CSE100: Design and Analysis of Algorithms
Lecture 21 – Dynamic Programming (wrap up) and More Dynamic Programming

Apr 12th 2022

Bellman-Ford, Floyd-Warshall, Longest Common Subsequences, Knapsack, and
(if time) Independent Sets in Trees

Bellman-Ford* algorithm (review)

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

Bellman-Ford*(G,s):

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

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

Today (part 1)

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

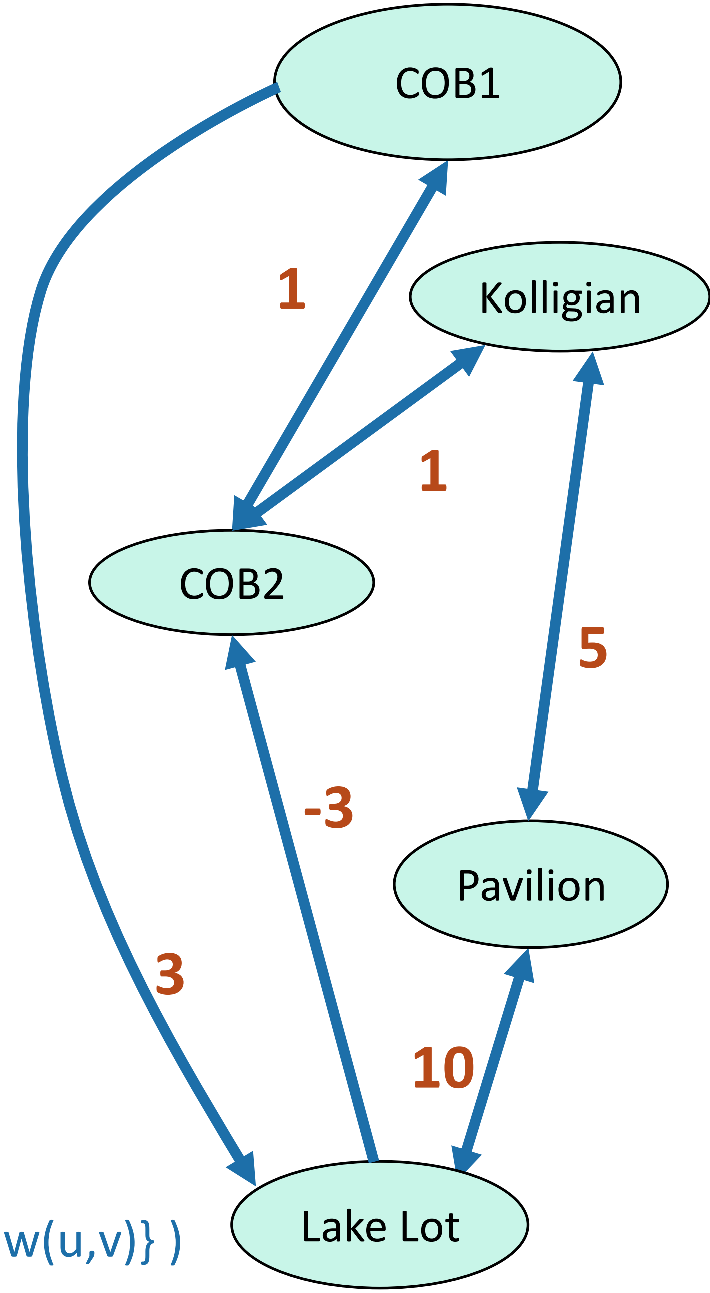
Bellman-Ford take-aways

- Running time is O(mn)
  - For each of n rounds, update m edges.
- Works fine with negative edges.
- Does not work with negative cycles.
  - But it can detect negative cycles!

Go through the slides, or CLRS, and understand how to modify Bellman-Ford to handle negative cycles!

Negative edge weights

	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**3**
$d^{(2)}$	0	**0**	**2**	**13**	3
$d^{(3)}$	0	0	**1**	**7**	3
$d^{(4)}$	0	0	1	**6**	3

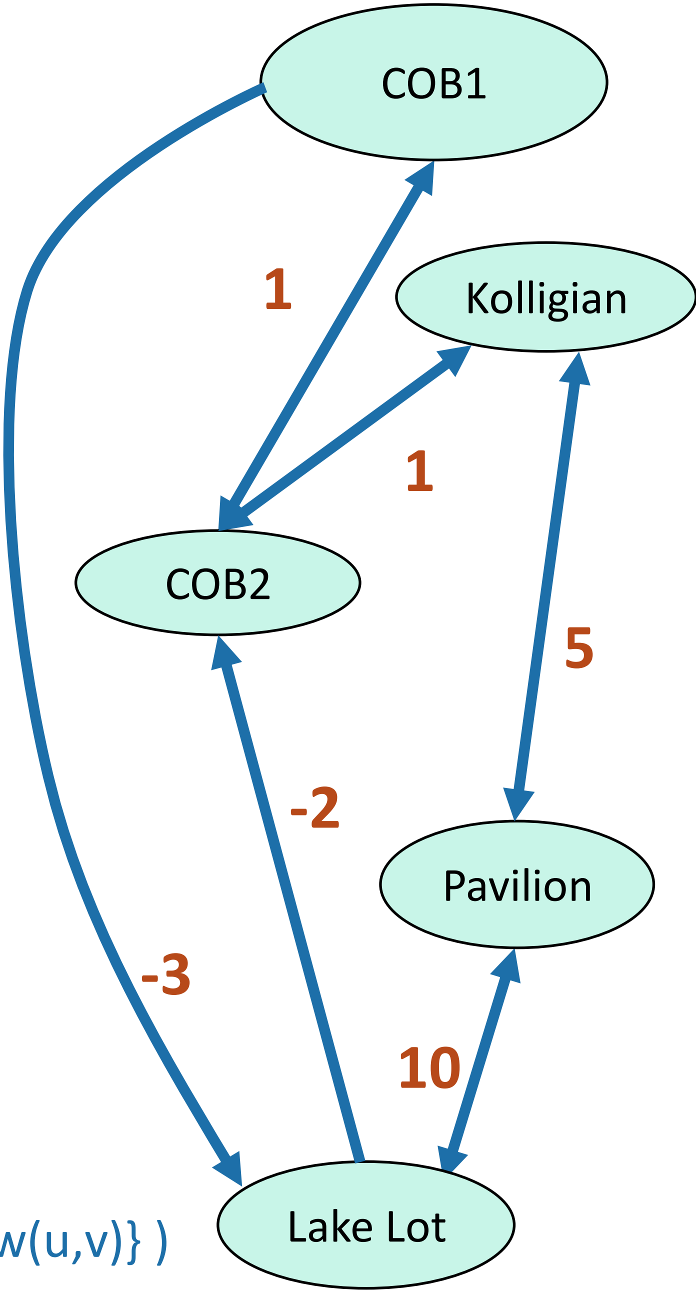


```
• For i=0,…,n-2:
    • For v in V:
        • $d^{(i+1)}[v] \leftarrow \min( d^{(i)}[v] , \min_{u \text{ in v.inNbrs}}\{d^{(i)}[u] + w(u,v)\} )$
```

B-F with negative cycles

	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**-3**
$d^{(2)}$	0	**-5**	**2**	**7**	-3
$d^{(3)}$	**-4**	-5	**-4**	7	-3

This is not looking good!



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

B-F with negative cycles

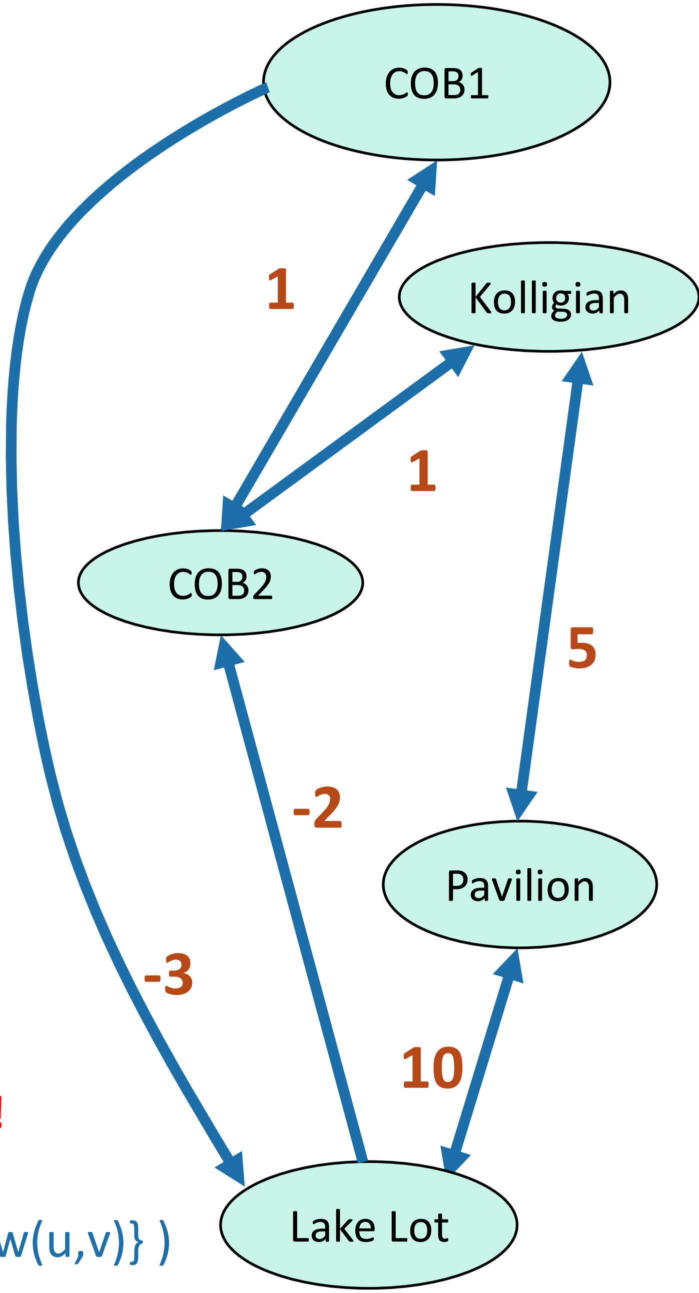
	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**-3**
$d^{(2)}$	0	**-5**	**2**	**7**	-3
$d^{(3)}$	**-4**	-5	**-4**	7	-3
$d^{(4)}$	-4	-5	-4	7	**-7**

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

$d^{(5)}$	-4	**-9**	-4	**3**	-7

Some stuff changed!

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



How Bellman-Ford deals with negative cycles

- If there are no negative cycles:
    - Everything works as it should.
    - The algorithm stabilizes after n-1 rounds.
    - Note: Negative ***edges*** are okay!!
- If there are negative cycles:
    - Not everything works as it should…
        - Note: it couldn’t possibly work, since shortest paths aren’t well-defined if there are negative cycles.
    - The d[v] values will keep changing.
- Solution:
    - Go one round more and see if things change.
        - If so, return NEGATIVE CYCLE ☹
    - (Pseudocode on next slide)

Bellman-Ford algorithm

Bellman-Ford*(G,s):

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

Running time: O(mn)

Summary

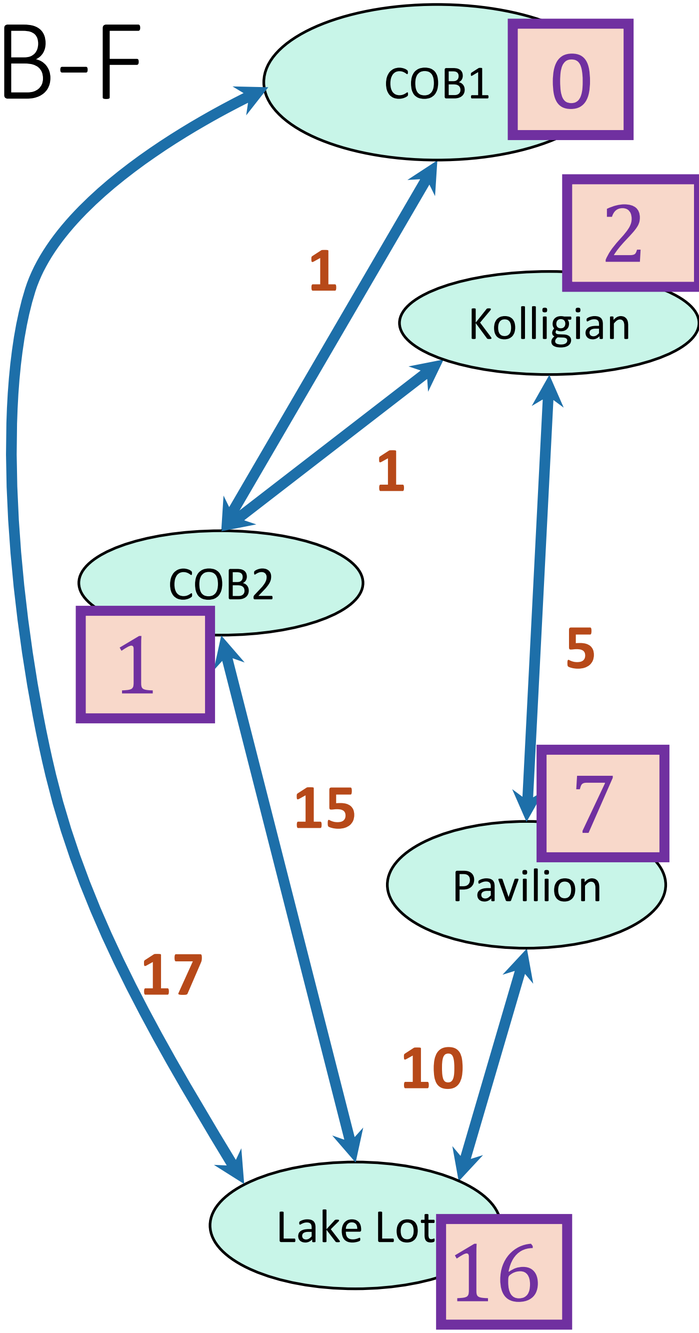
- The Bellman-Ford algorithm:
  - Finds shortest paths in weighted graphs with negative edge weights
  - runs in time O(nm) on a graph G with n vertices and m edges.
- If there are no negative cycles in G:
  - the BF algorithm terminates with $d^{(n-1)}[v] = d(s,v)$.
- If there are negative cycles in G:
  - the BF algorithm returns `negative cycle`.

Important thing about B-F

for the rest of this lecture

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

	COB1	COB2	Kolligian	Pavilion	Lake
$d^{(0)}$	0	∞	∞	∞	∞
$d^{(1)}$	0	**1**	∞	∞	**17**
$d^{(2)}$	0	1	**2**	**27**	**16**
$d^{(3)}$	0	1	2	**7**	16
$d^{(4)}$	0	1	2	7	16



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

How not to compute Fibonacci Numbers

- Definition:
  - F(n) = F(n-1) + F(n-2), with F(0) = F(1) = 1.
  - The first several are:
    - 1
    - 1
    - 2
    - 3
    - 5
    - 8
    - 13, 21, 34, 55, 89, 144,…
- Question:
  - Given n, what is F(n)?

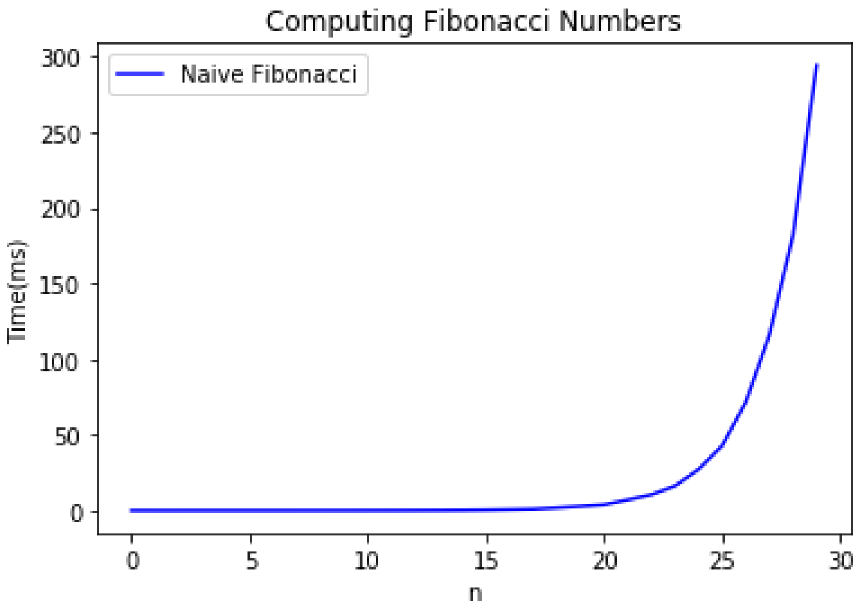
Candidate algorithm

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

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

Running time?

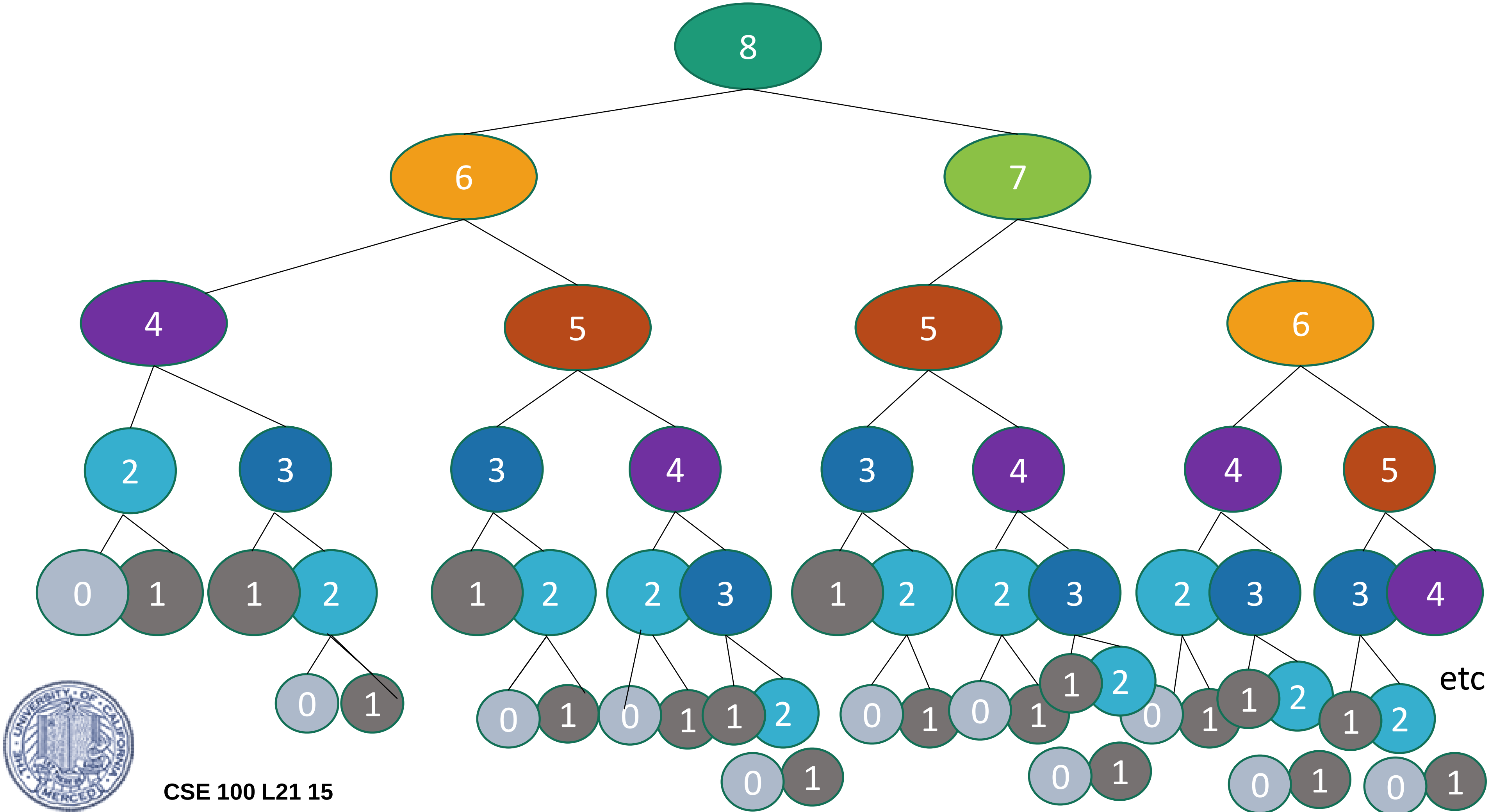
- T(n) = T(n-1) + T(n-2) + O(1)
- T(n) ≥ T(n-1) + T(n-2) for n ≥ 2
- So T(n) grows *at least* as fast as the Fibonacci numbers themselves…
- Fun fact, that's like ϕ^n where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.
- aka, **EXPONENTIALLY QUICKLY** ☹



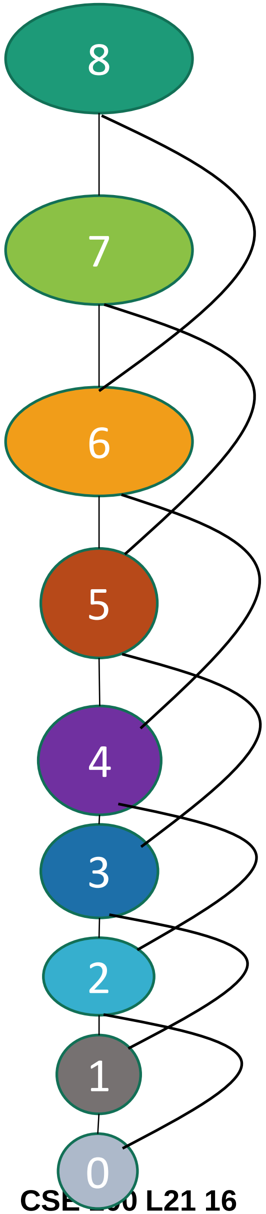
What’s going on?

Consider Fib(8)

That’s a lot of repeated computation!

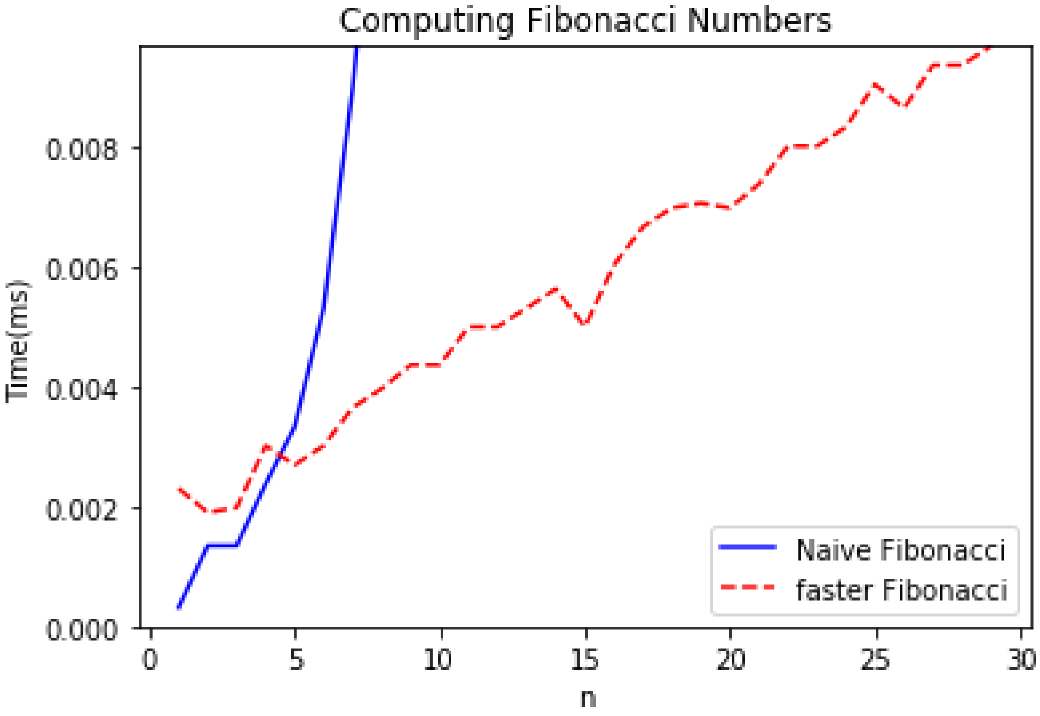


Maybe this would be better:



```
def fasterFibonacci(n):
    • F = [1, 1, None, None, …, None ]
        • \\ F has length n + 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**
  - E.g., ***shortest*** path
  - (Fibonacci numbers aren’t an optimization problem, but they are a good example…)

Elements of dynamic programming

1. Optimal sub-structure:

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

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

  - Bellman-Ford:

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

Shortest path with at most i edges from s to v

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

Elements of dynamic programming

2. Overlapping sub-problems:

- The sub-problems overlap.
  - 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 $d^{(i)}[v]$.

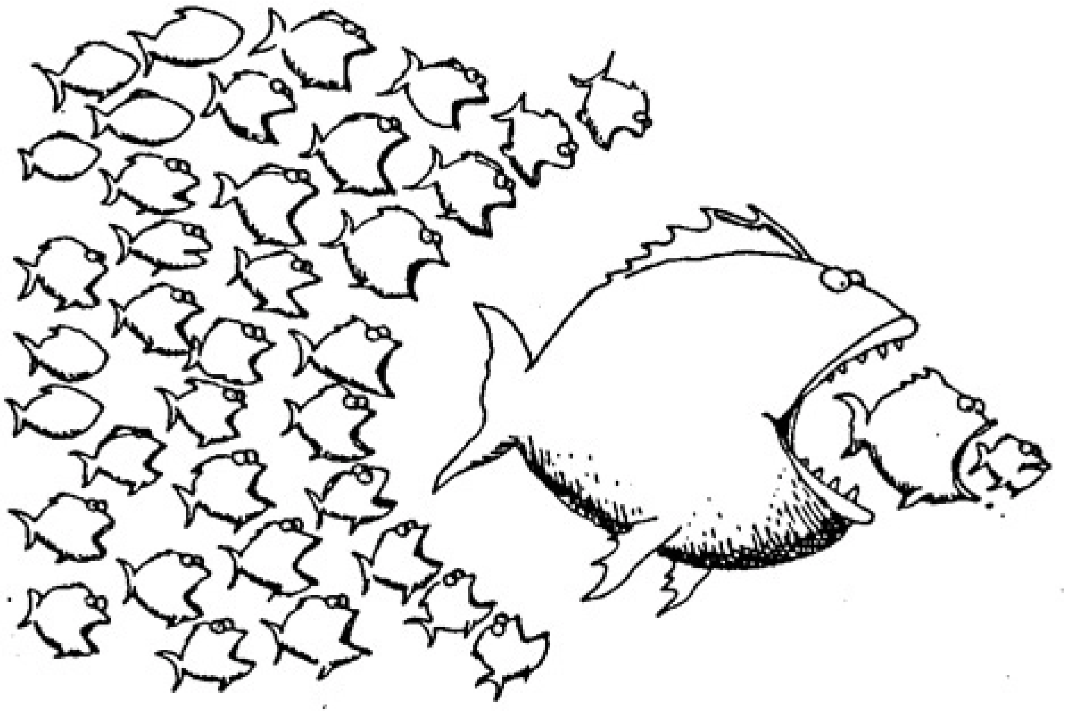
  - This means that we can save time by solving a sub-problem just once and storing the answer.

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

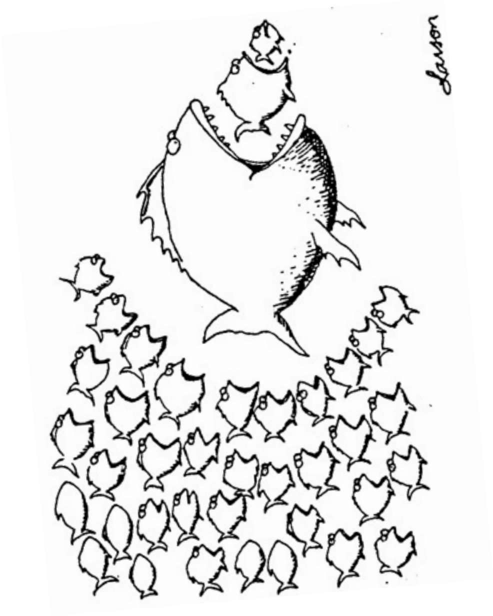


This picture isn't hugely relevant but I like it.

Bottom up approach

what we just saw.

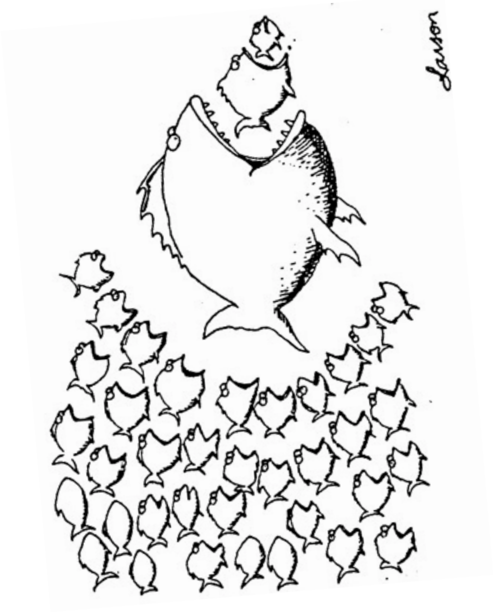
- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
  - fill in F[2]
- …
- Then bigger problems
  - fill in F[n-1]
- Then finally solve the real problem.
  - fill in F[n]



Bottom up approach

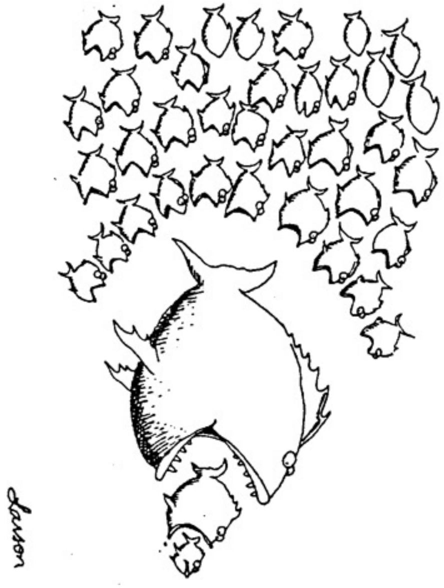
what we just saw.

- 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..

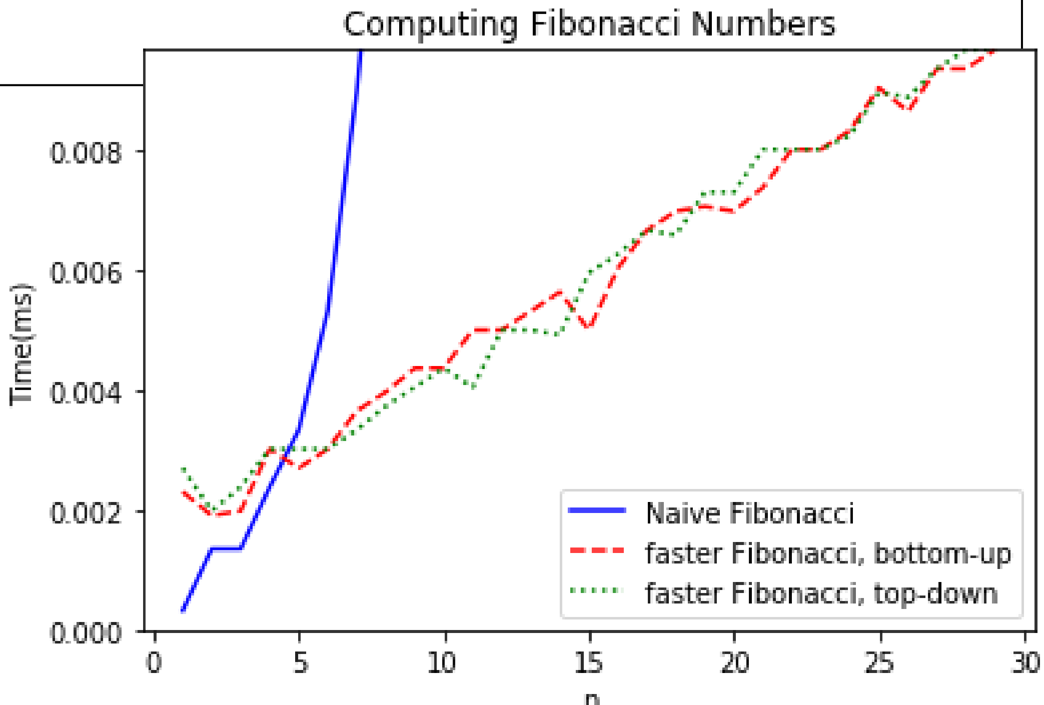


- The difference from divide and conquer:
    - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.
    - Aka, "**memo-ization**"

Example of top-down Fibonacci

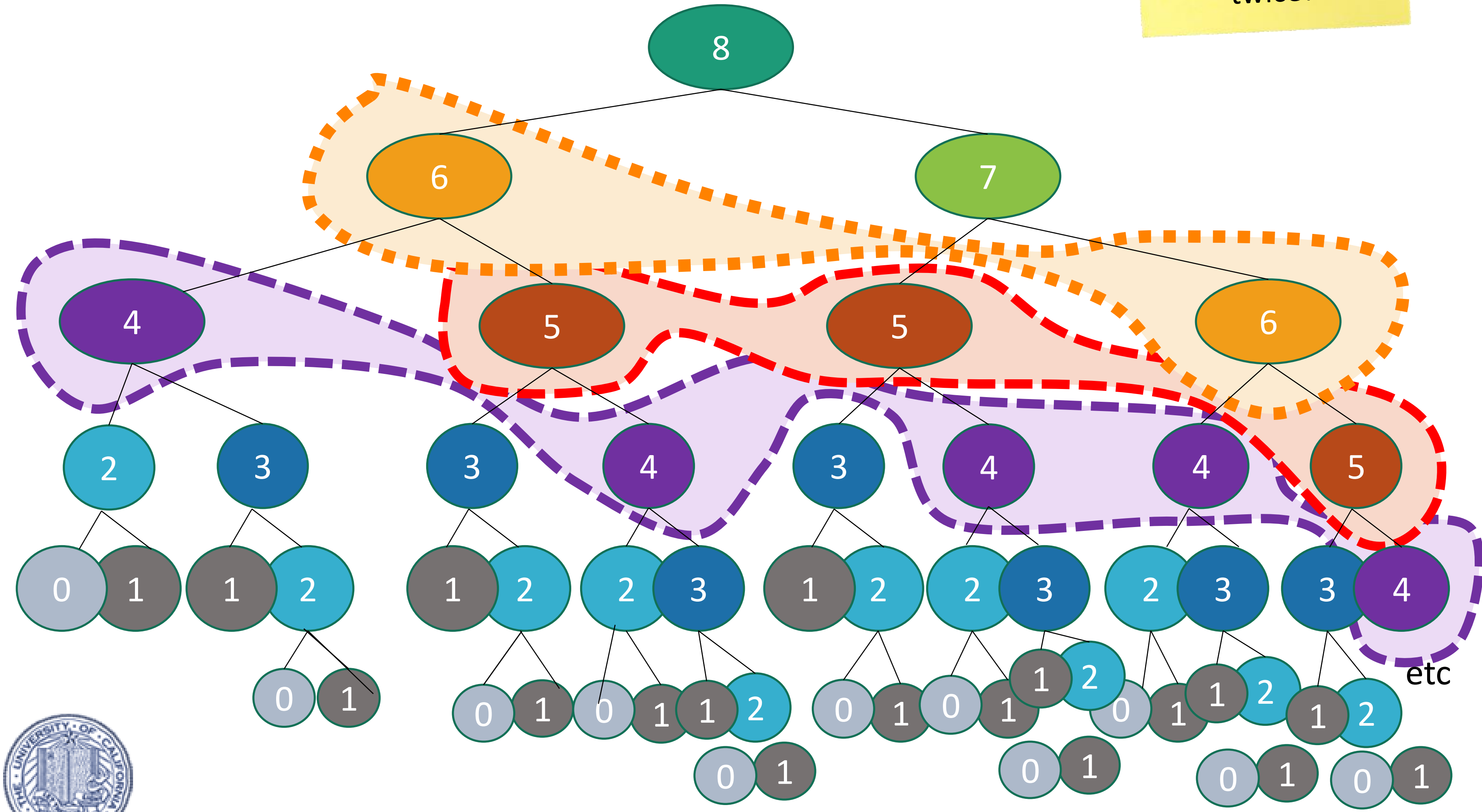
```
• define a global list F = [1,1,None, None, …, None]
• def Fibonacci(n):
    • if F[n] != None:
        • return F[n]
    • else:
        • F[n] = Fibonacci(n-1) + Fibonacci(n-2)
    • return F[n]
```

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



Memo-ization visualization

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

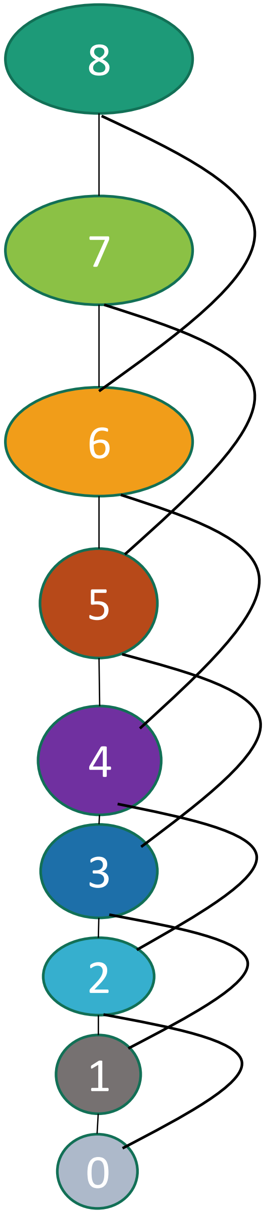


Memo-ization Visualization
contd.

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:

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

Why “*dynamic programming*” ?

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

Manipulating computer code in an action movie?

Why “*dynamic programming*” ?

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

Floyd-Warshall Algorithm

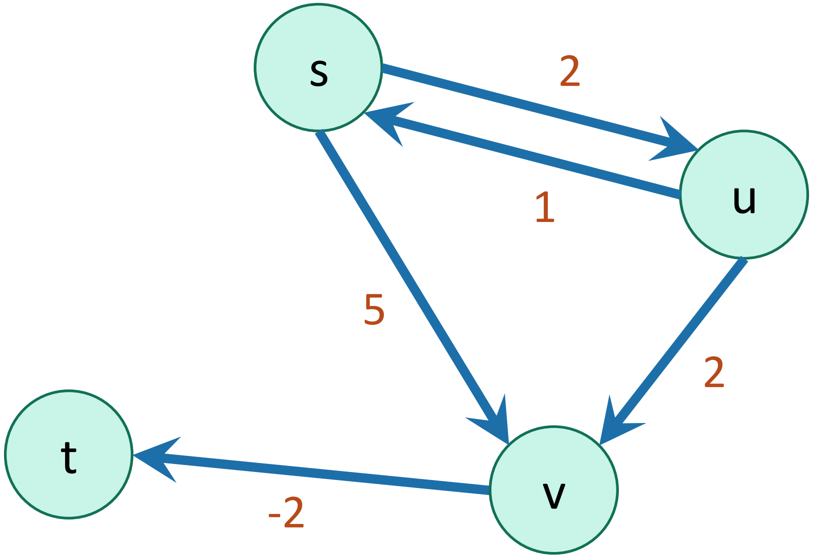
Another example of DP

- 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

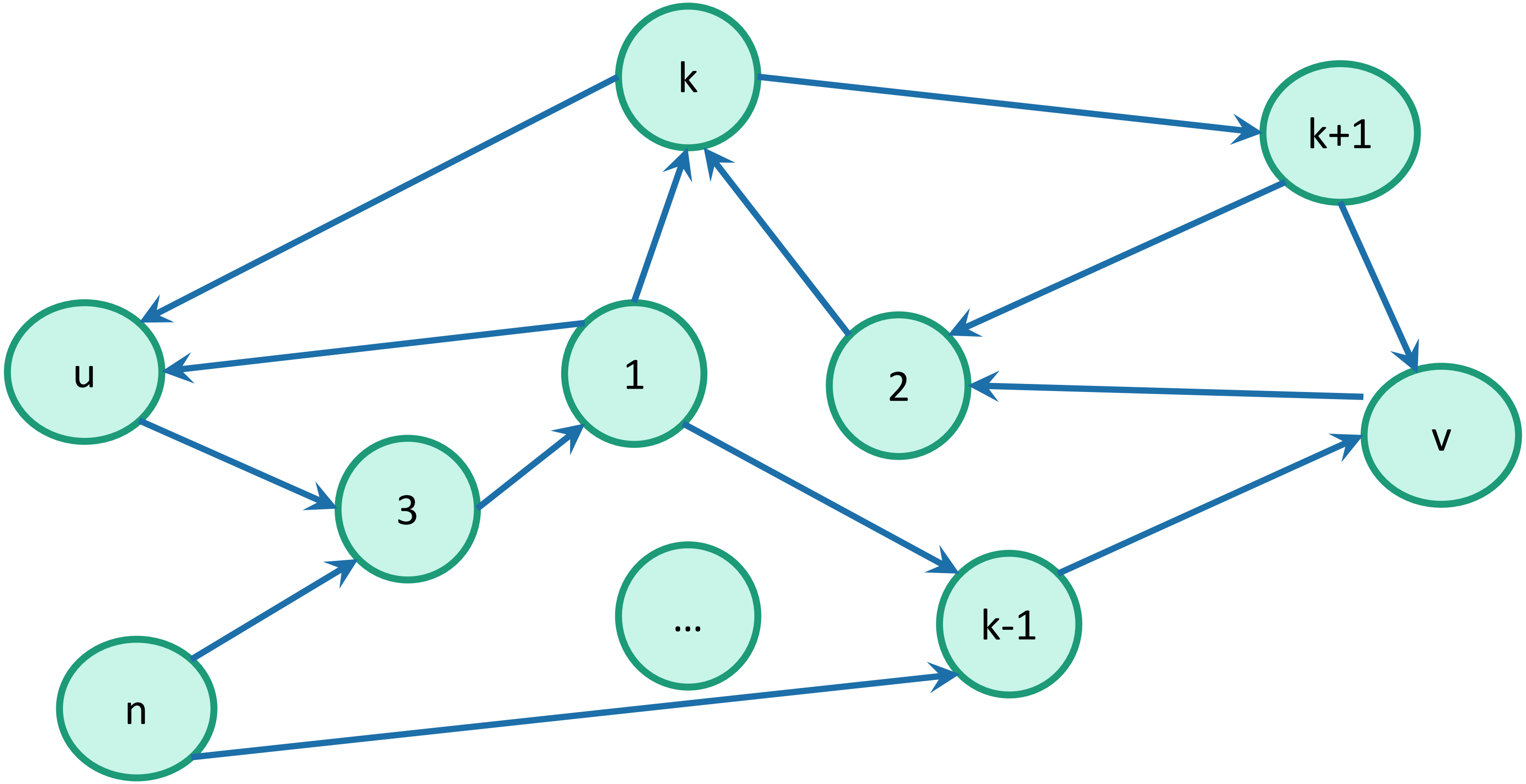
Another example of DP

- 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?

Label the vertices 1,2,…,n

Optimal substructure



Optimal substructure

Label the vertices 1,2,…,n

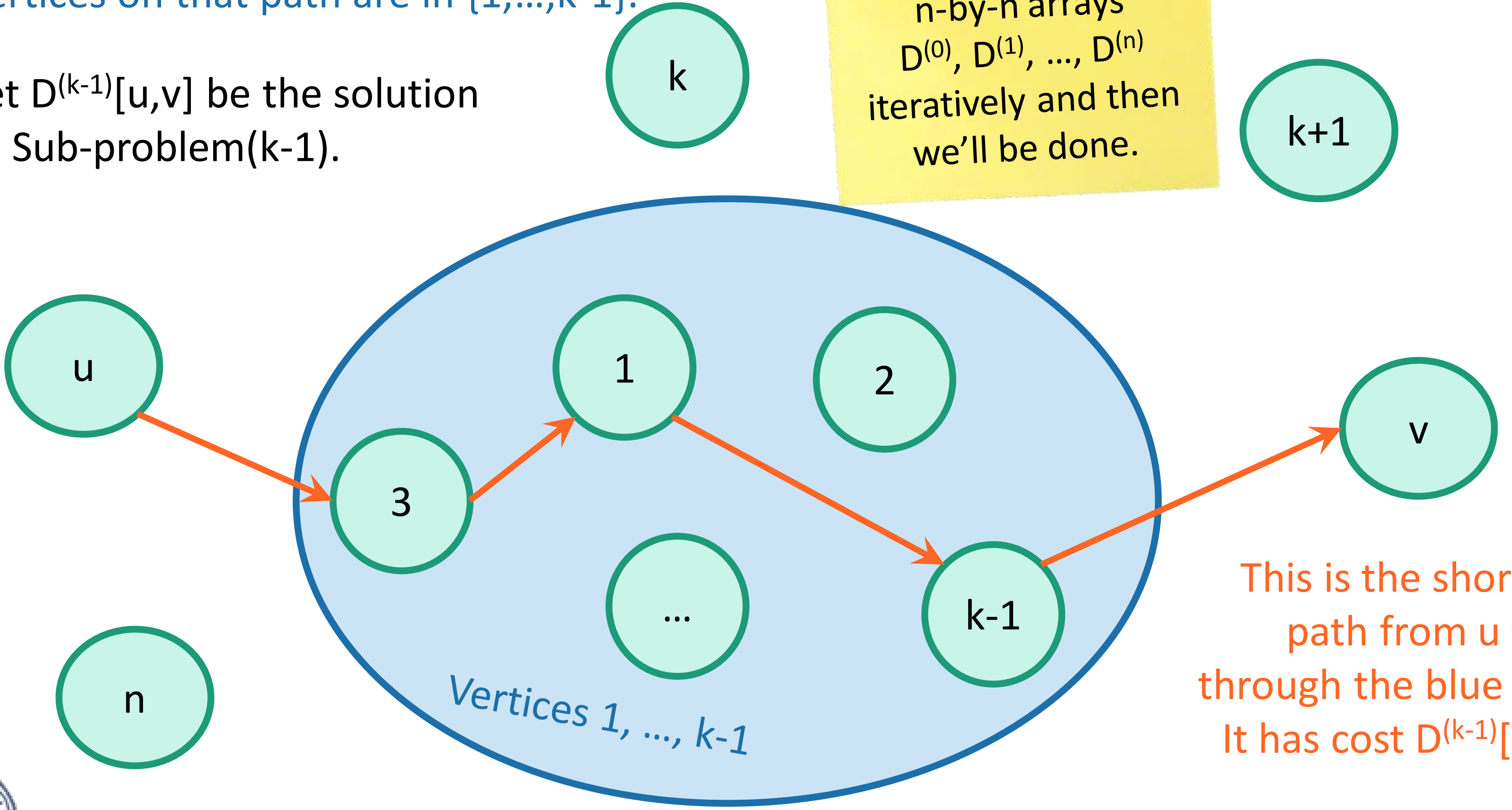
(We omit some edges in the picture below – meant to be a cartoon, not an example).

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).

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).

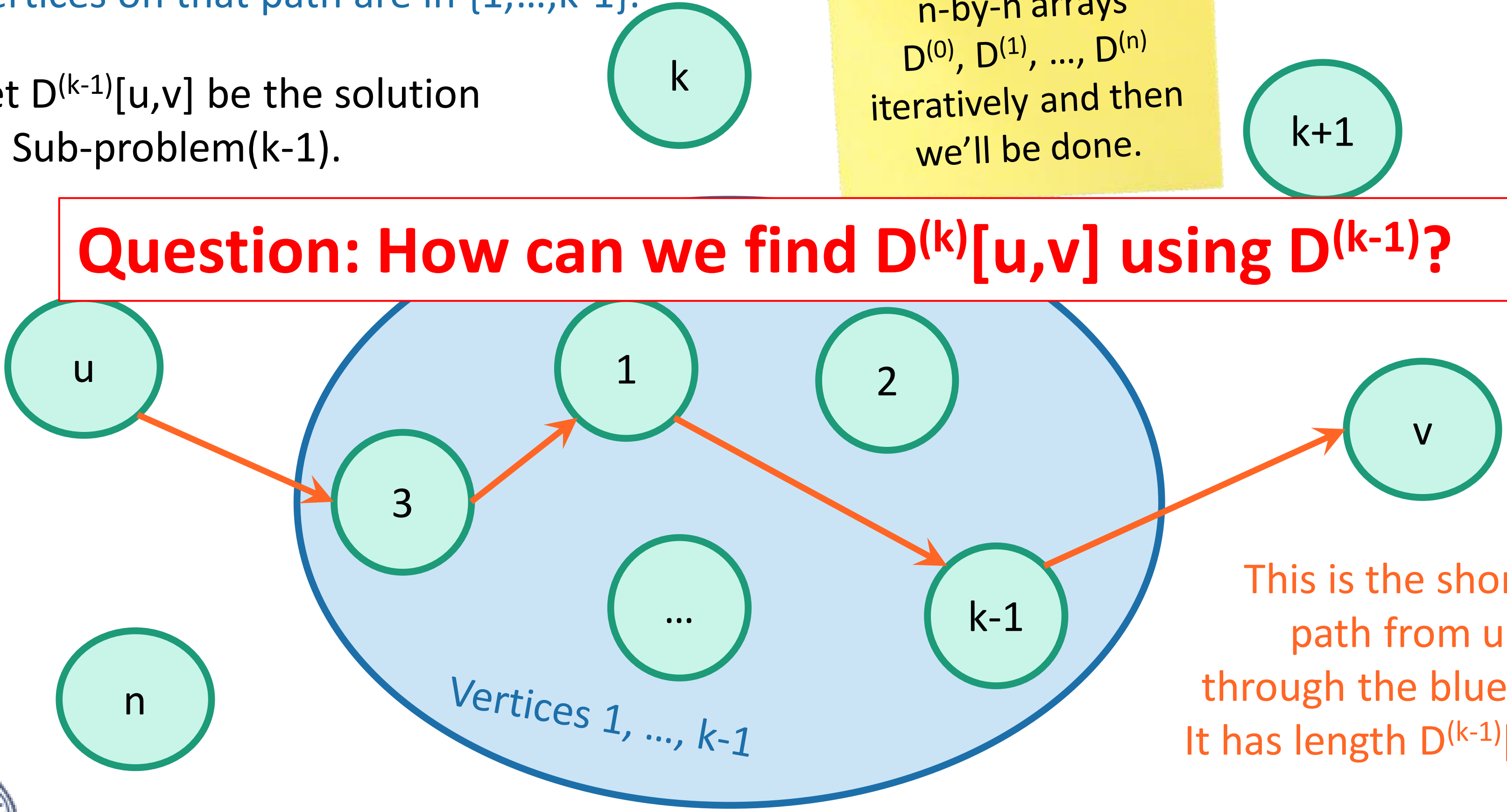
Sub-problem(k-1):

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.

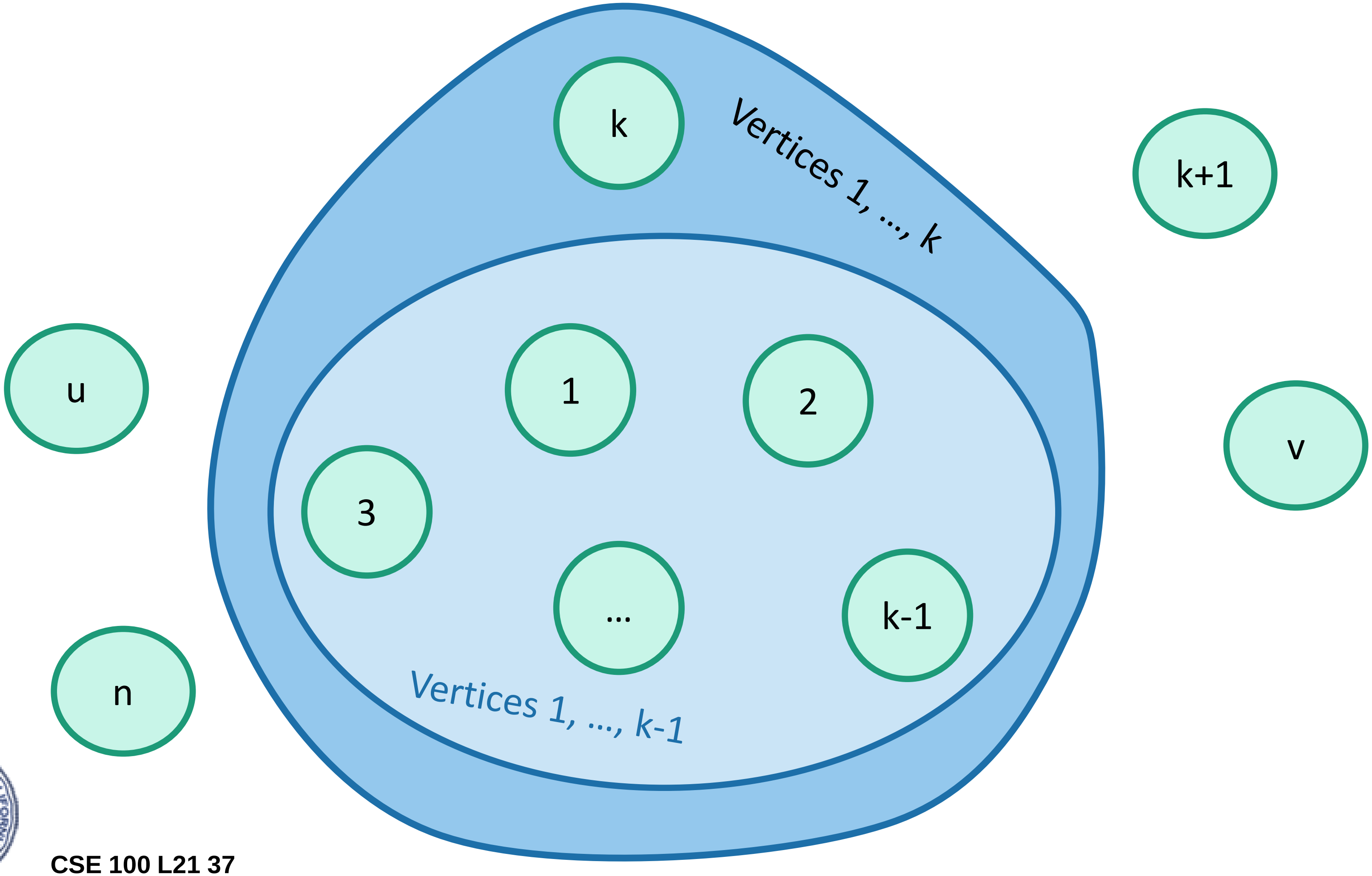
Question: How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?



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

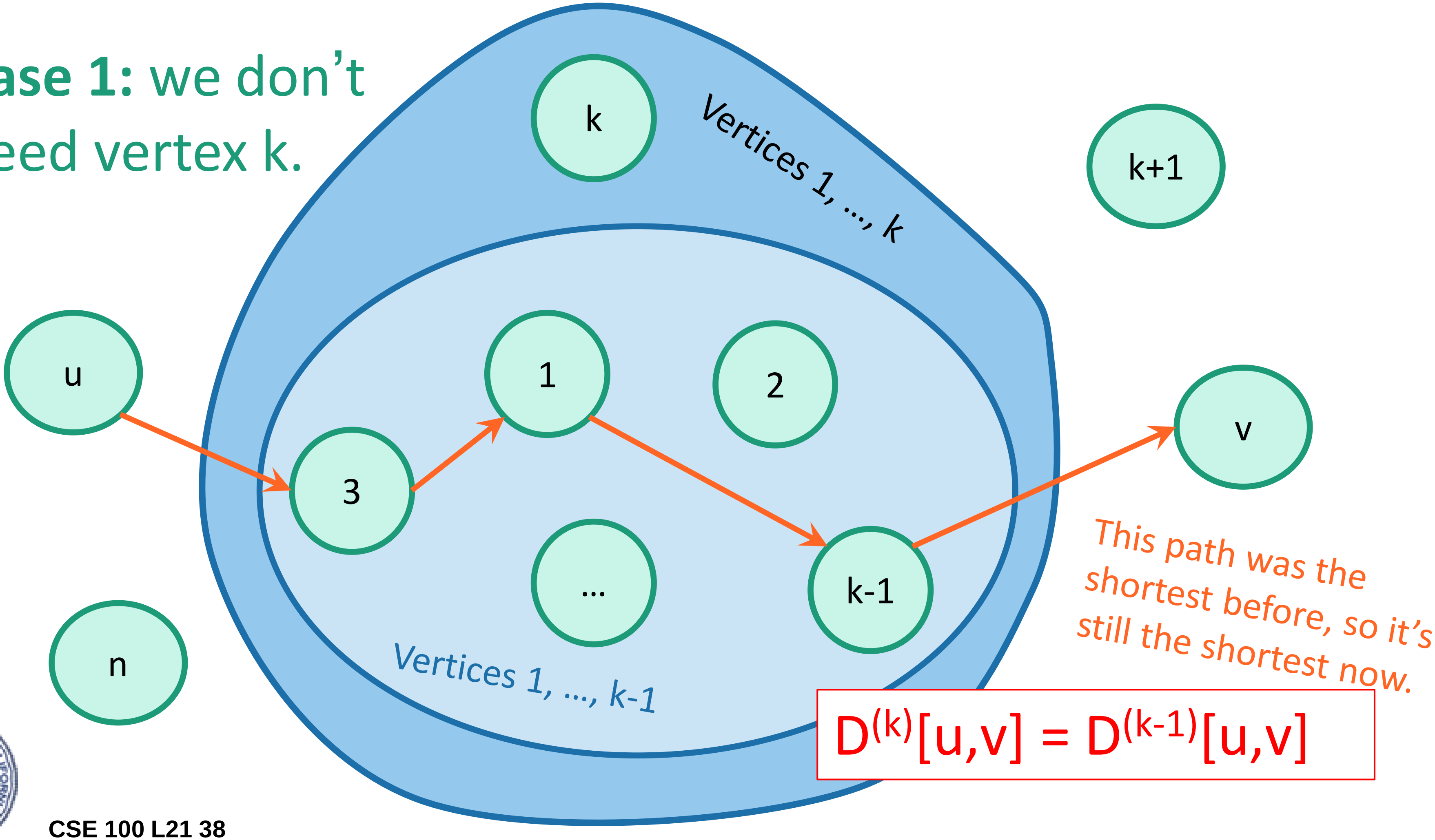
$D^{(k)}[u,v]$ is the cost of the shortest path from u to v so that all internal vertices on that path are in {1, …, k}.



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}.

Case 1: we don't need vertex k.



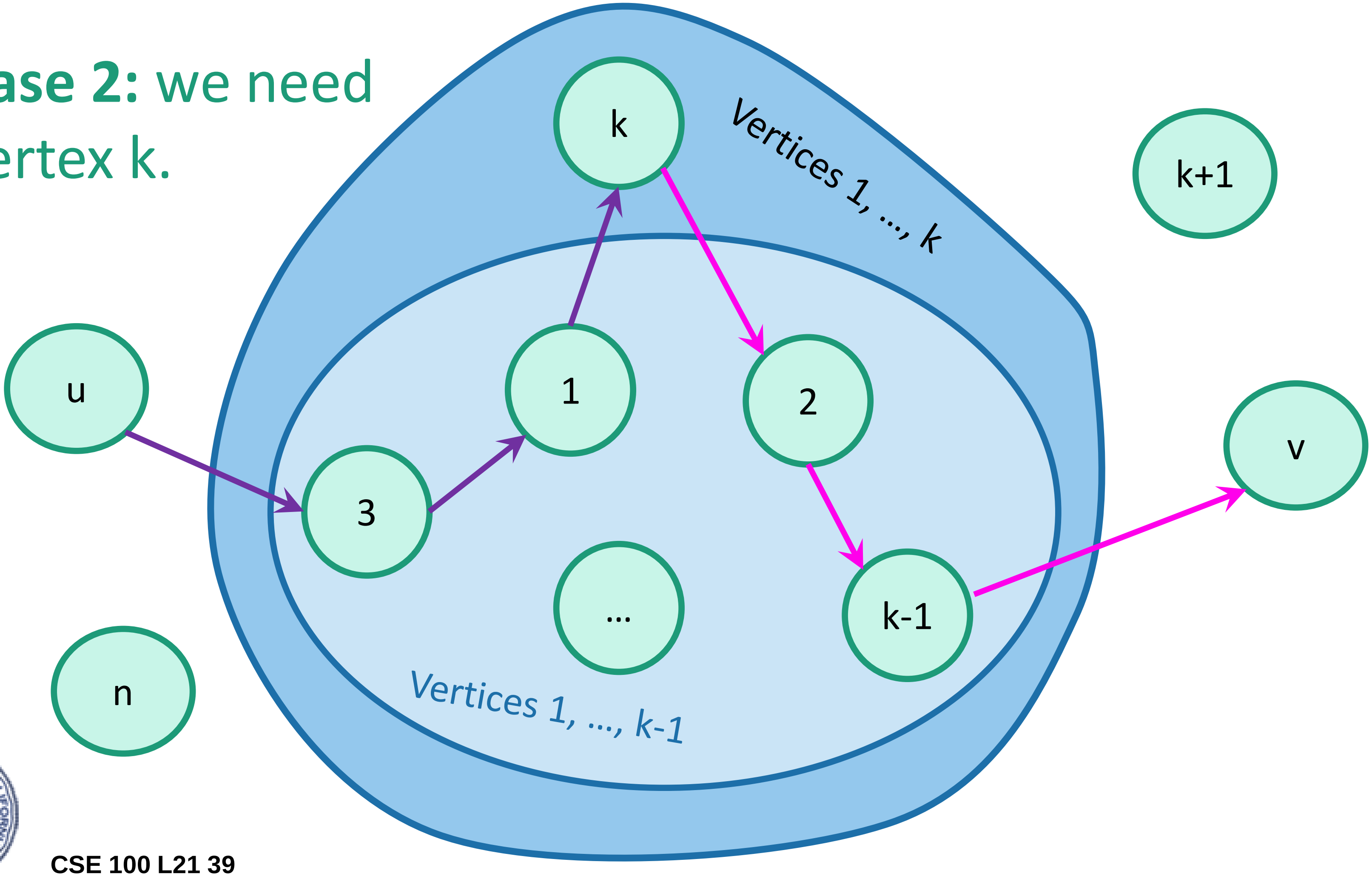
This path was the shortest before, so it's still the shortest now.

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

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.

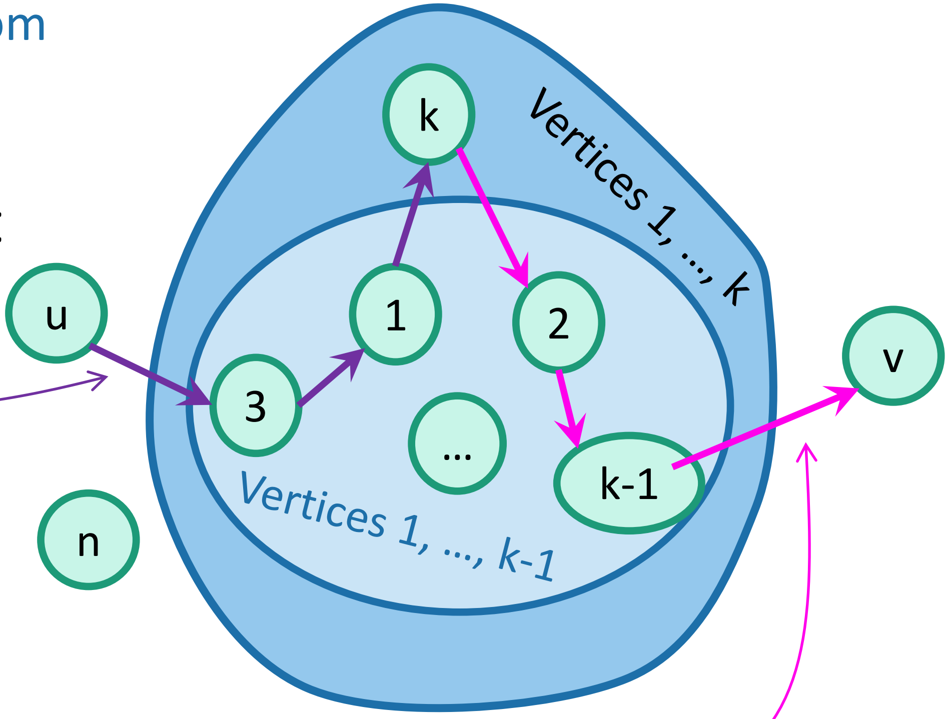


Case 2 continued

Without loss of generality

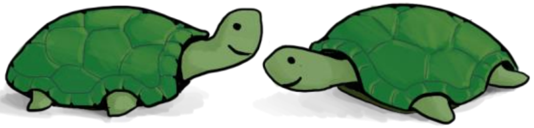
- Suppose there are no negative cycles.
  - Then WLOG the shortest path from u to v through {1,…,k} is **simple**.
- If **that path** passes through k, it must look like this:
- **This path** is the shortest path from u to k through {1,…,k-1}.
  - sub-paths of shortest paths are shortest paths
- Similarly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

Case 2: we need vertex k.

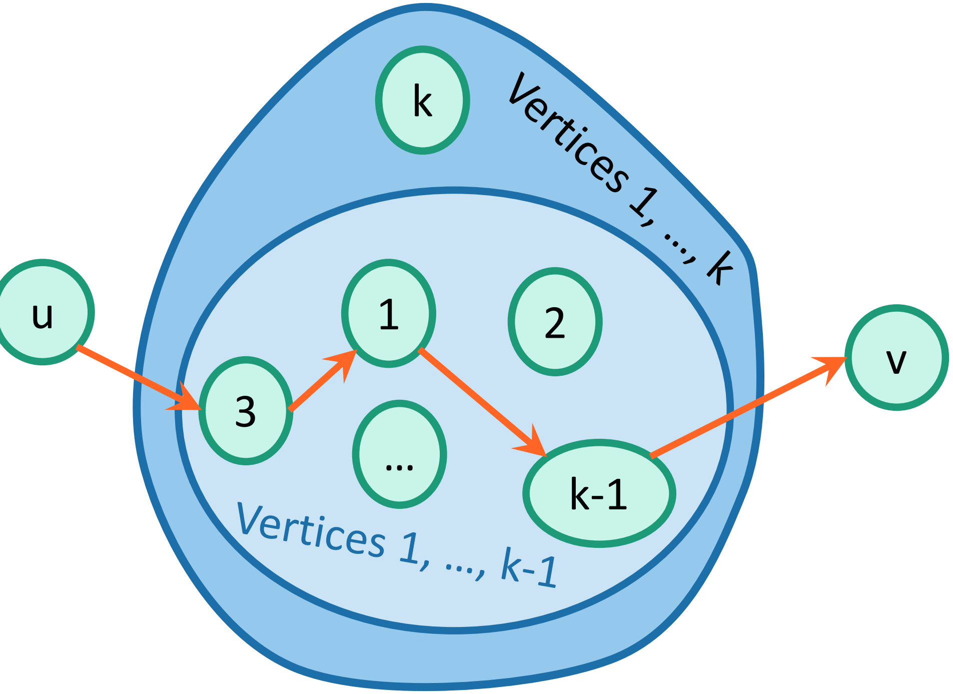


$$D^{(k)}[u,v] = D^{(k-1)}[u,k] + D^{(k-1)}[k,v]$$

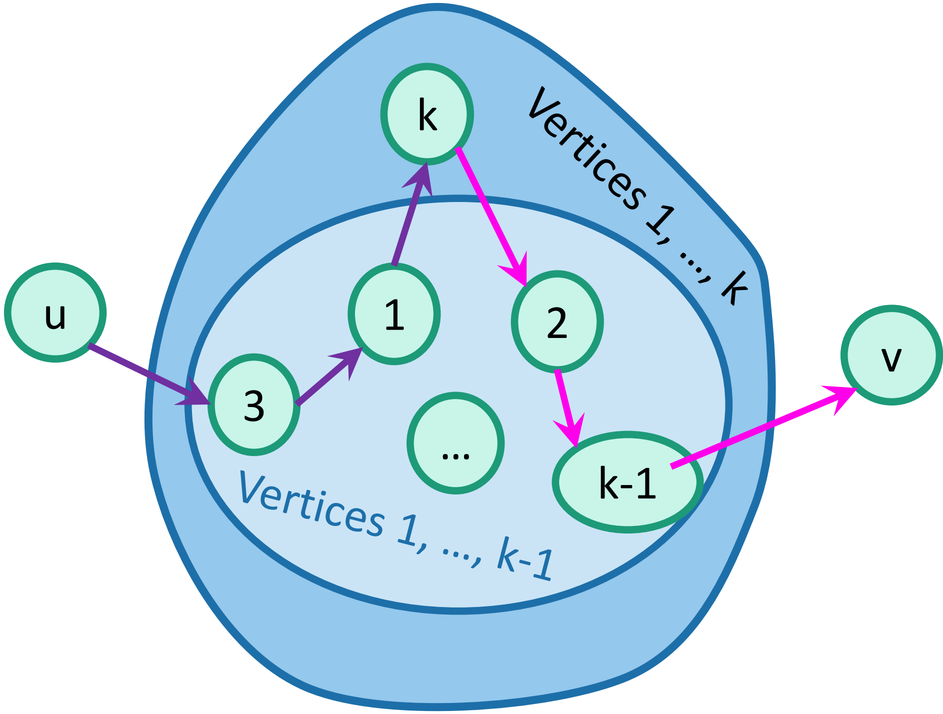
How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?



Case 1: we don’t need vertex k.



Case 2: we need vertex k.



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

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

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

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

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

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

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

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

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

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

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

- Using our ***Dynamic programming*** paradigm, this 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.
- **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.

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 by 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 ⇔ ∃ 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. $D^{(n)}[v,v] < 0$.

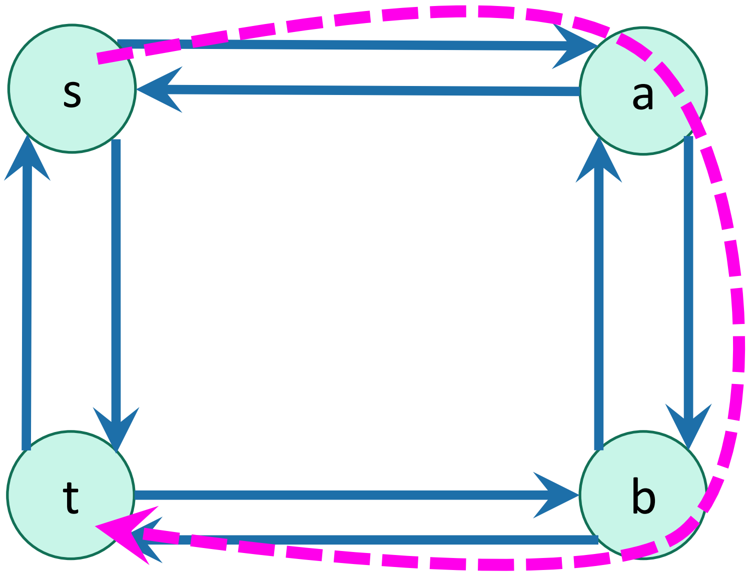
- Algorithm:
  - Run Floyd-Warshall as before.
  - If there is some v so that $D^{(n)}[v,v] < 0$:
    - **return** `negative cycle.`

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 $O(n^3)$.

Bonus: Another Example of DP?

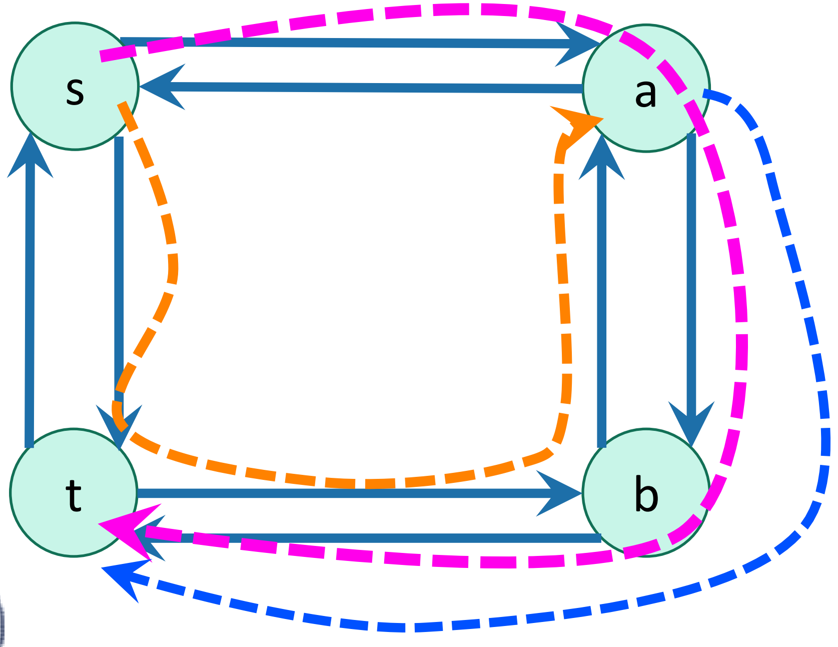
- Longest simple path (say all edge weights are 1):

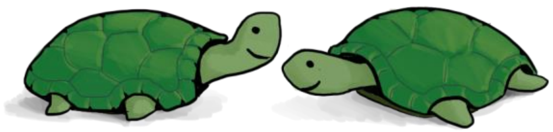


What is the longest simple path from s to t?

This is an optimization problem…

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



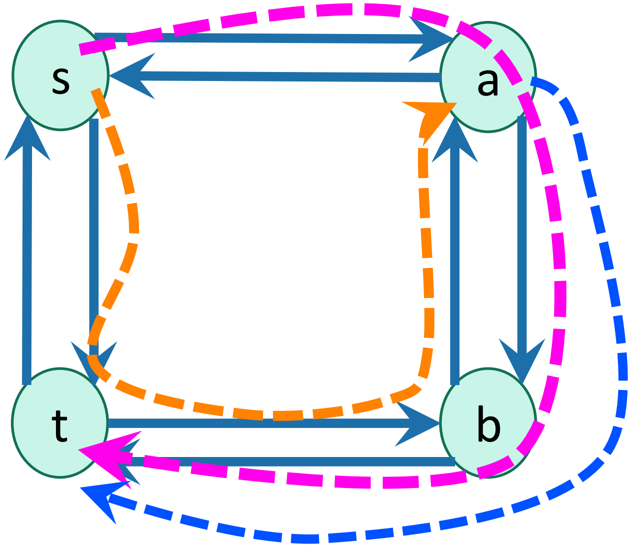


NOPE!

This doesn’t give optimal sub-structure

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

- The sub-problems we came up with aren’t independent:
  - Once we’ve chosen the longest path from a to t
    - which uses b,
  - our longest path from s to a shouldn’t be allowed to use b
    - since b was already used and that breaks the “simple-ness” of the combined path.
- Actually, the longest simple path problem is NP-complete.
  - We don’t know of any polynomial-time algorithms for it, DP or otherwise!



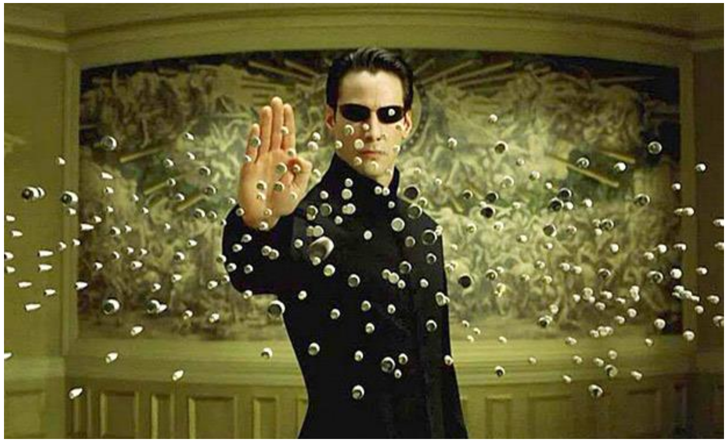
Recap

- Two 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 Part

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

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



Remember…

Dynamic Programming!

- Not coding in an action movie

T[illegible]se programs dynamically
in Mission Impossible

Last Lecture

Dynamic Programming!

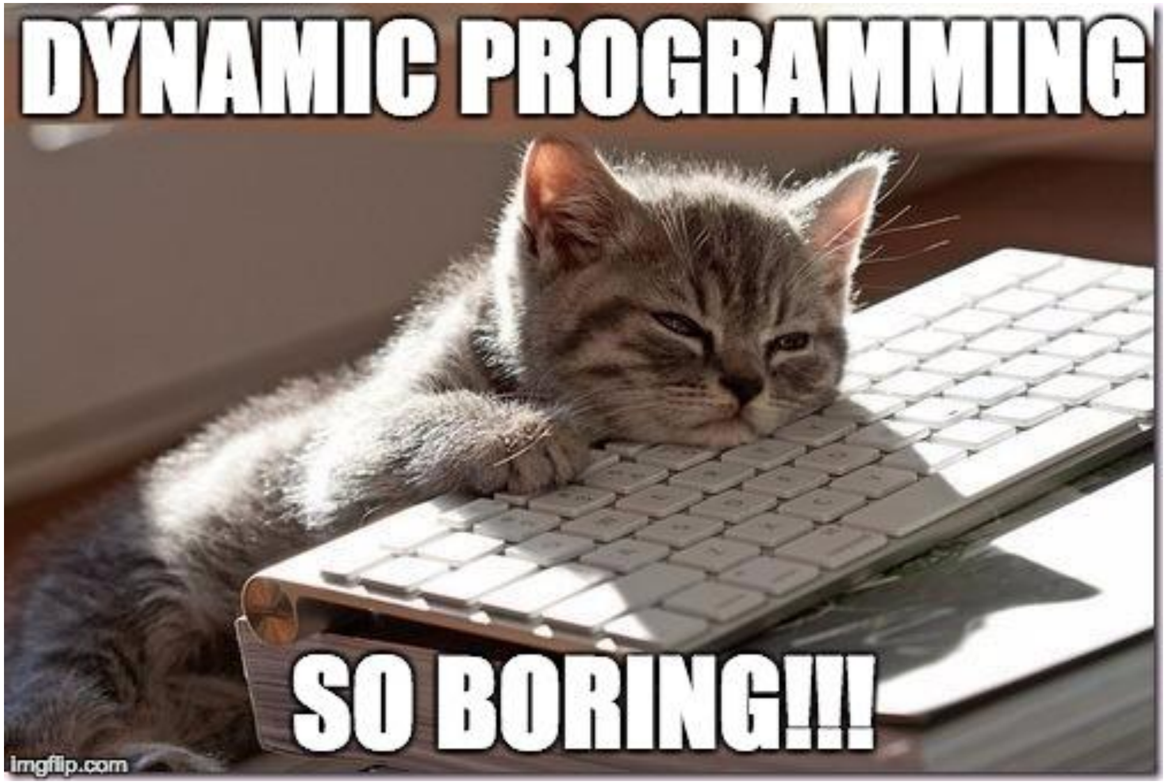
- Dynamic programming is an algorithm design paradigm.
- Basic idea:
  - Identify **optimal sub-structure**
    - Optimum to the big problem is built out of optima of small sub-problems
  - Take advantage of **overlapping sub-problems**
    - Only solve each sub-problem once, then use it again and again
  - Keep track of the solutions to sub-problems in a table as you build to the final solution.

Today (part 2)

- Examples of dynamic programming:
  1. Longest common subsequence
  2. Knapsack problem
     - Two versions!
  3. Independent sets in trees
     - If we have time…
     - (If not the slides will be there as a reference)

The remaining goal of today's lecture

- For you to get really bored of dynamic programming



Longest Common Subsequence

- How similar are these two species?

DNA:

AGCCCTAAGGGCTACCTAGCTT

DNA:

GACAGCCTACAAGCGTTAGCTTG

Longest Common Subsequence

- How similar are these two species?

DNA:

AGCCCTAAGGGCTACCTAGCTT

DNA:

GACAGCCTACAAGCGTTAGCTTG

- Pretty similar, their DNA has a long common subsequence:

AGCCTAAGCTTAGCTT

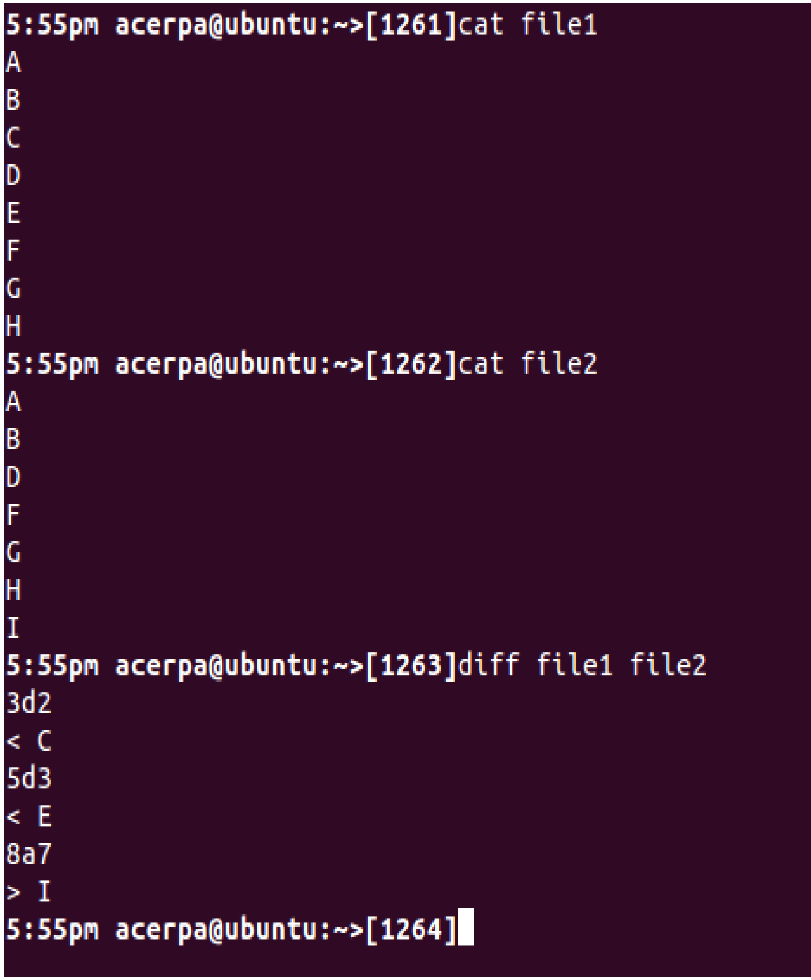
Longest Common Subsequence

- Subsequence:
  - BDFH is a **subsequence** of ABCDEFGH
- If X and Y are sequences, a **common subsequence** is a sequence which is a subsequence of both.
  - BDFH is a **common subsequence** of ABCDEFGH and of ABDFGHI
- A **longest common subsequence**…
  - …is a common subsequence that is longest.
  - The **longest common subsequence** of ABCDEFGH and ABDFGHI is ABDFGH.

We sometimes want to find these

- Applications in bioinformatics

- The unix command `diff`
- Merging in version control
  - svn, git, etc…



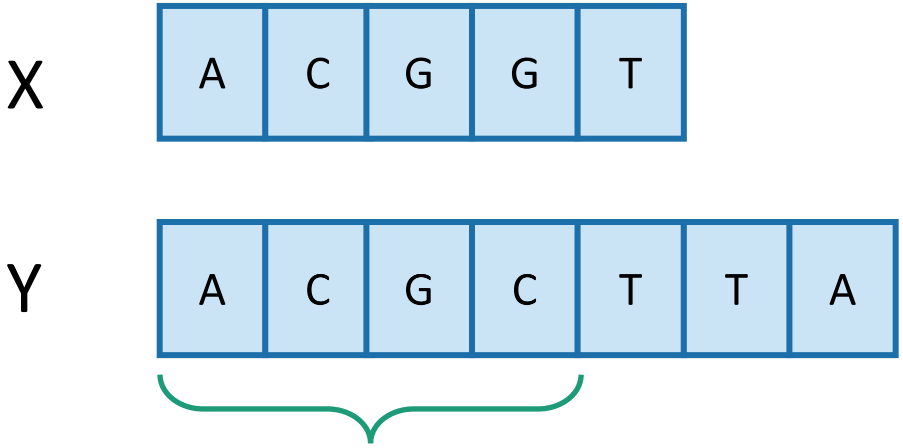
```
5:55pm acerpa@ubuntu:~>[1261]cat file1
A
B
C
D
E
F
G
H
5:55pm acerpa@ubuntu:~>[1262]cat file2
A
B
D
F
G
H
I
5:55pm acerpa@ubuntu:~>[1263]diff file1 file2
3d2
< C
5d3
< E
8a7
> I
5:55pm acerpa@ubuntu:~>[1264]
```

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

Step 1: Optimal substructure

Prefixes:



X	A	C	G	G	T		
Y	A	C	G	C	T	T	A

Notation: denote this prefix **ACGC** by Y_4

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i, j] = length_of_LCS(X_i, Y_j)

Examples: C[2,3] = 2
C[4,4] = 3

Optimal substructure ctd.

- Subproblem:
  - finding LCS's of prefixes of X and Y.

- Why is this a good choice?
  - As we will see, there's some relationship between LCS's of prefixes and LCS's of the whole things.
  - These subproblems overlap a lot.

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.