff you've already done.



Memo-ization visualization

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

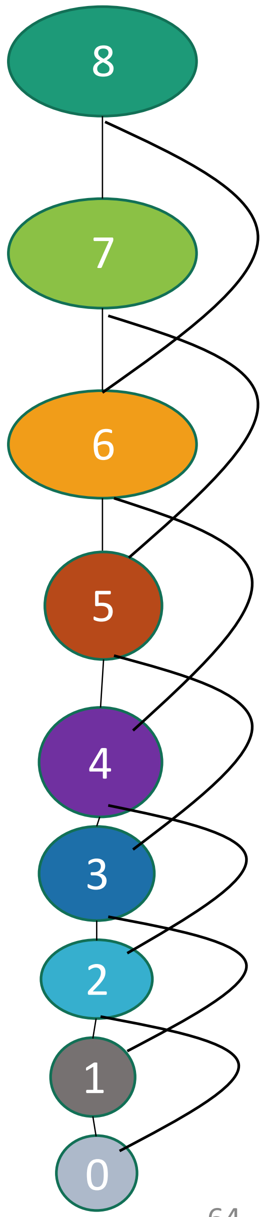


Memo-ization Visualization
ctd

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

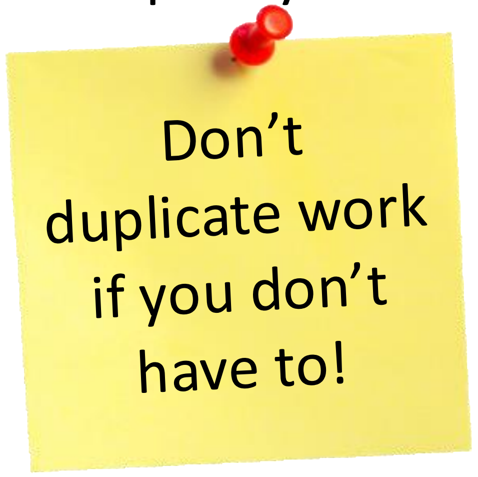
But otherwise treat it like the same old recursive algorithm.

```
• define a global list F = [0,1,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:


Don't duplicate work if you don't have to!

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

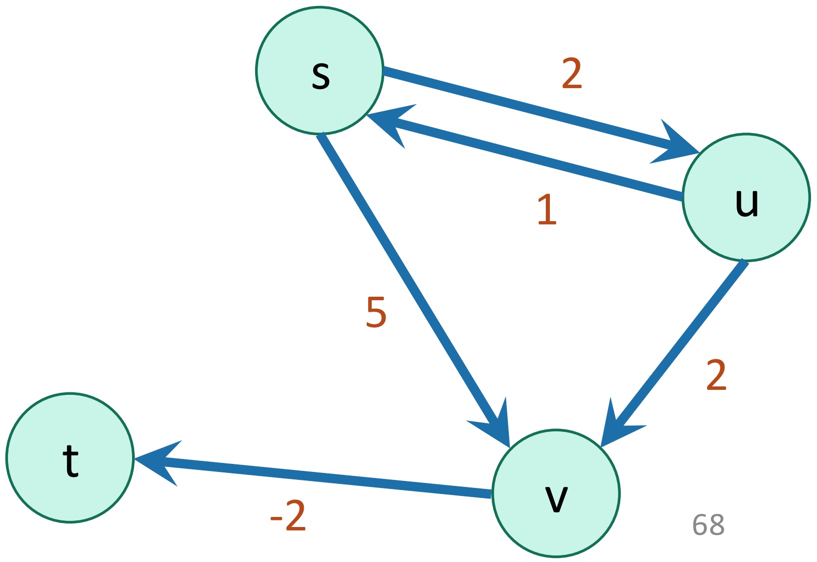
Another example of DP

- Solves **All-Pairs Shortest Paths** (APSP)
  - Goal in APSP: find 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

- Solves **All-Pairs Shortest Paths (**APSP)
  - Goal in APSP: find shortest path from u to v for **ALL pairs** u,v of vertices in the graph.

- Straightforward 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 $\Omega(n^4)$ if $m \approx n^2$

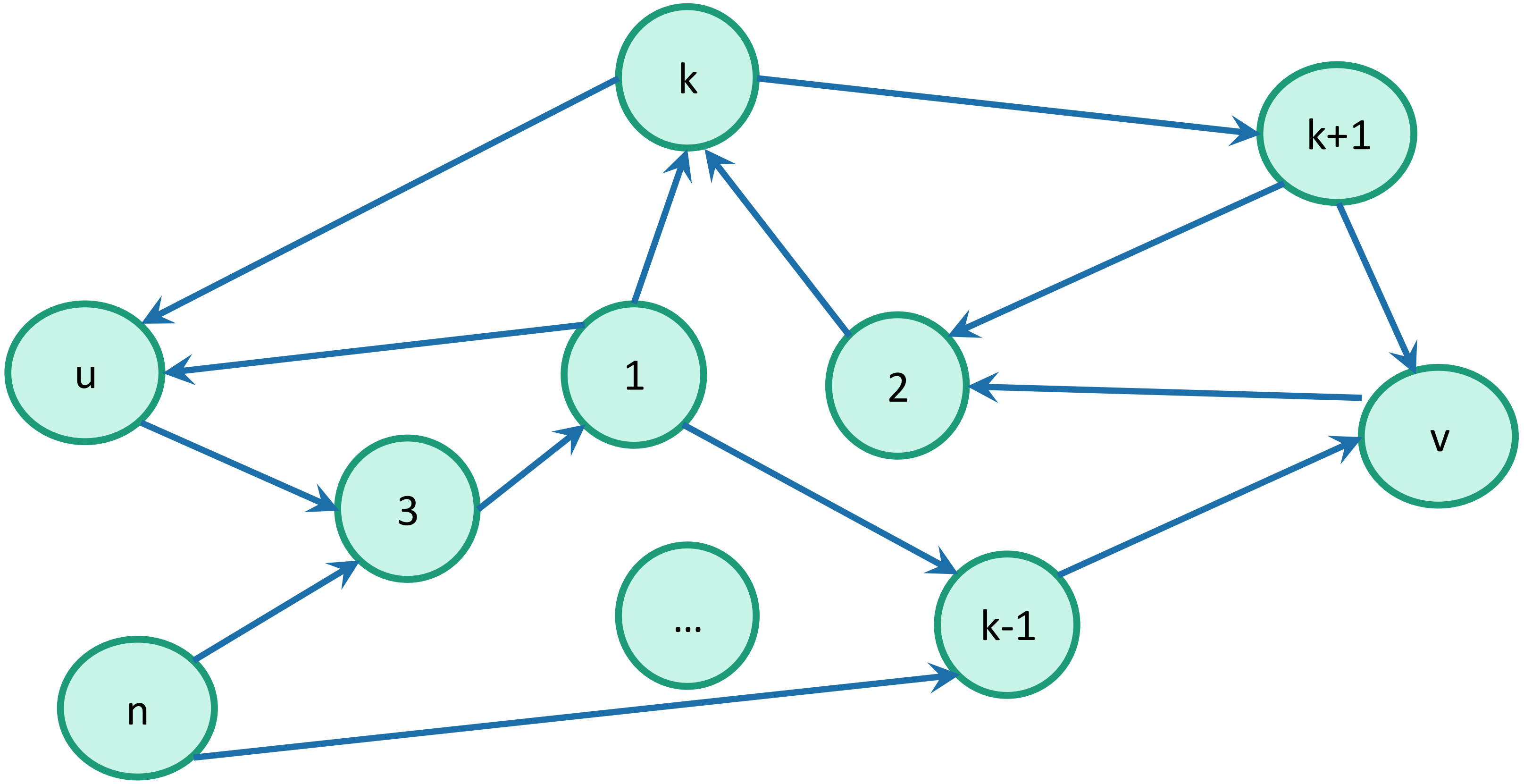
Can we do better?

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
  - What are our subproblems?
- **Step 2:** Find a recursive formulation for the subproblems
  - How can we solve larger problems using smaller ones?
- **Step 3:** Use dynamic programming to compute the thing you want.
  - Fill in a table, starting with the smallest sub-problems and building up.
- (**Steps 4 and 5** coming next lecture!)

Optimal substructure

Label the vertices 1,2,…,n

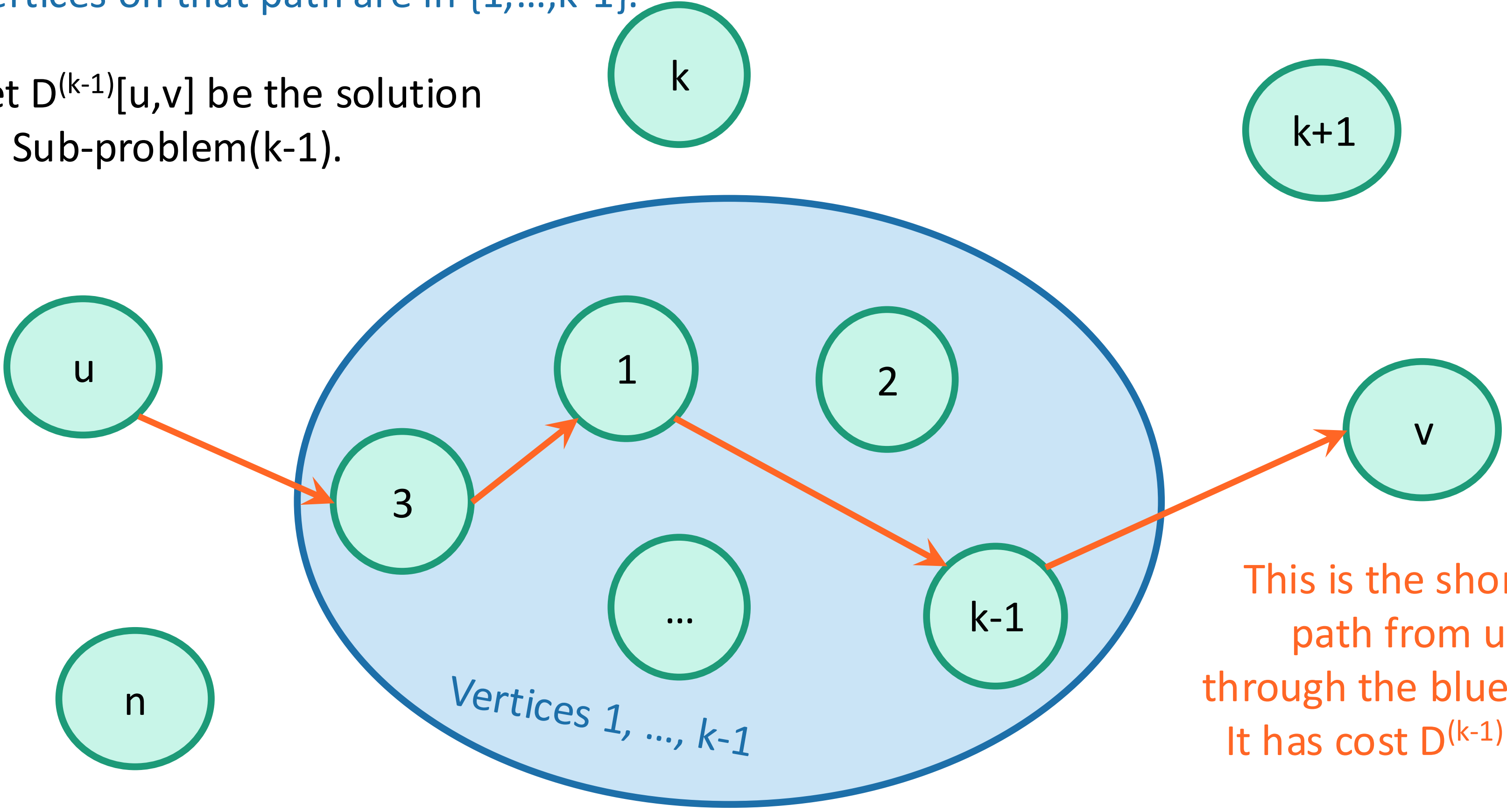


Optimal substructure

Label the vertices 1,2,…,n
(We omit some edges in the picture below – meant to be a cartoon, not an example).

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).



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

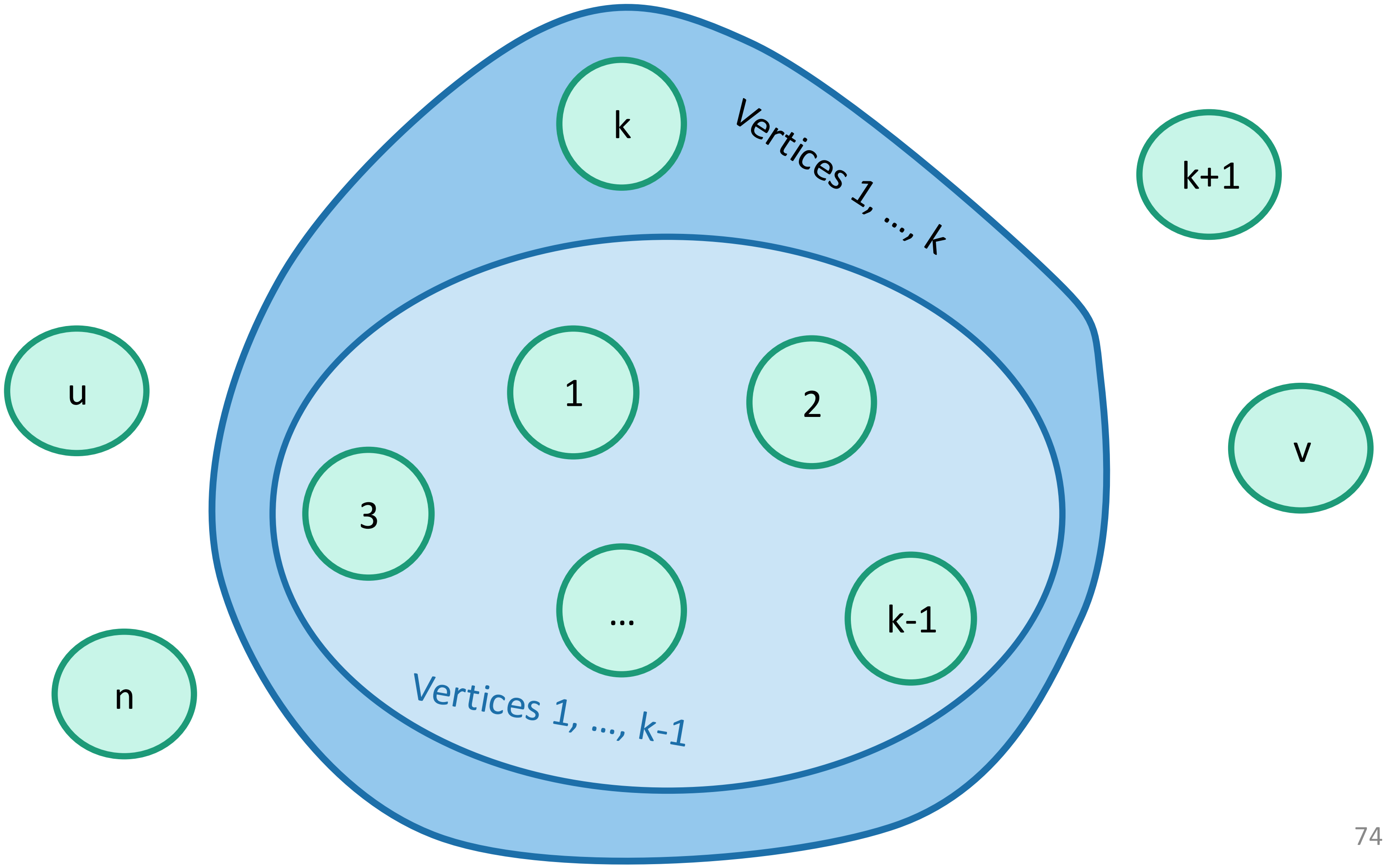
*Note that u and v might live inside the blue blob…that's okay, just a slightly different picture.

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
  - What are our subproblems?
- **Step 2:** Find a recursive formulation for the subproblems
  - How can we solve larger problems using smaller ones?
- **Step 3:** Use dynamic programming to compute the thing you want.
  - Fill in a table, starting with the smallest sub-problems and building up.

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

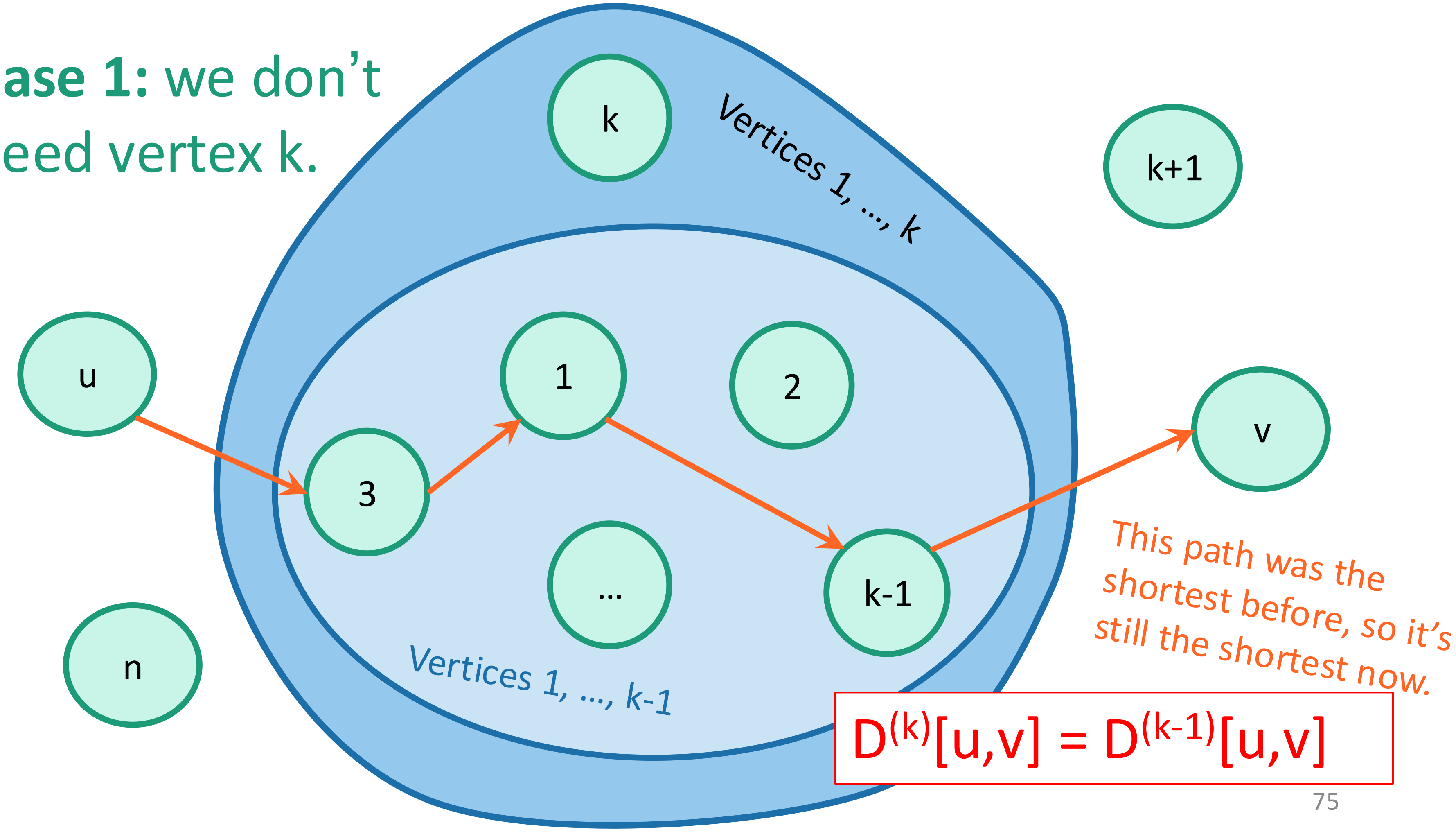
$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.



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.



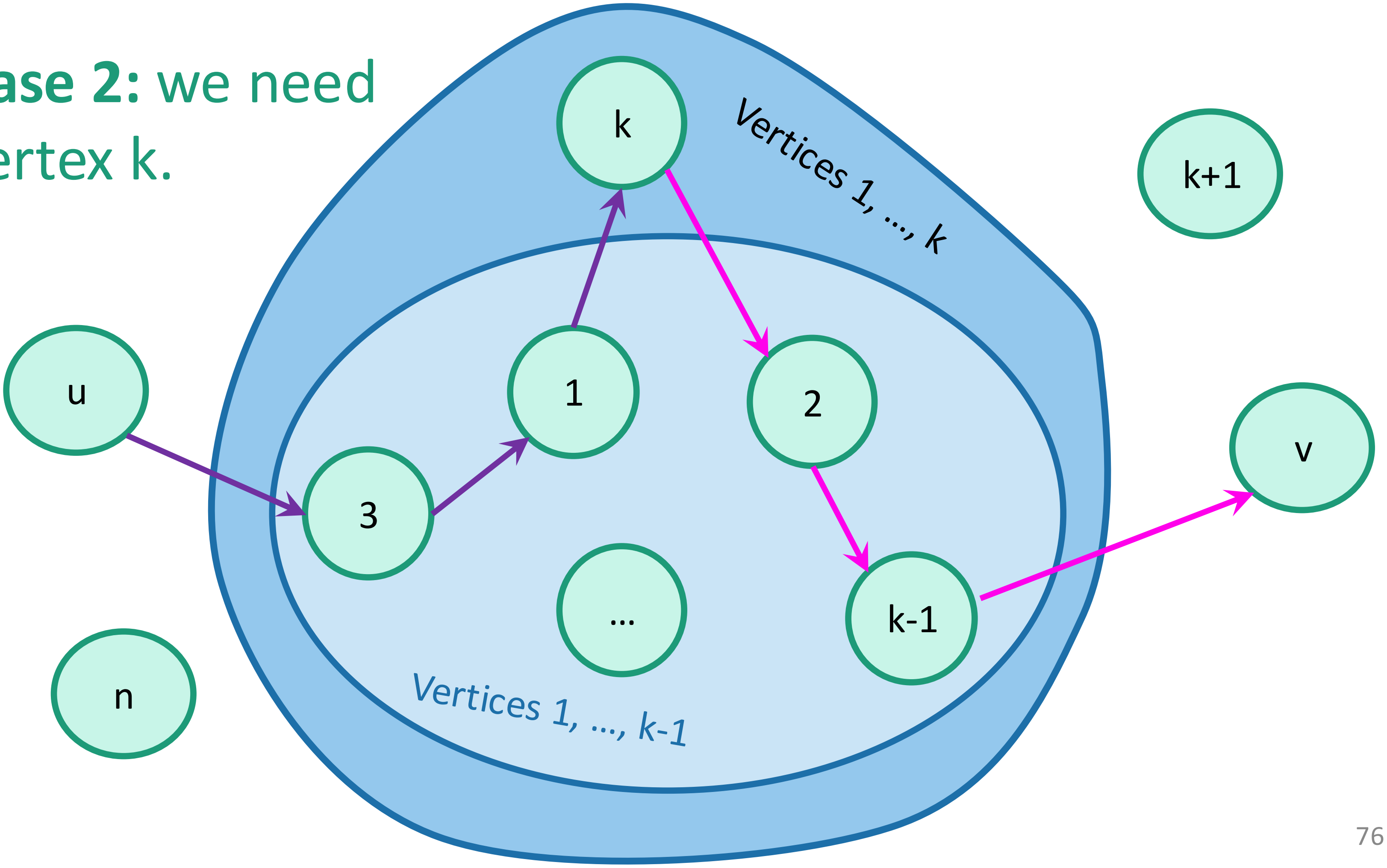
k
Vertices 1, …, k
k+1
u
1
2
v
3
…
k-1
n
Vertices 1, …, k-1
This path was the shortest before, so it's still the shortest now.

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

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.



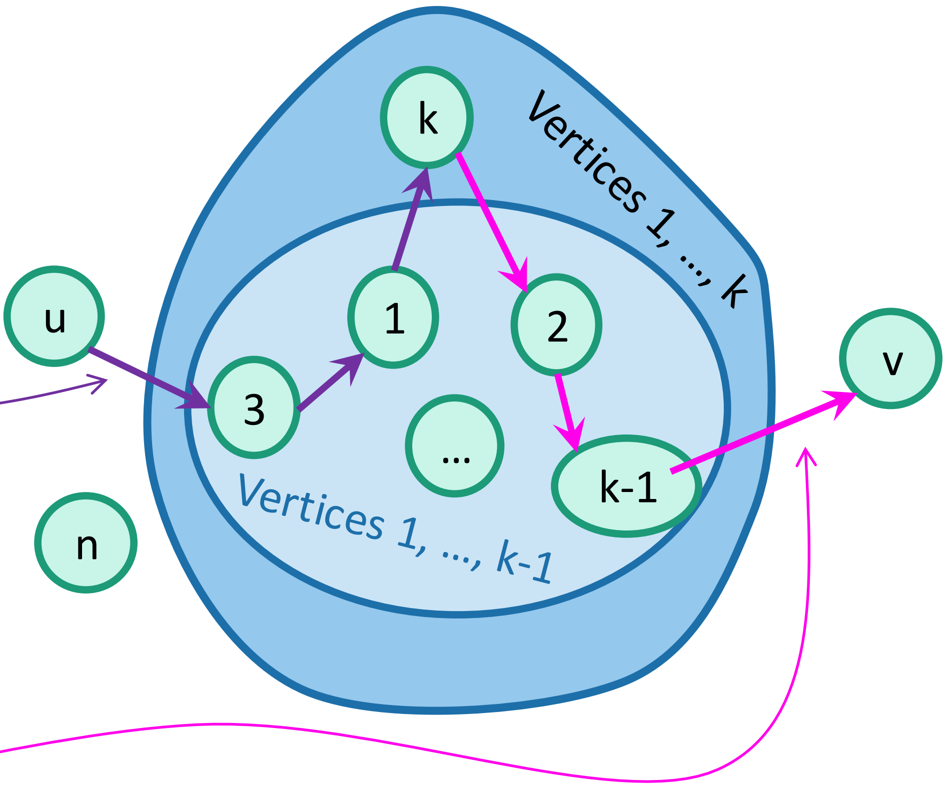
Case 2 continued

- Suppose there are no negative cycles. WLOG the shortest path from u to v through {1,…,k} is *simple* (aka, has no cycles).

- The shortest path from u to v looks like this: $\longrightarrow$

- **This path** is the shortest path from u to k through {1,…,k-1}.
  - sub-paths of shortest paths are shortest paths
- Similarly for **this path**.

Case 2: we need vertex k.

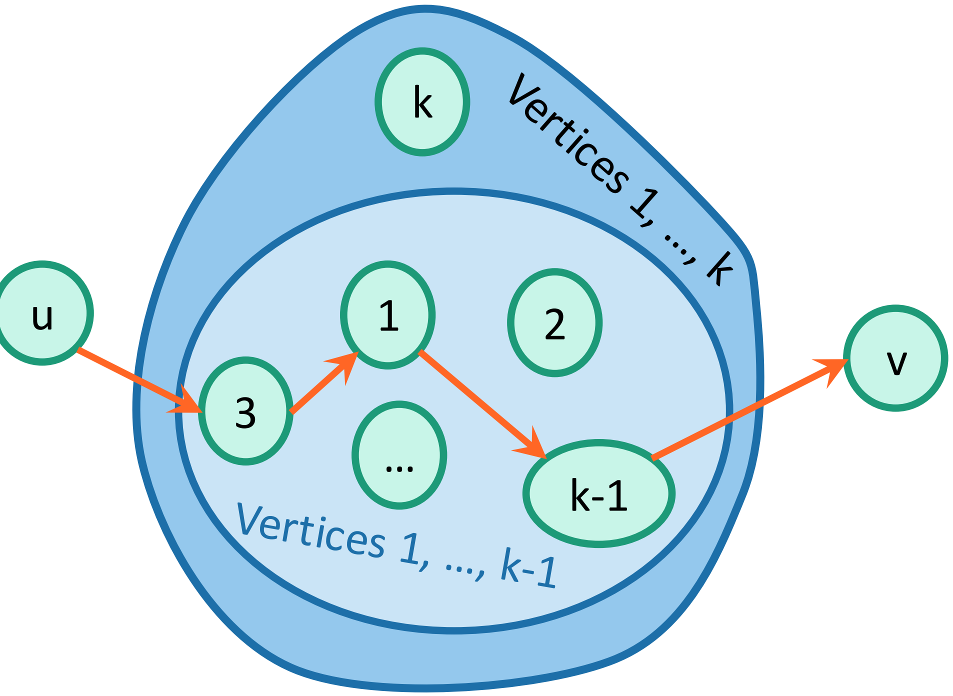


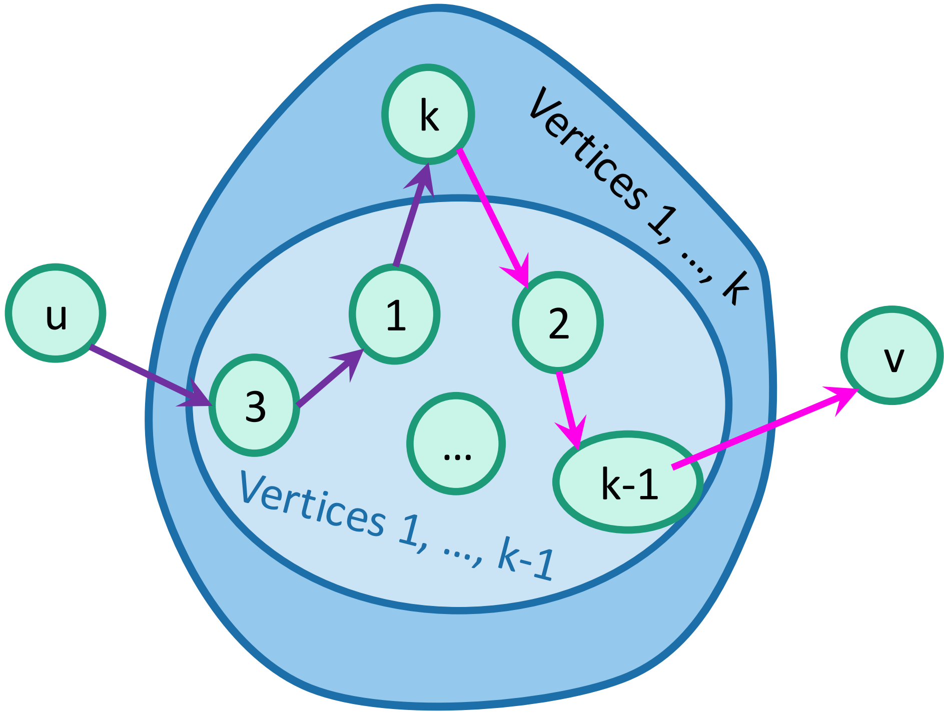
$$D^{(k)}[u,v] = D^{(k-1)}[u,k] + D^{(k-1)}[k,v]$$

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

Case 1: we don't need vertex k.

Case 2: we need vertex k.





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

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

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 solutions to 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 gives us an algorithm!

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
  - What are our subproblems?
- **Step 2:** Find a recursive formulation for the subproblems
  - How can we solve larger problems using smaller ones?
- **Step 3:** Use dynamic programming to compute the thing you want.
  - Fill in a table, starting with the smallest sub-problems and building up.

Floyd-Warshall algorithm

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

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

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

Our earlier logic shows

- 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:

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

Work out the details of a proof! (Hint: your inductive hypothesis should be that $D^{(i)}[u,v]$ has the interpretation we want it to have).

- Running time: $O(n^3)$
  - Better than running Bellman-Ford n times!

- 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 $D^{(k)}$, just the current one and the previous one.

What if there *are* negative cycles?

- Just like Bellman-Ford, Floyd-Warshall can detect negative cycles:
  - "Negative cycle" means that there's some v so that there is a path from v to v that has cost < 0.
  - Aka, $D^{(n)}[v,v] < 0$.

- 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 $O(n^3)$.

Can we do better than $O(n^3)$?

Nothing on this slide is required knowledge for this class

- There is an algorithm that runs in time $O(n^3/\log^{100}(n))$.
  - *[Williams, "Faster APSP via Circuit Complexity", STOC 2014]*
- If you can come up with an algorithm for All-Pairs-Shortest-Path that runs in time $O(n^{2.99})$, that would be a really big deal.
  - Let me know if you can!
  - See *[Abboud, Vassilevska-Williams, "Popular conjectures imply strong lower bounds for dynamic problems", FOCS 2014]* for some evidence that this is a very difficult problem!

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the thing you want.
  - E.g, length of shortest paths
- **Step 3:** Use dynamic programming to compute the thing you want.
  - Fill in a table, starting with the smallest sub-problems and building up.
- (**Steps 4 and 5** coming next lecture…)

Recap

- Two 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 not repeating work!

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 for Lecture 13!