All Pair Shortest Path Algorithm

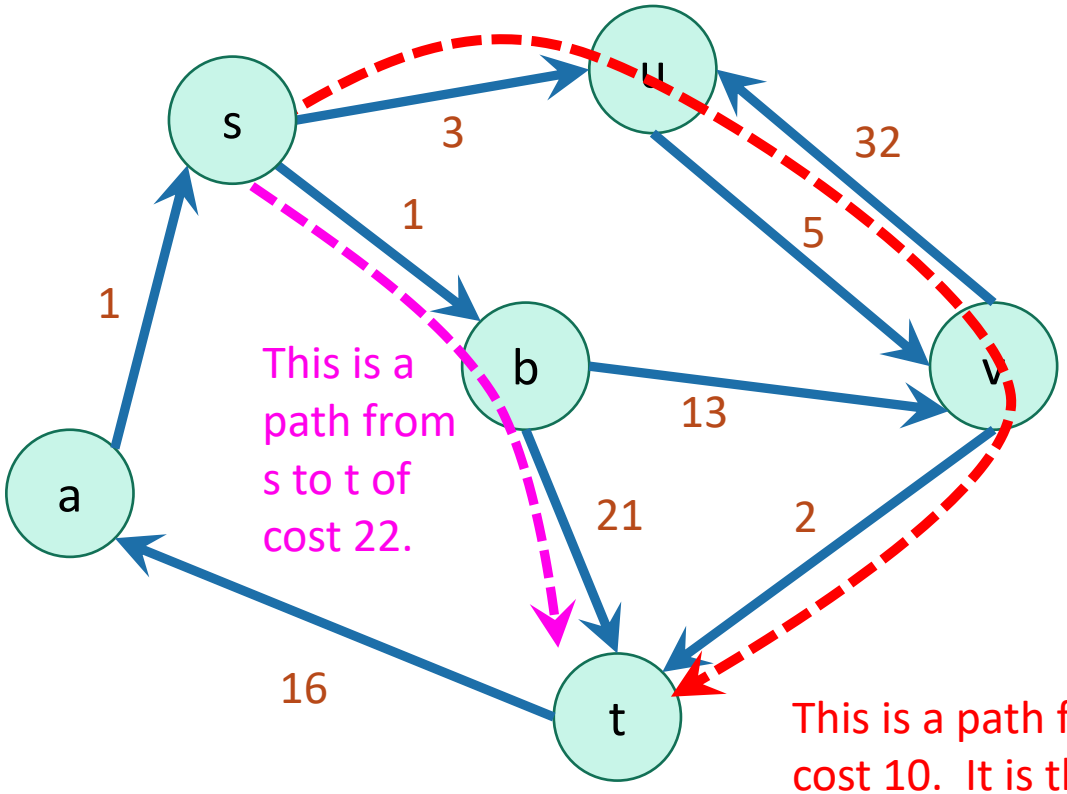
Floyd-Warshall

Adapted from

Stanford CS161 course

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!
  - Solves the single-source shortest path problem in weighted graphs.
- Bellman-Ford algorithm!
  - **ALSO** solves the single-source shortest path problem in weighted graphs.

Recap: shortest paths

- BFS:
  - (+) O(n+m)
  - (-) only unweighted graphs
- Dijkstra's algorithm:
  - (+) weighted graphs
  - (+) O(nlog(n) + m) if you implement it with a Fibonacci heap
  - (-) no negative edge weights
  - (-) very "centralized" (need to keep track of all the vertices to know which to update).
- 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)

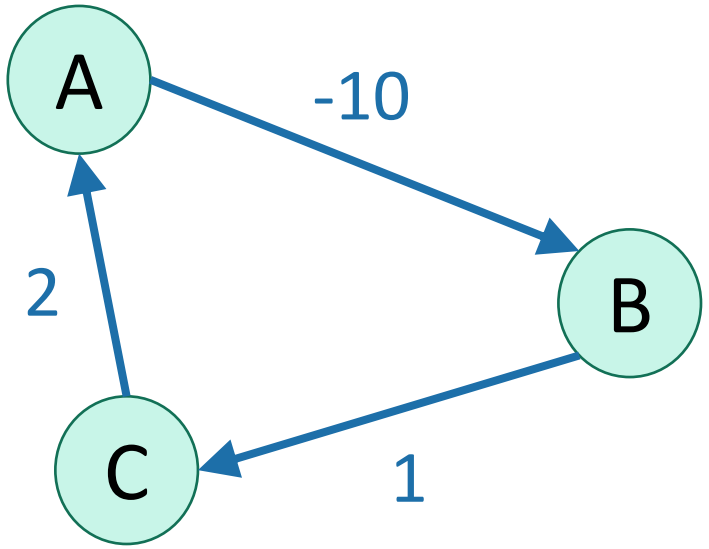
Bellman-Ford vs. Dijkstra

- Dijkstra:
  - Find the u with the smallest d[u]
  - Update u's neighbors: d[v] = min(d[v], d[u] + w(u,v))

- Bellman-Ford:
  - Don't bother finding the u with the smallest d[u]
    - Everyone updates!
  - Slower, but more flexible:
    - Can handle negative edge weights (as long as there aren't negative cycles)
    - Can do updates in a decentralized way.

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 vs. Dijkstra

- Dijkstra:
  - Find the u with the smallest d[u]
  - Update u's neighbors: d[v] = min(d[v], d[u] + w(u,v))

- Bellman-Ford:
  - Don't bother finding the u with the smallest d[u]
    - Everyone updates!
  - Slower, but more flexible:
    - Can handle negative edge weights (as long as there aren't negative cycles)
    - Can do updates in a decentralized way.

Bellman-Ford* algorithm

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

Bellman-Ford*(G,s):

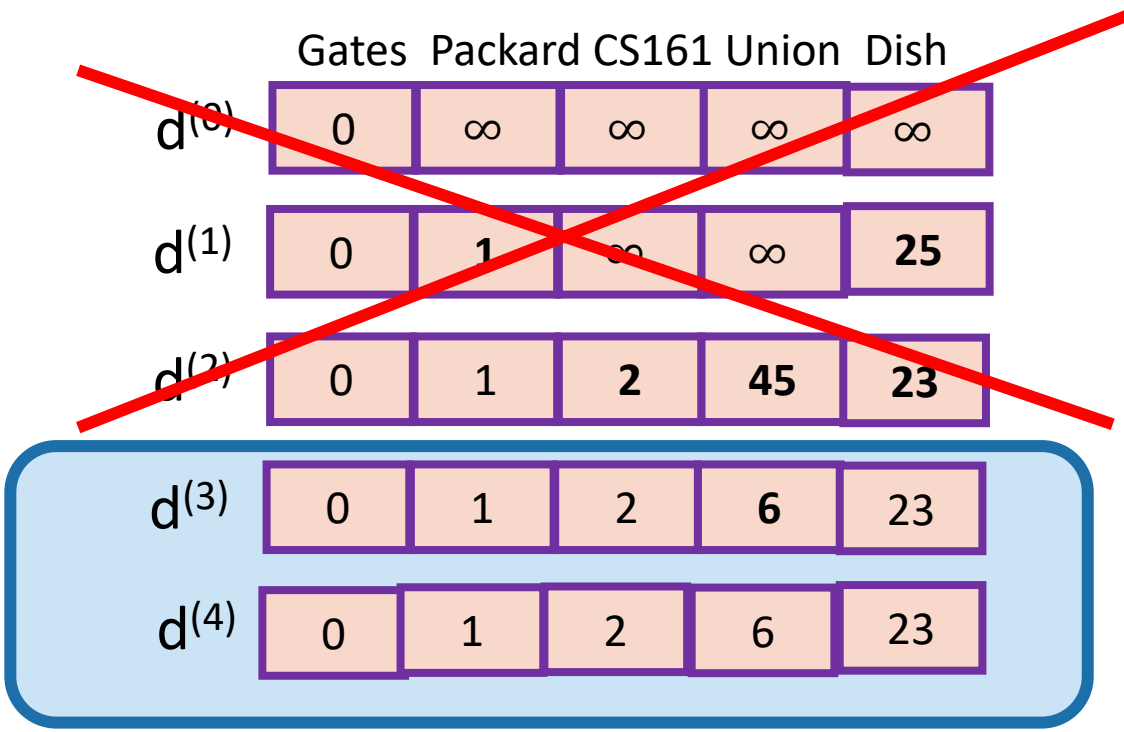
- Initialize arrays $d^{(0)},\dots,d^{(n-1)}$ of length n
- $d^{(0)}[v] = \infty$ for all v in V
- $d^{(0)}[s] = 0$
- **For** i=0,…,n-2:
  - **For** v in V:
    - $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , \min_{u \text{ in } v.inNbrs}\{d^{(i)}[u] + w(u,v)\})$
- Now, dist(s,v) = $d^{(n-1)}[v]$ for all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

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

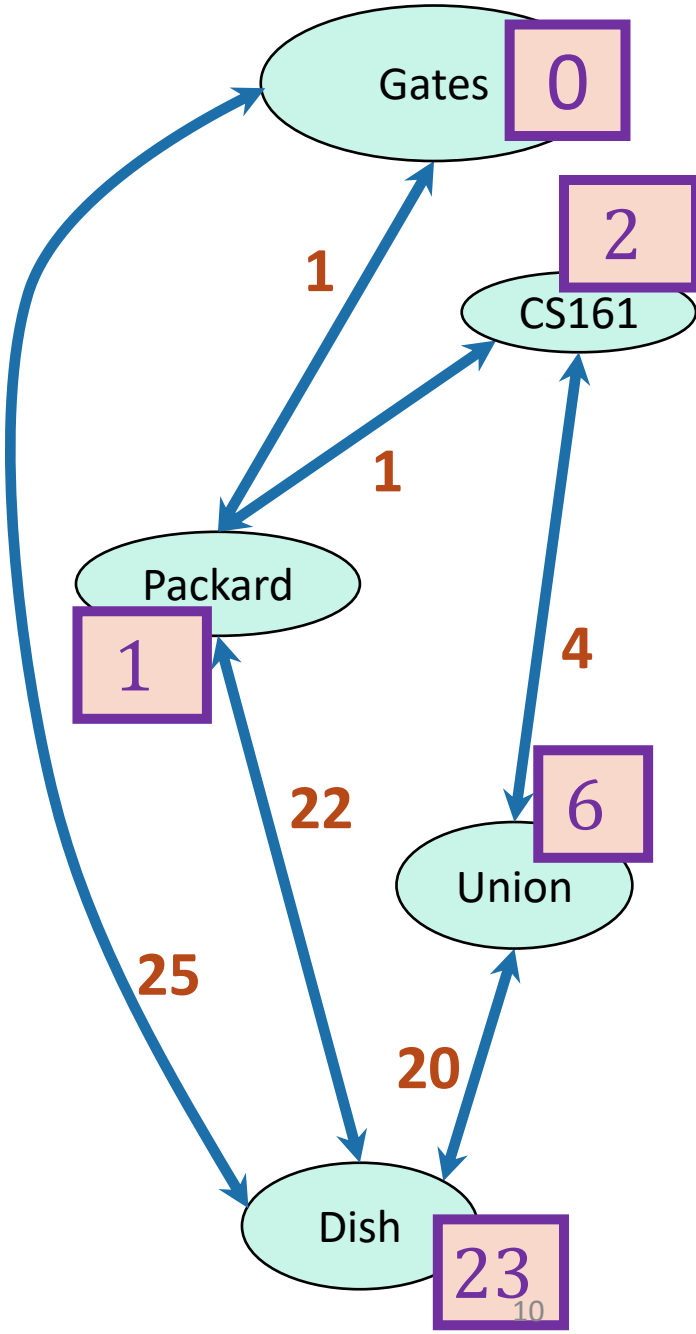


Only need these two in order to compute $d^{(4)}$

Interpretation of $d^{(i)}$

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**.

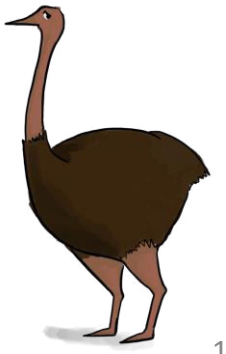
	Gates	Packard	CS161	Union	Dish
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**25**
$d^{(2)}$	0	1	**2**	**45**	**23**
$d^{(3)}$	0	1	2	**6**	23
$d^{(4)}$	0	1	2	6	23



Why does Bellman-Ford work?

- Inductive hypothesis:
  - For all v, $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v **with at most i edges**.
- Conclusion:
  - For all v, $d^{(n-1)}[v]$ is equal to the cost of the shortest path between s and v **with at most n-1 edges**.

Do the base case and inductive step!

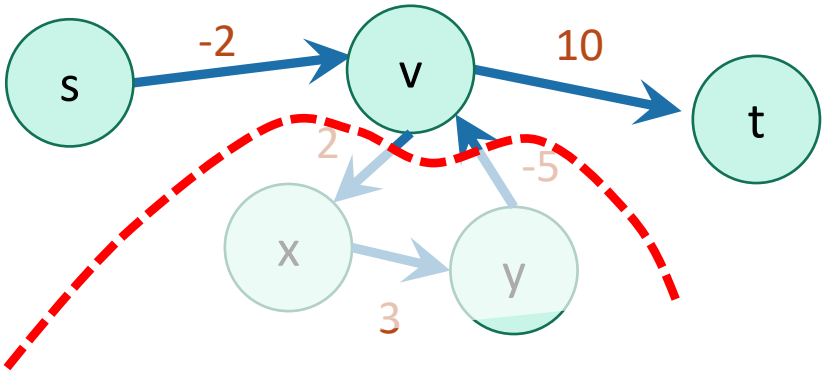


Aside: simple paths

Assume there is no negative cycle.

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

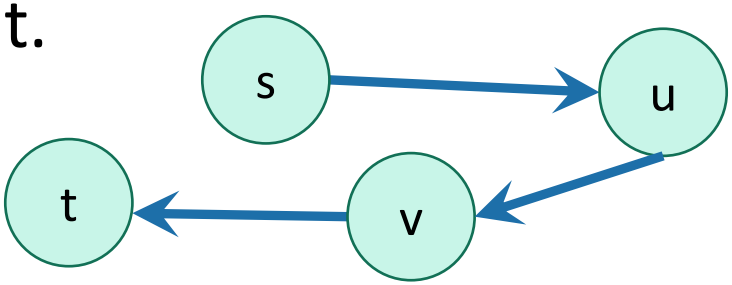
- Then there is a shortest path from s to t, and moreover there is a simple shortest path.



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.

Can't add another edge without making a cycle!



- So there is a shortest path with at most n-1 edges

Why does Bellman-Ford work?

- Inductive hypothesis:
  - For all v, $d^{(i)}[v]$ is equal to the cost of the shortest path between s and v **with at most i edges**.
- Conclusion:
  - For all v, $d^{(n-1)}[v]$ 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.

Bellman-Ford take-aways

- Running time is O(mn)
  - For each of n rounds, update m edges.
- Works fine with negative edges.
- Does not work with negative cycles.
  - No algorithm can – shortest paths aren't defined if there are negative cycles.
- B-F can detect negative cycles!
  - Fun exercise! (Hint, what happens if you run Bellman-Ford for longer than n-1 iterations?)

- What class of algorithm does Dijkstra fall in?

  Greedy Algorithm

- What about Bellman-Ford?

Bellman-Ford is an example of…

Dynamic Programming!

What about Folyd-Warshall??

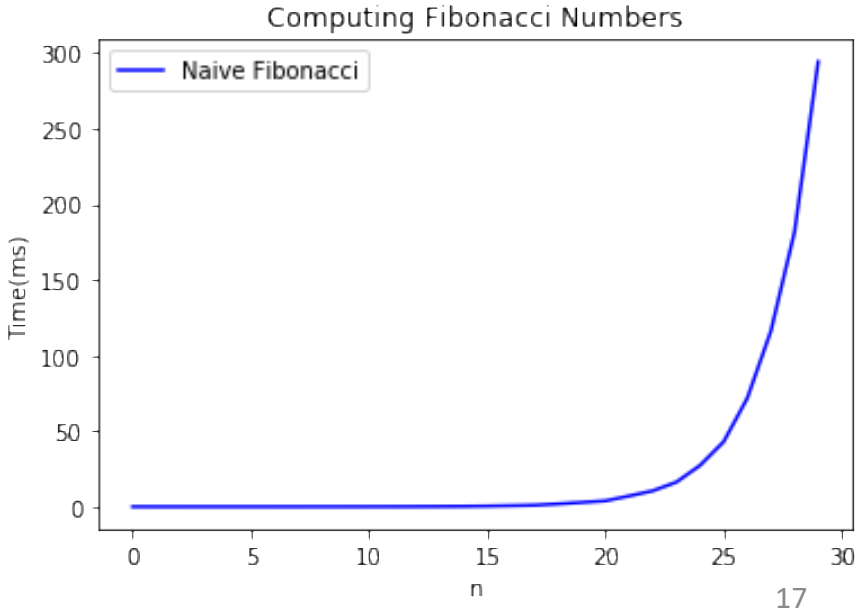
Also, Dynamic Programming

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!**

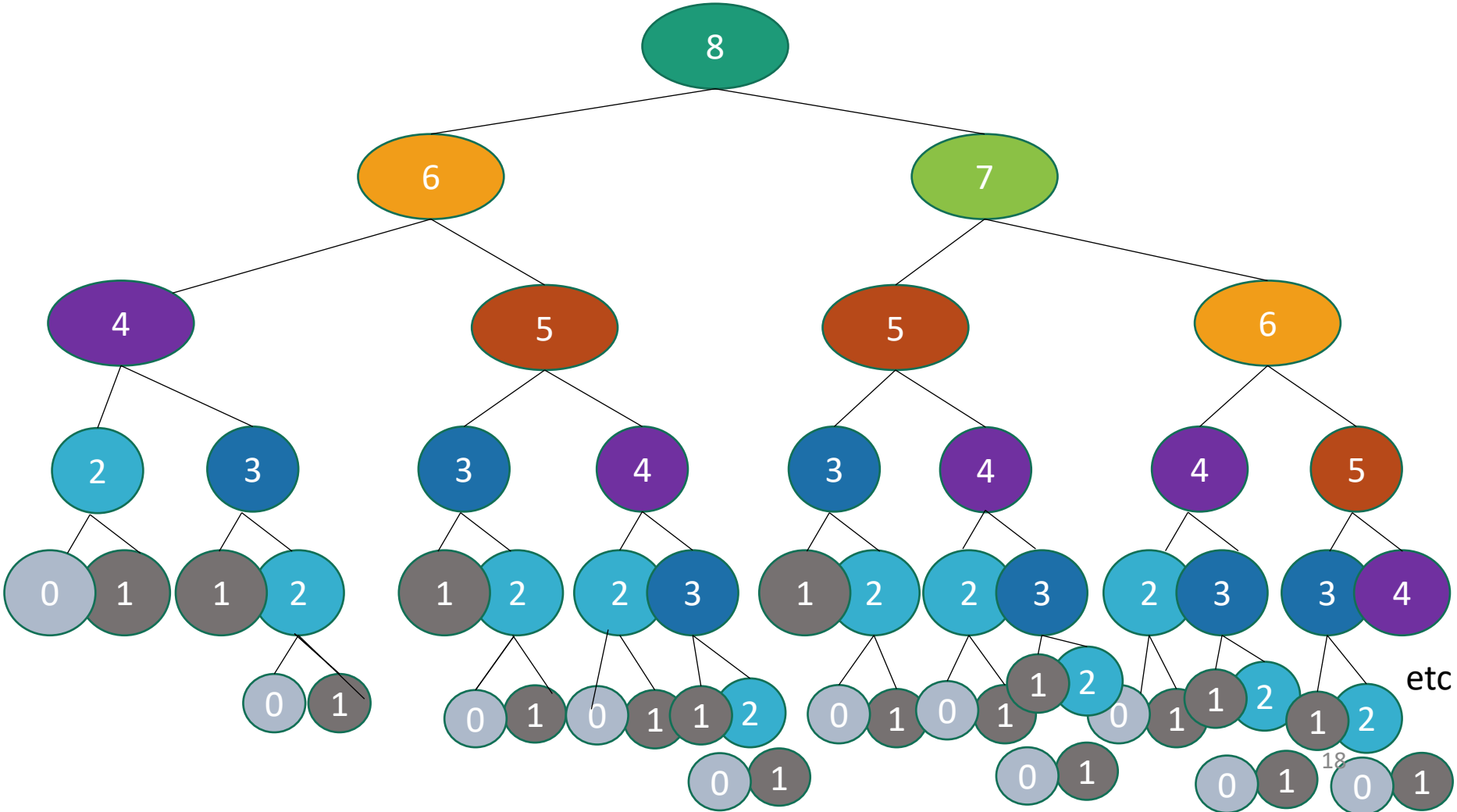


See IPython notebook for lecture 12

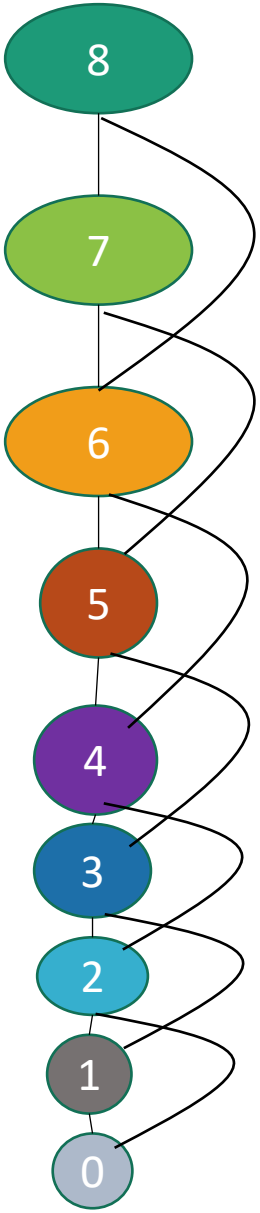
What's going on?

Consider Fib(8)

That's a lot of repeated computation!

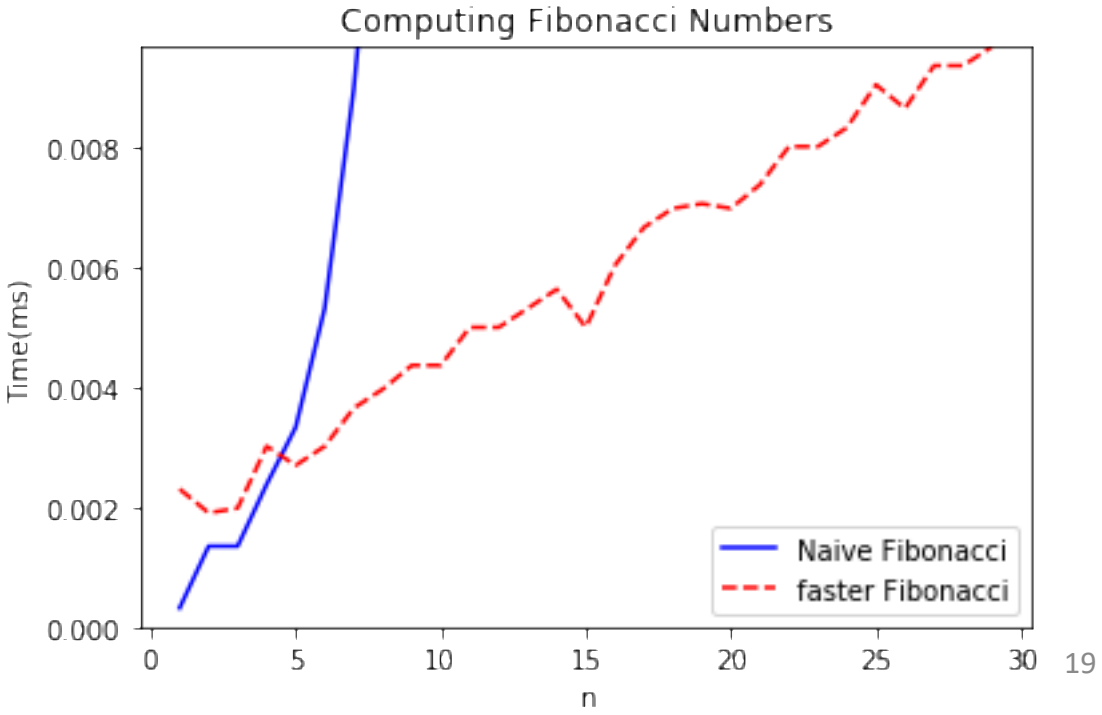


Maybe this would be better:




```
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.
- Usually it is for solving **optimization problems**
  - eg, ***shortest*** path
  - (Fibonacci numbers aren't an optimization problem, but they are a good example of DP anyway…)

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:

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

Shortest path with at most i edges from s to v

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

*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

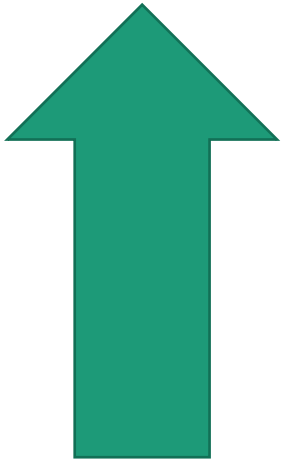
- Using these properties, we can design a ***dynamic programming*** algorithm:
  - Keep a table of solutions to the smaller problems.
  - Use the solutions in the table to solve bigger problems.
  - At the end we can use information we collected along the way to find the solution to the whole thing.

Two ways to think about and/or implement DP algorithms

- Top down
- Bottom up

Bottom up approach

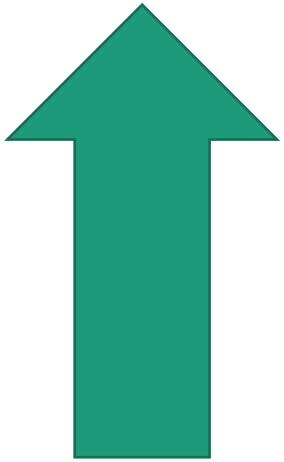
what we just saw.



- 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

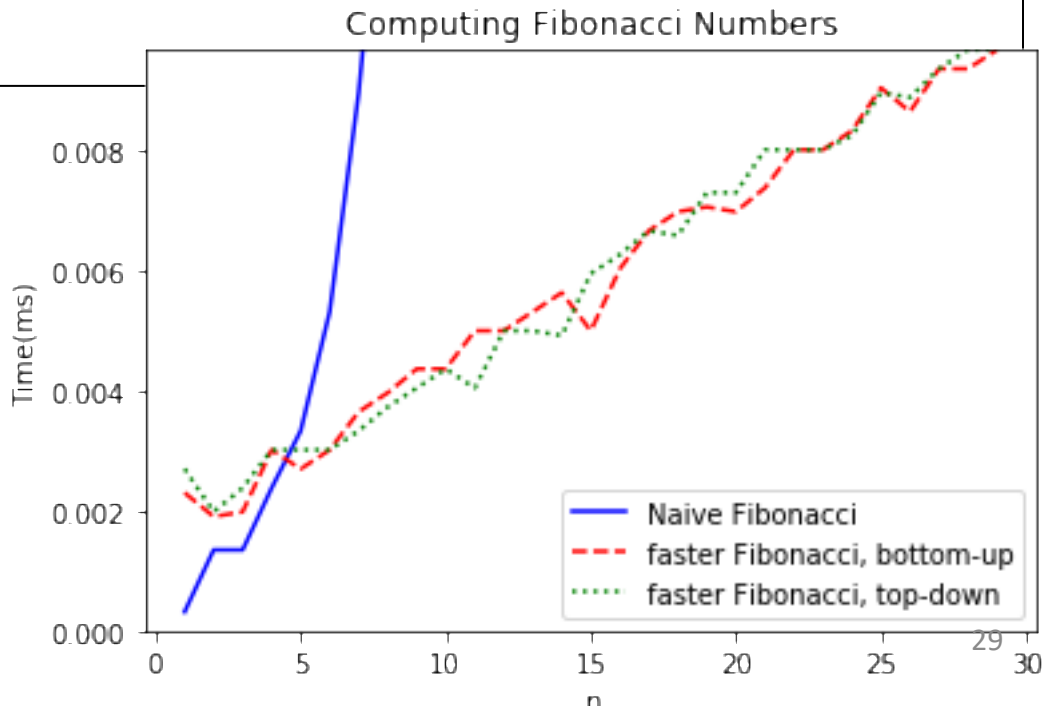
- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
      - etc..

- 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.



What have we learned?

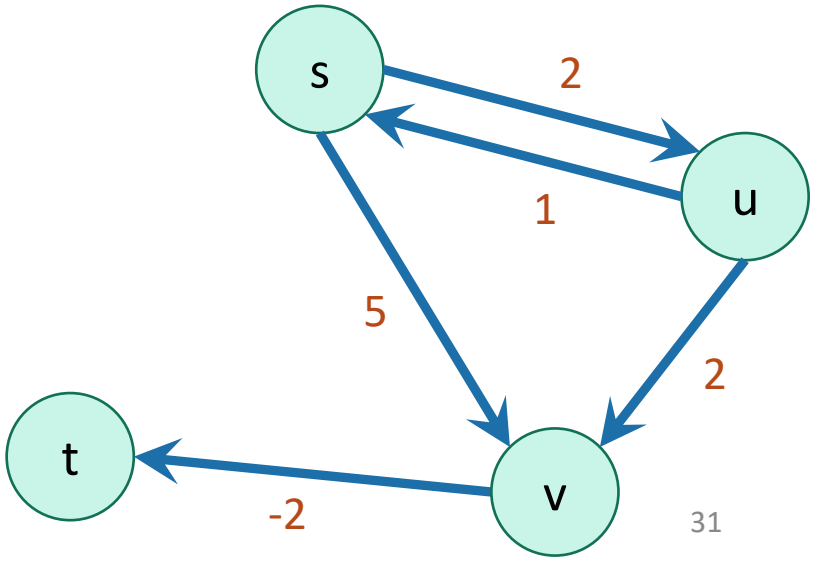
- ***Dynamic programming:***
  - Paradigm in algorithm design.
  - Uses **optimal substructure**
  - Uses **overlapping subproblems**
  - Can be implemented **bottom-up** or **top-down**.
  - It's a fancy name for a pretty common-sense idea:

Floyd-Warshall Algorithm

- 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 $O(n \cdot nm) = O(n^2m)$,
    - may be as bad as n^4 if $m=n^2$

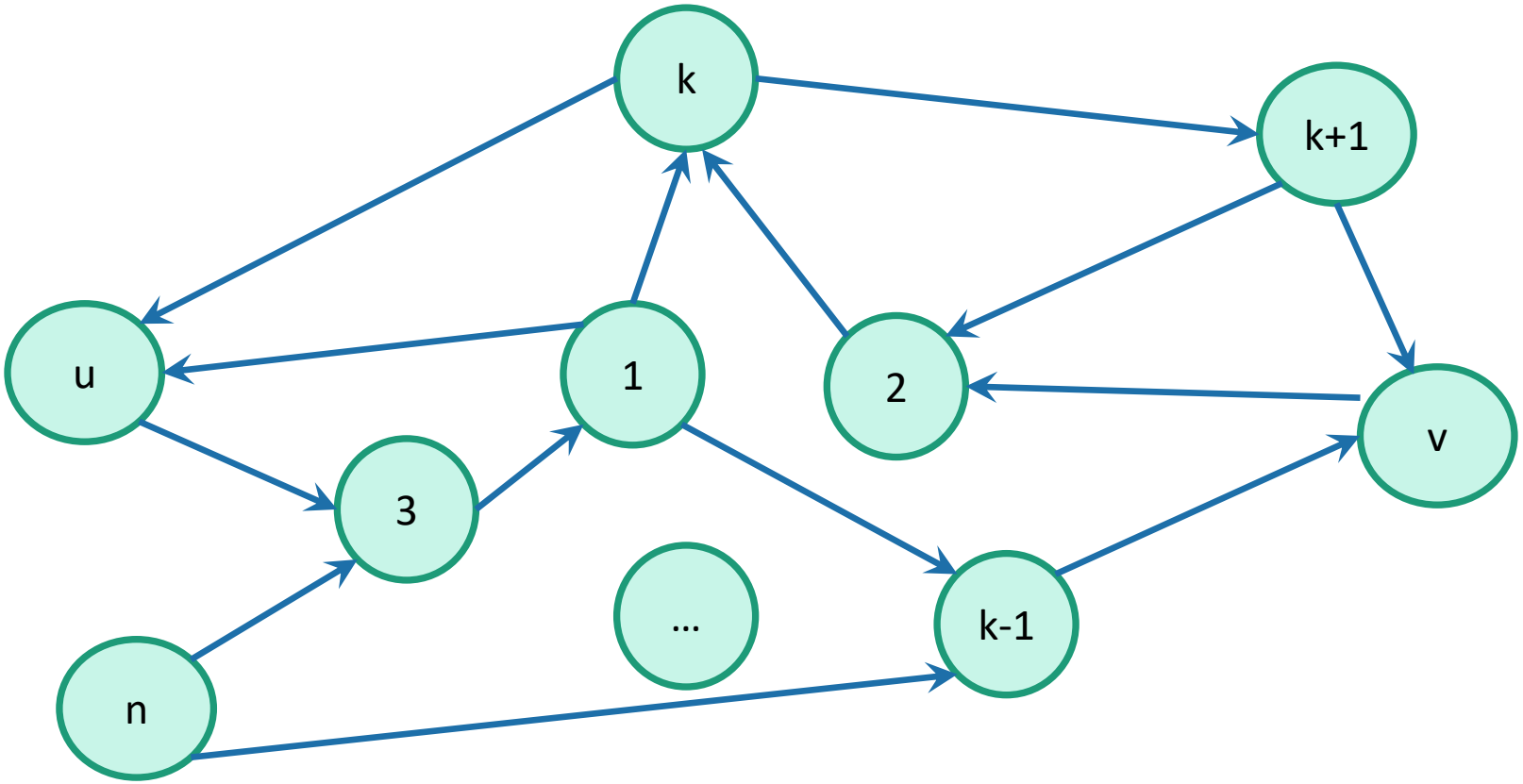
Can we do better?

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 find the thing you want.
  - Fill in a table, starting with the smallest sub-problems and building up.

Label the vertices 1,2,…,n

Optimal substructure

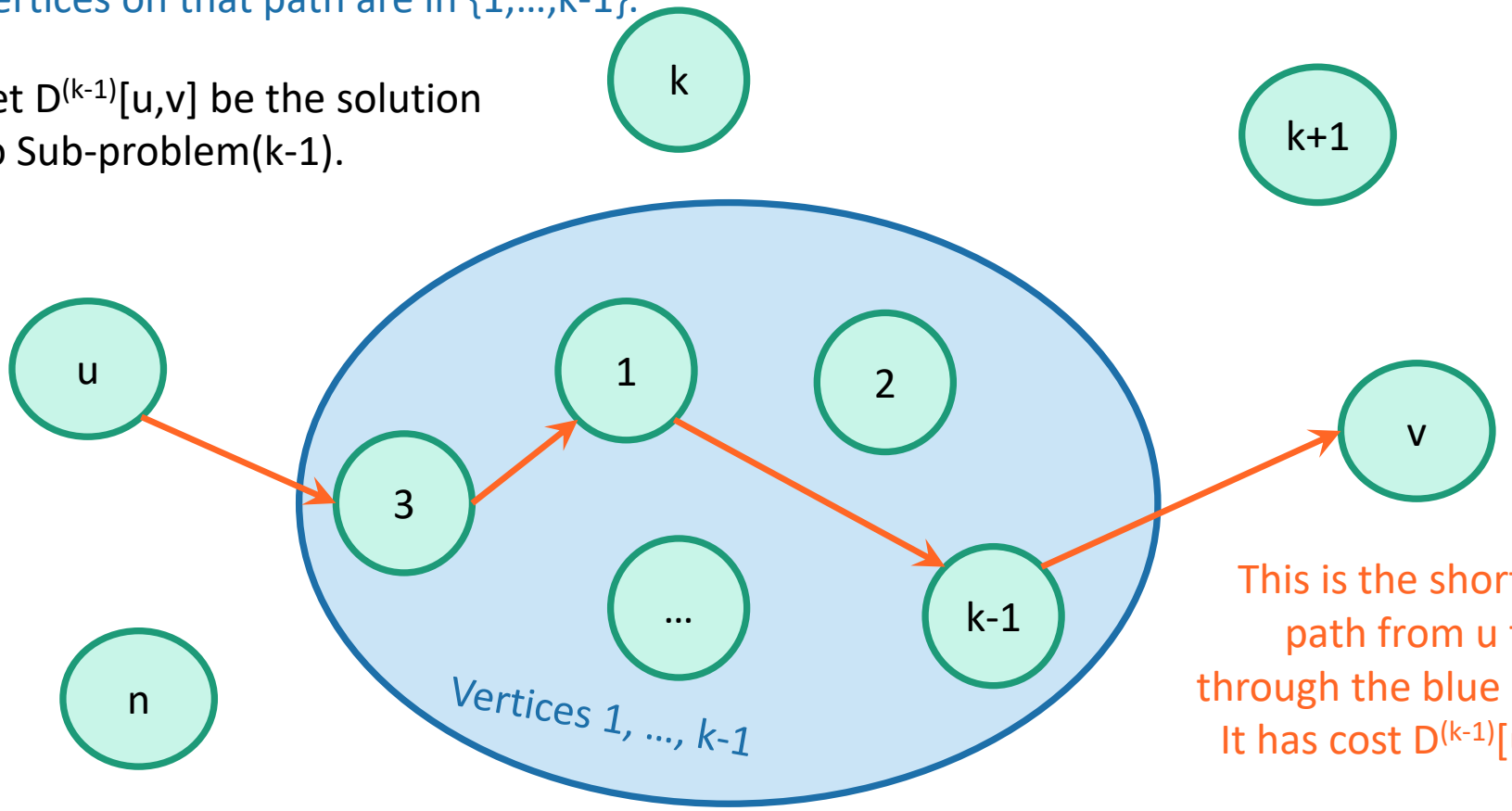


Optimal substructure

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).



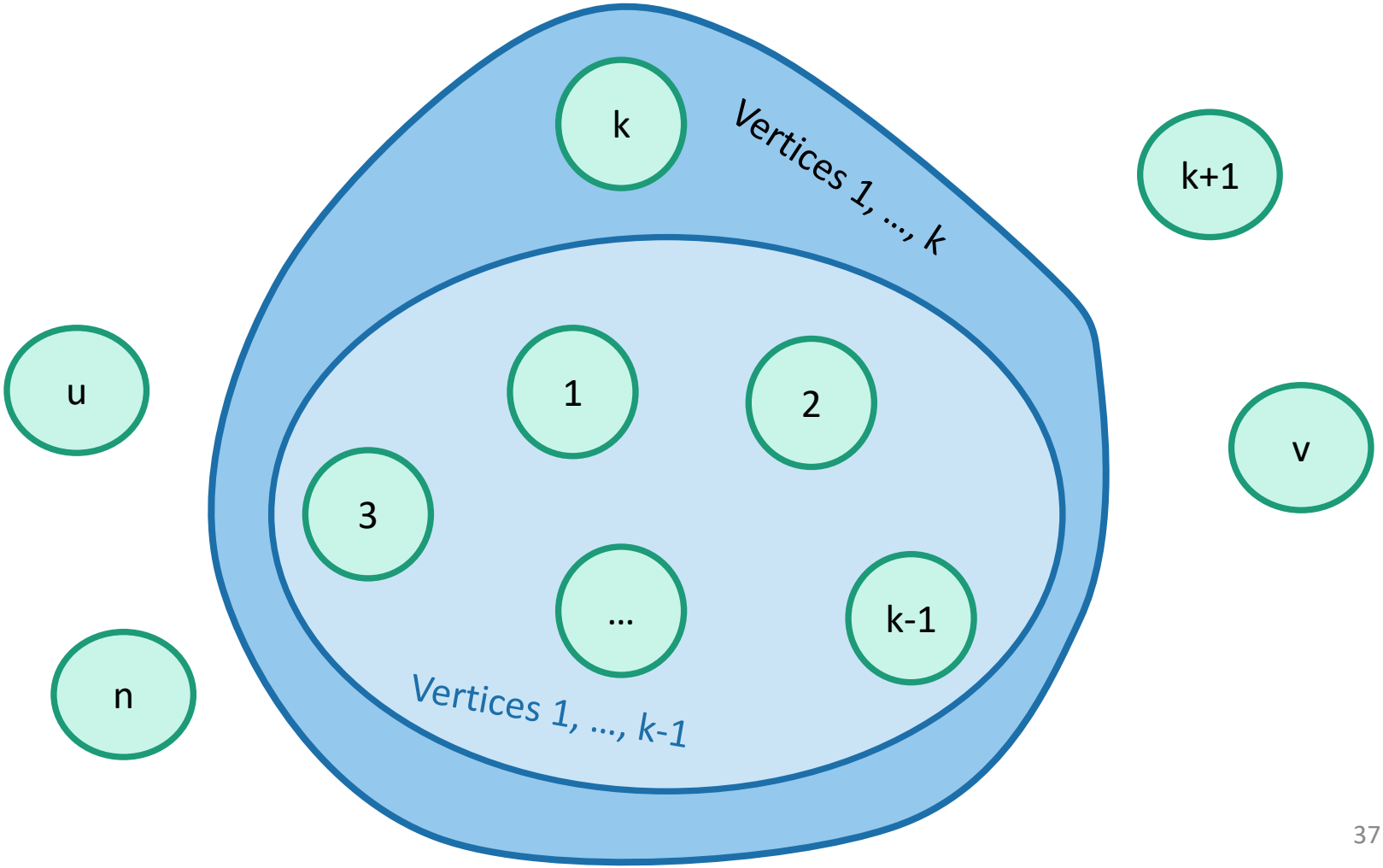
This is the shortest path from u to v through the blue set. It has cost $D^{(k-1)}[u,v]$

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 find 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 $D^{(k-1)}$?

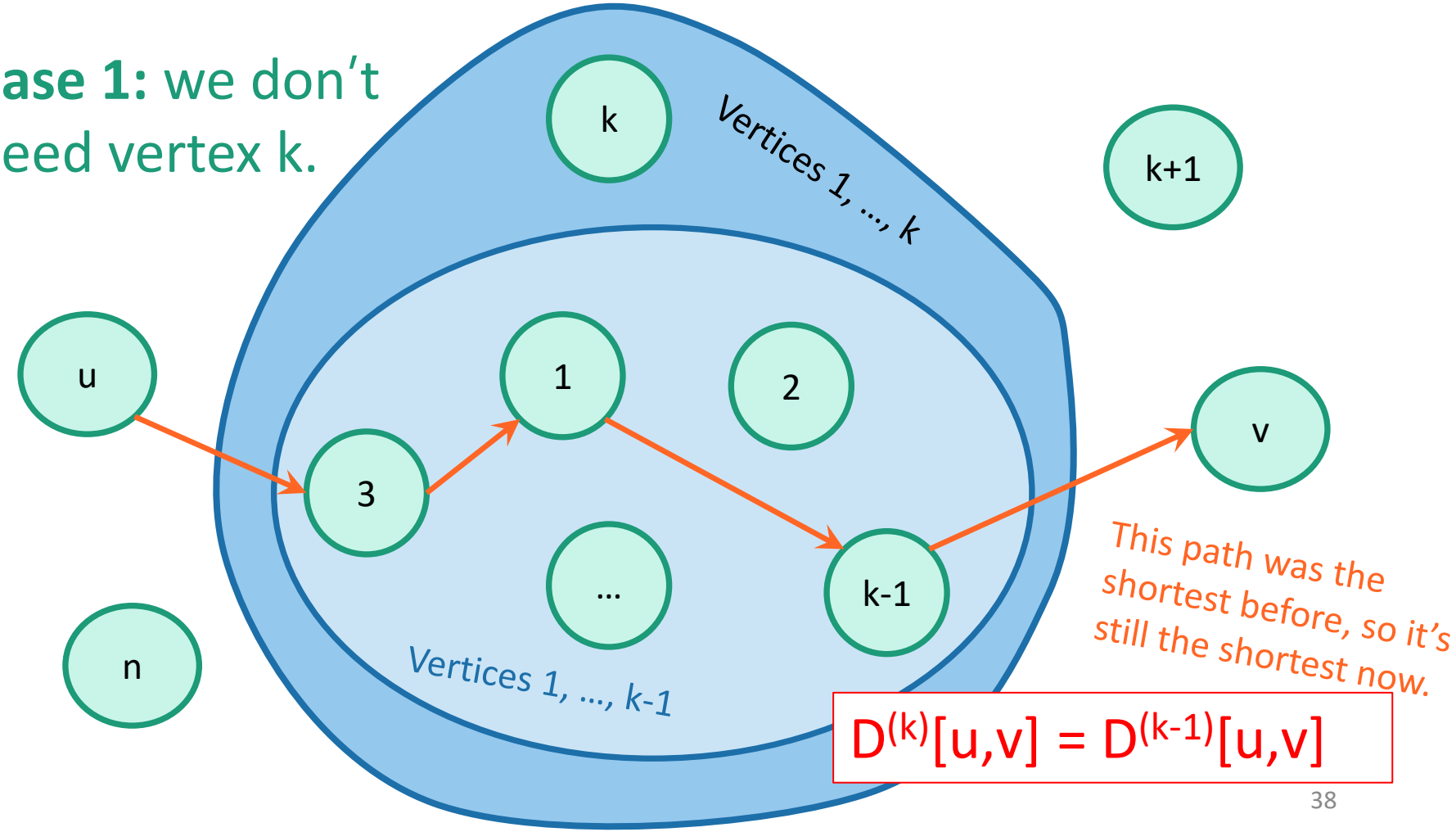
$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.



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.

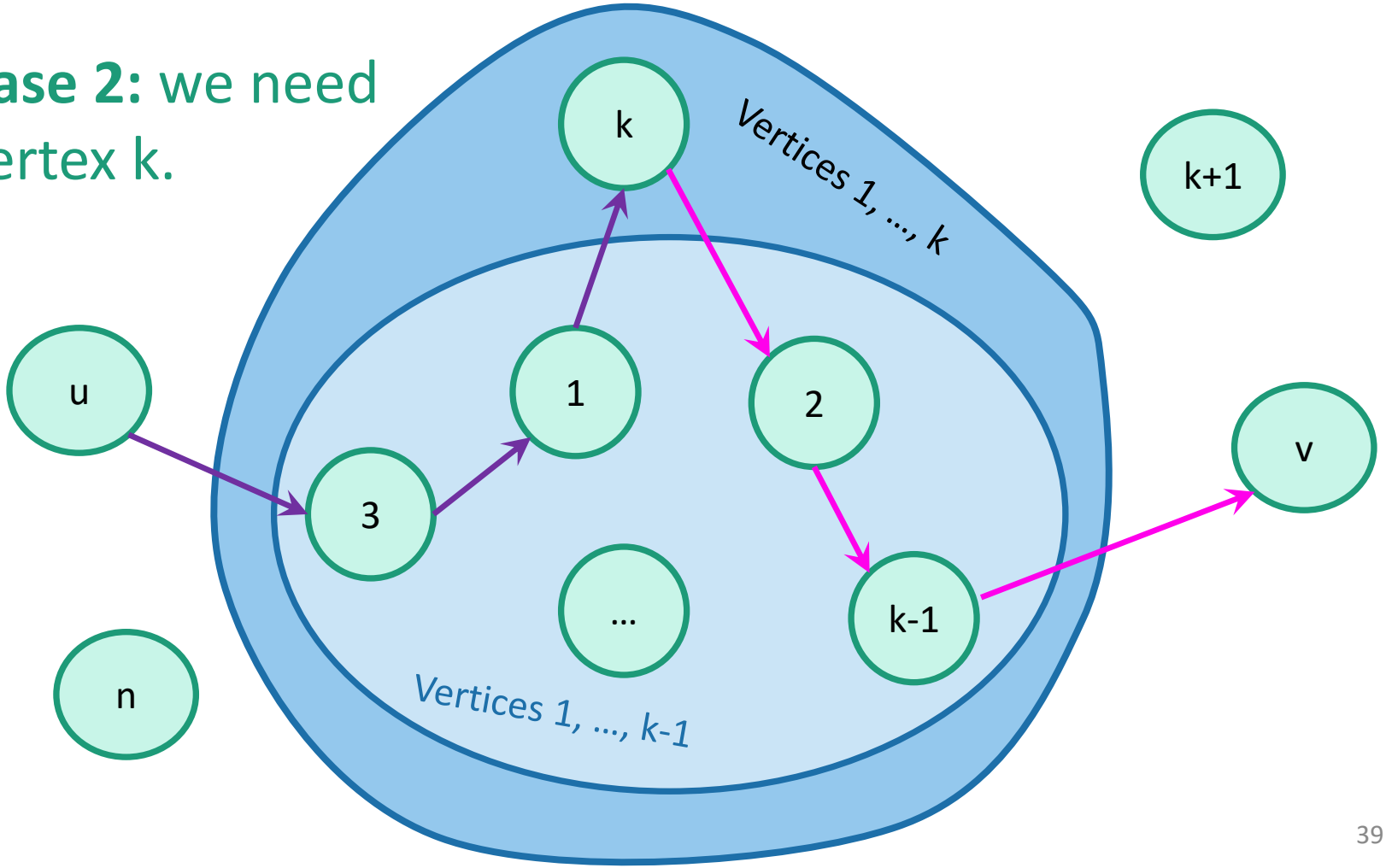


$$D^{(k)}[u,v] = D^{(k-1)}[u,v]$$

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}.

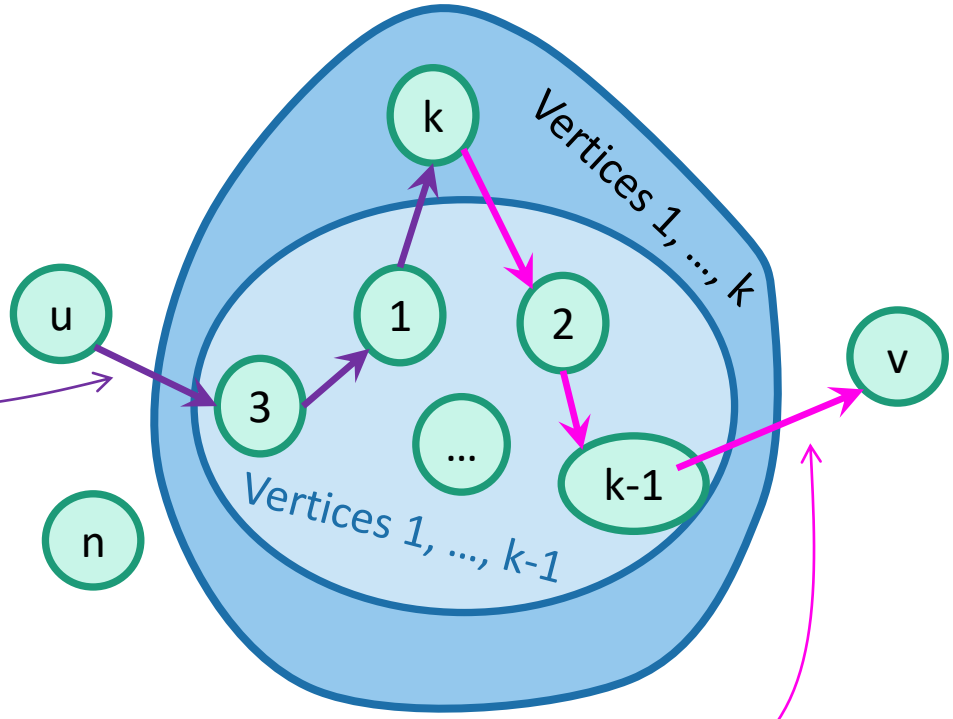
Case 2: we need vertex k.



Case 2 continued

- Suppose there are no negative cycles. WLOG the shortest path from u to v through {1,…,k} is simple.
- The shortest path from u to v looks like this:
- **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**.

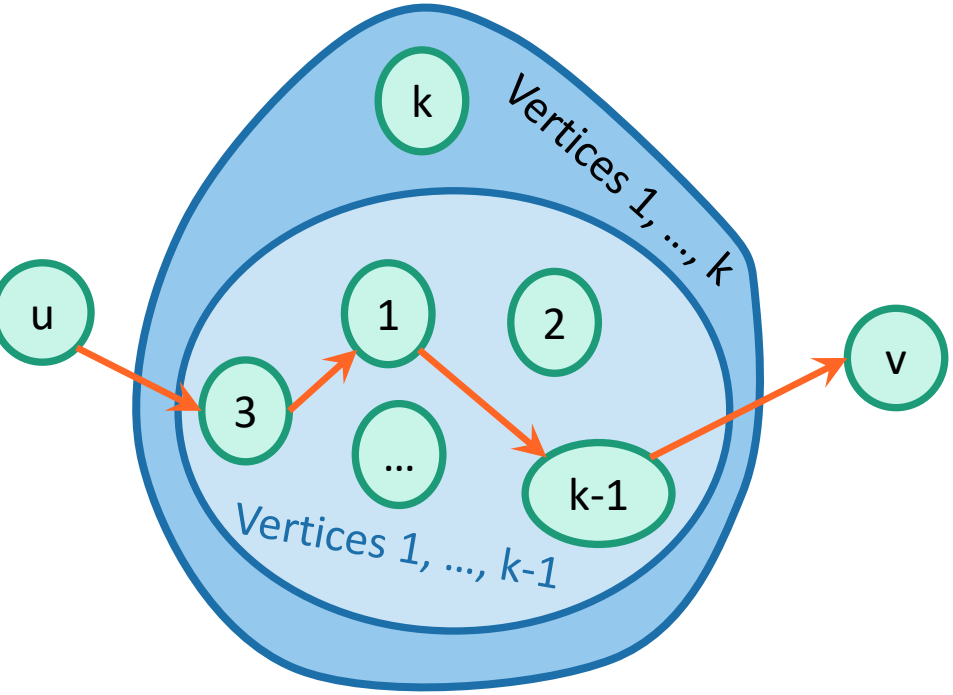
Case 2: we need vertex k.



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

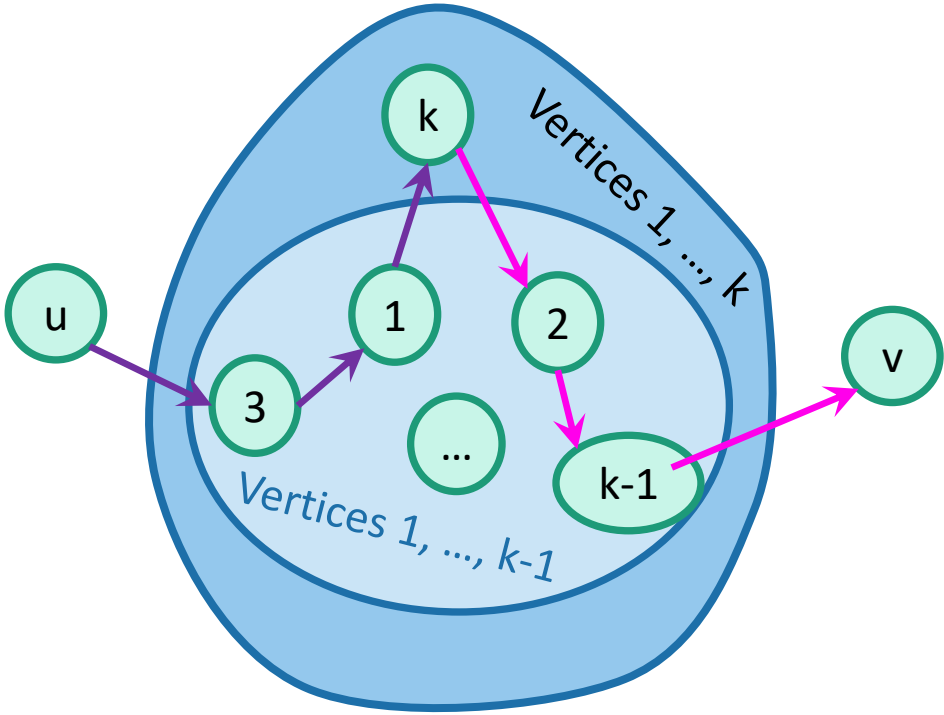
How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

Case 1: we don't need vertex k.



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

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 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 find 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
  - $D^{(0)}[u,v]$ = 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.

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

Work out the details of a proof!

- 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" 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.`

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 find the thing you want.
  - Fill in a table, starting with the smallest sub-problems and building up.
- (**Steps 4 and 5** coming next lecture…)