Advanced Data Structures and Algorithms

Dynamic Programming and

Floyd-Warshall (All-Pairs Shortest Paths: APSP)

Today

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

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

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,…

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

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

Candidate algorithm

- **def** Fibonacci(n):
  - **if** n == 0 or 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 $n \geq 2$

Candidate algorithm

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

Running time?

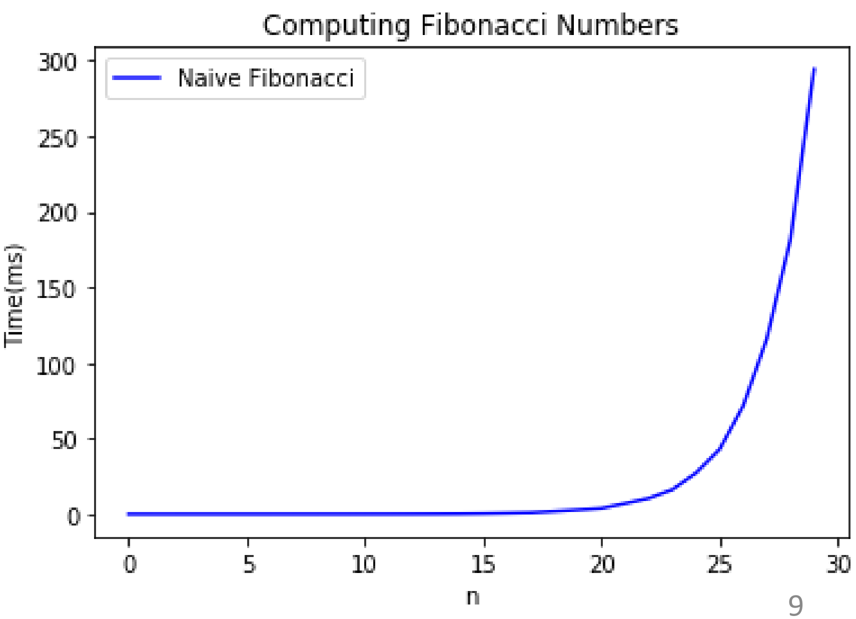
- T(n) = T(n-1) + T(n-2) + O(1)
- T(n) $\geq$ T(n-1) + T(n-2) for n $\geq$ 2
- So T(n) grows *at least* as fast as the Fibonacci numbers themselves…
- Fun fact, that's like ϕ^n where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Candidate algorithm

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

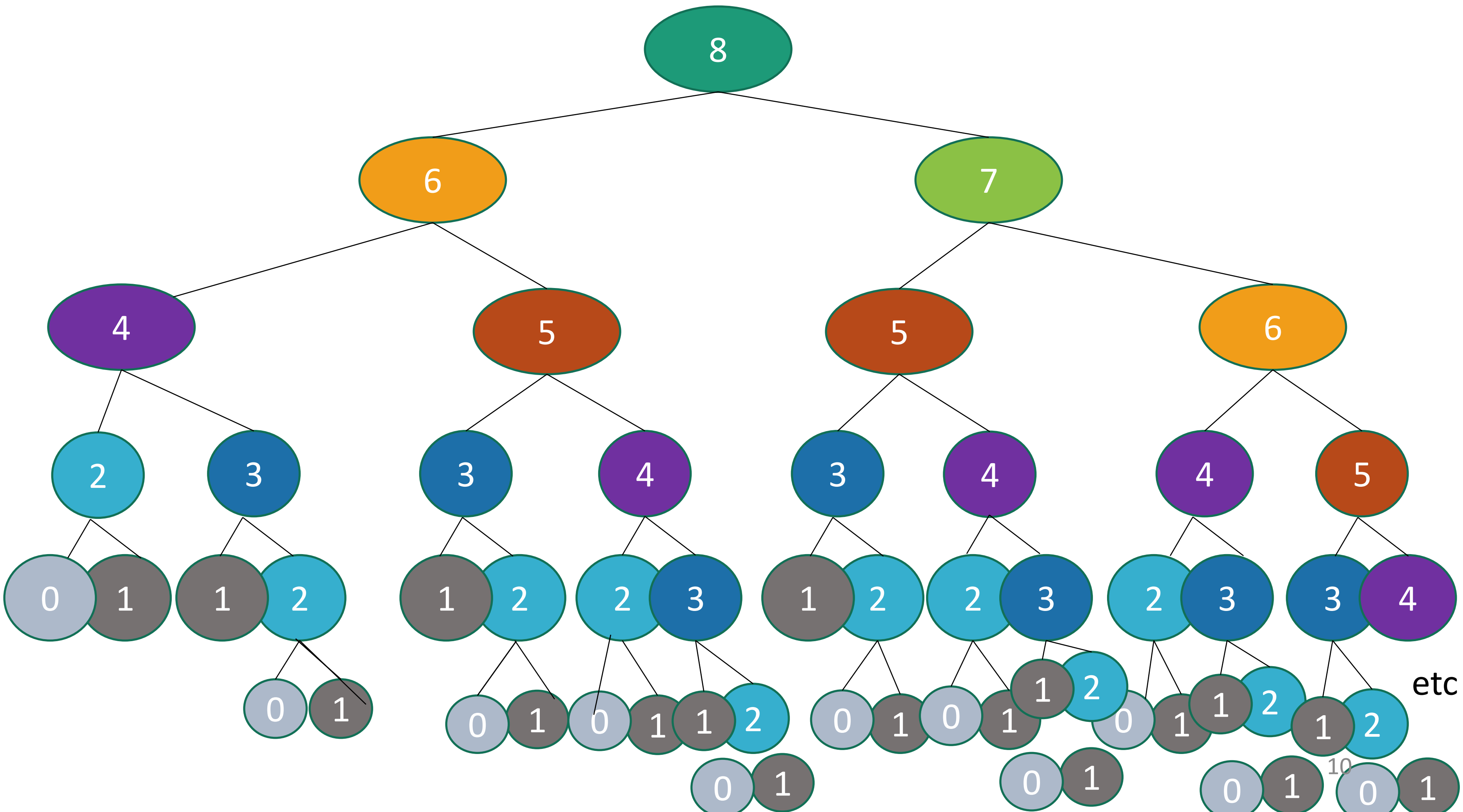
Running time?

- $T(n) = T(n\text{-}1) + T(n\text{-}2) + O(1)$
- $T(n) \geq T(n\text{-}1) + T(n\text{-}2)$ for $n \geq 2$
- 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?

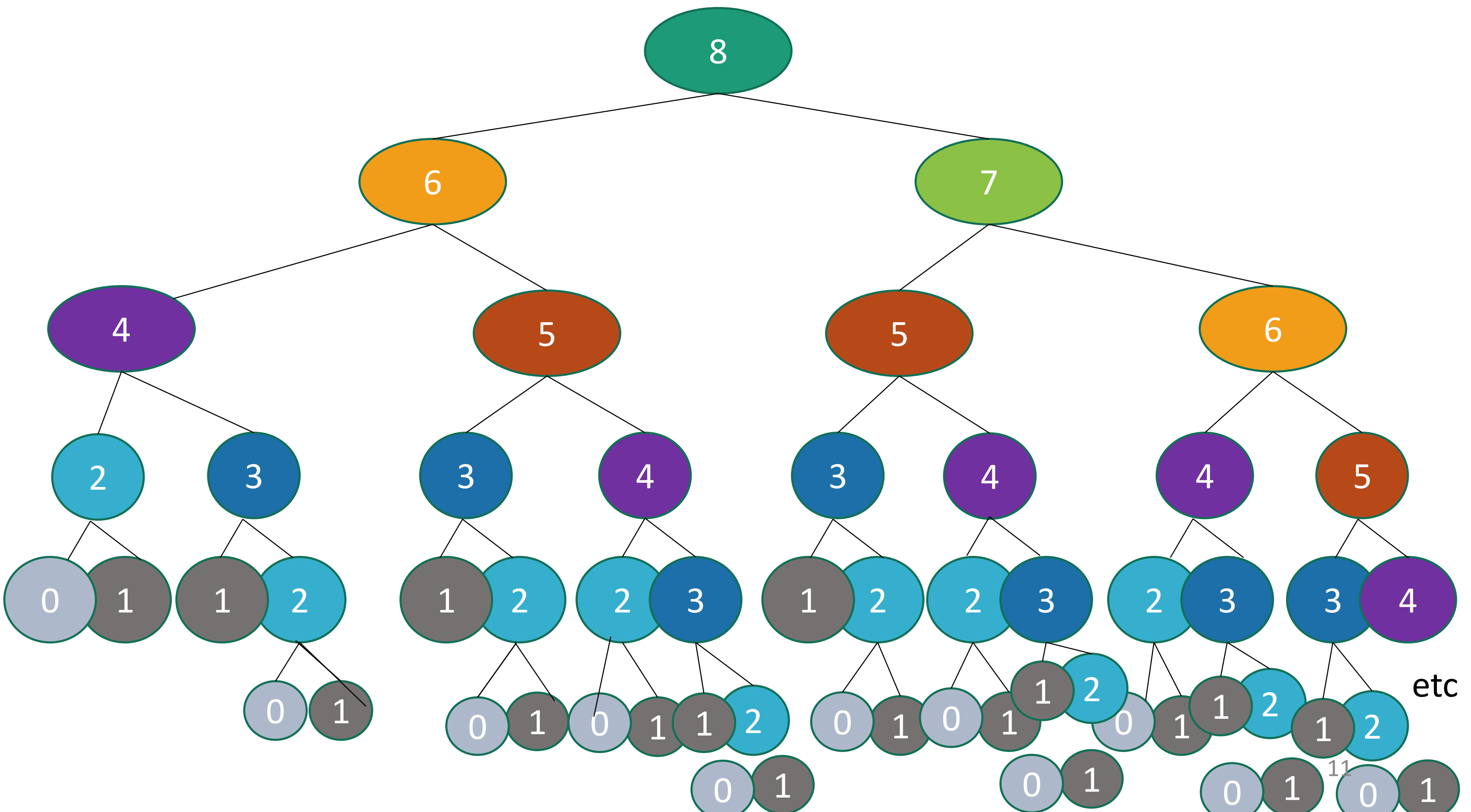
Consider Fib(8)



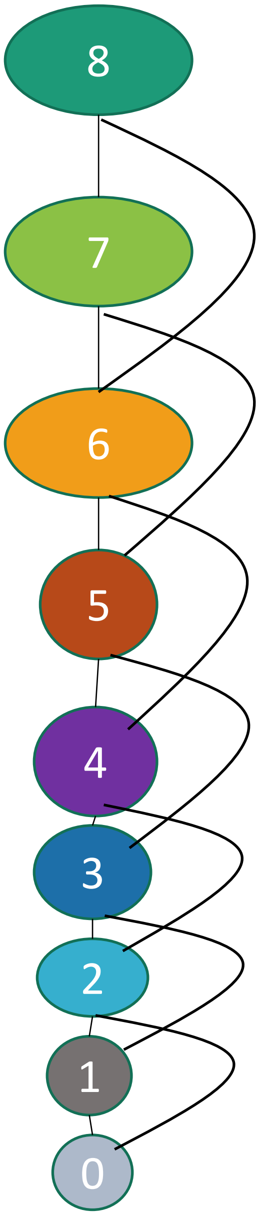
What's going on?

Consider Fib(8)

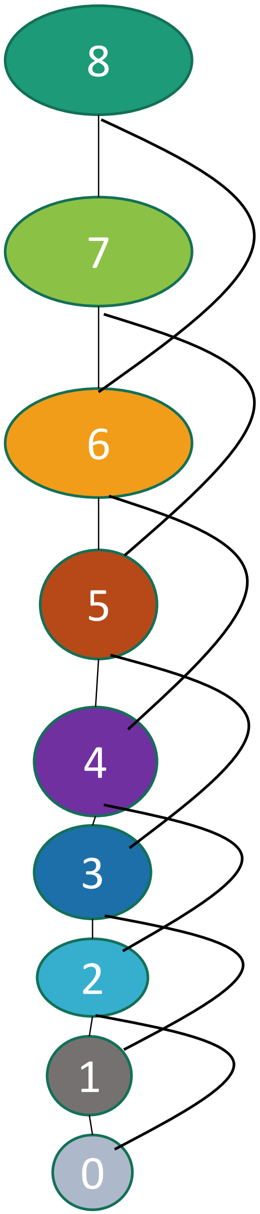
That's a lot of repeated computation!



Maybe this would be better:



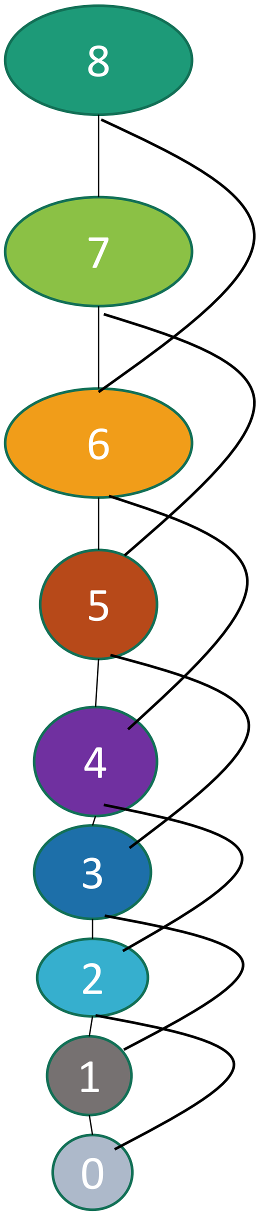
Maybe this would be better:



def fasterFibonacci(n):

- F = [1, 1, None, None, …, None]
  - \\ F has length n + 1
- **for** i = 2, …, n:
  - F[i] = F[i-1] + F[i-2]
- **return** F[n]

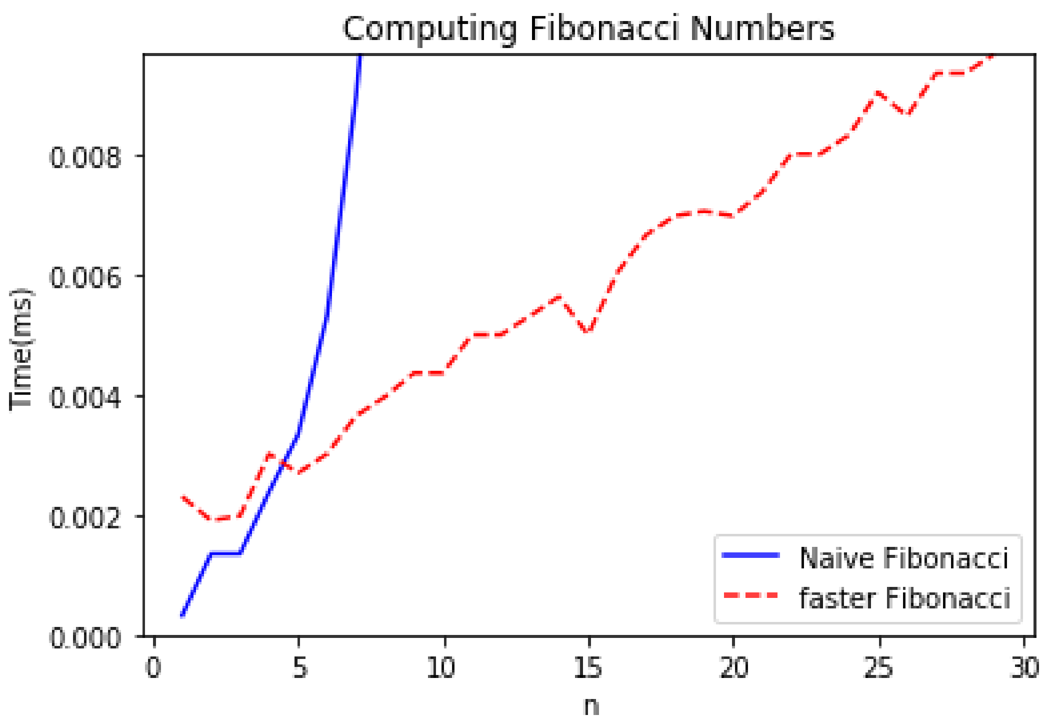
Maybe this would be better:



def fasterFibonacci(n):

- F = [1, 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*?

What is *dynamic programming*?

- It is an algorithm design paradigm
  - like divide-and-conquer is an algorithm design paradigm.

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:

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 $i \leq n$

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.

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:

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

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.

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].

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 $d^{(i)}[v]$.

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

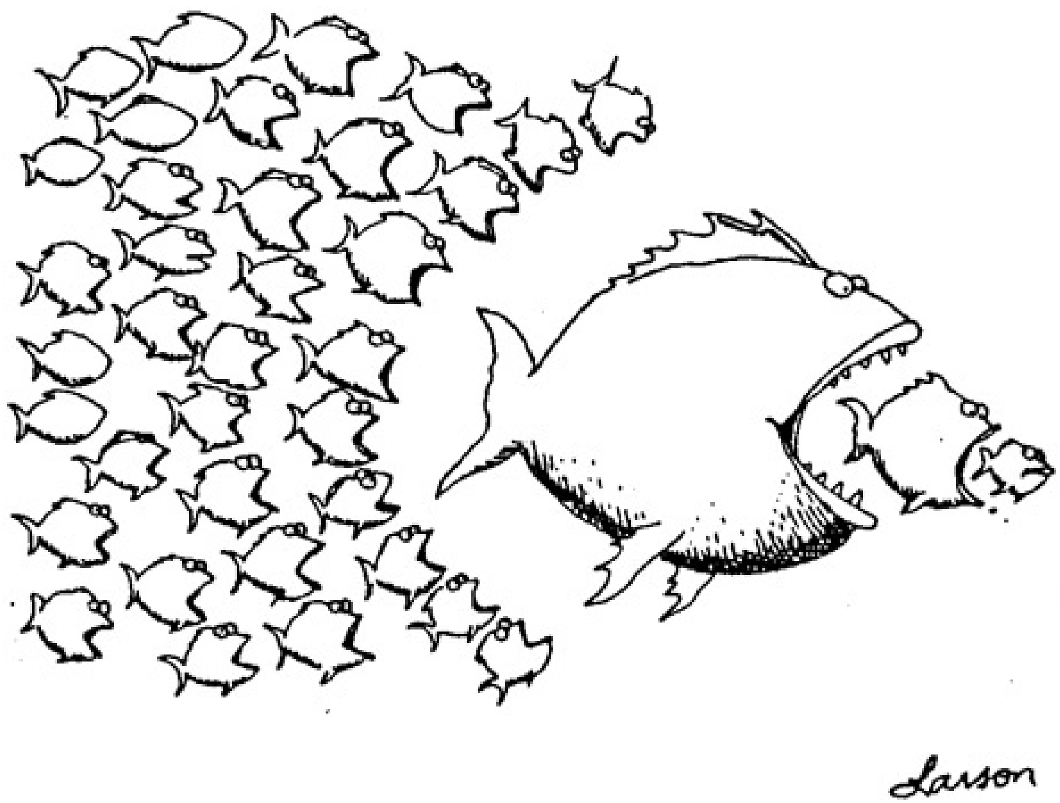
Elements of dynamic programming

- 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.

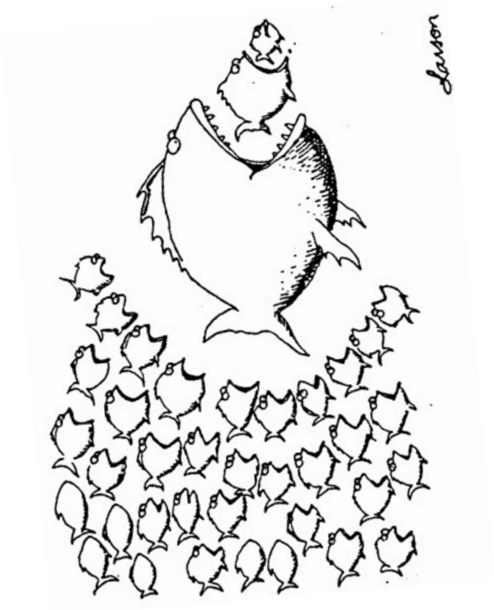
Two ways to think about and/or implement DP algorithms

- Top down
- Bottom up



Bottom up approach

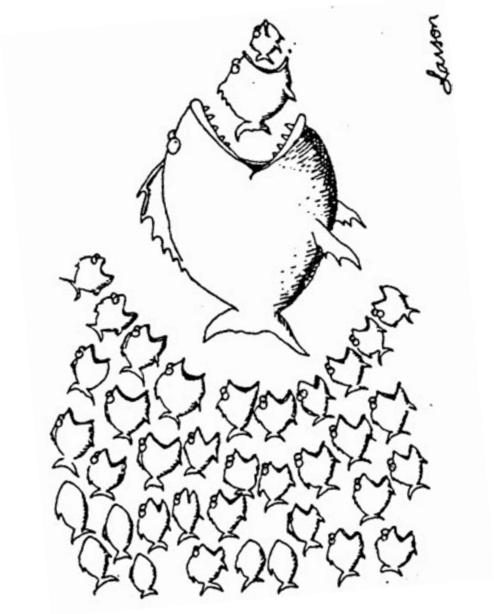
what we just saw.



Bottom up approach

what we just saw.

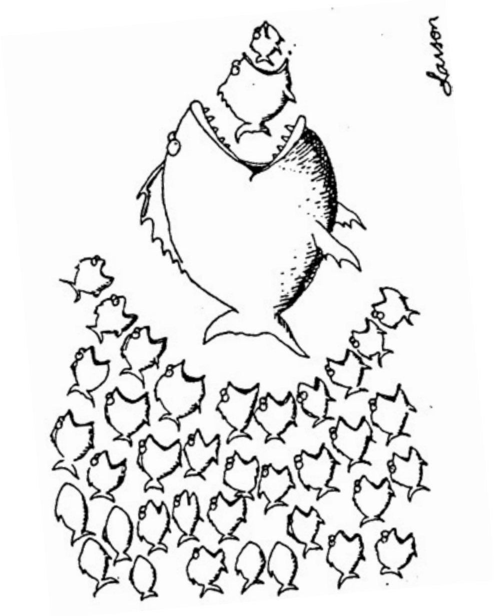
- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]



Bottom up approach

what we just saw.

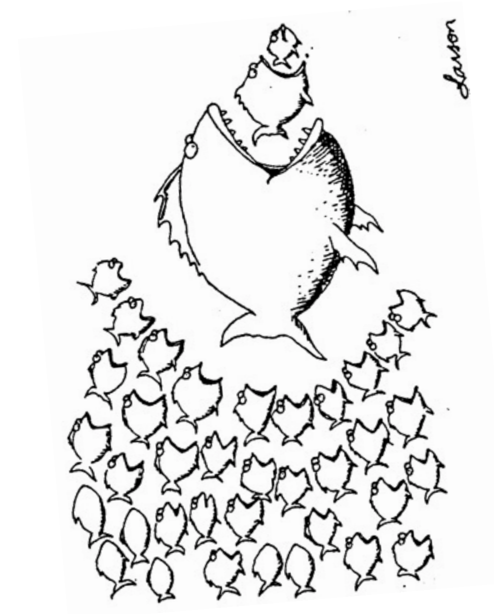
- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
  - fill in F[2]



Bottom up approach

what we just saw.

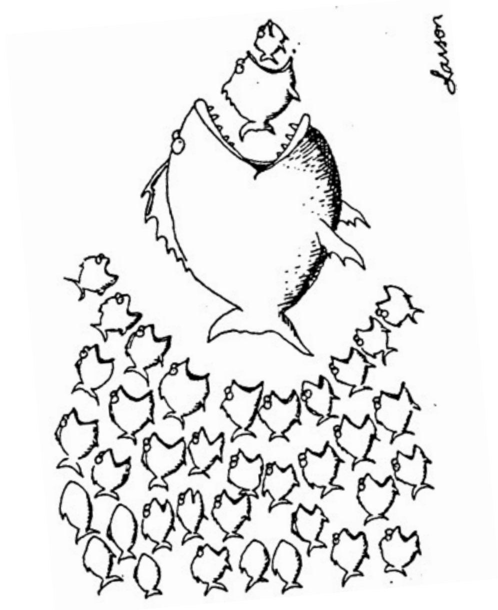
- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
  - fill in F[2]
- …
- Then bigger problems
  - fill in F[n-1]



Bottom up approach

what we just saw.

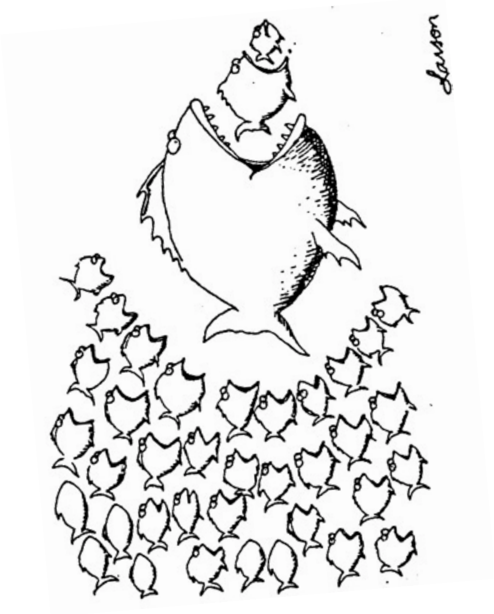
- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
  - fill in F[2]
- …
- Then bigger problems
  - fill in F[n-1]
- Then finally solve the real problem.
  - fill in F[n]



Bottom up approach

what we just saw.

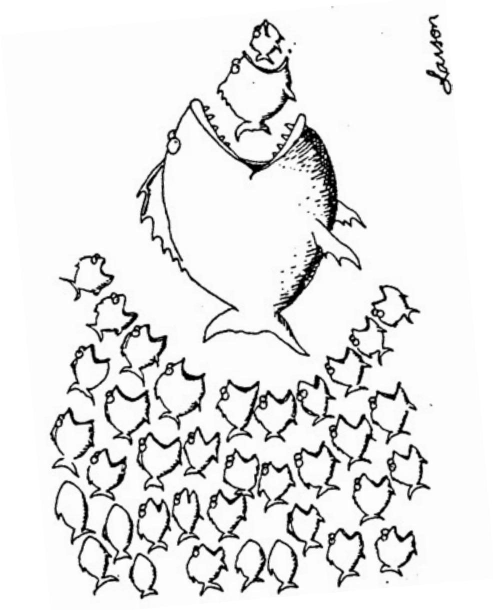
- For Bellman-Ford:
- Solve the small problems first
  - fill in $d^{(0)}$



Bottom up approach

what we just saw.

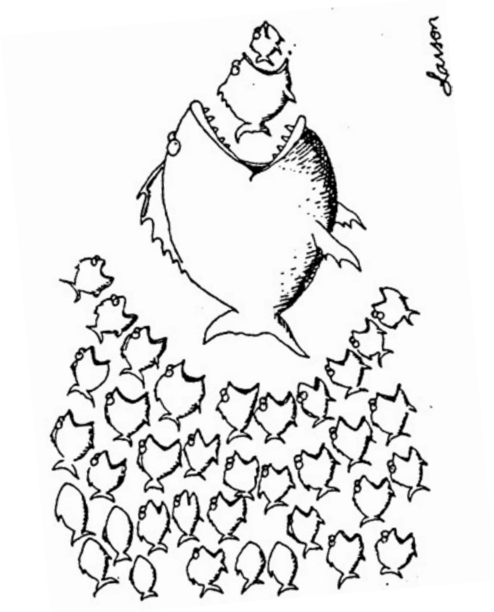
- For Bellman-Ford:
- Solve the small problems first
  - fill in $d^{(0)}$
- Then bigger problems
  - fill in $d^{(1)}$



Bottom up approach

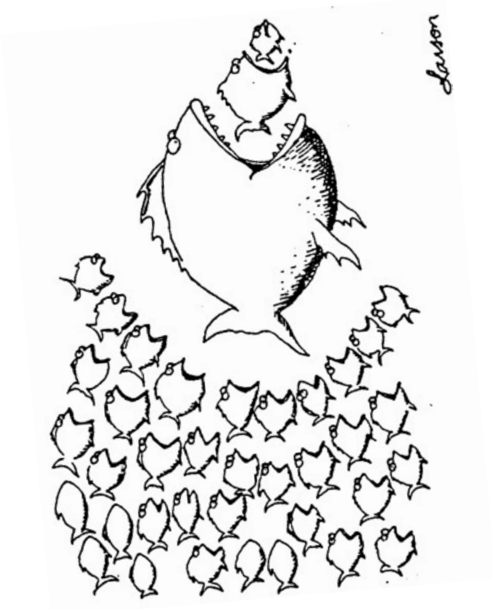
what we just saw.

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



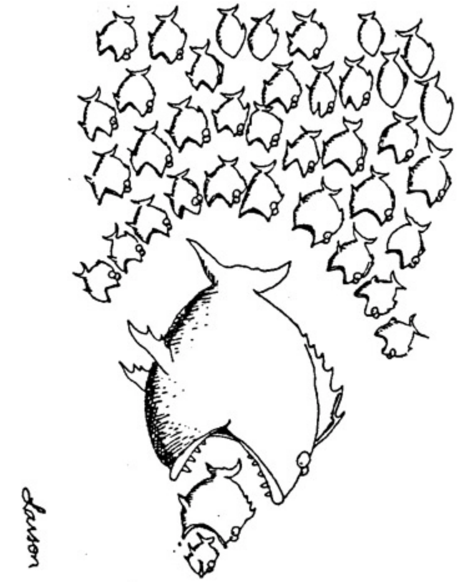
Bottom up approach

what we just saw.

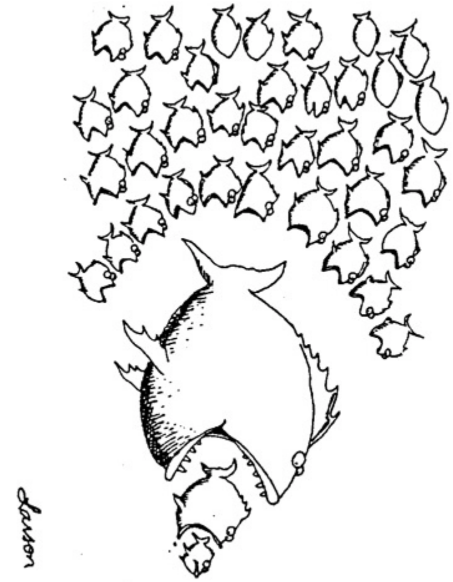


- For Bellman-Ford:
- Solve the small problems first
  - fill in $d^{(0)}$
- Then bigger problems
  - fill in $d^{(1)}$
- …
- Then bigger problems
  - fill in $d^{(n-2)}$
- Then finally solve the real problem.
  - fill in $d^{(n-1)}$

Top down approach

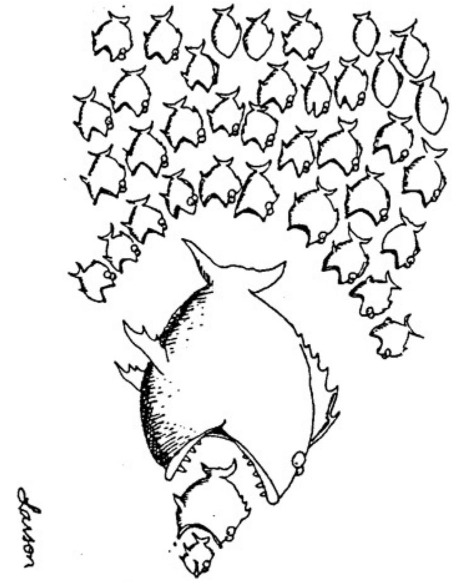


Top down approach



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

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..

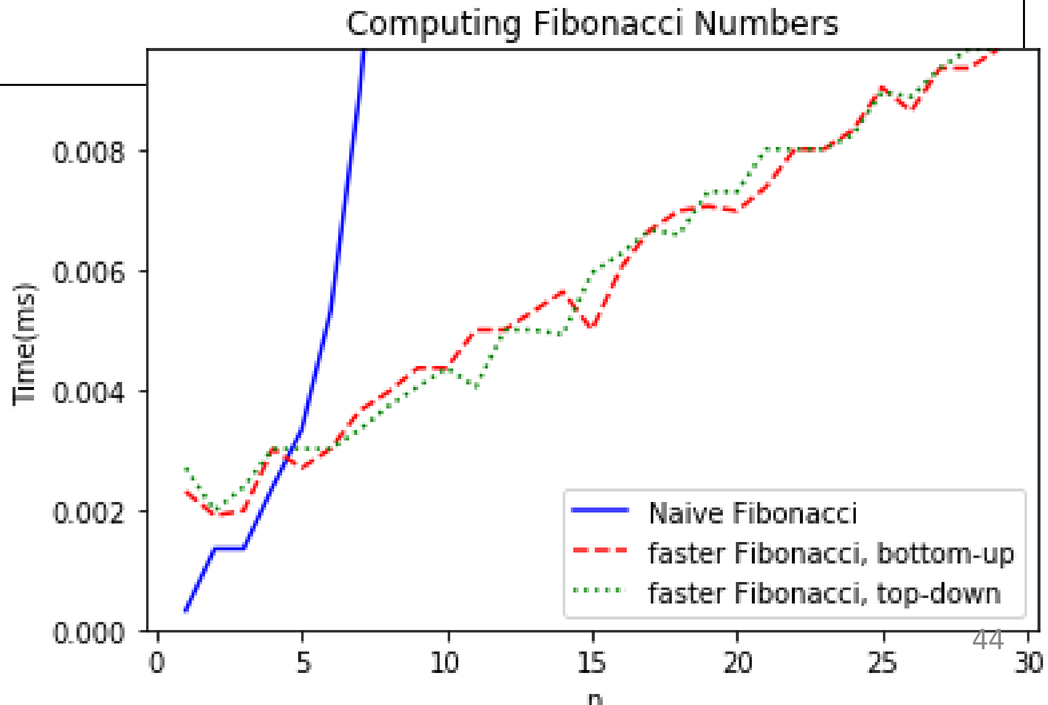
- The difference from divide and conquer:
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.
  - Aka, "**memorization**"

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]

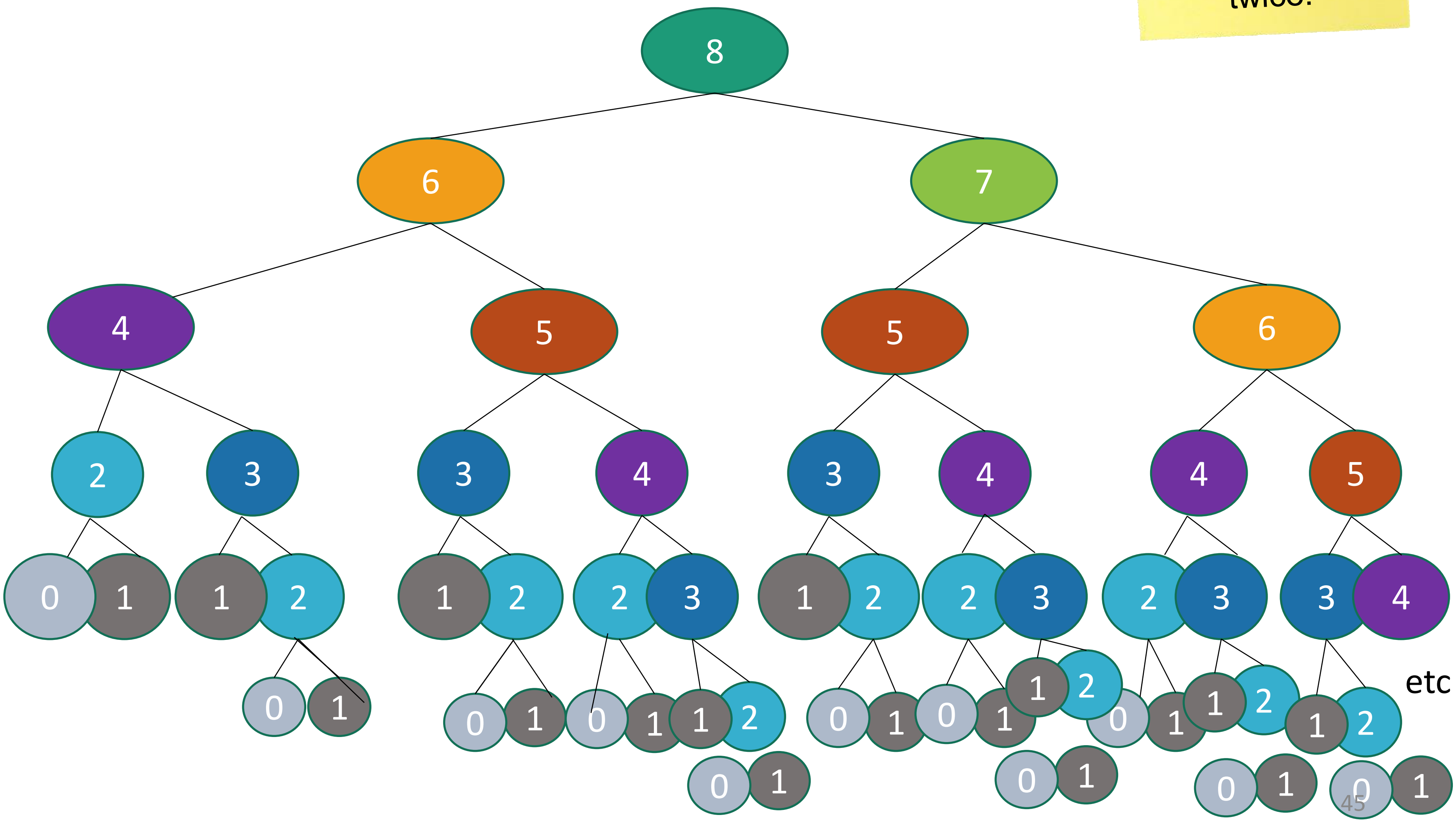
Example of top-down Fibonacci

- 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]



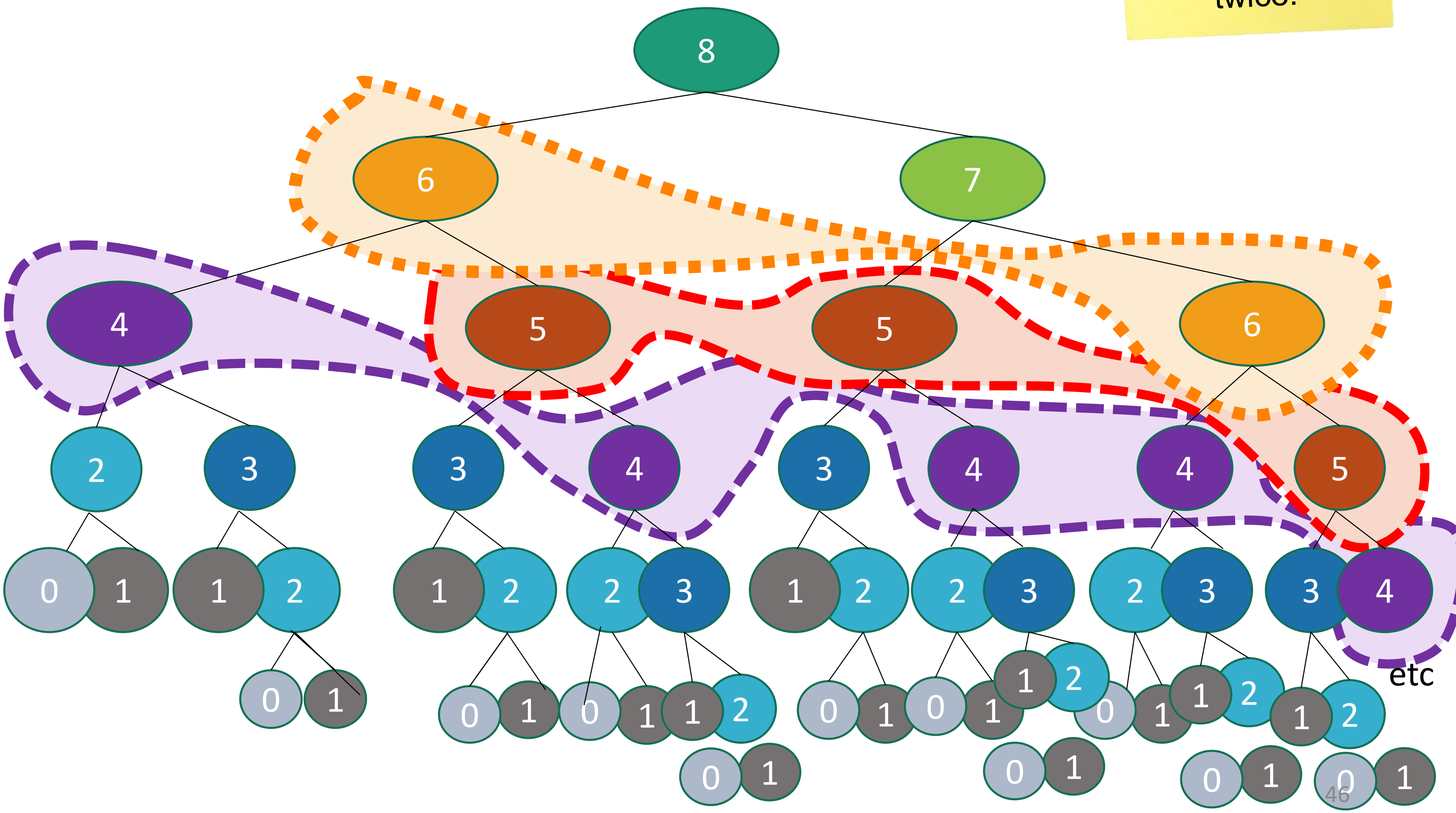
Memorization visualization

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



Memorization visualization

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



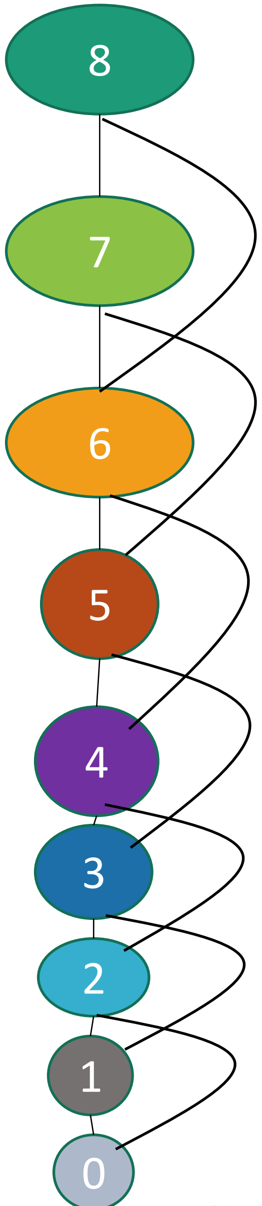
Memorization Visualization

ctd

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

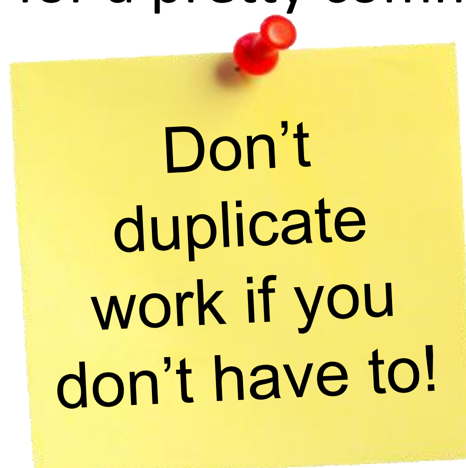
But otherwise treat it like the same old recursive algorithm.

- 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*” ?

Why “*dynamic programming*” ?



Manipulating computer code in an action movie?

Why "*dynamic programming*" ?

- Programming refers to finding the optimal "program."
  - as in, a shortest route is a *plan* aka a *program*.

Manipulating computer code in an action movie?

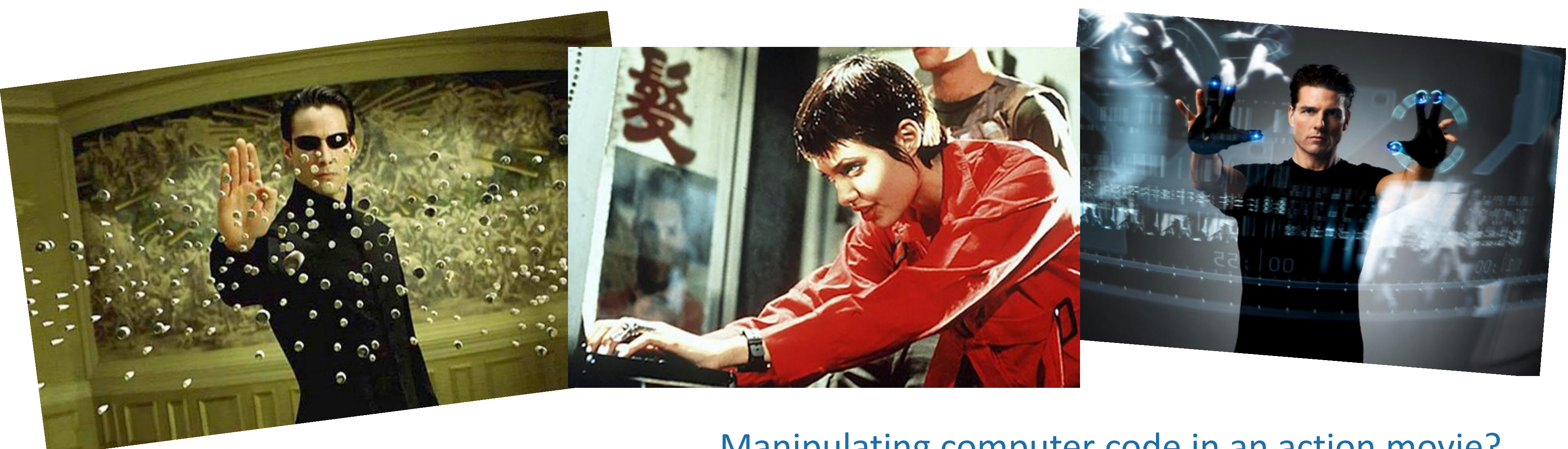
Why “*dynamic programming*” ?

- 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.

Manipulating computer code in an action movie?

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, and projects needed flashy names to get funded.

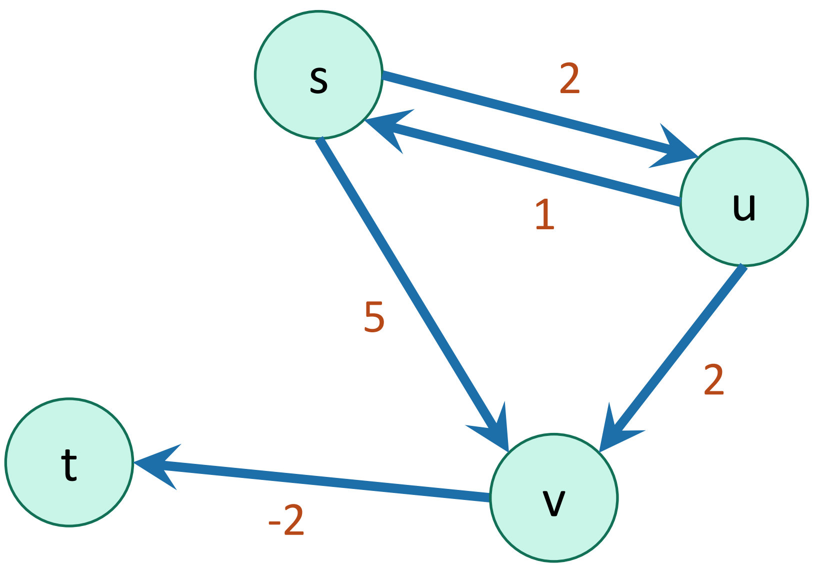
Floyd-Warshall Algorithm

Another example of Dynamic Programming

Floyd-Warshall Algorithm

Another example of Dynamic Programming

- 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.



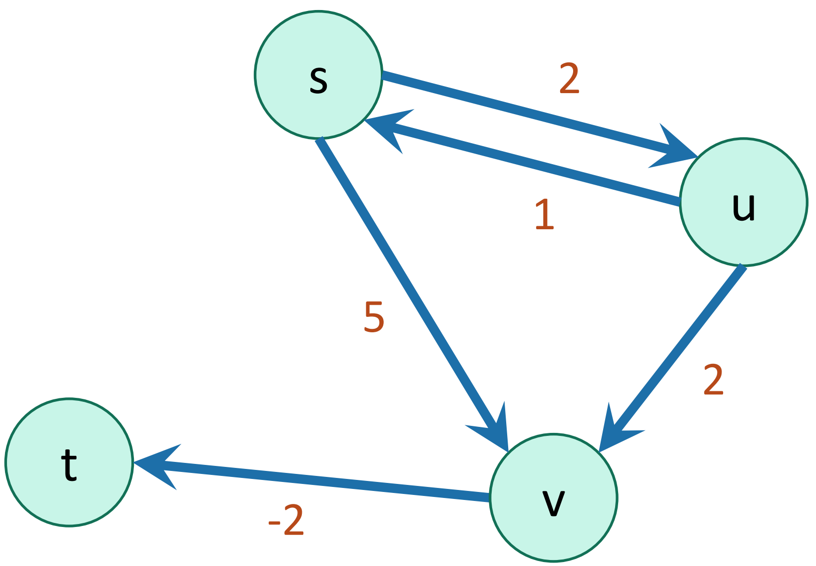
Floyd-Warshall Algorithm

Another example of Dynamic Programming

- 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 Dynamic Programming

- 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.

Floyd-Warshall Algorithm
Another example of Dynamic Programming

- 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 ?

Floyd-Warshall Algorithm
Another example of Dynamic Programming

- 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 $m=n^2$

Floyd-Warshall Algorithm
Another example of Dynamic Programming

- 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.

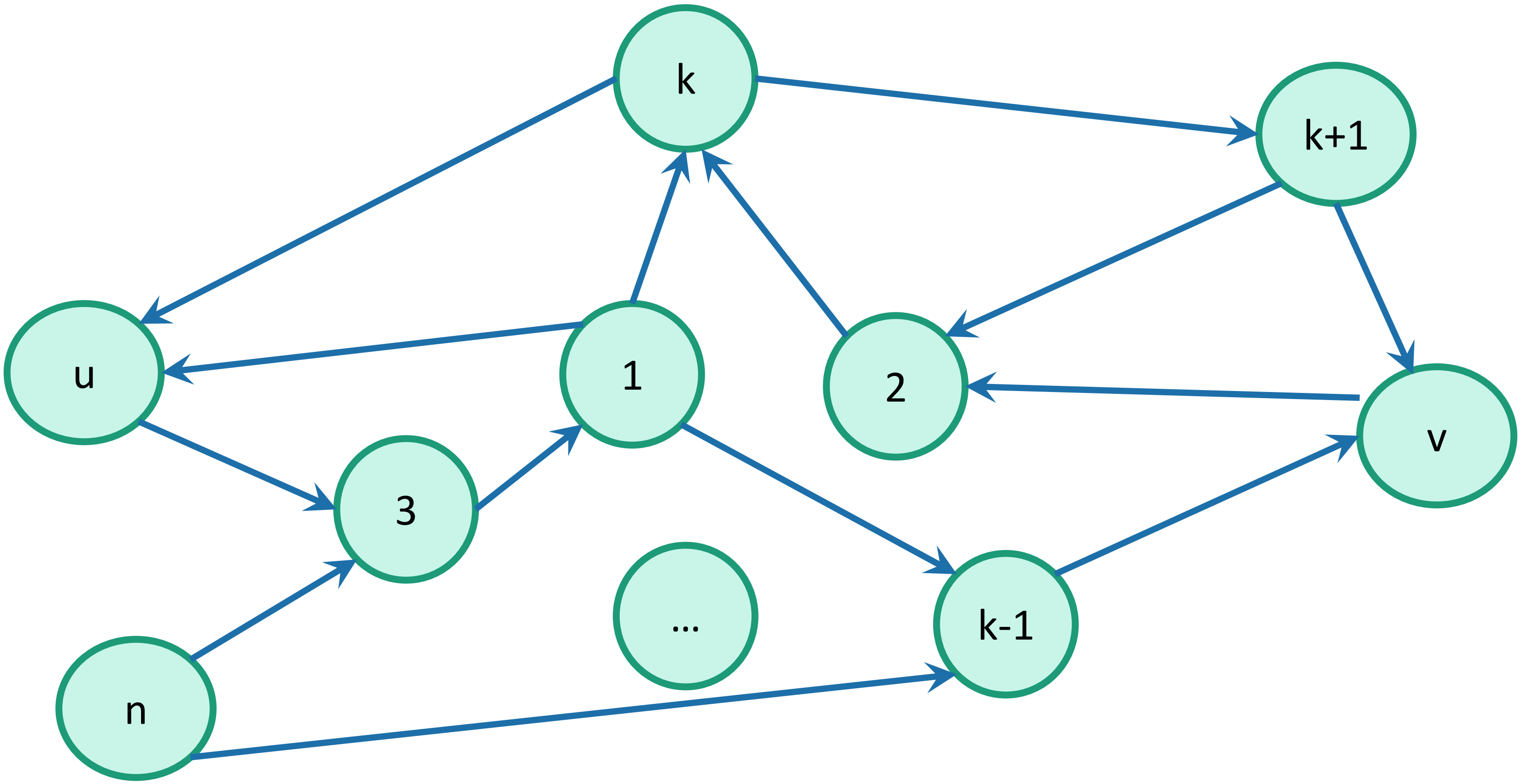
- Naïve solution (if we want to handle negative edge weights):
  - For all s in G:
    - Run Bellman-Ford on G starting at s.

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

Can we do better?

Label the vertices 1,2,…,n

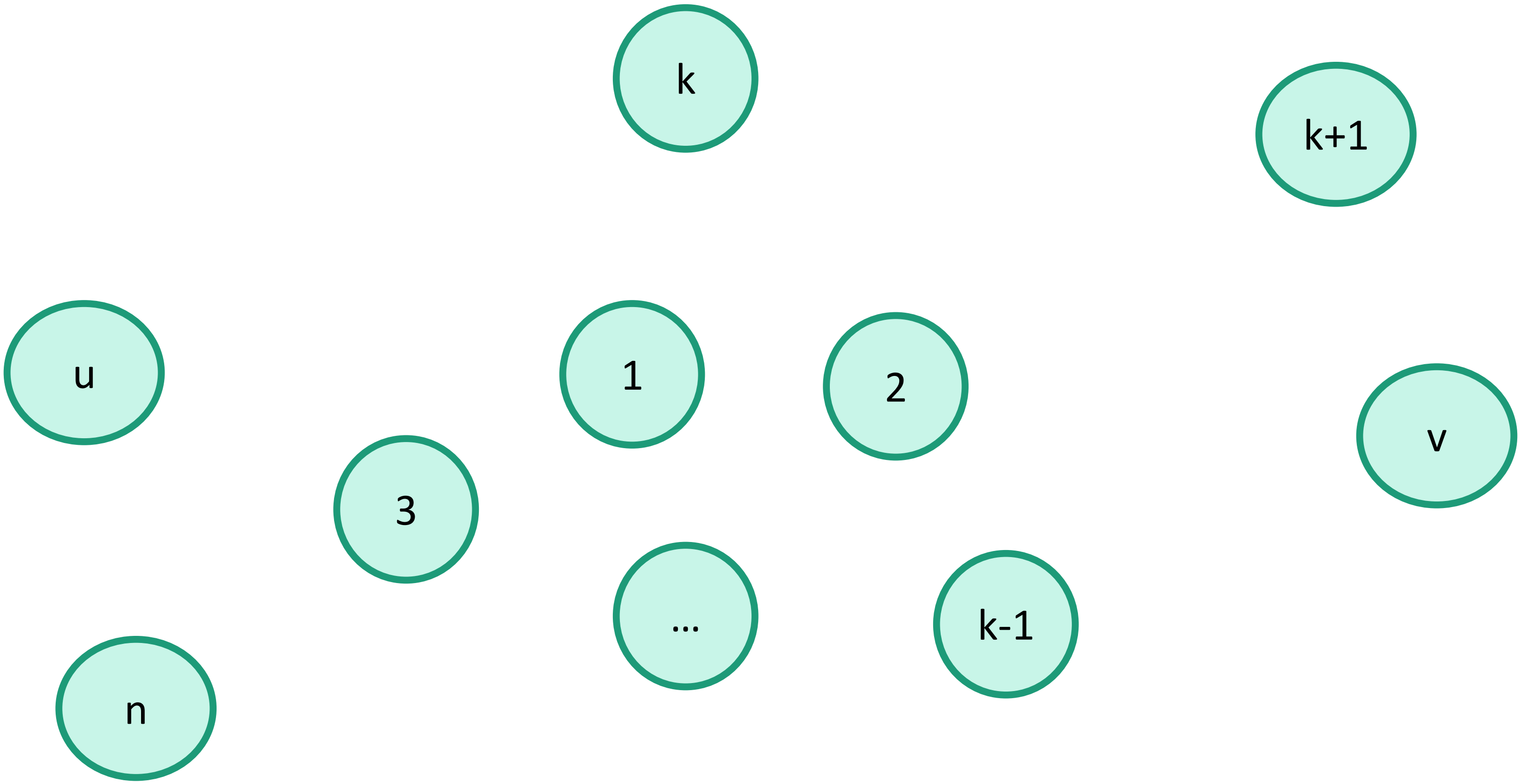
Optimal substructure



Optimal substructure

Label the vertices 1,2,…,n

(We omit some edges in the picture below – meant to be a cartoon, not an example).



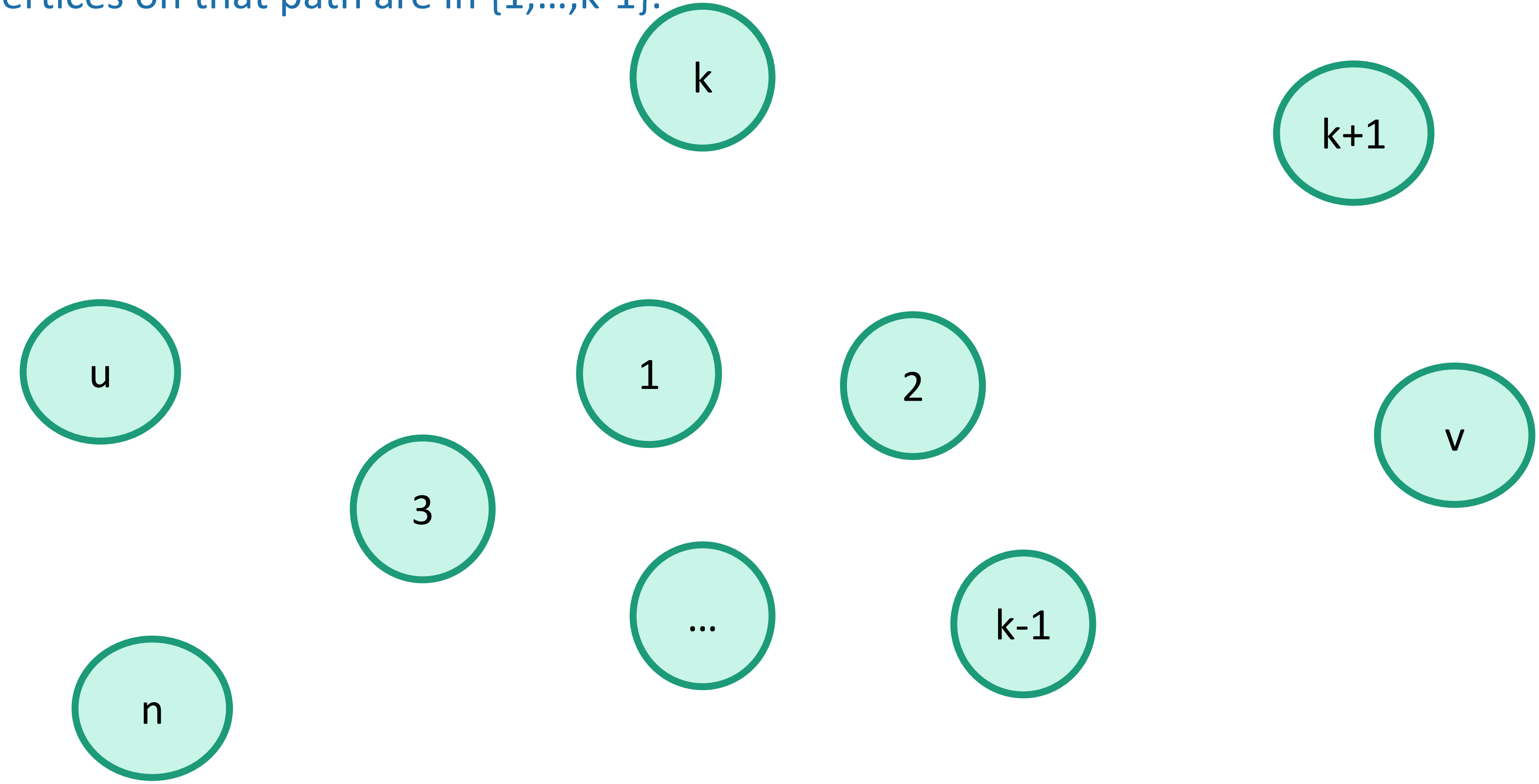
Optimal substructure

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}.



Optimal substructure

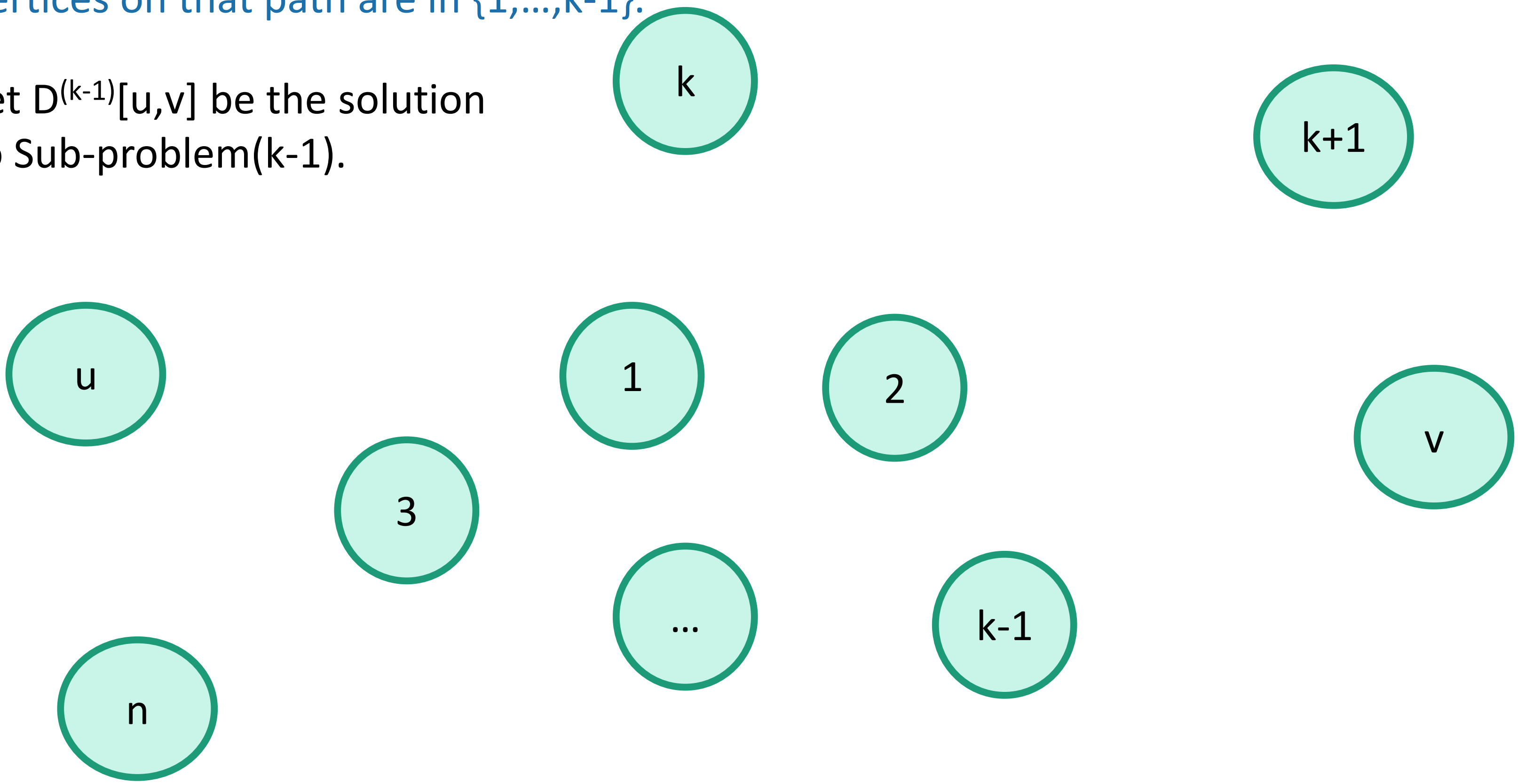
Label the vertices 1,2,…,n

(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).



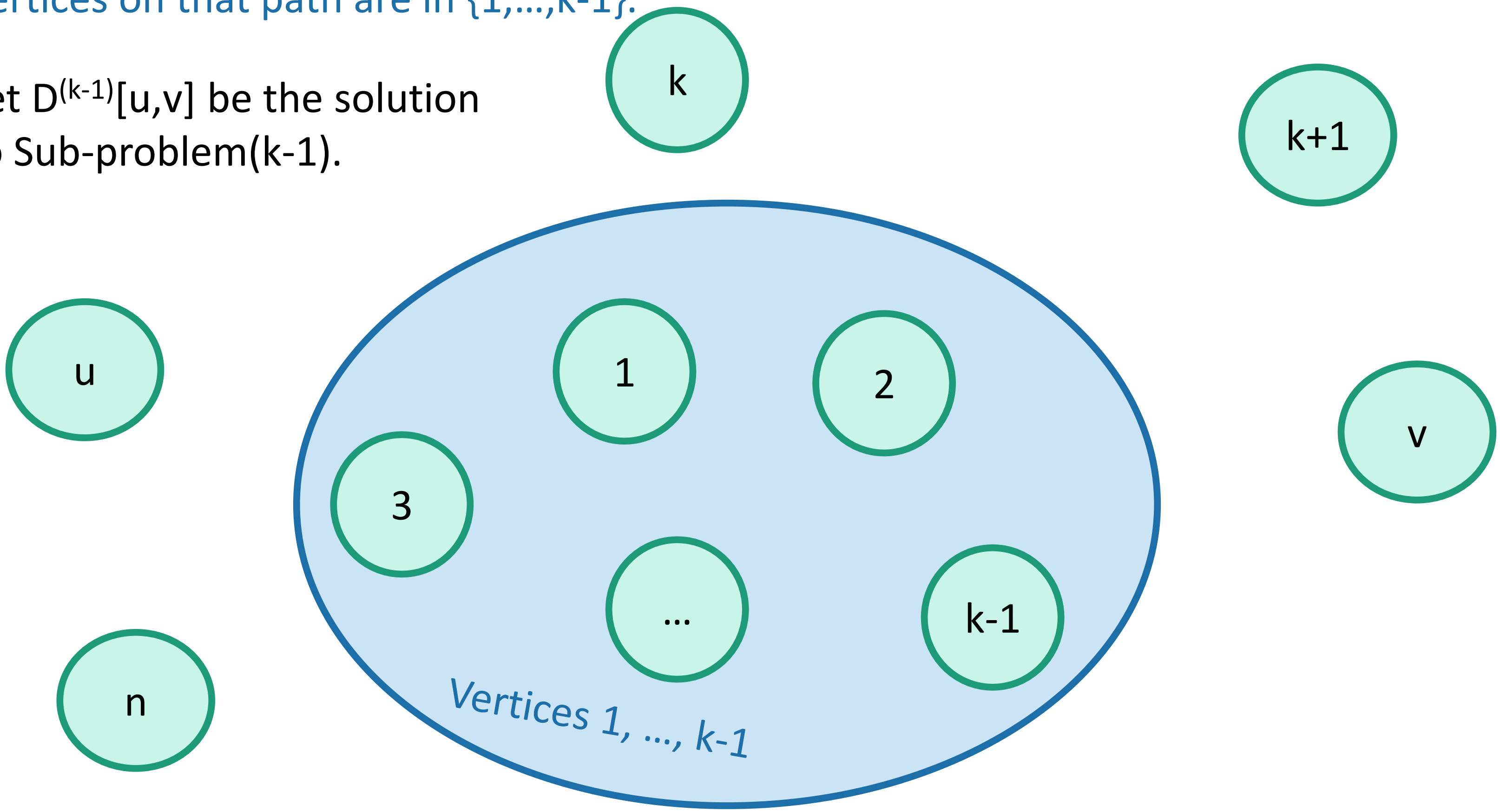
Optimal substructure

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).



Optimal substructure

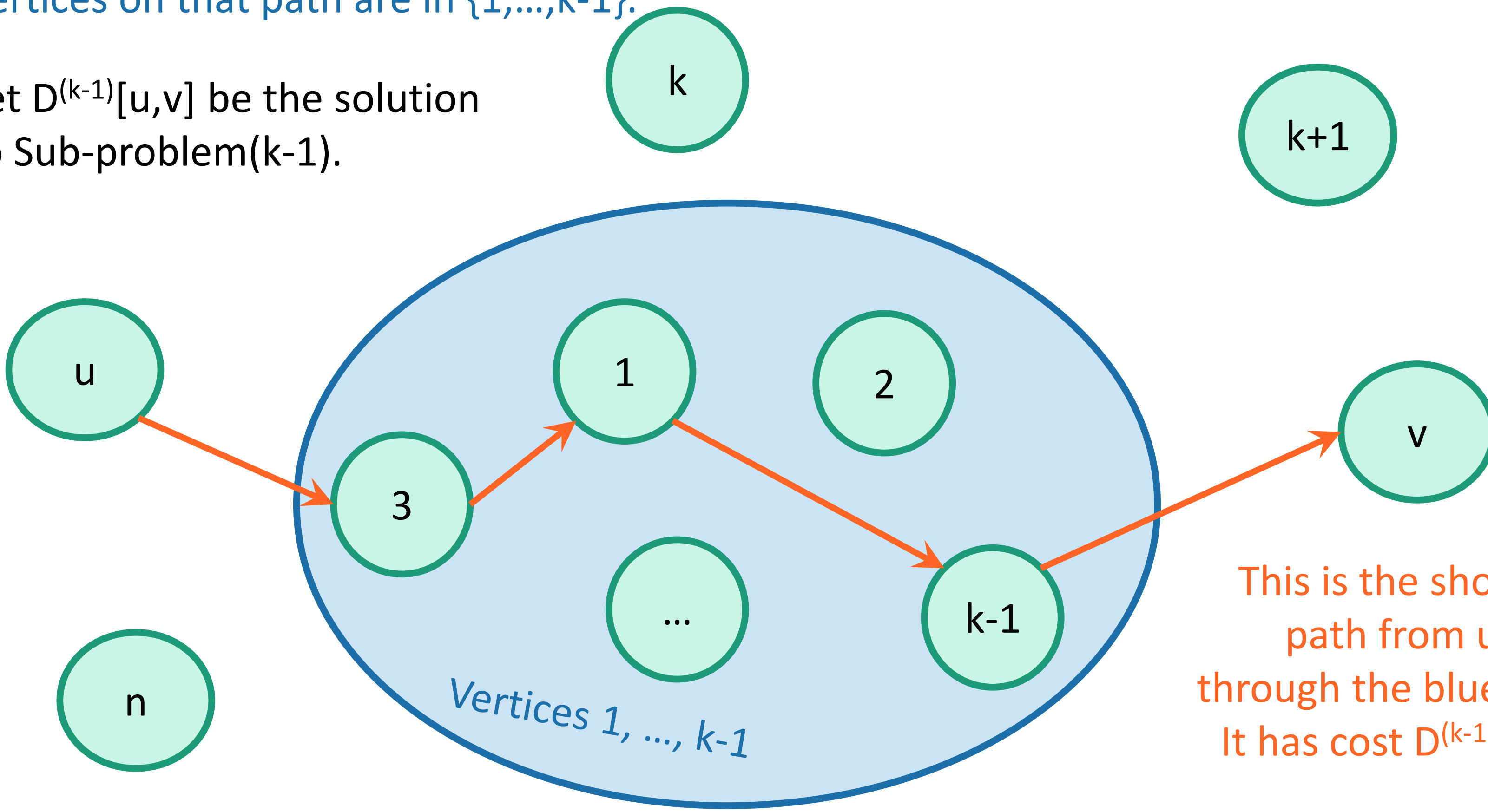
Label the vertices 1,2,…,n

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

Optimal substructure

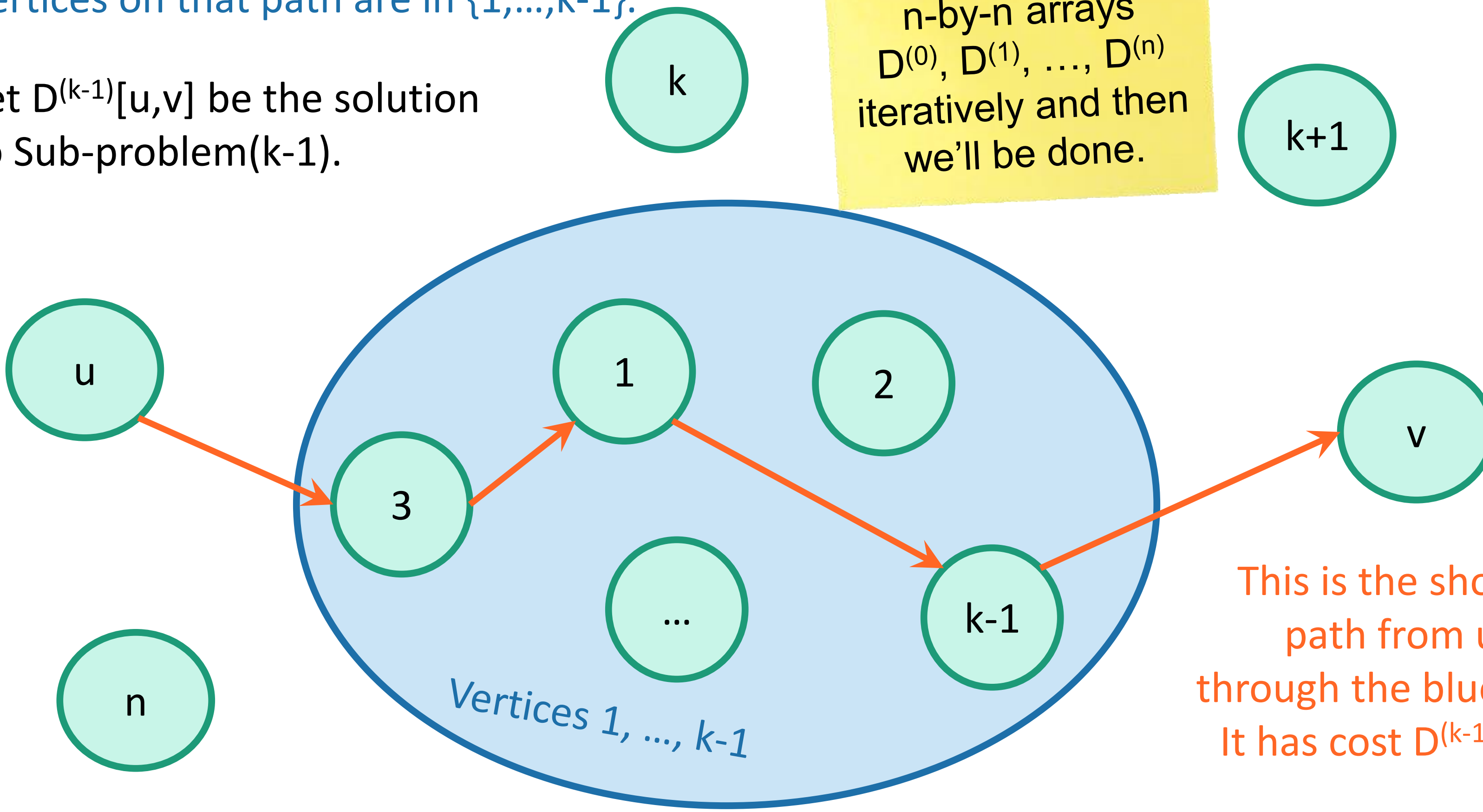
Label the vertices 1,2,…,n

(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).

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.



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

Optimal substructure

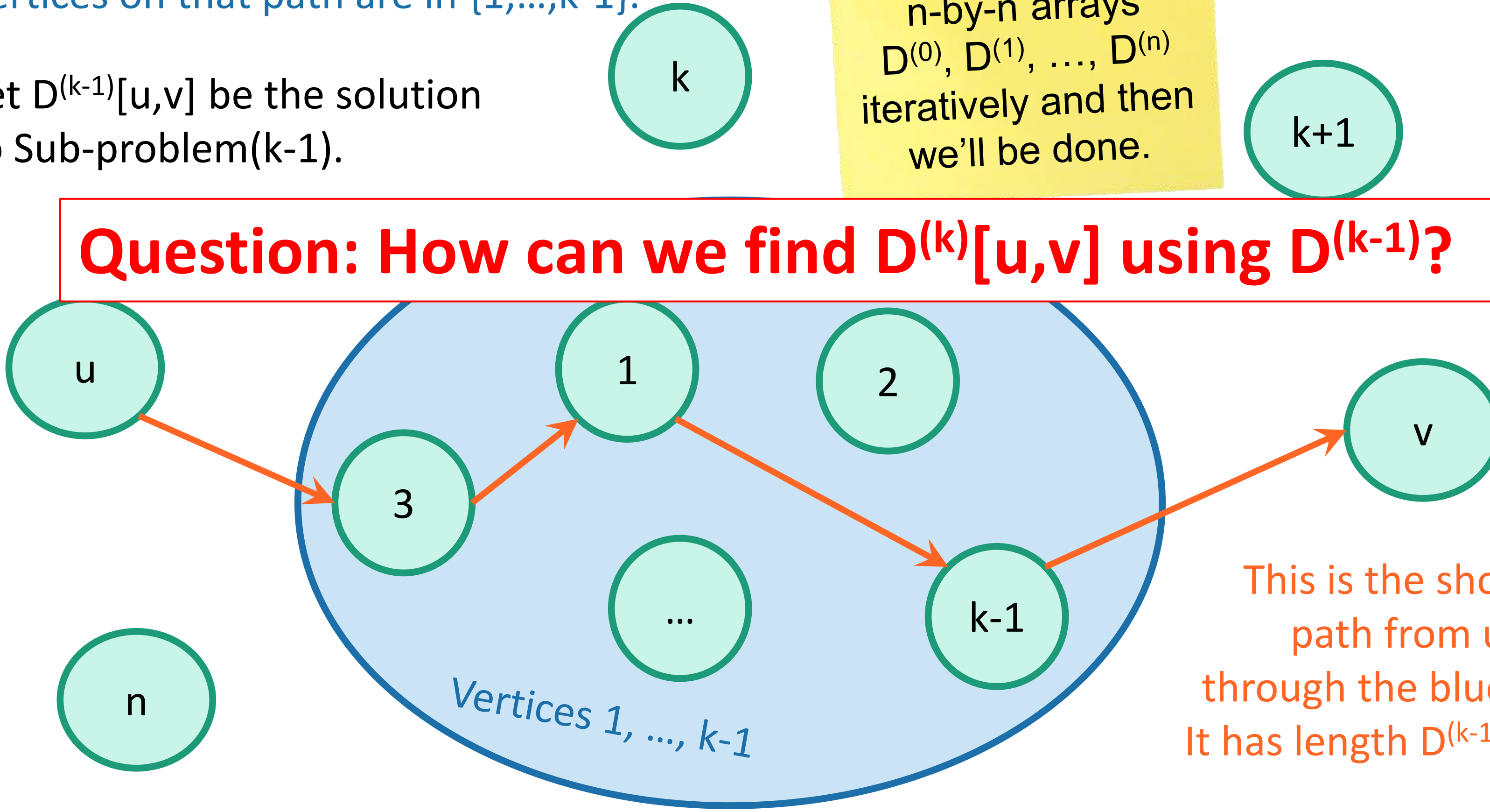
Label the vertices 1,2,…,n
(We omit some edges in the picture below – meant to be a cartoon, not an example).

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).

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.

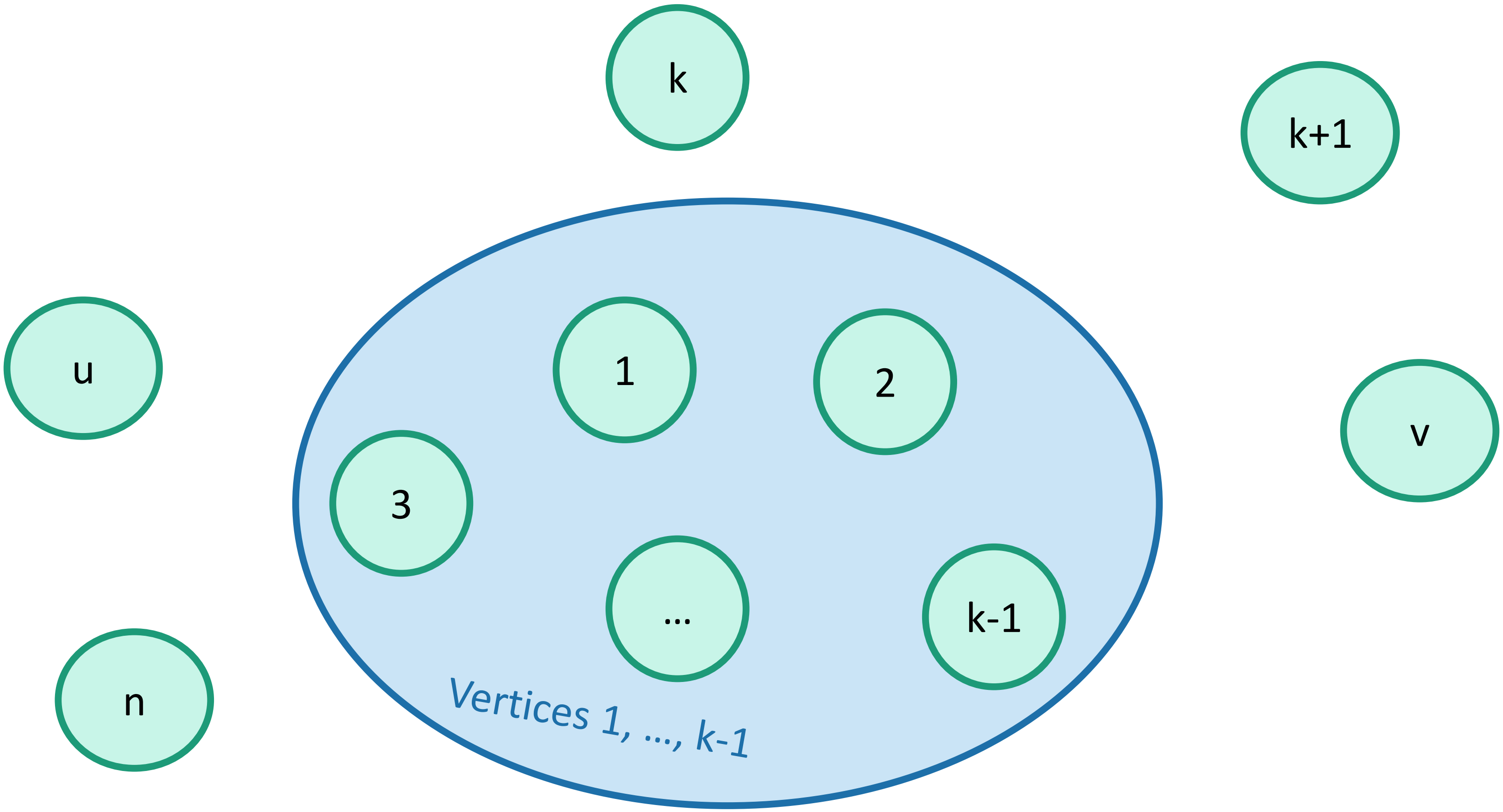
Question: How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?



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

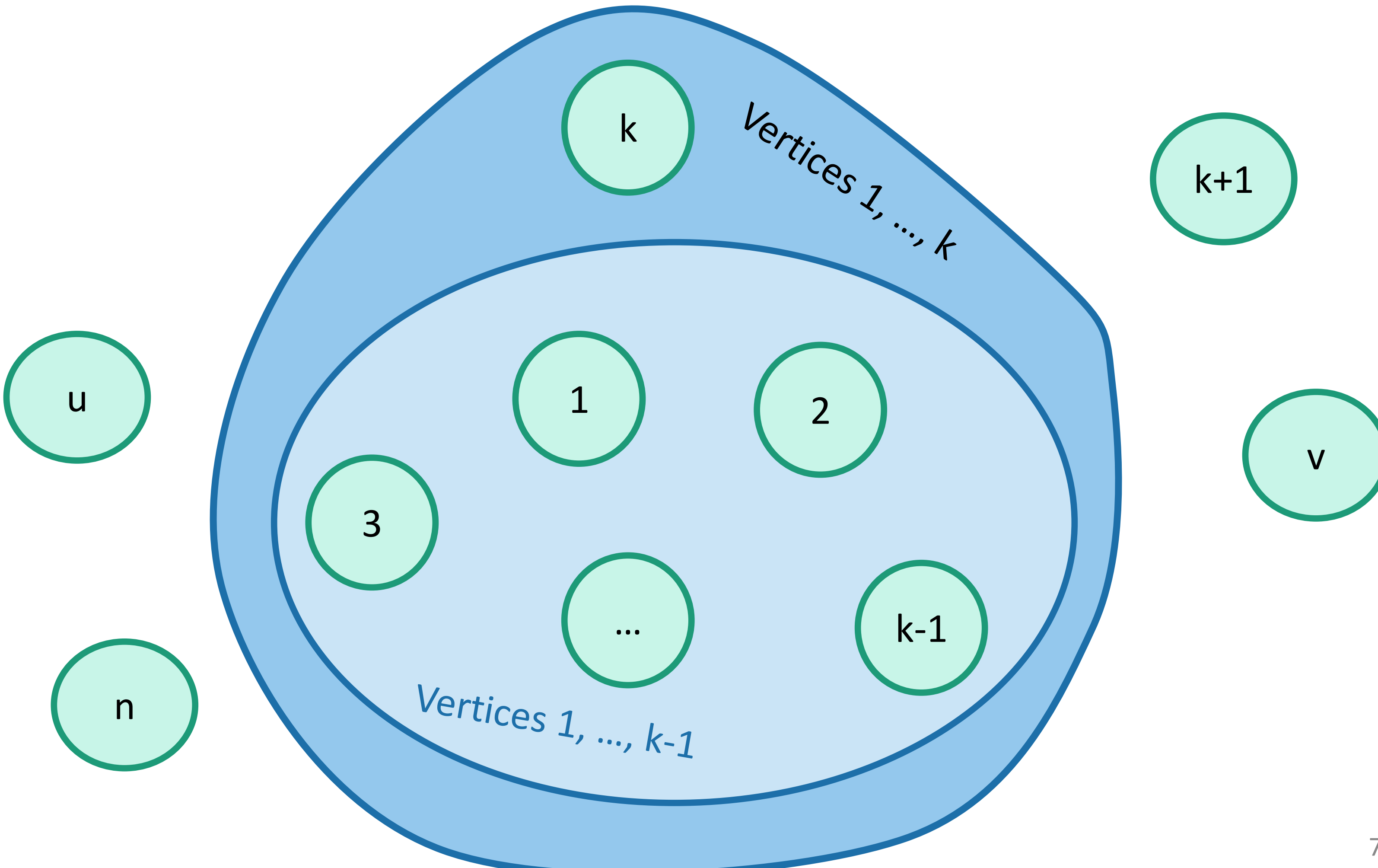
How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

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

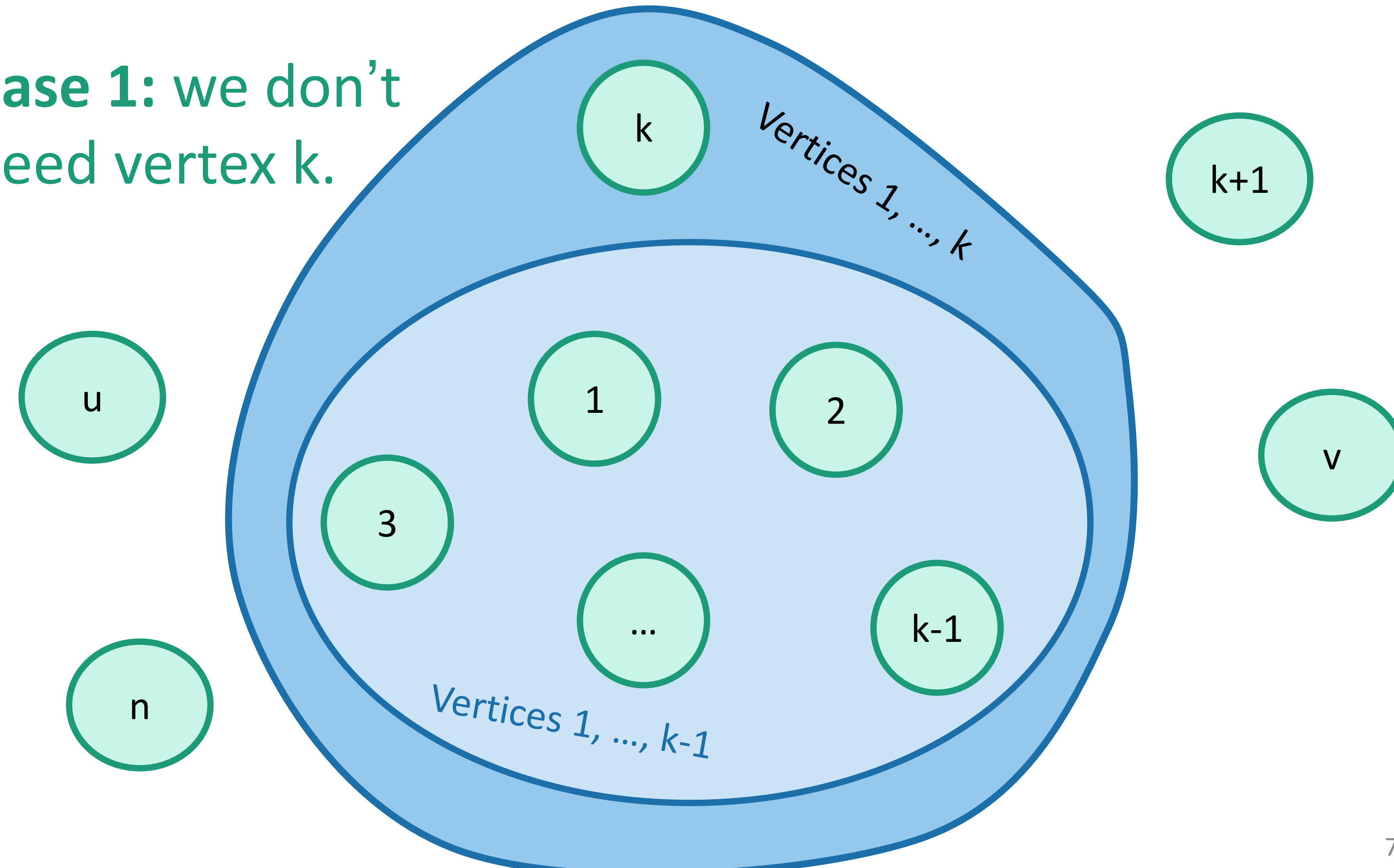
$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.



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}.

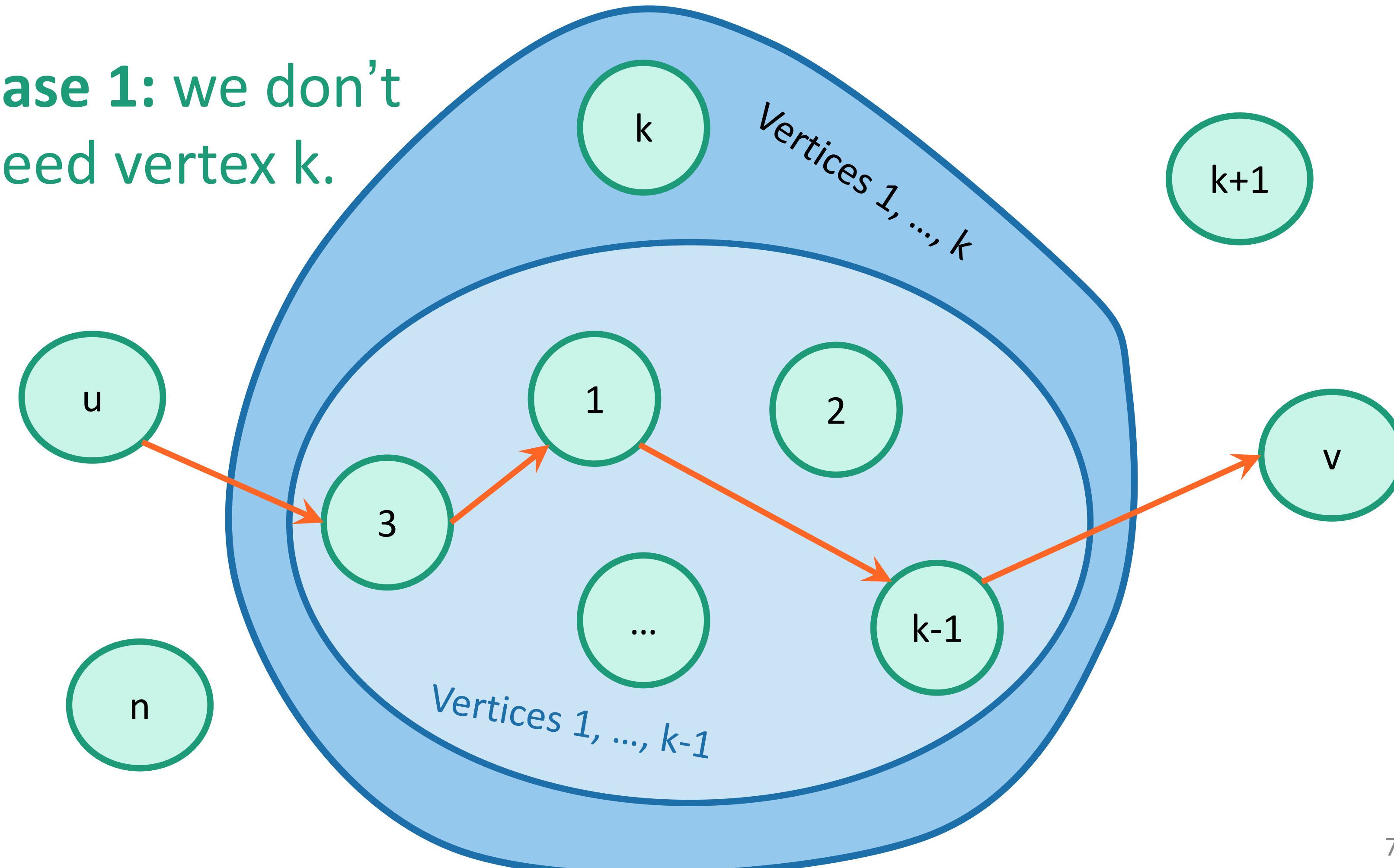
Case 1: we don't need vertex k.



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}.

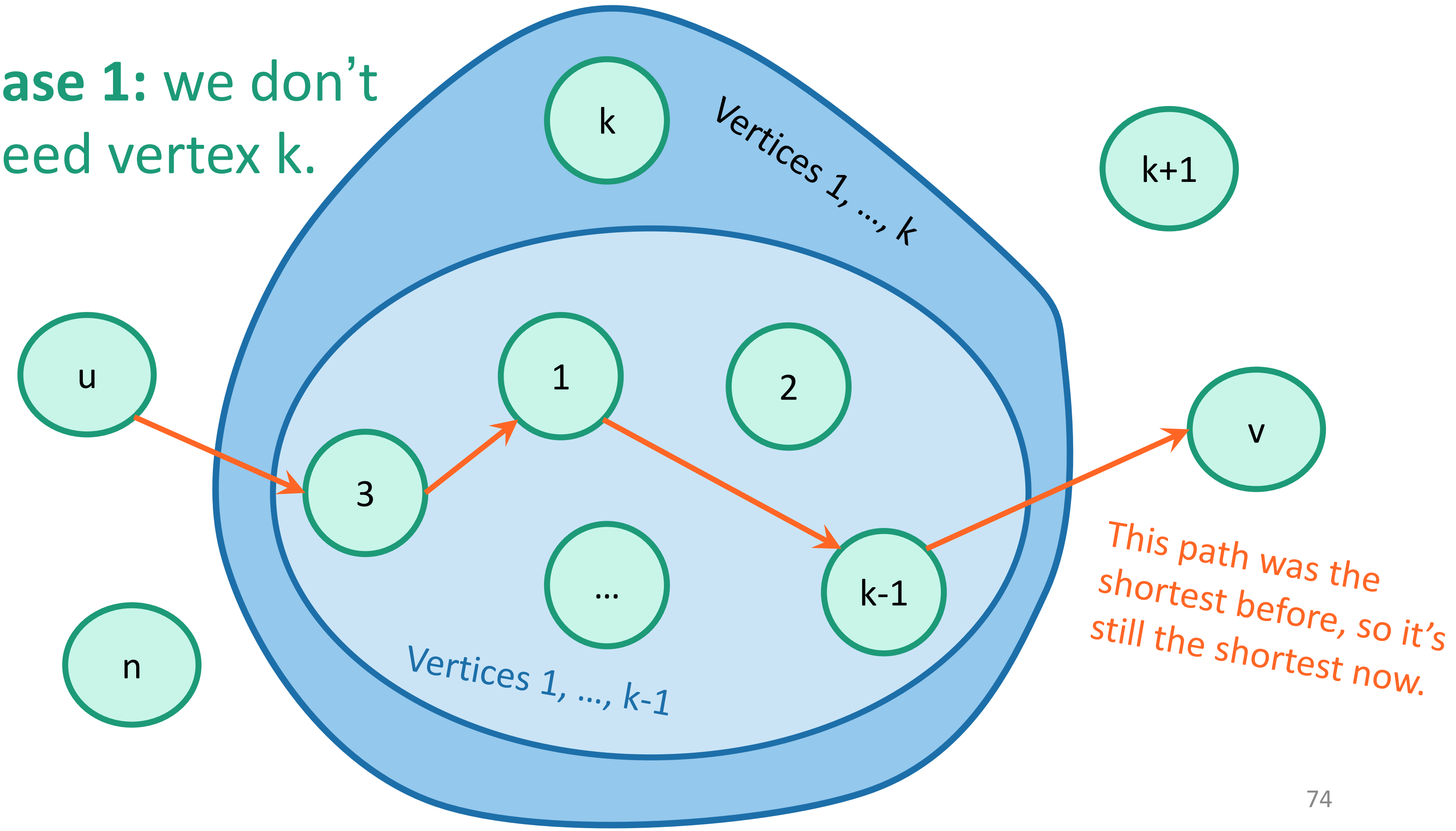
Case 1: we don't need vertex k.



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}.

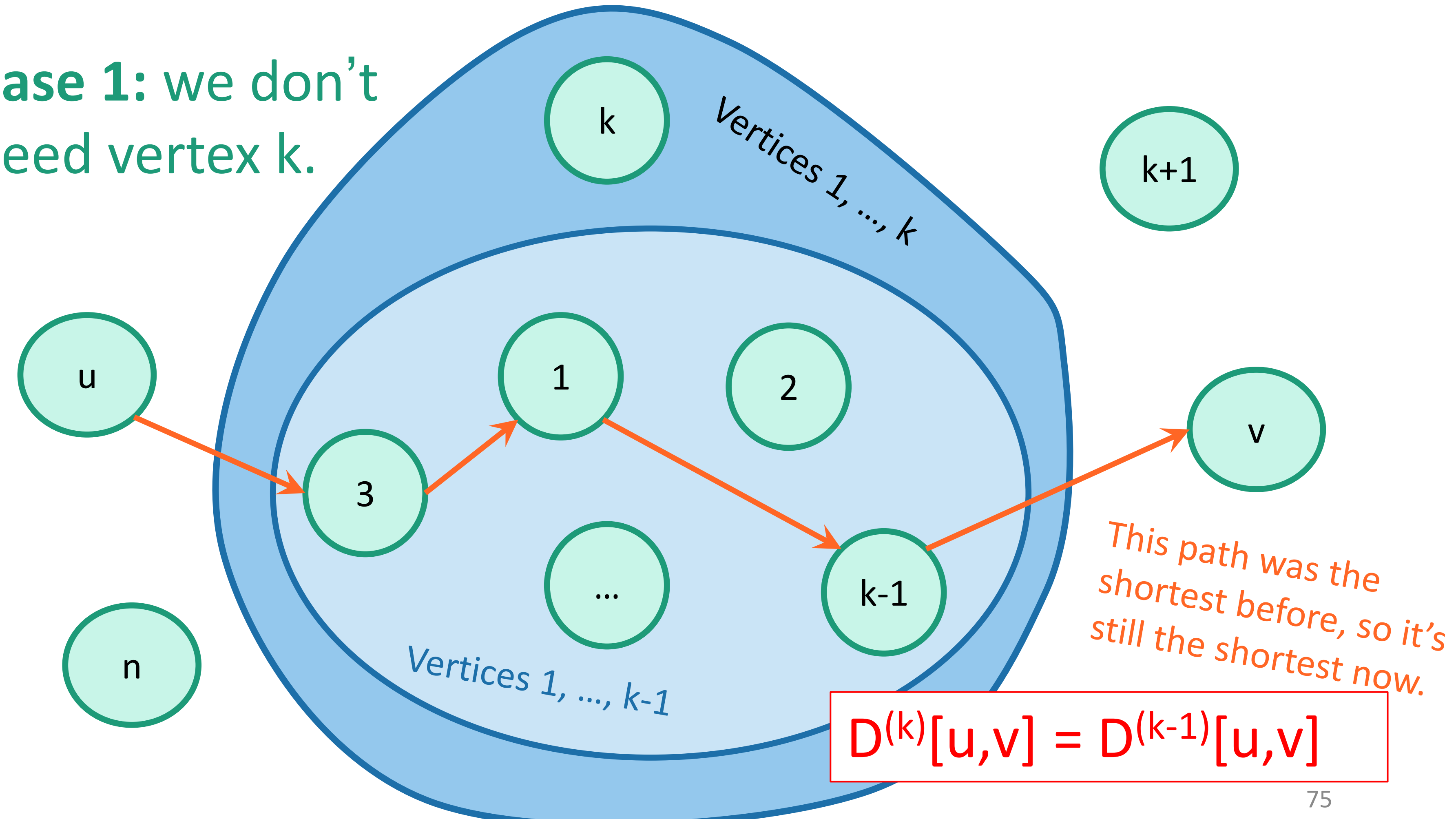
Case 1: we don't need vertex k.



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.

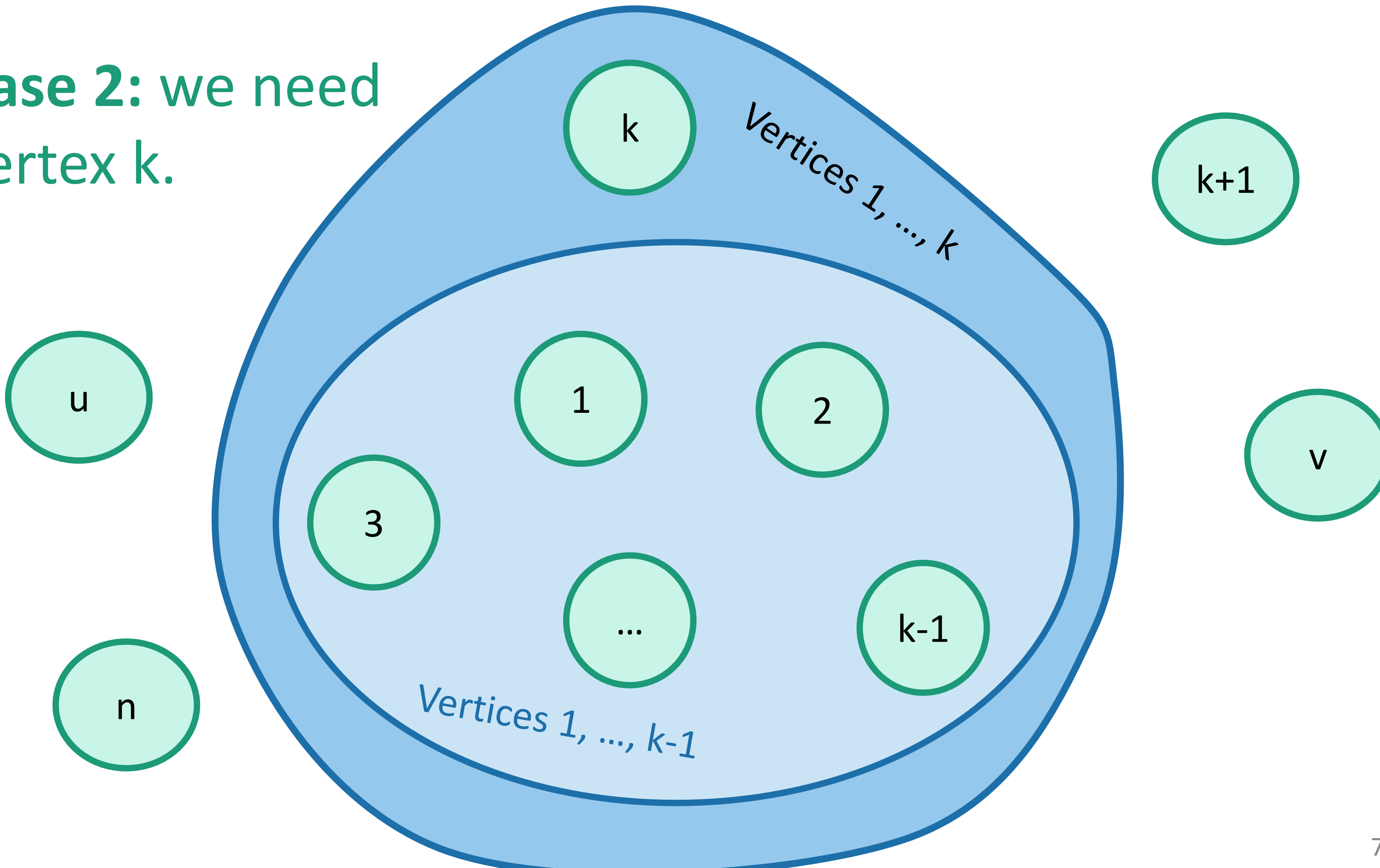


$D^{(k)}[u,v] = D^{(k-1)}[u,v]$

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}.

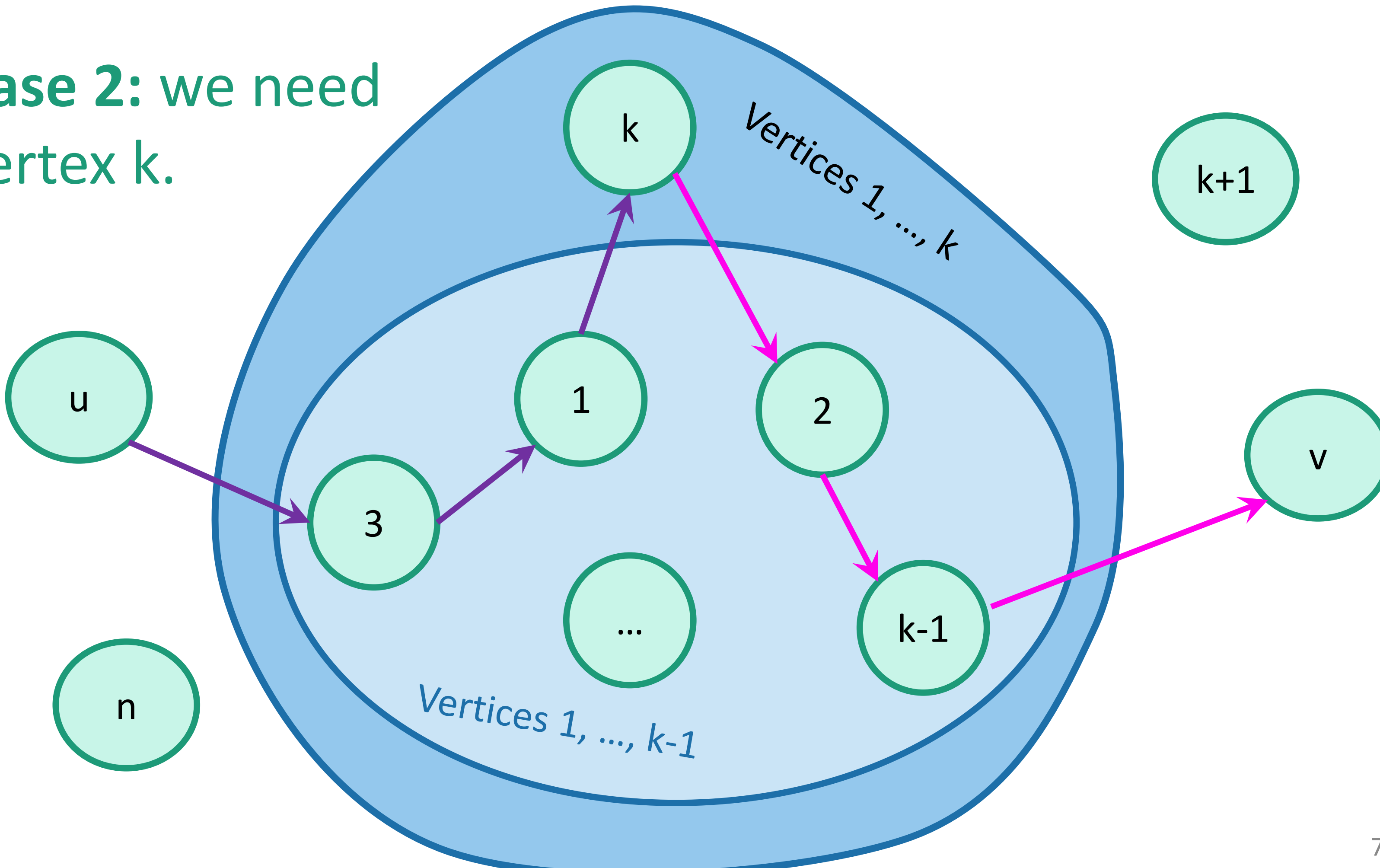
Case 2: we need vertex k.



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

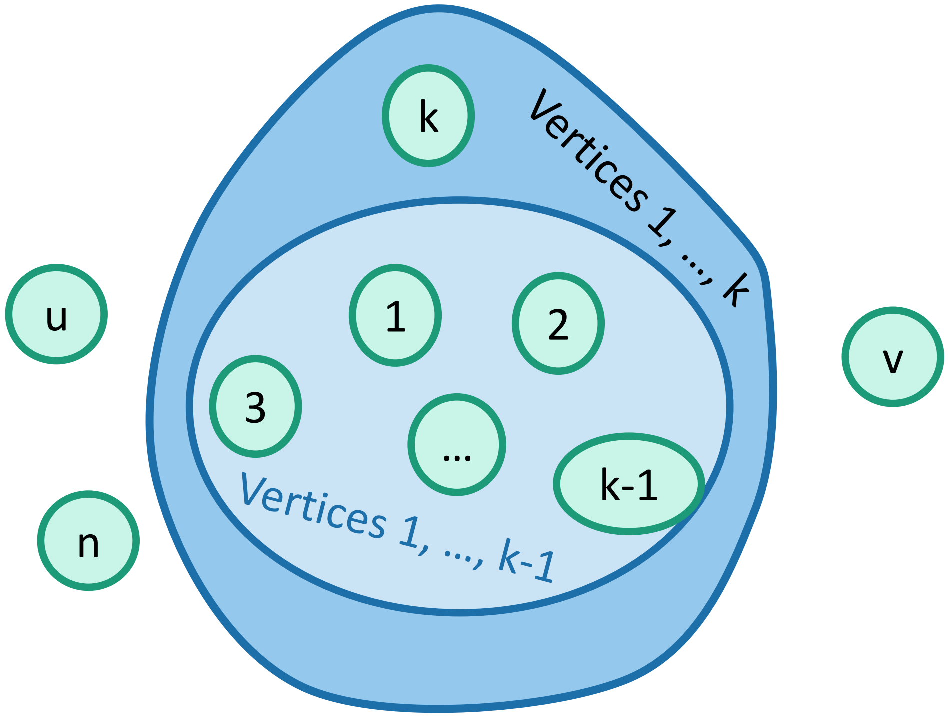
$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.

Case 2: we need vertex k.



Case 2 continued

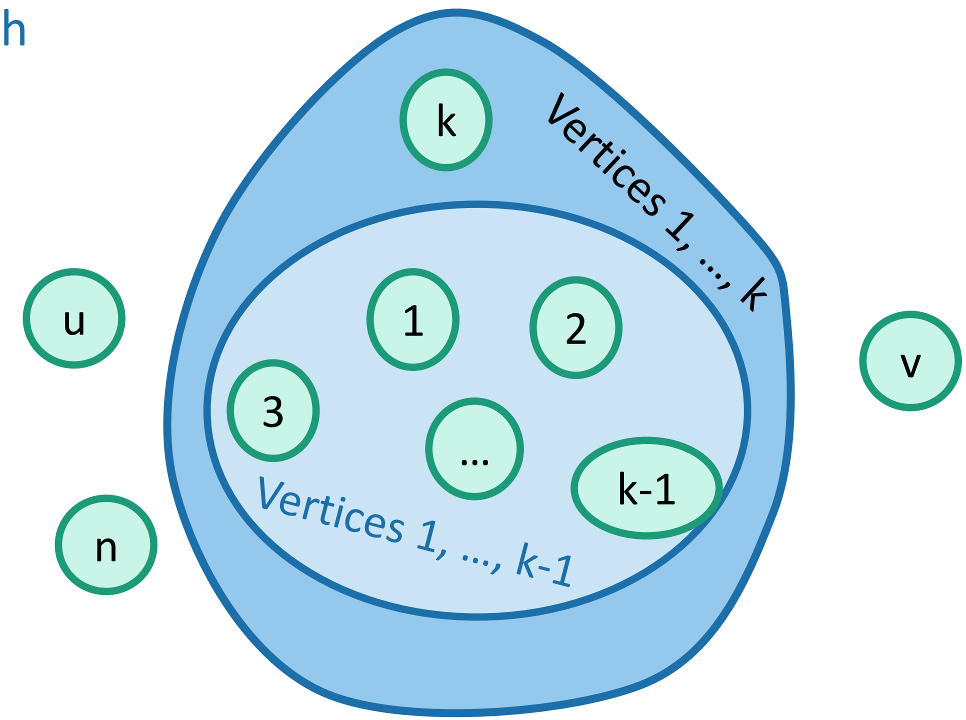
Case 2: we need vertex k.



Case 2 continued

- Suppose there are no negative cycles.
  - Then without loss of generality the shortest path from u to v through {1,…,k} is **simple**.

Case 2: we need vertex k.

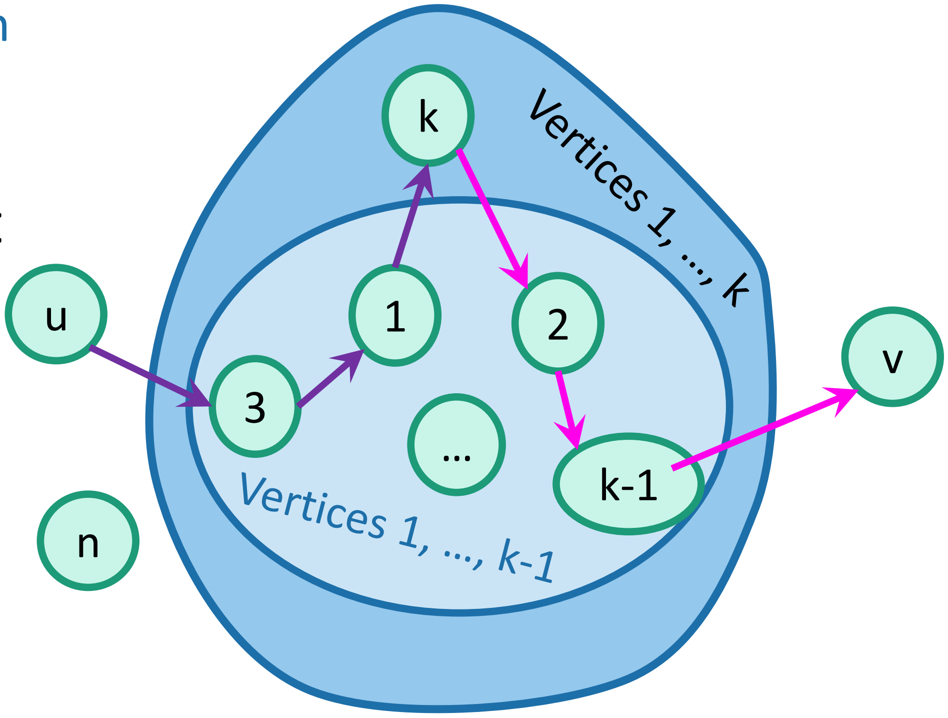


Case 2 continued

- Suppose there are no negative cycles.
  - Then without loss of generality the shortest path from u to v through {1,…,k} is **simple**.

- If **that path** passes through k, it must look like this:

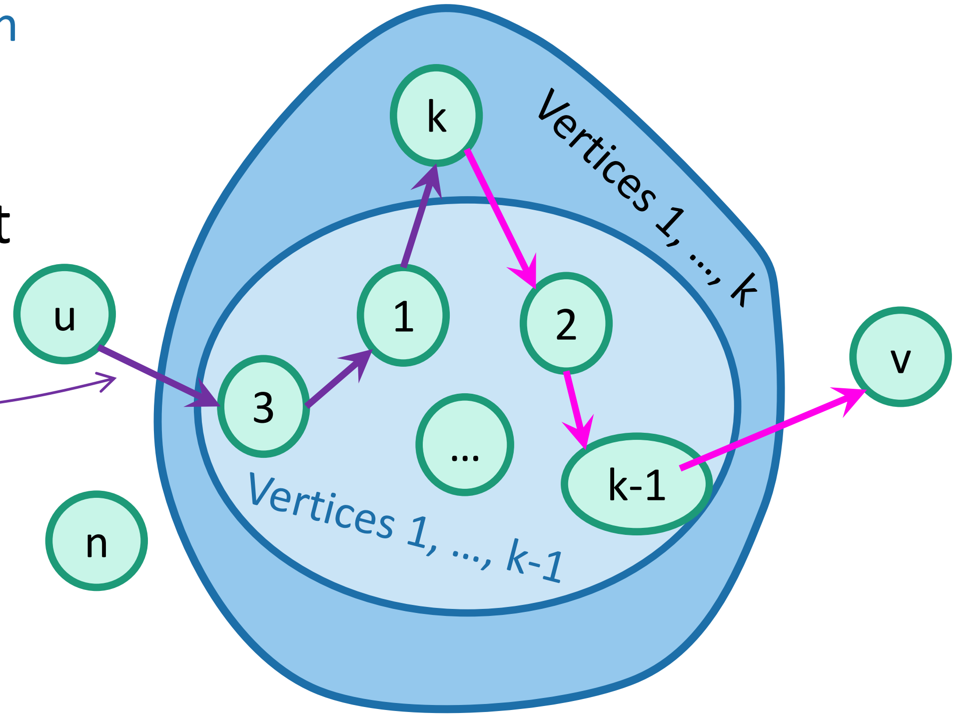
Case 2: we need vertex k.



Case 2 continued

- Suppose there are no negative cycles.
  - Then without loss of generality 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

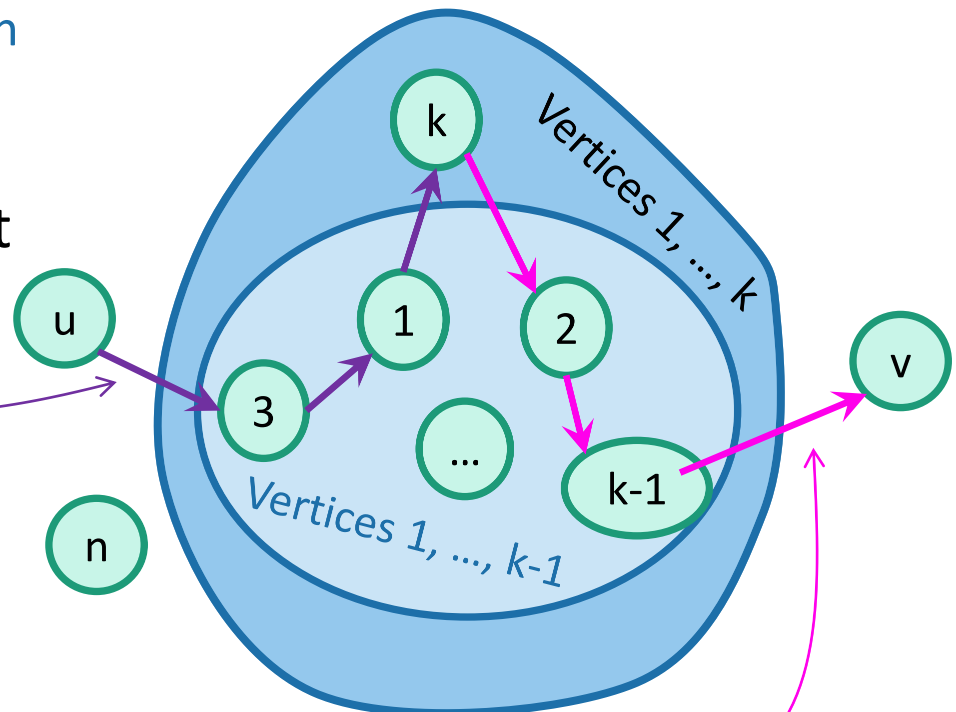
Case 2: we need vertex k.



Case 2 continued

- Suppose there are no negative cycles.
  - Then without loss of generality 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
- Similarly for **this path**.

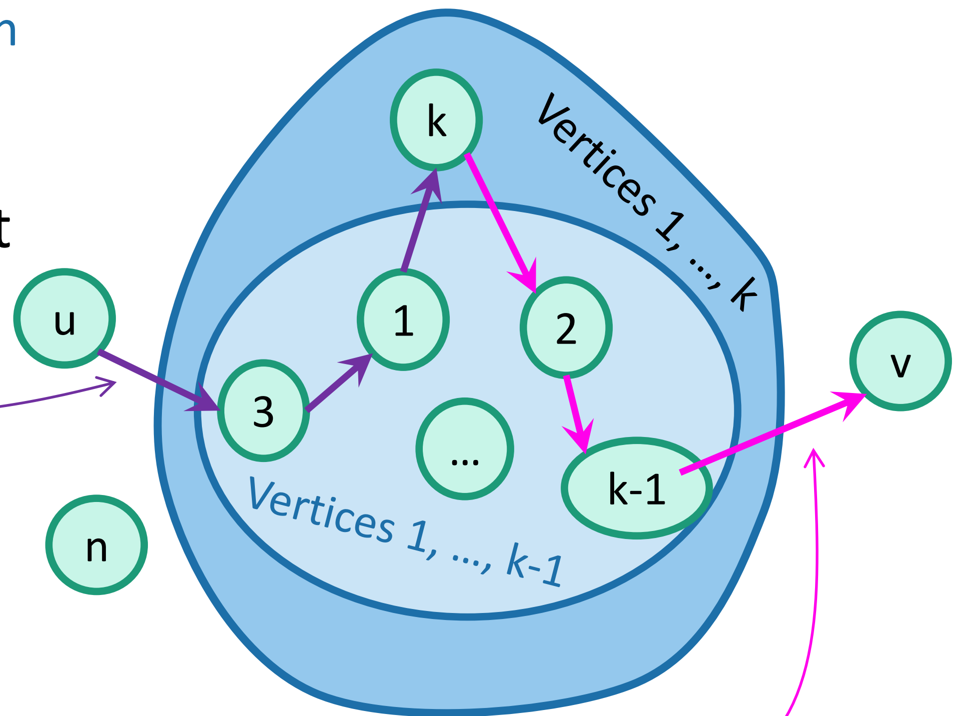
Case 2: we need vertex k.



Case 2 continued

- Suppose there are no negative cycles.
  - Then without loss of generality 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
- Similarly for **this path**.

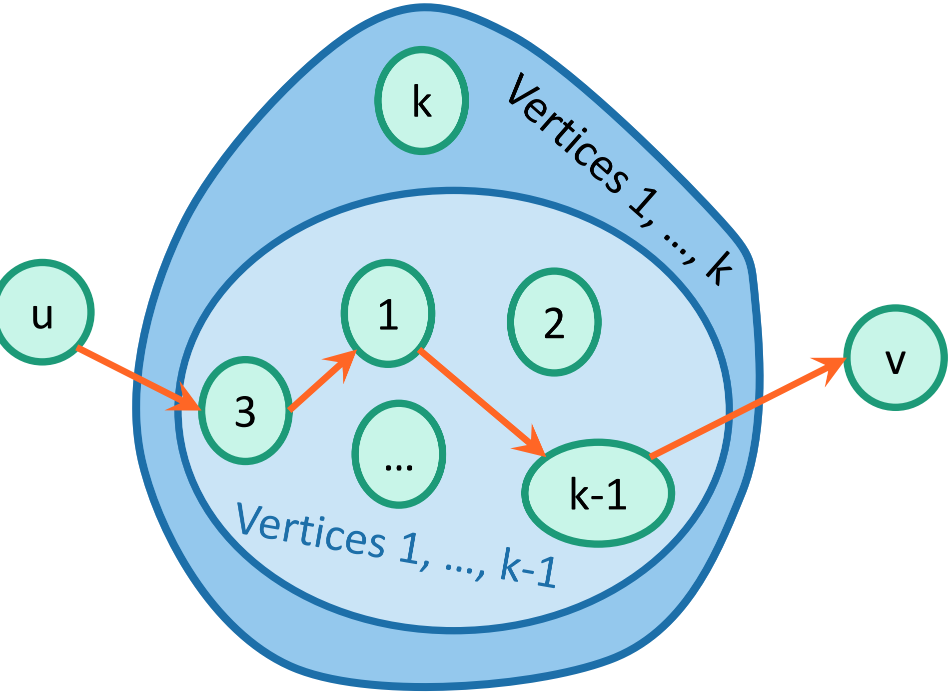
Case 2: we need vertex k.



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

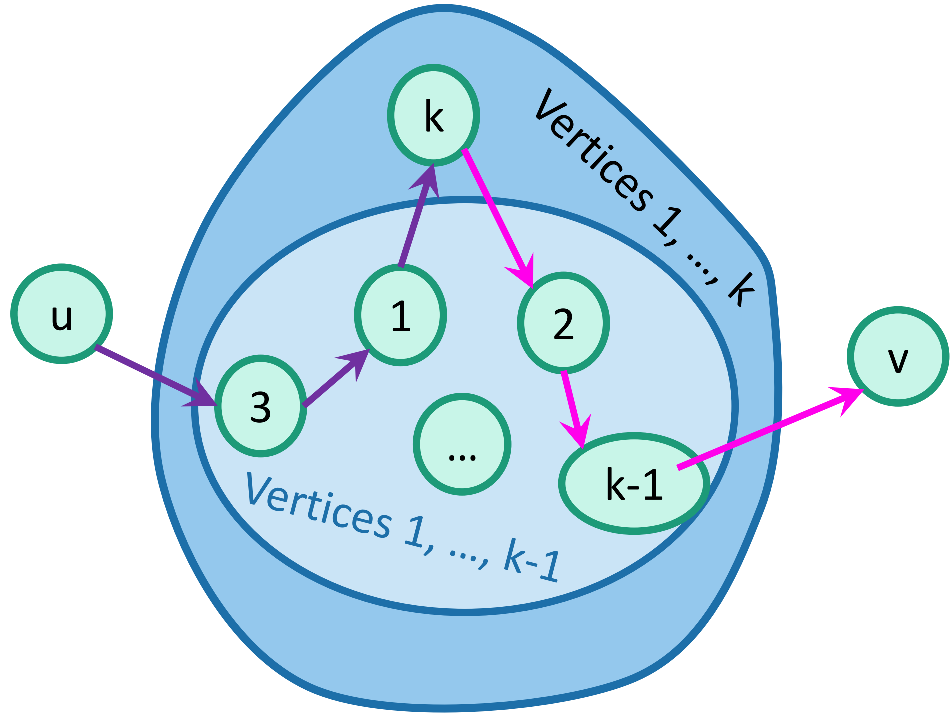
How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

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

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

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}

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}

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 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.

Floyd-Warshall algorithm

- Initialize n-by-n arrays $D^{(k)}$ for $k = 0,\ldots,n$
  - $D^{(k)}[u,u] = 0$ 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.

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.
- **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.

Floyd-Warshall algorithm

- Initialize n-by-n arrays $D^{(k)}$ for $k = 0,\ldots,n$
  - $D^{(k)}[u,u] = 0$ 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.
- **For** $k = 1, \ldots, n$:
  - **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.

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:

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

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:

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

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

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:

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

- 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.

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:

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

- 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 $D^{(k)}$.

What if there *are* negative cycles?

What if there *are* negative cycles?

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

What if there *are* negative cycles?

- Just like Bellman-Ford, Floyd-Warshall can detect negative cycles:
  - Negative cycle ⇔ ∃ v s.t. there is a path from v to v that goes through all n vertices that has cost < 0.
  - Negative cycle ⇔ ∃ 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 $O(n^3)$.

Recap

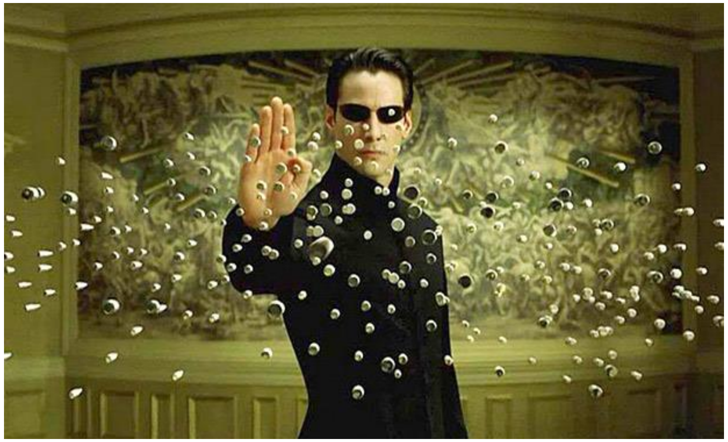
- ***Two shortest-path algorithms:***
  - Bellman-Ford for single-source shortest path
  - Floyd-Warshall for all-pairs shortest path

- ***Dynamic programming!***
  - This is a fancy name for:
    - 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.



Acknowledgement

- Stanford University