Lec14: Dynamic Programming II

Algorithm I
COMP319-003
Spring 2023

Instructor: Jiyeon Lee

School of Computer Science and Engineering

Kyungpook National University **(KNU)**

Last time: Bellman-Ford Algorithm

- (–) Slower than Dijkstra's algorithm
- (+) Can handle negative edge weights.
  - Can be useful if you want to say that some edges are actively good to take, rather than costly.
  - Can be useful as a building block in other algorithms.
- (+) Allows for some flexibility if the weights change.

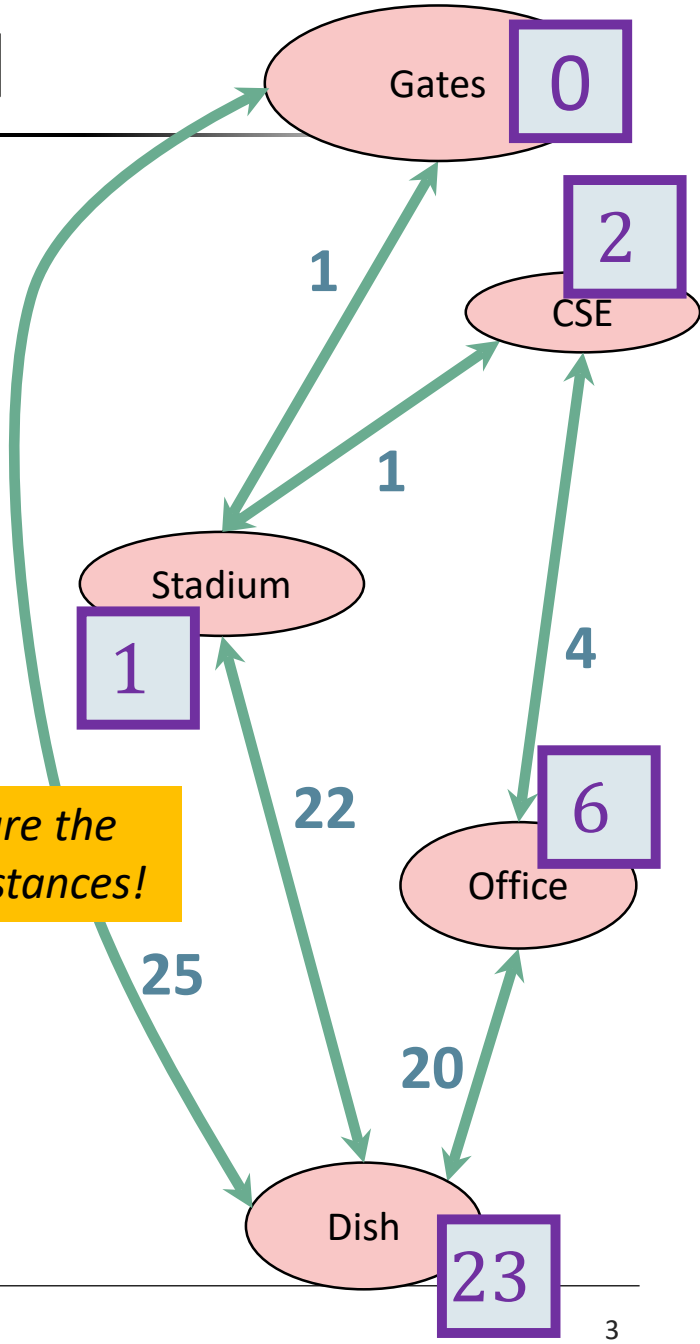
Bellman-Ford

- **How far is a node from Gates?**

	Gates	Stadium	CSE	Office	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!

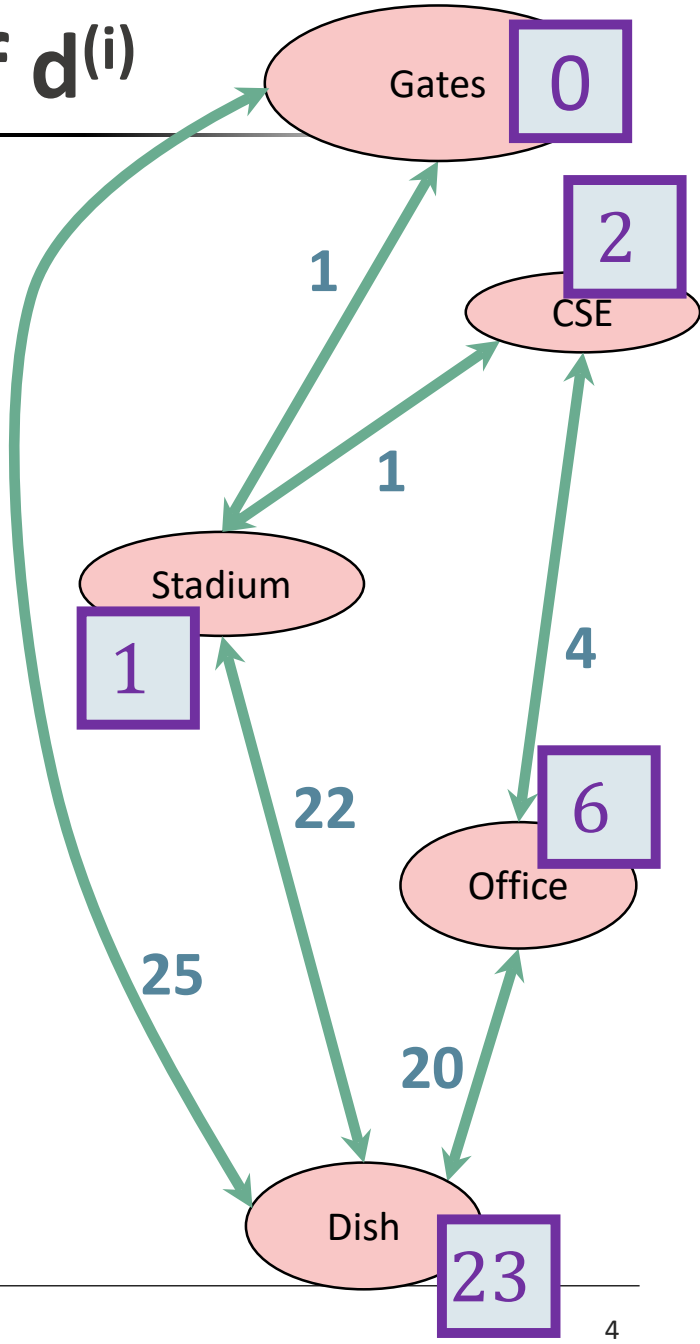
For i=0,…,n-2:
  For v in V:
    $d^{(i+1)}[v] \leftarrow \min(d^{(i)}[v] , d^{(i)}[u] + w(u,v))$
    where we are also taking the min over all u in v.inNeighbors



Interpretation of $d^{(i)}$

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

	Gates	Stadium	CSE	Office	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



Outline

1. Dynamic programming
   - Warm-up example: Fibonacci numbers
2. Floyd-Warshall Algorithm

- *Reading: CLRS 15.3, 25.2*

And B-F is also an example of…

Dynamic Programming!

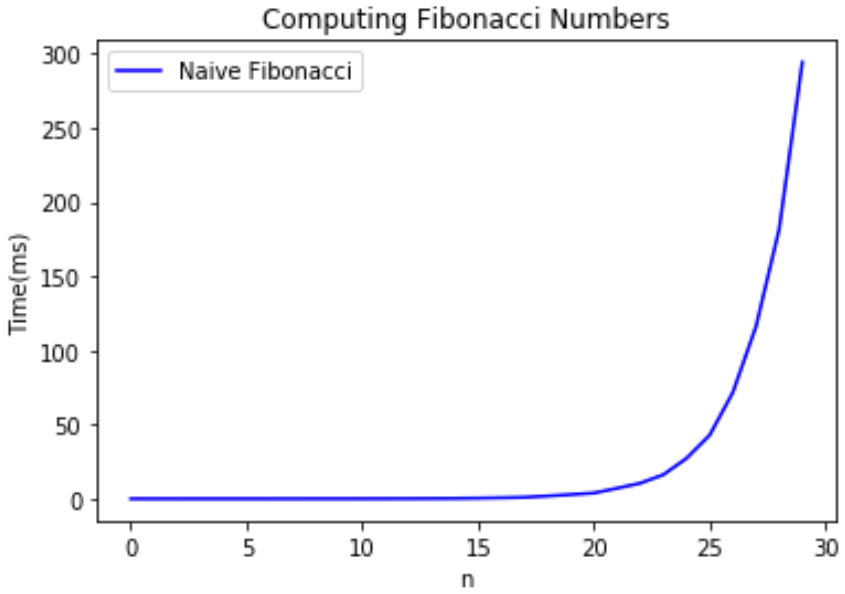
Let's start with a simple example

- How 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)
```

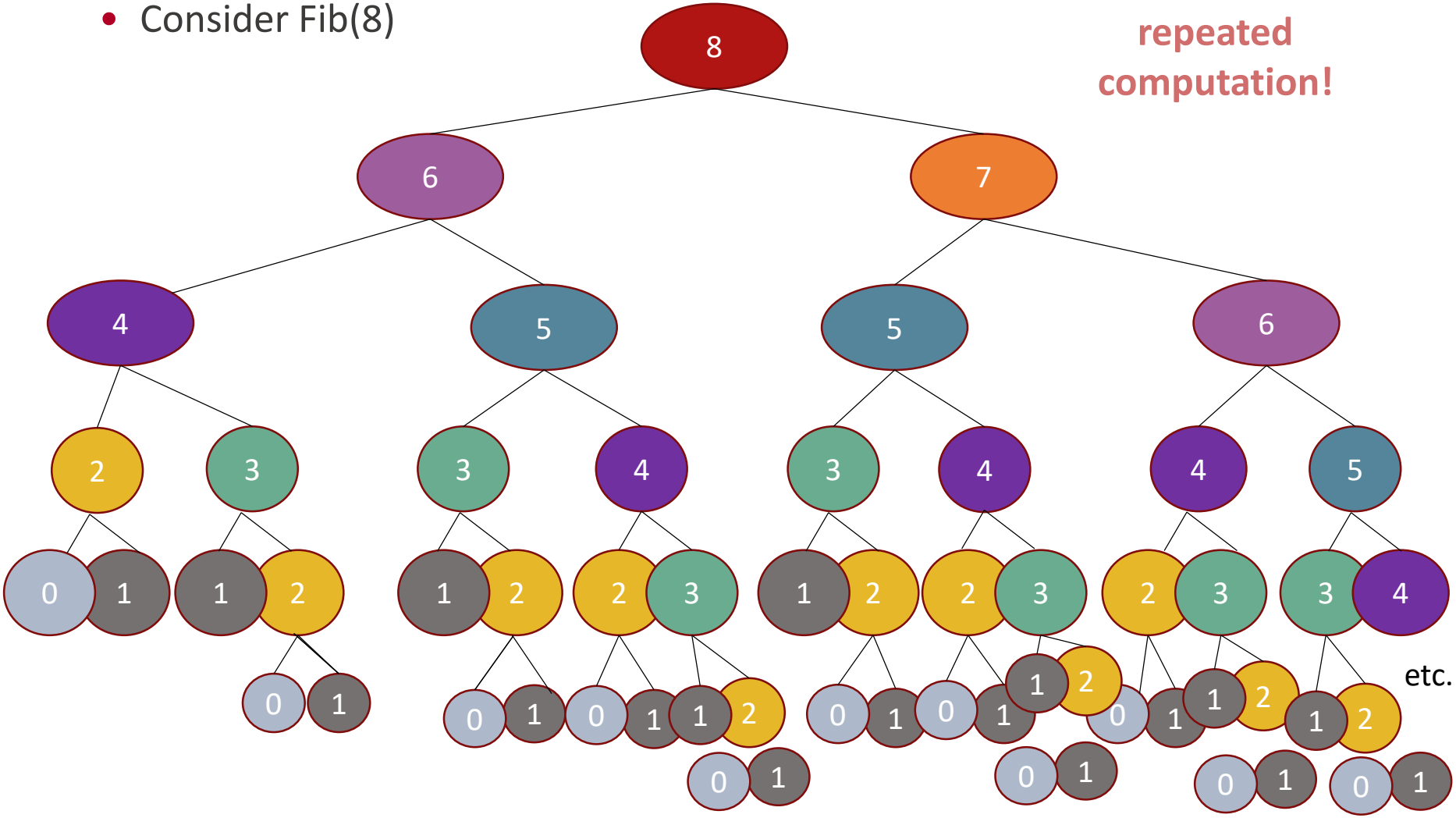
- Running time?
- T(n) ≥ T(n-1) + T(n-2) for n ≥2
- This is **EXPONENTIALLY QUICKLY!**
  - $T(n) \geq 2T(n-2)$ implies $T(n) \geq \Omega(2^{n/2})$.



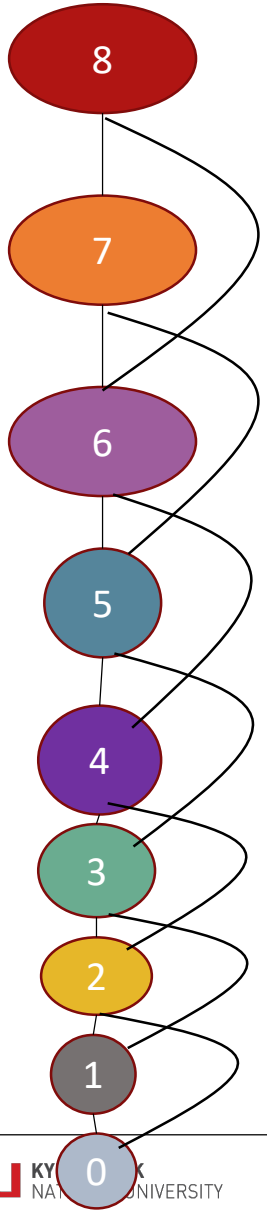
What's going on?

- Consider Fib(8)

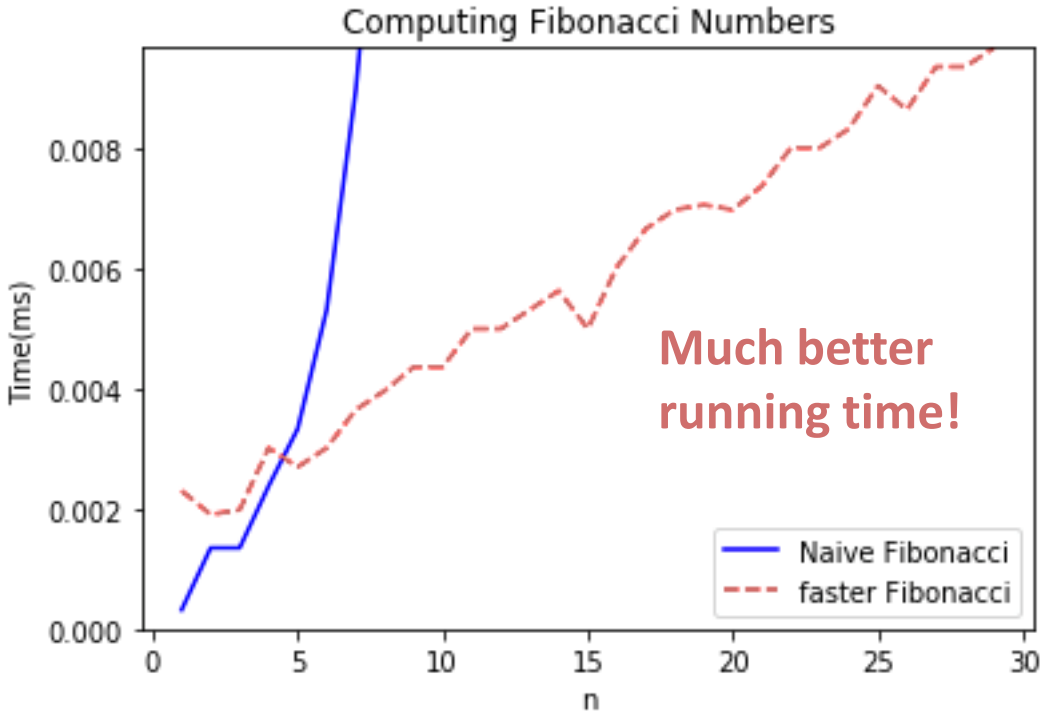
That's a lot of repeated computation!



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]
```



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.**
  - If the answer to a large problem includes the answer to a smaller problem, it is said to have an optimal structure.
  - E.g., ***shortest*** path

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

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

1. **Optimal substructure.**
   - Optimal solutions to sub-problems can be used to find the optimal solution of the original problem.
2. **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.

Aside: Comparison of strategies

- **Divide-and-conquer**
  - Break the problem into sub-problems and independently solve each problem.
  - The solution of the original problem is obtained by combining the solutions of each part of the problem.
- **Dynamic Programming**
  - Break the problem into overlapping partial problems and solve each problem.
  - Seeking the final solution by finding answers to bigger problems with partial problems.

- **Greedy** (which will cover later.)
  - Always making the choice that looks best at the moment.

DP algorithms

- Two ways to think about and/or implement
  - 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

- **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

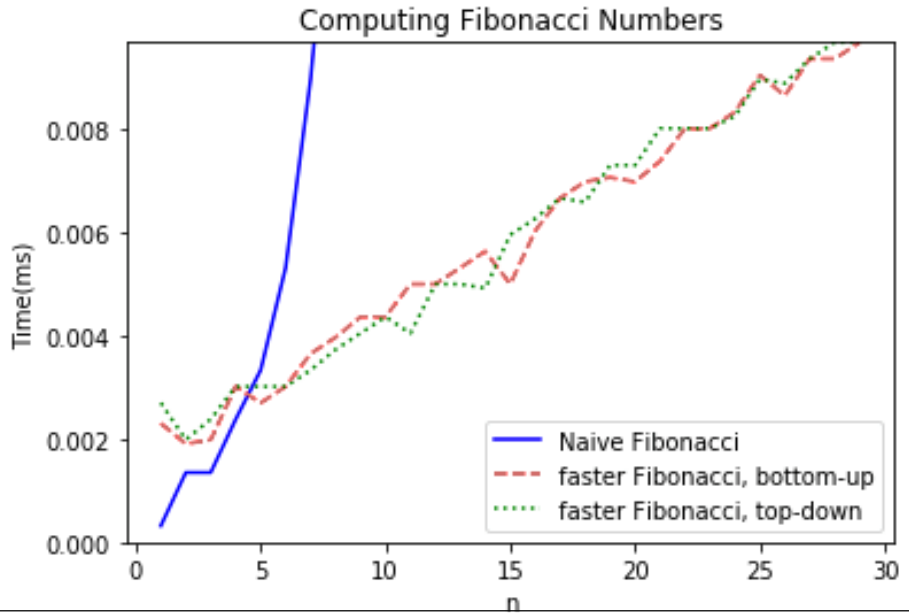
- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
    - etc..

- 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"

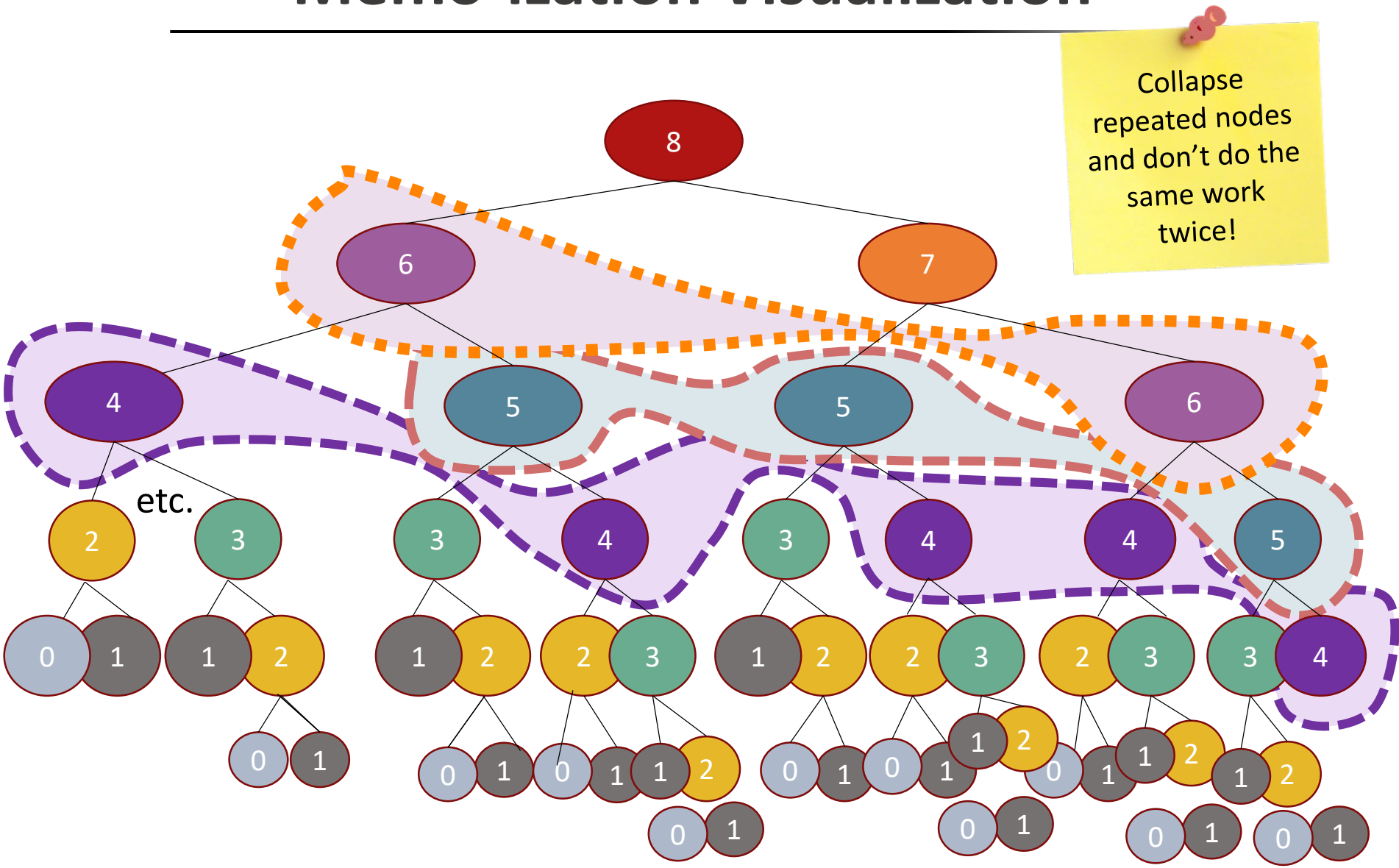
Example of top-down Fibonacci

```
• 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

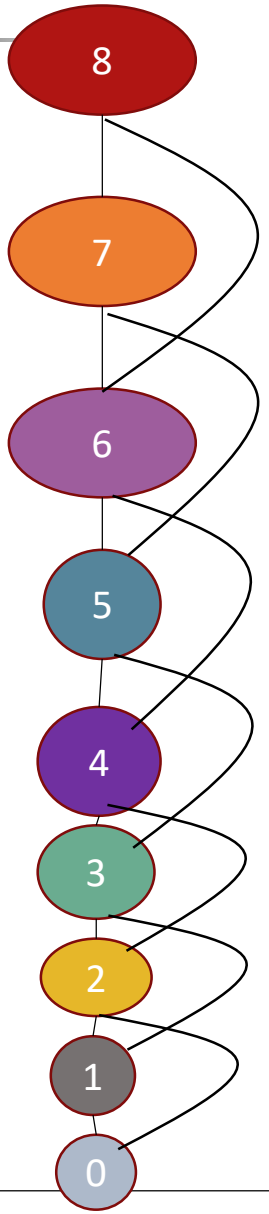


Memo-ization Visualization

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.

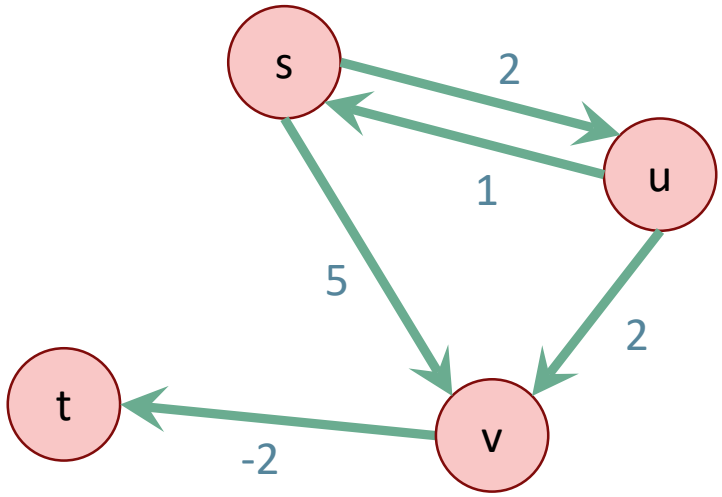
Floyd-Warshall Algorithm

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

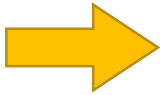


Floyd-Warshall Algorithm

- 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.
- **Naive 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?**

Recipe for applying DP



Step 1: Identify optimal substructure.

Step 2: Find a recursive formulation for the value of the optimal solution.

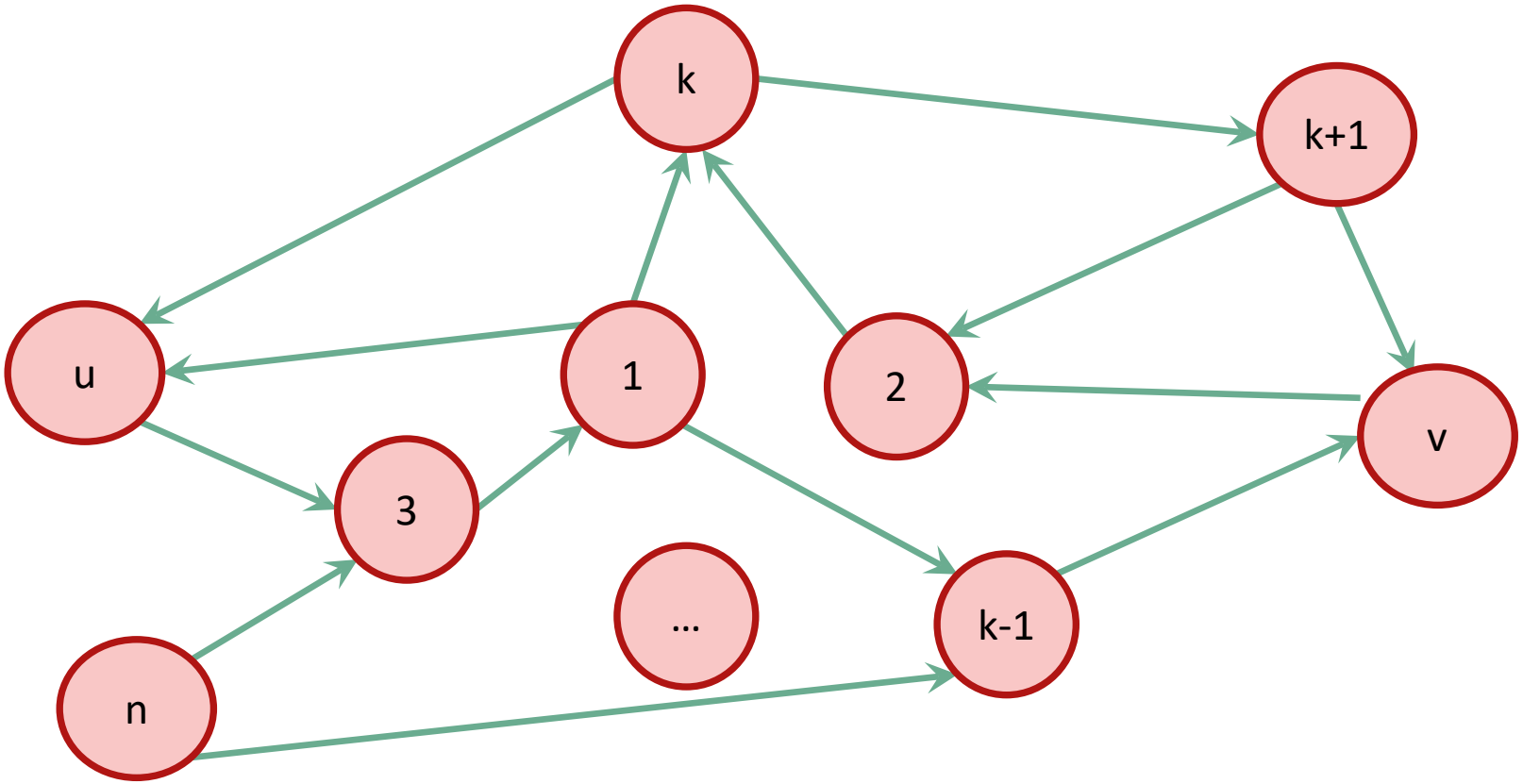
Step 3: Use dynamic programming to find the value of the optimal solution.

Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.

Step 5: If needed, code this up.

Optimal substructure

Label the vertices 1, 2, …, n

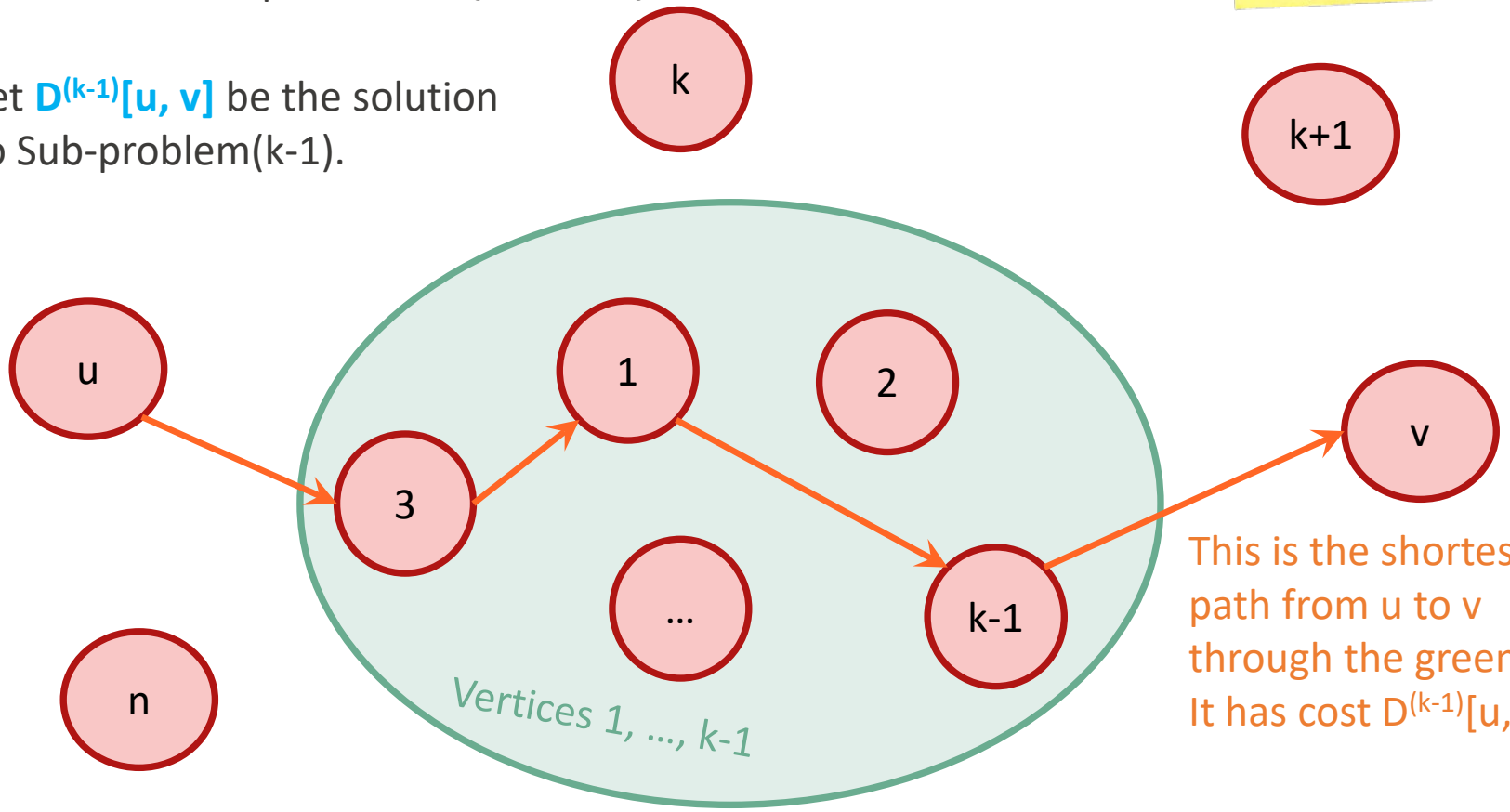


Optimal substructure

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

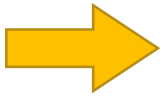
Sub-problem(k-1):
For all pairs, u, v, find the cost of the shortest path from u to v, so that all the internal vertices on that path are in {1, …, k-1}.

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 green set. It has cost $D^{(k-1)}[u, v]$.

Recipe for applying DP



Step 1: Identify optimal substructure.

Step 2: Find a recursive formulation for the value of the optimal solution.

Step 3: Use dynamic programming to find the value of the optimal solution.

Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.

Step 5: If needed, code this up.

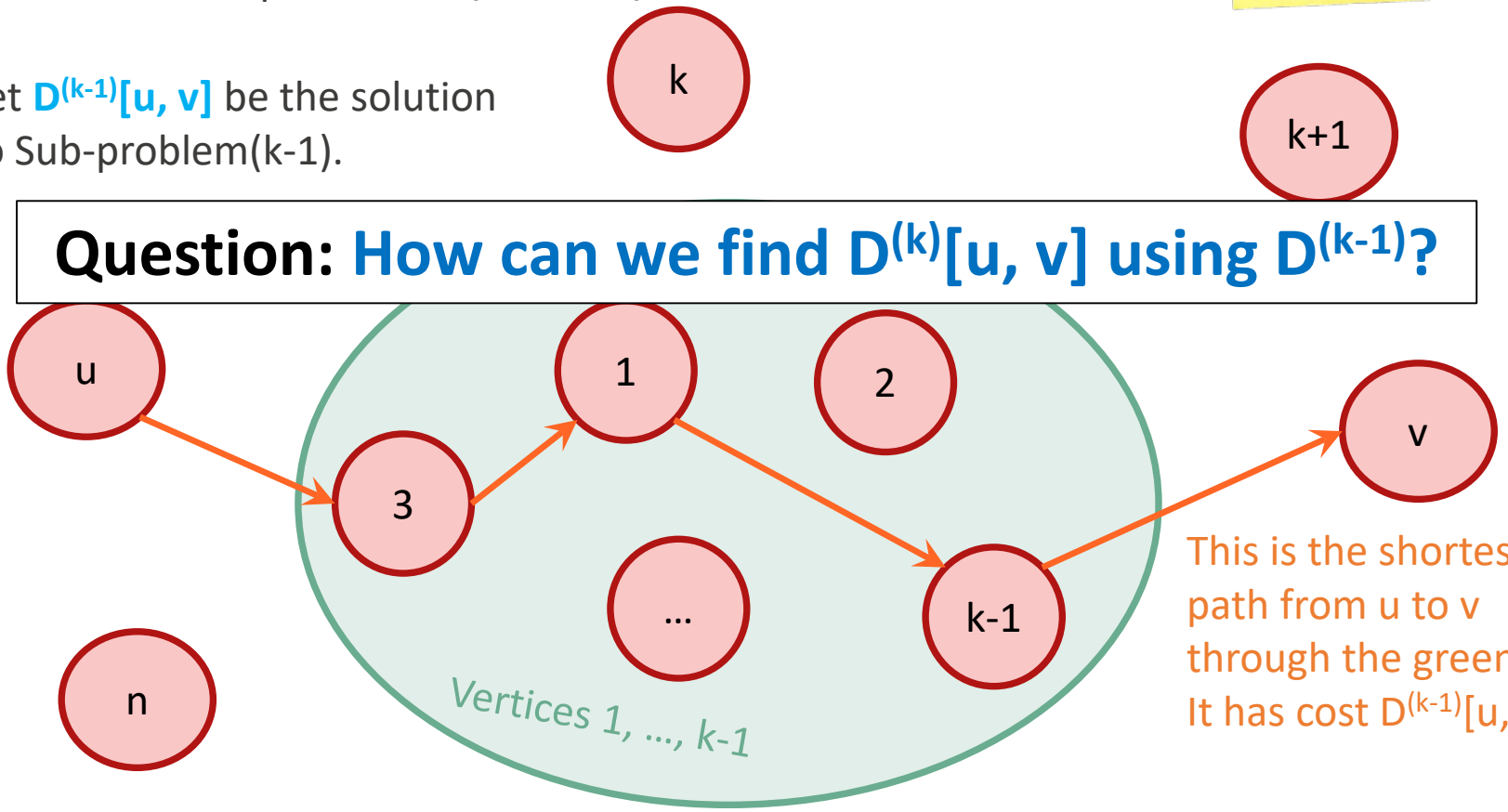
Optimal substructure

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.

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}.

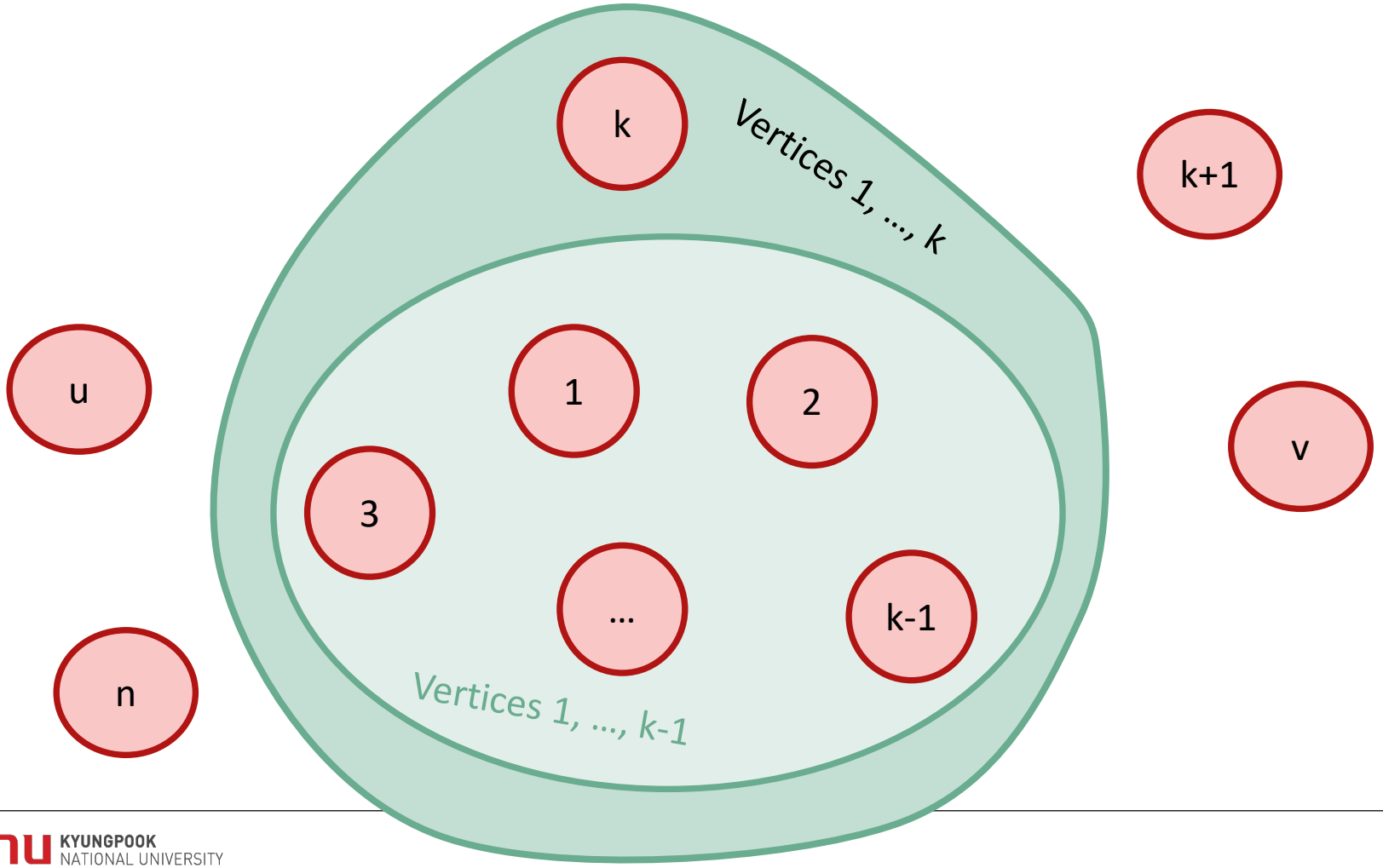
Let $D^{(k-1)}[u, v]$ be the solution to Sub-problem(k-1).

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



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}.

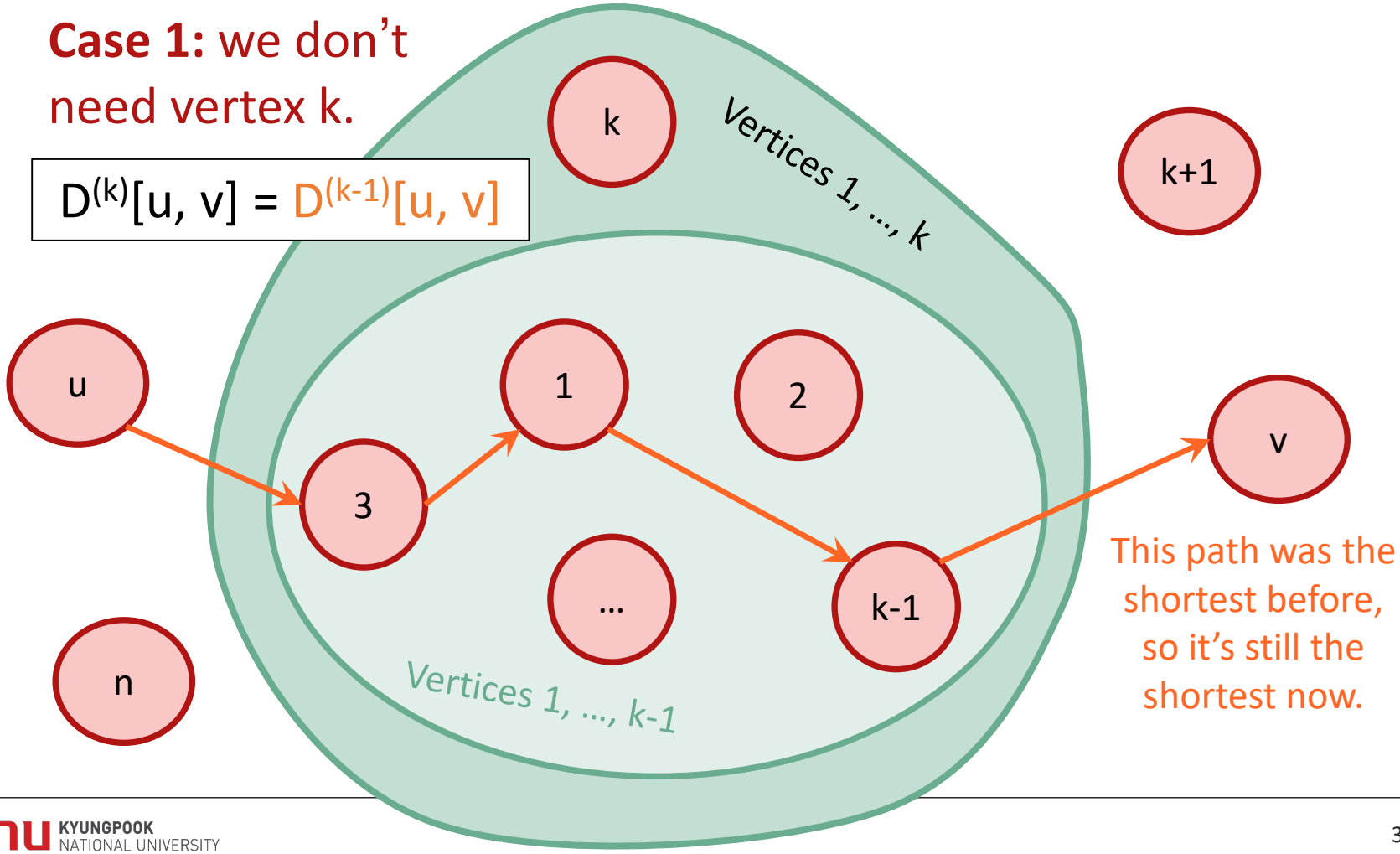


How can we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

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

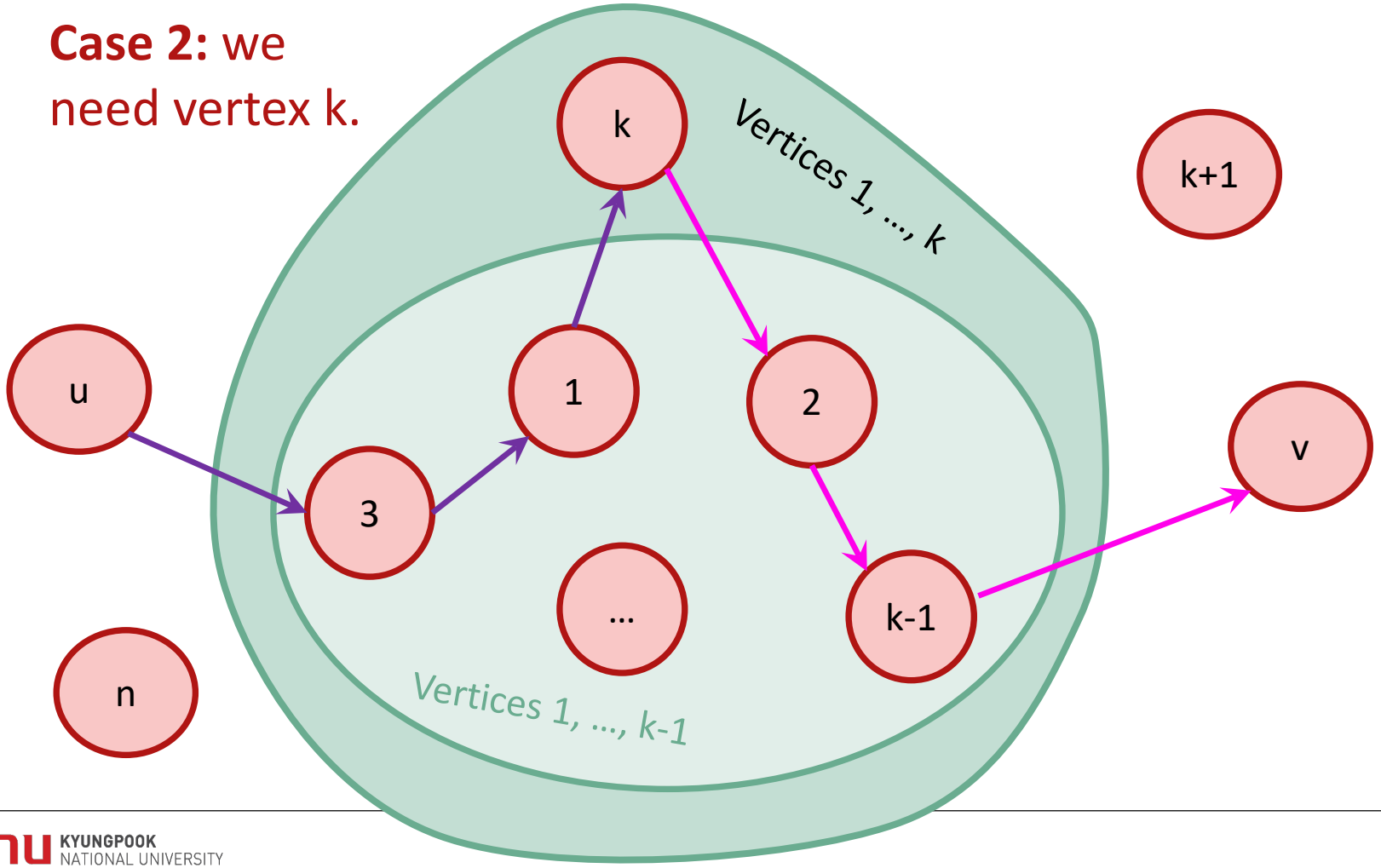
$$D^{(k)}[u, v] = D^{(k-1)}[u, v]$$



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}.

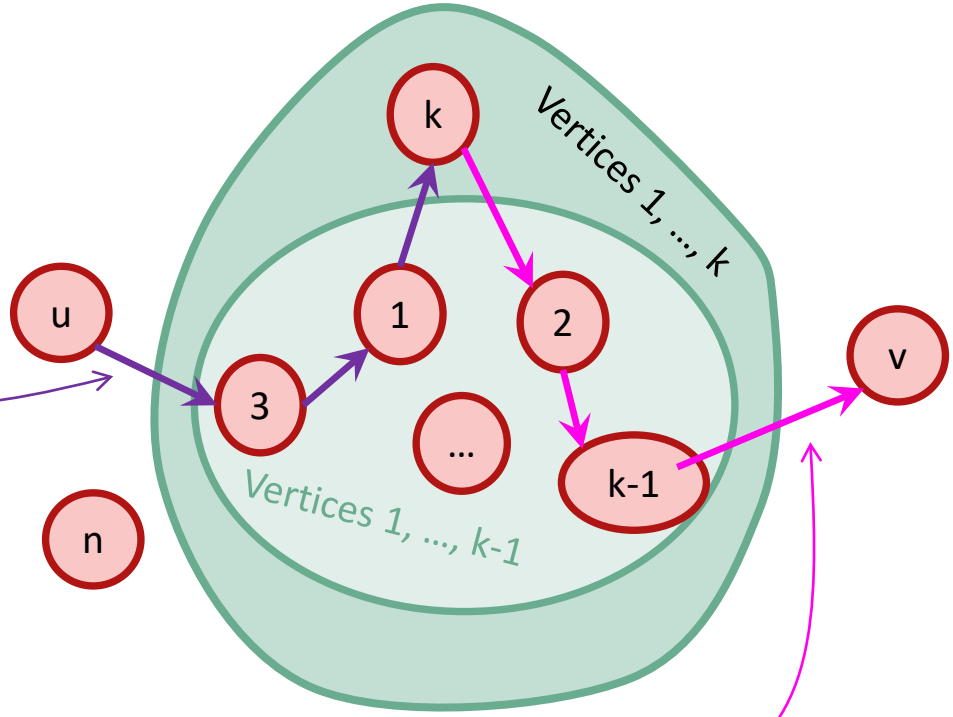
Case 2: we need vertex k.



Case 2 continued

Case 2: we need vertex k.

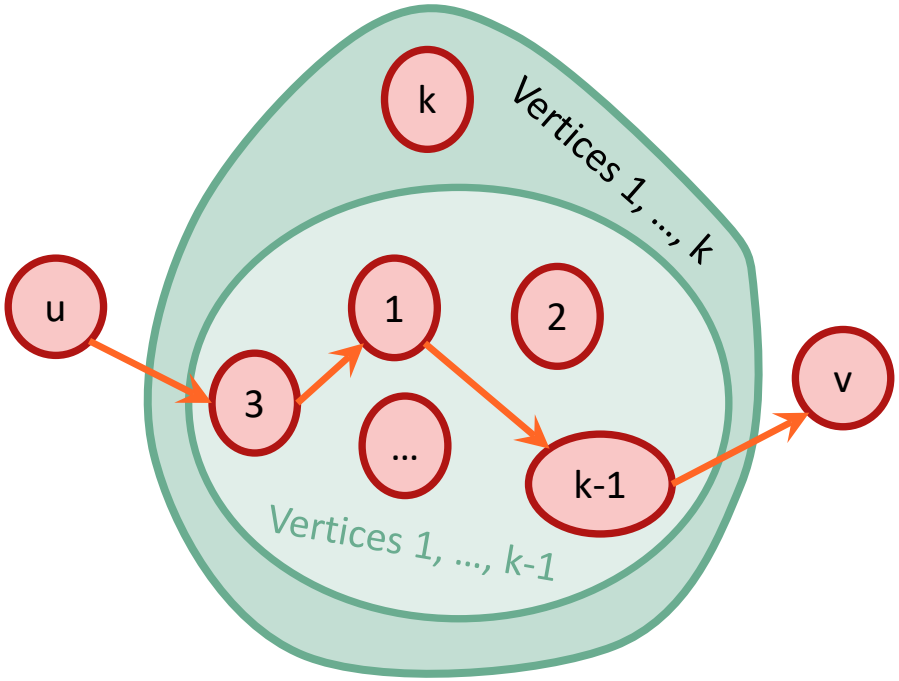
- Suppose there are **no negative cycles.**
  - Then without loss of generality, the shortest path from u to v through {1, …, k} is simple.
- If that path passes through k, it must look like this: ---------->
- **This path** is the shortest path from u to k through {1, …, k-1}.
  - Sub-paths of shortest paths are shortest paths.
- Similarly for **this path.**
- Thus,



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

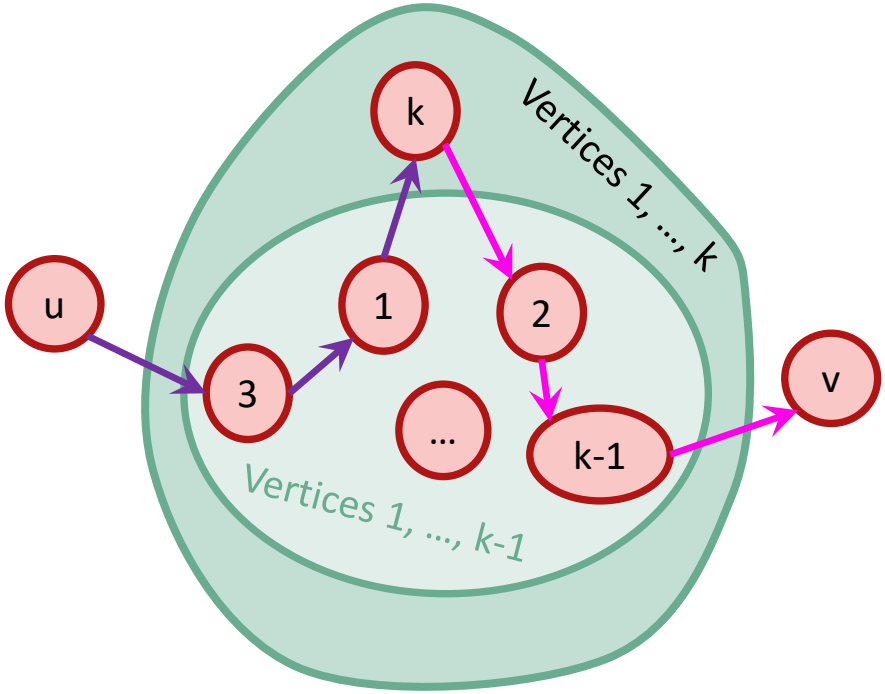
How can we find $D^{(k)}[u, v]$ using $D^{(k-1)}$?

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 from u to v through {1,…,k-1}

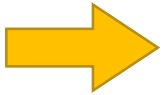
Case 2: Cost of shortest path from u to k and then from k to v through {1,…,k-1}

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

Recipe for applying DP

Step 1: Identify optimal substructure.

Step 2: Find a recursive formulation for the value of the optimal solution.



Step 3: Use dynamic programming to find the value of the optimal solution.

Step 4: If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.

Step 5: If needed, code this up.

Floyd-Warshall algorithm

- **Floyd-Warshall:**
  - Initialize n-by-n arrays
    - $D^{(0)}[u, u] = 0$ for all u, for all k
    - $D^{(0)}[u, v] = \infty$ 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.

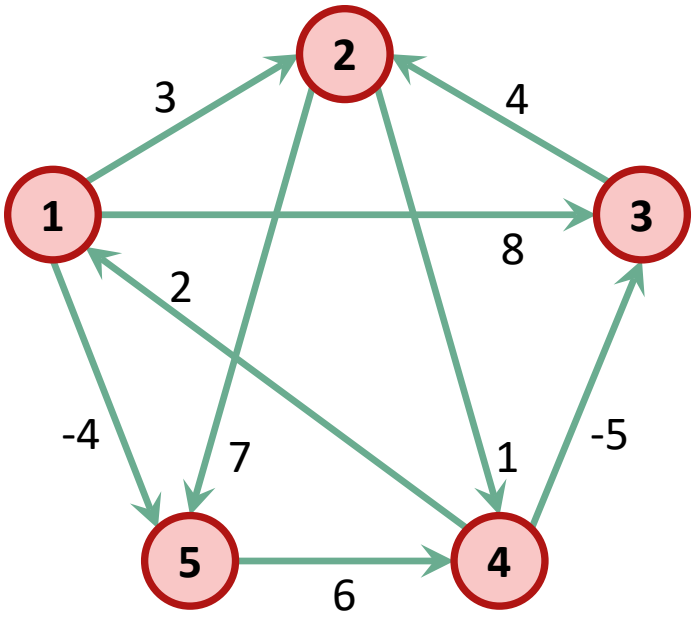
- This is a **bottom-up** *Dynamic programming* algorithm.

Example of F-W

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

$D^{(0)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

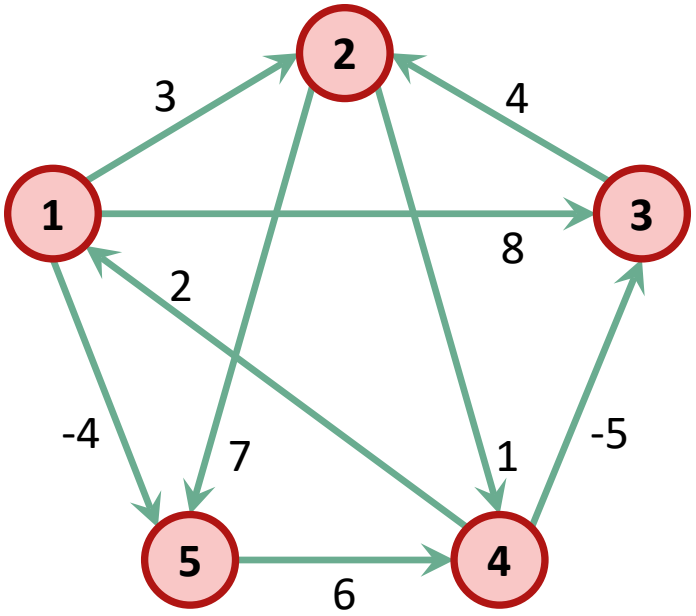


Example of F-W

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

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0



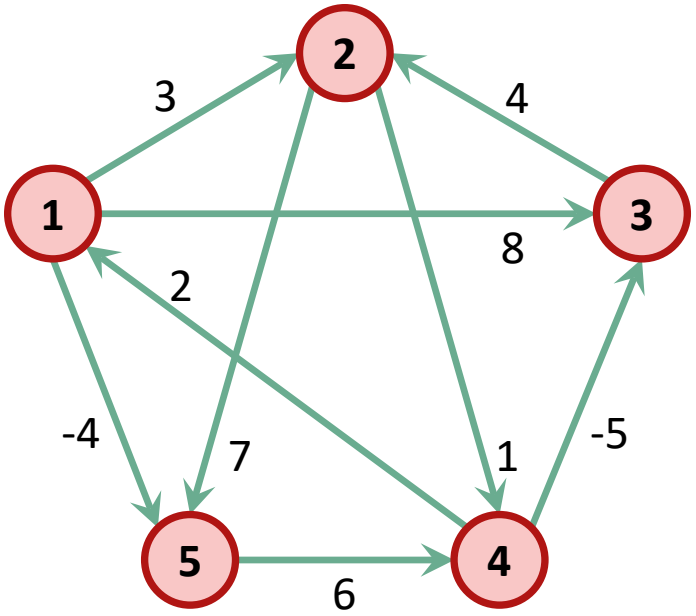
Example of F-W

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

Check: min { 2 -> 3 (∞), 2 -> 1 -> 3 (∞ + 8) } = ∞

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0



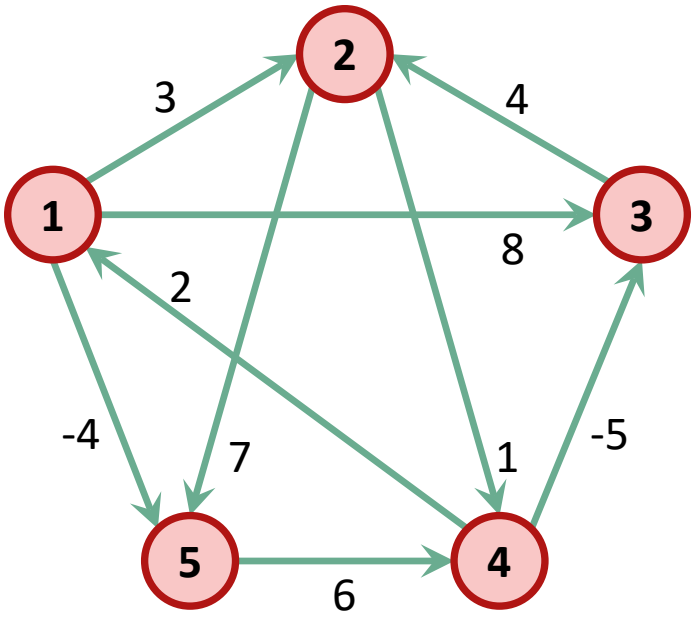
Example of F-W

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

Check: min { 2 -> 4 (1), 2 -> 1 -> 4 (∞ + ∞) } = 1

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0



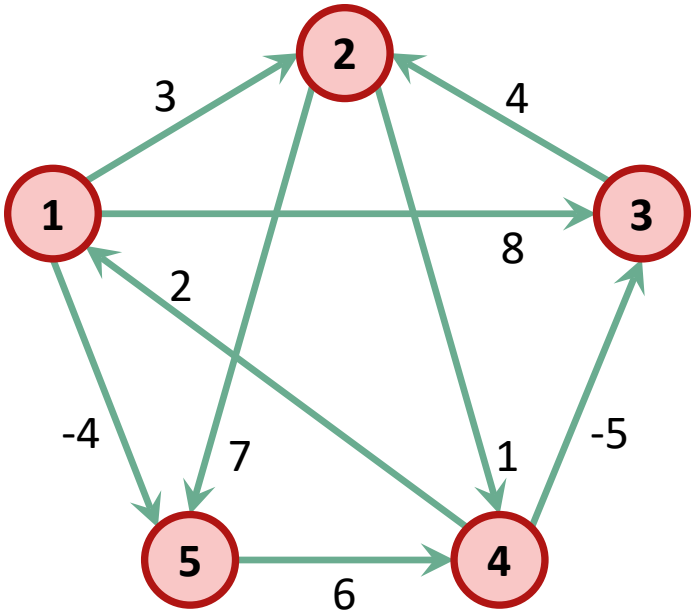
Example of F-W

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

Check: min { 2 -> 5 (7), 2 -> 1 -> 5 (∞ – 4) } = 7

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0



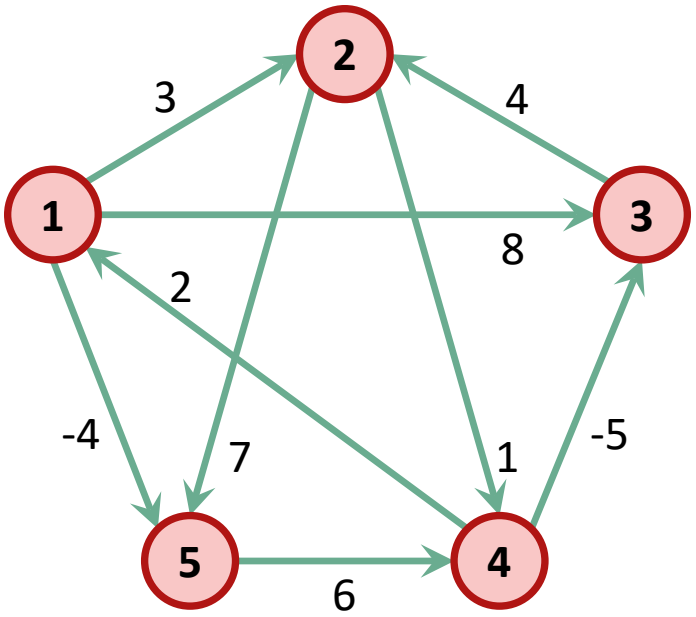
Example of F-W

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

Check: min { 3 -> 2 (4), 3 -> 1 -> 2 (∞ + 3) } = 4

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0



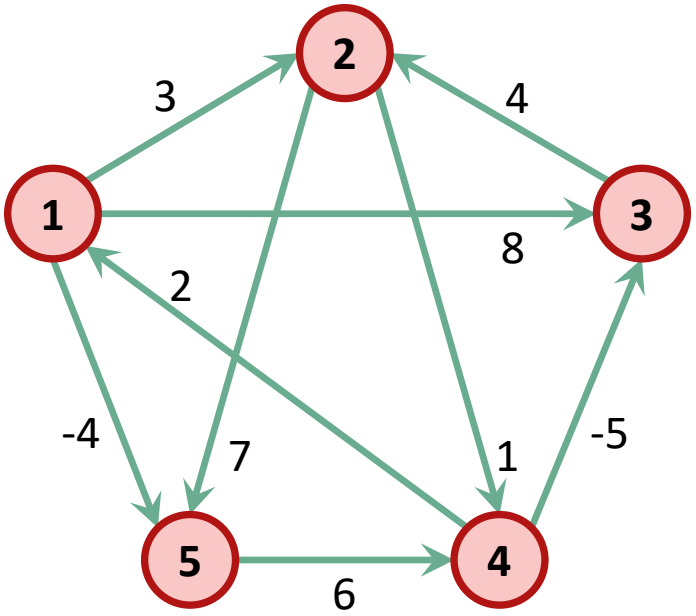
Example of F-W

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

Check: min { 3 -> 4 (∞), 3 -> 1 -> 4 (∞ + ∞) } = ∞

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0



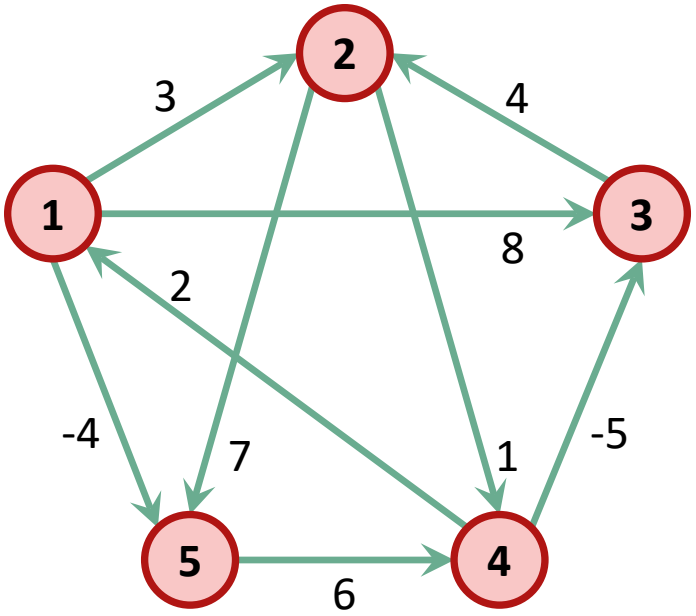
Example of F-W

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

Check: min { 3 -> 5 (∞), 3 -> 1 -> 5 (∞ – 4) } = ∞

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0



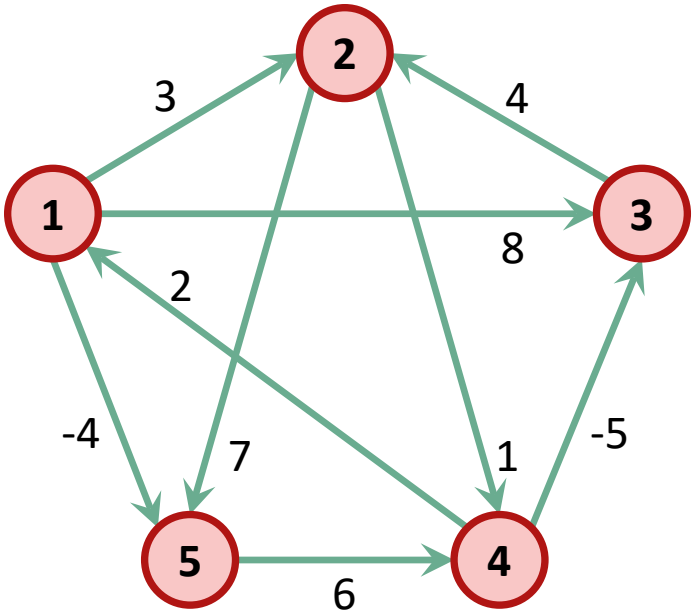
Example of F-W

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

Check: min { 4 -> 2 (∞), 4 -> 1 -> 2 (2 + 3) } = 5

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0



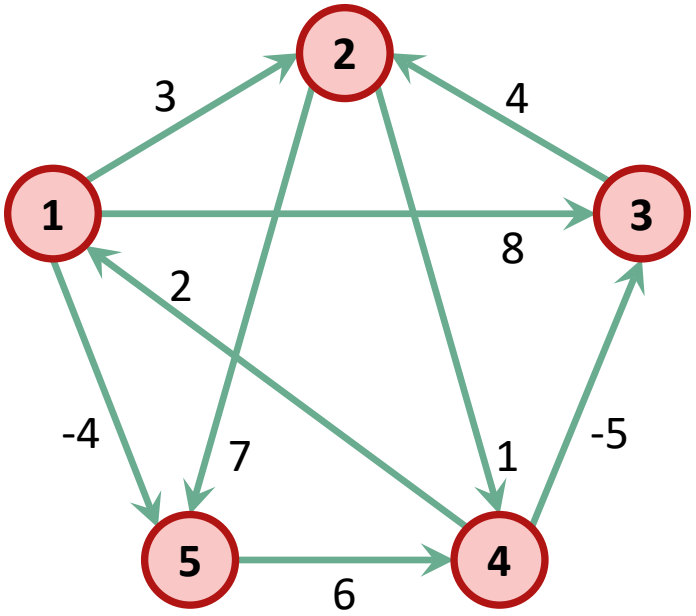
Example of F-W

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

Check: min { 4 -> 2 (∞), 4 -> 1 -> 2 (2 + 3) } = 5

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	∞
5	∞	∞	∞	6	0



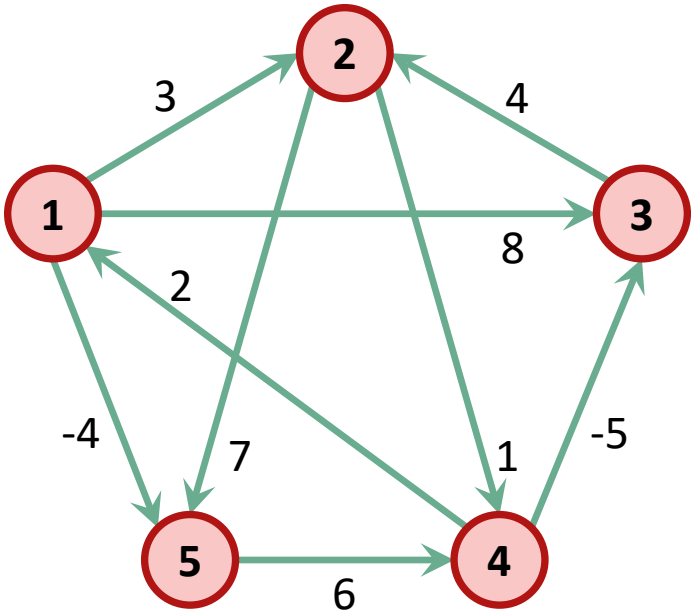
Example of F-W

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

Check: min { 4 -> 3 (-5), 4 -> 1 -> 3 (2 + 8) } = -5

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	∞
5	∞	∞	∞	6	0



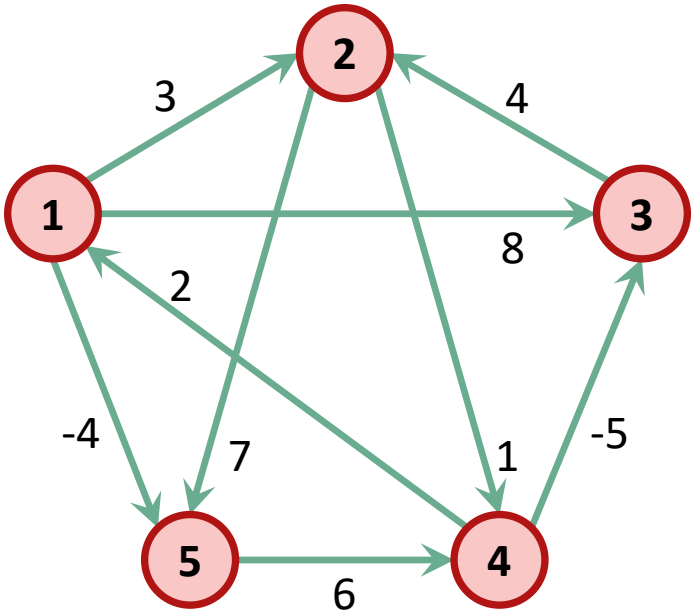
Example of F-W

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

Check: min { 4 -> 5 (∞), 4 -> 1 -> 5 (2 – 4) } = -2

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	∞
5	∞	∞	∞	6	0



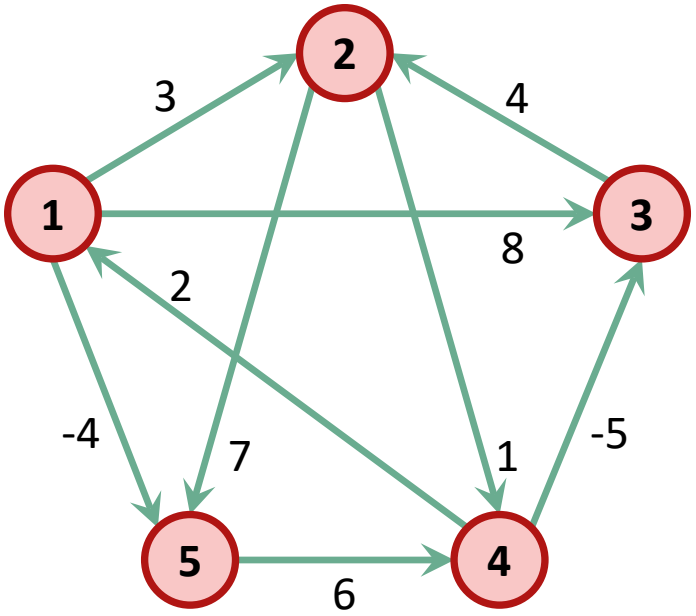
Example of F-W

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

Check: min { 4 -> 5 (∞), 4 -> 1 -> 5 (2 – 4) } = -2

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

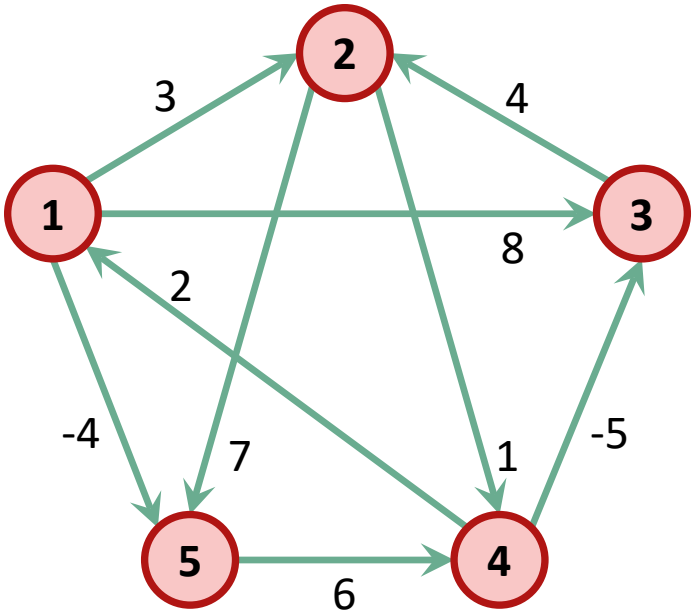


Example of F-W

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

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

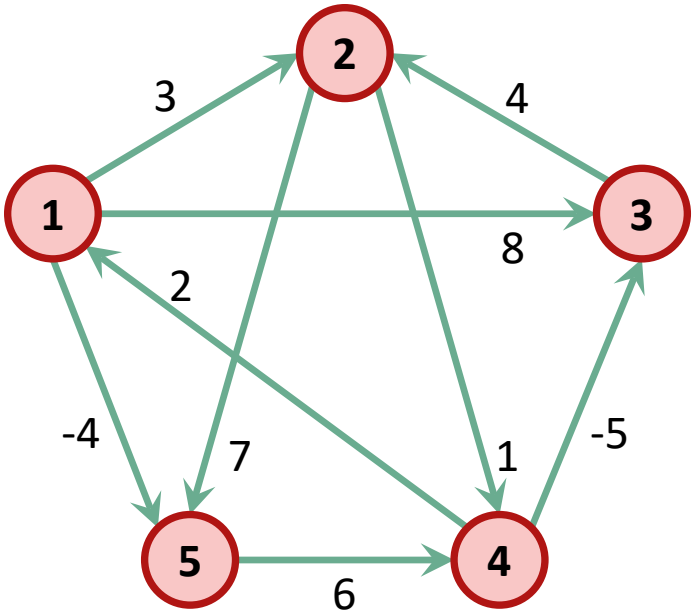


Example of F-W

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

$D^{(2)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

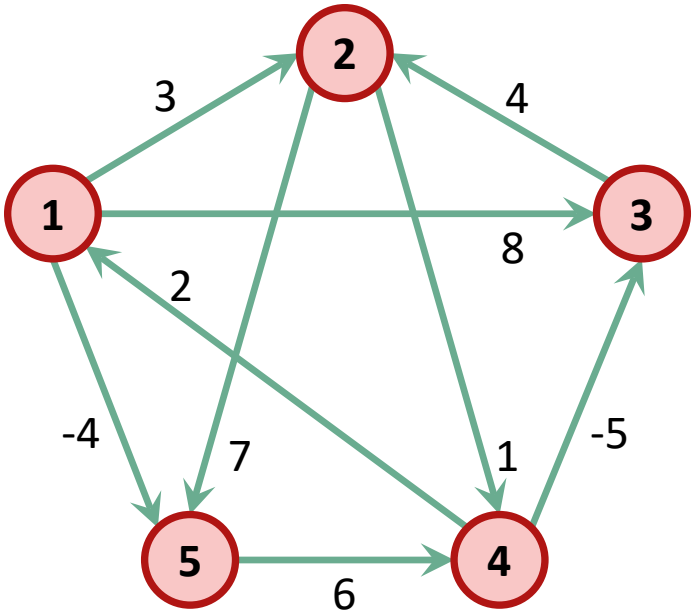


Example of F-W

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

$D^{(2)}$

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

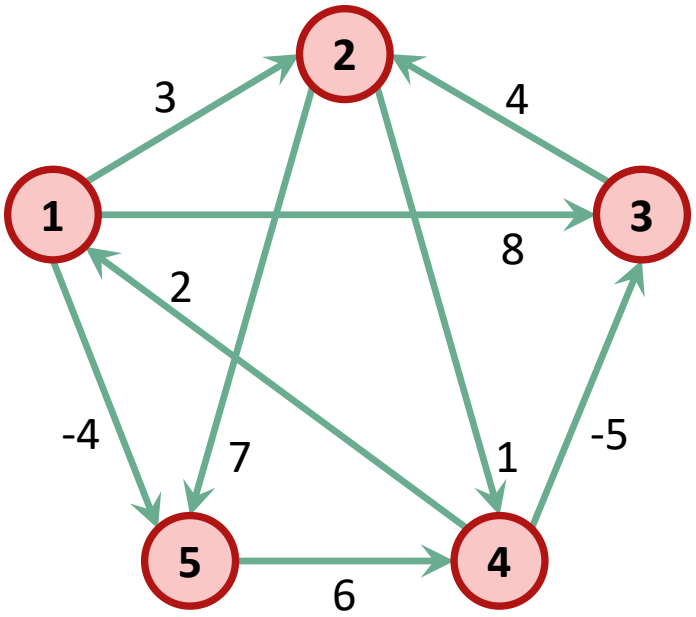


Example of F-W

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

$D^{(3)}$

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

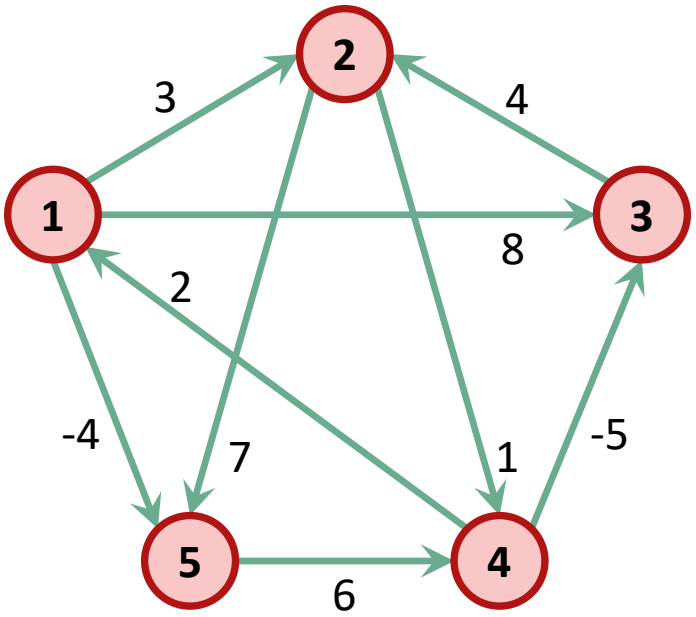


Example of F-W

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

$D^{(3)}$

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

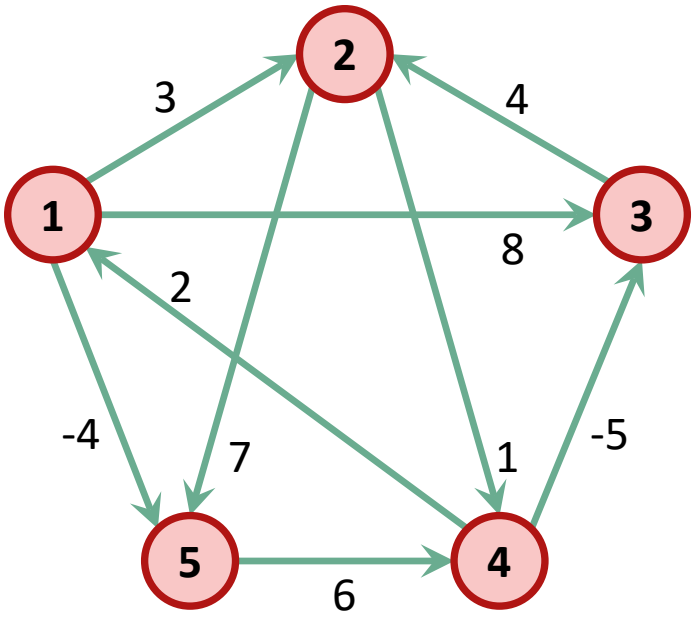


Example of F-W

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

$D^{(4)}$

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

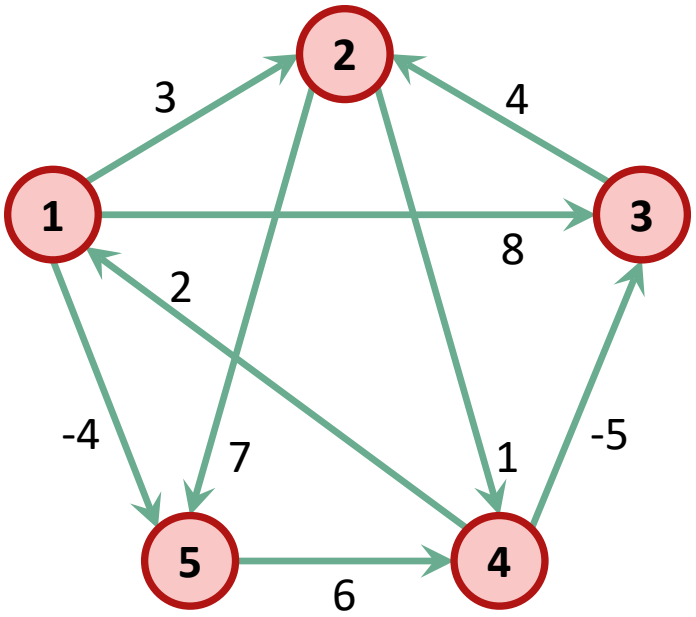


Example of F-W

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

$D^{(4)}$

	1	2	3	4	5
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

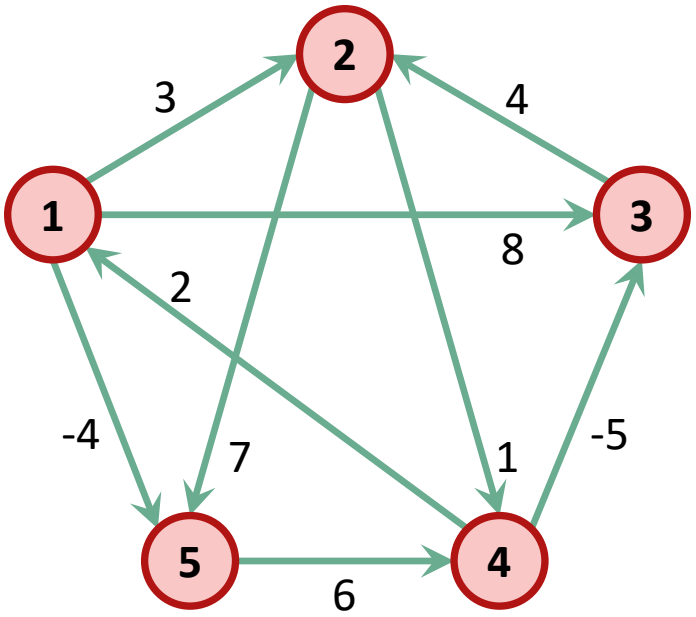


Example of F-W

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

$D^{(5)}$

	1	2	3	4	5
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

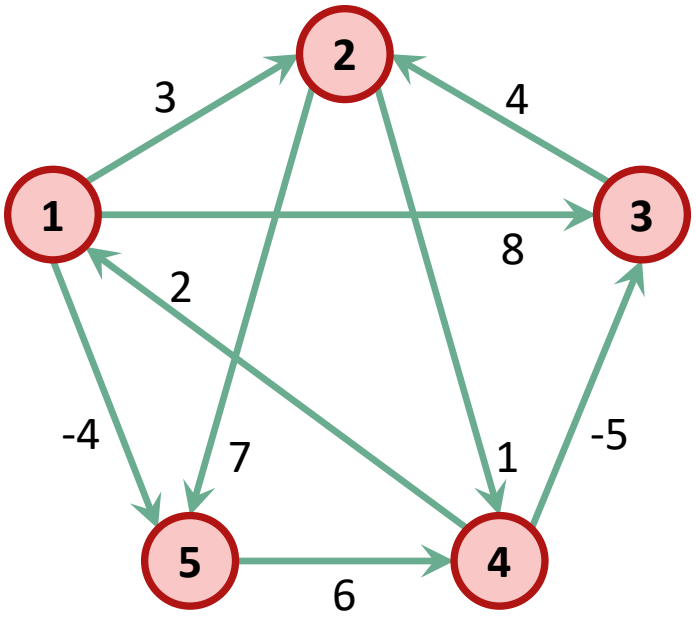


Example of F-W

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

$D^{(5)}$

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0



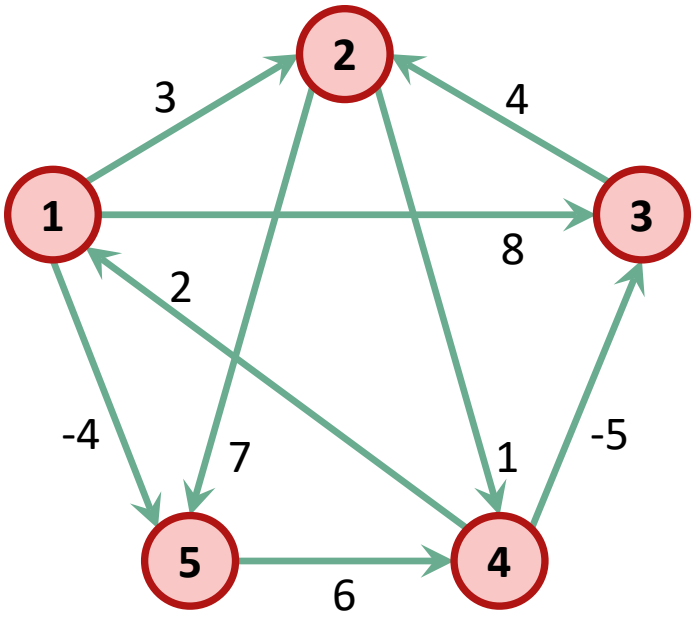
Example of F-W

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

DONE!

$D^{(5)}$

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0



Floyd-Warshall algorithm

- **Correctness**
  - We've just shown in last time.
  - **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:

    $D^{(n)}[u, v]$ = a simple shortest path 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.
  - 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 $O(n^3)$?

- 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.

What have we learned?

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

Graph Algorithms

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

Recap

- ***Dynamic programming!***
  - This is a fancy name for:
    - Break up an optimization problem into smaller problems.
    - Take advantage of overlapping sub-problems.
  - Two shortest-path algorithms:
    - Bellman-Ford for single-source shortest path
    - Floyd-Warshall for all-pairs shortest path

- **Next time:**
  - More examples of ***Dynamic programming!***


Any Question?