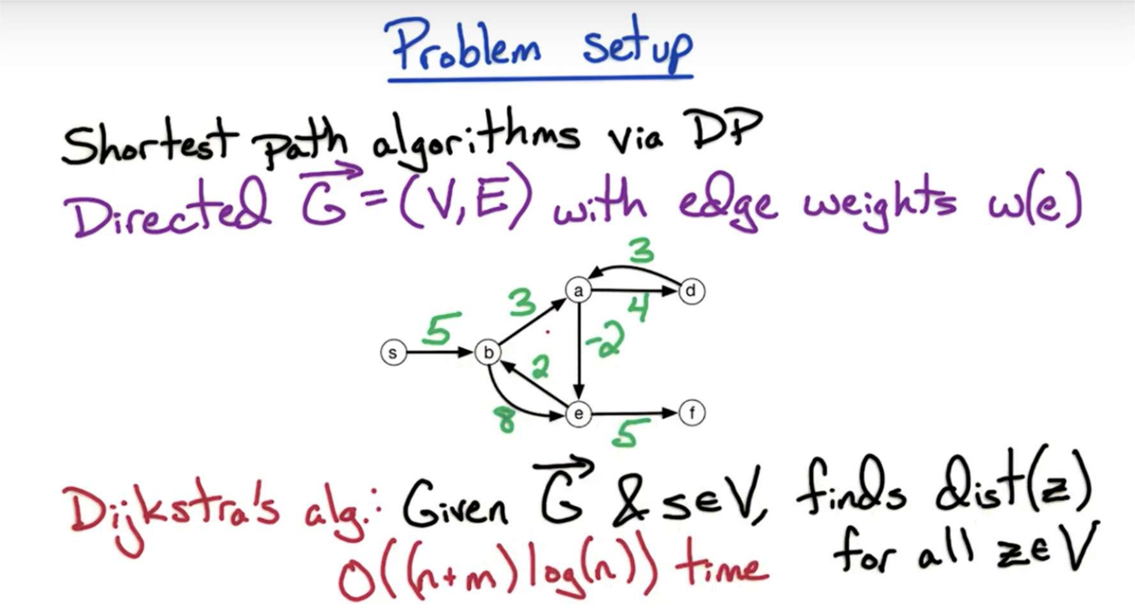
(Slide 1) Shortest Paths via DP
***



Problem setup

Shortest path algorithms via DP

Directed $\overrightarrow{G}=(V,E)$ with edge weights $w(e)$

Dijkstra's alg.: Given $\overrightarrow{G}$ & $s \in V$, finds dist(z) for all $z \in V$

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

In this lecture, we'll look at several versions of shortest path problems and we'll use dynamic programming to design fast algorithms for these problems. The setting is that we have a directed graph G and we'll put the arrow on top to denote that it's a directed graph. And in addition, we have weights on the edges which is denoted by w(e).

Here's an example of a directed graph on six vertices. Let's add in some edge weights. So this edge from s to b has weight 5 … 3 and some of the edges will have negative weights. So let's assign this edge from a to e weight -2 and we can have these anti-parallel edges. So we have an edge from a to d and from d to a and they might have the same or different weights. Now, these anti-parallel edges such as from a to d and d to a are quite useful. They allow us to encode an undirected graph as a directed graph. So if we have an undirected graph, we can replace the edge between a and d by this pair of anti-parallel edges. And, in this way, this directed graph problem is more general than the undirected graph problem because we can encode any undirected graph as a directed graph by replacing each edge in this undirected graph by this pair of anti-parallel edges.

In our first problem, we have a designated start vertex which will denote as s. So let's fix our start vertex s and we're going to look at the length of the shortest path from s to every other vertex in this graph. Therefore, we're going to define the following function. So every other vertex in the graph will be denoted by z. We're going to define this function dist(z). This is defined as the length of the shortest path from s to z. Now, dist(z) is defined for every vertex in the graph. So it's an array of length n. And our goal is to compute this array. To compute the distance of z for every vertex z. So we want to output this array of length n.

Let's take a look at this function for this example. The simplest case is dist(s). What's the length of the shortest path from s to itself? Well, this is length zero. What's the length from s to b? Well, it's 5. s to a, is of length 8, 5 + 3. The shortest path length to e is of length 6 and so on. The shortest path to d is 12. The shortest path to f is 11.

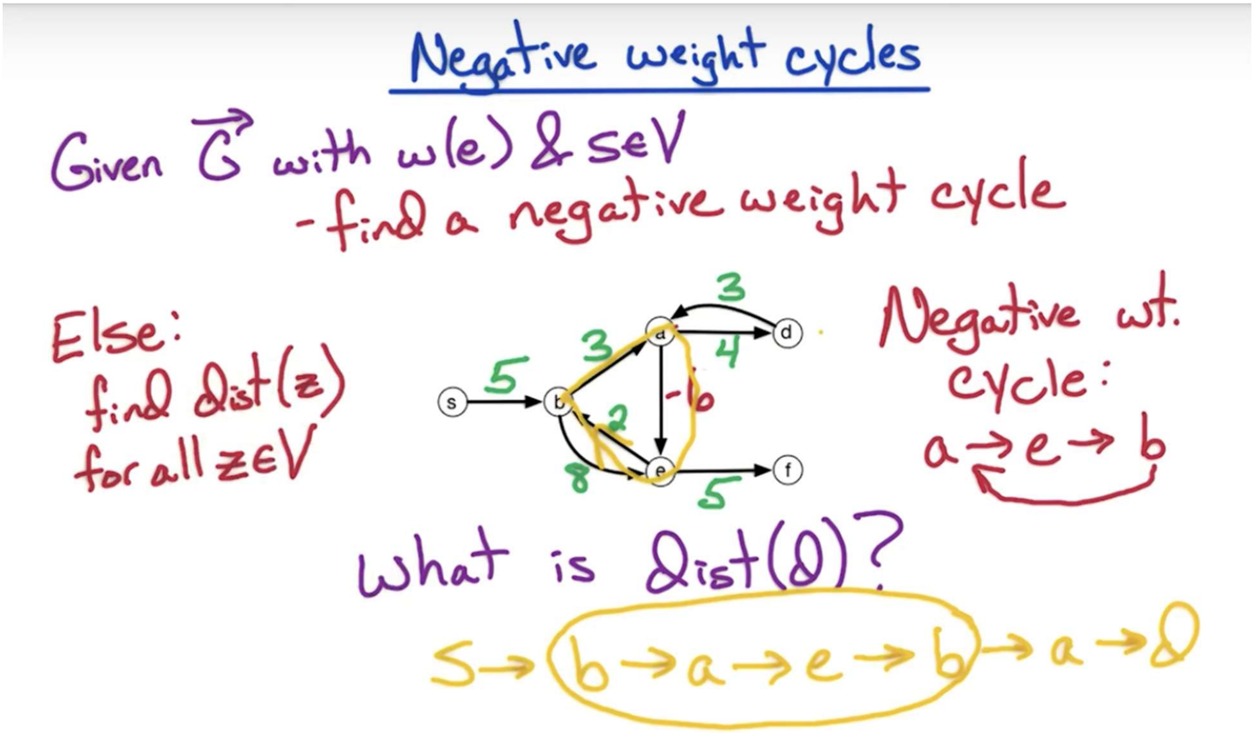
The classical algorithm for this problem is Dijkstra's algorithm. You probably seen it many times before. Now, I'm not going to subject you to another lecture about Dijkstra's algorithm but let me give you a quick recap or a quick reminder about what Dijkstra's algorithm accomplishes.

So Dijkstra's algorithm takes a directed graph with edge weights and a designated start vertex. That's the input to Dijkstra's algorithm. The output is this distance array. It outputs the distance of z for all vertices in the graph. Once again, this is the length of the shortest path from s to this vertex z. Now, how does Dijkstra's algorithm work? Well, it works in a manner similar to BFS, Breath First Search. It explores the graph in this layered approach. Now, recall BFS, as is DFS, takes linear time, so takes O(n+m) time, where n is the number of vertices and m is the number of edges. Now, in Dijkstra's algorithm, we have weights on the edges so we have to use a min heap data structure or priority queue. Each operation in these data structures takes O(log(n)) time. So there's an additional overhead over BFS and Dijkstra's algorithm takes O((n+m) log (n)) time. And this is the total runtime for Dijkstra's algorithm to find this distance array.

Now, there is one big limitation in Dijkstra's algorithm - it requires that all the edge weights are positive. Now, why does it require positive edge lengths? Well, because it might find this distance to b from s to be 5. But if there are negative edge weights, then it might find a shorter path to b. But once it outputs its distance, it doesn't recompute a shorter path - if it finds a new shorter path to b then it has to re-explore the edges out of that vertex again. So Dijkstra's algorithm is not guaranteed to produce the correct output when the edge weights are allowed to be negative.

We're going to look at this more general problem where the edge weights are allowed to be negative, such as in this example. And we're going to design a dynamic programming algorithm to solve the shortest path problem when the edge weights are allowed to be negative because Dijkstra's algorithm no longer is guaranteed to work.

(Slide 2) Negative weight cycles
***



Negative weight cycles

Given $\vec{G}$ with $w(e)$ & $s \in V$

- find a negative weight cycle

Else:
find $dist(z)$
for all $z \in V$

Negative wt. cycle:
$a \rightarrow e \rightarrow b$

What is $dist(d)$?

$s \rightarrow (b \rightarrow a \rightarrow e \rightarrow b) \rightarrow a \rightarrow d$

We're looking once again at a directed graph with edge weights where the edge weights are allowed to be positive or negative such as in this example. And for a designated start vertex s, we want to find the shortest path from s to every other vertex.

And our first question is whether this problem is well-defined. In this example, it is. We already looked at what the distance is from s to every other vertex. Let's modify this example a little bit. Let's make this negative weight a little bit more extreme. Let me change the length of this edge from a to e. From length, -2 and we'll make it -6.

Now, let's take a look at this example. And what is the length of the shortest path from s to d? Where previously it was length 12, because we went s to b to a to d. That has 5+3+4, so it's length, 12. But look at this cycle which goes b, a, e, b. What's the length of this cycle? It's length -1. So every time we go around this cycle, our length goes down by one. So an alternative route from s to d is to go s to b to a to e to b to a to d. So we're gonna go here. We're gonna go round the cycle and then follow this path. What's the length of this walk? It's length 11 because we went down by one by going around this cycle. Similarly, we can repeat this cycle many times. Every time we repeat it, our length goes down by -1. So the shortest route from s to d goes around this cycle infinite number of times. Now, this is a walk, it's not a path. Because we're allowed to repeat vertices in this case.

Now, the shortest path in this example is s to b to a to d. That's if we're only allowed to visit a vertex at most once. But if we're allowed to repeat vertices, I mean why not in this example? Then, we want to go around this cycle infinite number of times. This cycle is a negative weight cycle. Cycle going from a to e to b and back to a … a to e to b and back to a. The sum of the weights along this cycle is less than zero. So it's a negative weight cycle (which we define it as).

Now when a graph has a negative weight cycle, then the shortest path problem is not well-defined any longer. But if it has such a negative weight cycle, then it's interesting to find such a negative weight cycle. So let's change the problem.

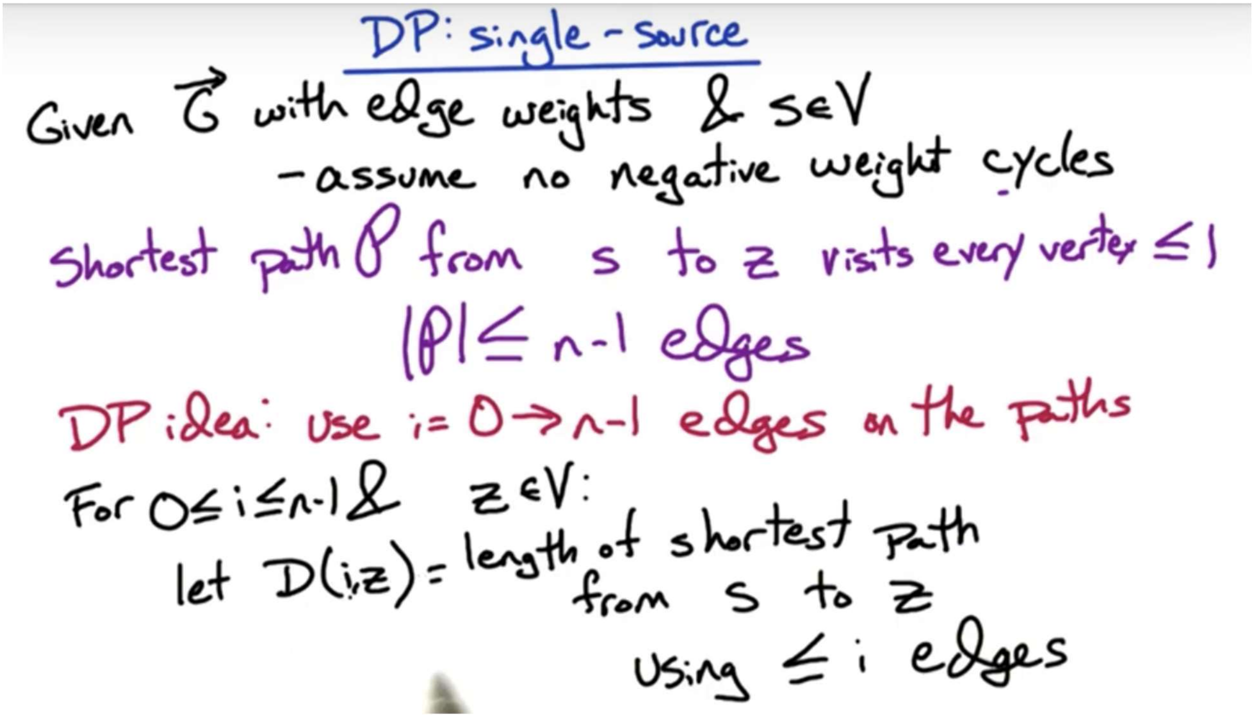
Let's look at the more general problem. As given, a directed graph with edge weights, let's find a negative weight cycle if one exists in the graph. What if there is no negative weight cycle? Well then, the shortest path problem is well defined and we'll solve the shortest path problem.

So we're going to solve this more general problem. We're giving a directed graph G with edge weights and the edge weights can be allowed to be positive or negative. And we're also given this designated start vertex s. And our goal is to find a negative weight cycle if one exists in the graph. Actually, to be more precise, we're going to find a negative weight cycle which is reachable from s. If there's a negative weight cycle in the graph but it's not reachable from s, then it doesn't have play any role in these distance vectors.

So we're going to find a negative weight cycle if one exists in the graph. And what if there's no negative weight cycle in this graph? Then for every vertex in the graph, we're going to find its distance. The shortest path length from s to this vertex. So we're going to output this array of size n. And let's look at how to use dynamic programming to solve this problem.

(Slide 3) Single Source: Subproblem

***



DP: single-source

Given $\vec{G}$ with edge weights & $s \in V$

- assume no negative weight cycles

Shortest path P from s to z visits every vertex ≤ 1

$|P| \le n-1$ edges

DP idea: use $i = 0 \rightarrow n-1$ edges on the paths

For $0 \le i \le n-1$ & $z \in V$:

let $D(i,z) =$ length of shortest path from s to z using $\le i$ edges

So let's design a dynamic programming algorithm for the single source shortest path problem. We're given as input a graph and it's a directed graph. And the edges of the graph are weighted, and they are weighted arbitrarily some can be positive, some can be negative, so we can no longer apply Dijkstra's algorithm. And we have some specified start vertex s, and our goal is to find the shortest path from s to every other vertex.

Now what about negative weight cycles? Let's assume for now that there are no negative weight cycles in the graph and therefore, the shortest path length from s to every other vertex is well-defined. We're going to visit every vertex at most once. We'll see how to solve this problem and then we'll see a slight tweak of the algorithm will detect whether there exists a negative weight cycle or not. But for now, let's assume there's no negative weight cycles in the graph. The problem is well-defined.

Since there are no negative weight cycles in the graph, as we just noticed, the shortest path from the start vertex s to any other particular vertex Z visits every vertex at most once. There's no reason to repeat a vertex because the cycles all have positive length. So let P denote the path - the particular path which is of shortest length from s to z. If there's multiple paths, let P be any particular one of shortest length. Since we visit every vertex at most once, what do we know about the length of P? P contains at most n-1 edges because we visit every vertex at most once.

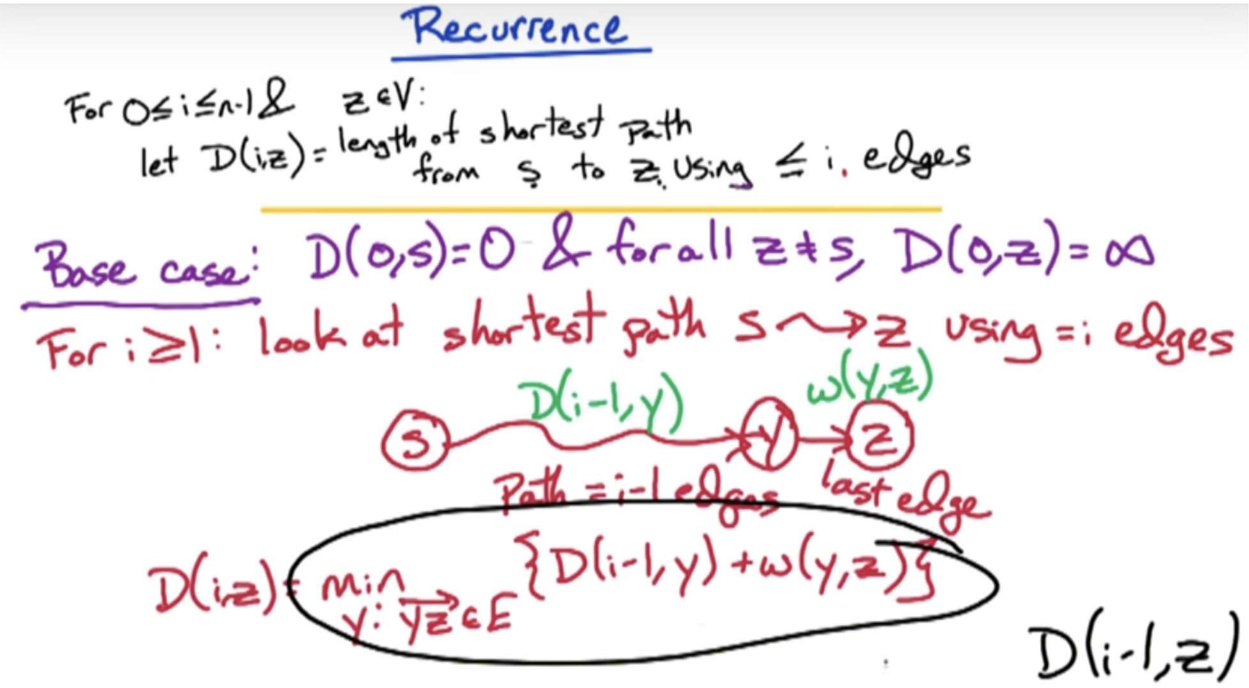
Now let's try to design a dynamic programming algorithm for this single source shortest path problem. Normally we try to use a prefix of the input in our dynamic programming algorithm. Here, it's going to be a little different type of solution. Notice that the path length is at most n-1 edges. Let's try to use a prefix of the path. … What do we mean by that? Let's try to condition on the number of edges in the path. What we're going to do is to introduce a variable i which is going to vary from 0 to n-1. And this is going to be the number of edges that we allow on the path that we consider. When i=n-1, then we're going to allow the path to be of length at most n-1 edges and that's going to solve the shortest path problem. It's going to be the final solution.

At the beginning, the base case i=0 - we don't allow any edges.

Let's more formally define our subproblem for the dynamic programming algorithm. We're going to have two parameters, i and z. i is going to go between 0 and n-1 and z is a vertex of the graph. And we're going to find the function D(i,z). This is going to denote the length of the shortest path from s to z, but we only consider paths which use at most i edges. So when i=n-1, then this is the final solution that we're looking for and we're going to build up our solutions starting from i=0 and building it up to n-1.

So now let's try to write a recurrence for D(i,z). Our goal is to express D(i,z) in terms of D(i-1,) (Slide 4) Single Source: Recurrence

***



Recurrence

For $0 \le i \le n-1$ & $z \in V$:

let $D(i,z)$ = length of shortest path from s to z using $\le$ i edges

Base case: $D(0,s) = 0$ & for all $z \ne s$, $D(0,z) = \infty$

For $i \ge 1$: look at shortest path $s \rightsquigarrow z$ using = i edges

$$D(i,z) = \min_{y: \vec{yz} \in E} \{D(i-1,y) + w(y,z)\}$$

$D(i-1,z)$

Recall our subproblem definition from the previous slide. For each i between 0 and n-1 and each vertex z, we define D(i,z) as the length of the shortest path from s to that chosen vertex z, where we only consider paths which use the most i edges. The final solution that we're trying to find is when i=n-1.

Therefore, the base case is i=0. In this case, we're going from s and we're not allowed to use any edges. So, the only vertex we can reach is s itself and, therefore, D(0,s) = 0. The length to go from s to itself is length 0, and the length to go to any other vertex is infinite. So, for every other vertex except s, the length is infinite. That defines the base case.

Now, let's try to do the case when i is at least 1. Now, let's look at the shortest path from s to z using exactly i edges. So, this path starts at s and ends at z. There's some penultimate vertex, let's call it y. So, there's a path of length i-1 from s to y. And then there's an edge from y to z. So, y to z is the last edge on this path and the prefix of the path is of length i-1 since the entire path is length i.

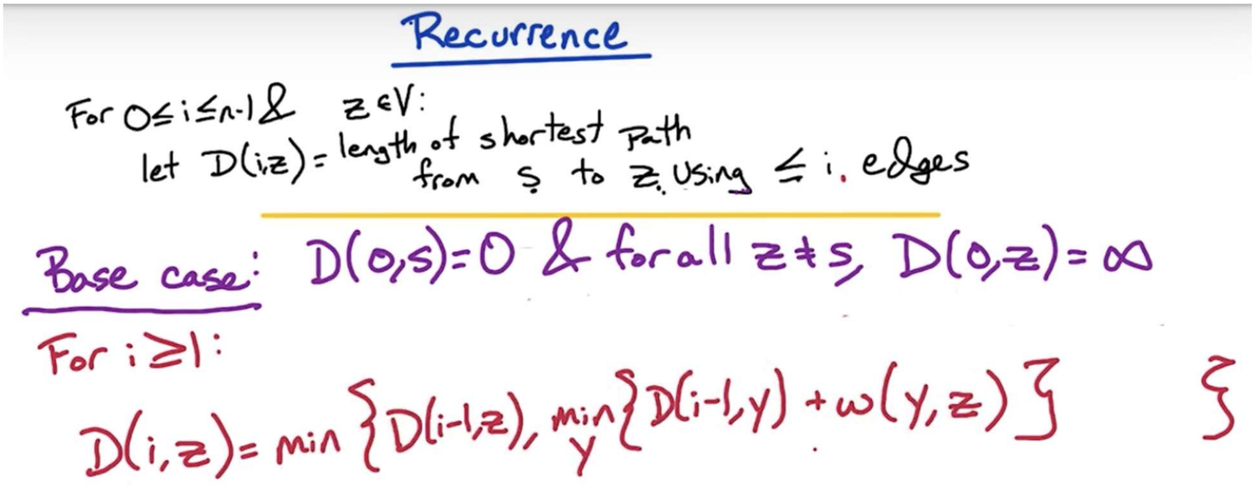
Now, notice our subproblem is slightly different from what we proposed before. Now, we're talking about exactly i edges, whereas, before we were talking at most i edges. Let's just look at this version for now. Can we write a recurrence now for D(i,z)? What we're going to do is we're going to try all possibilities for the last - the penultimate vertex on the path - and we're going to take the best of those choices for y. What is the best? The best means shortest, which means minimum. So, we're going to minimize over the choices of y.

What are the choices for y? Well, y has to have an edge to z. This is a directed graph, so we're considering those y where the directed edge from y to z … in our input graph. What is the length of this path which goes through y? Well, from s to y it's allowed to use exactly i-1 edges. Therefore, it is D(i-1,y) because it's the shortest path using exactly i-1 edges from s to y. Notice we're talking about exactly i-1. And again, slightly different from our original version, which said "at most i". We're just going to ignore that slight difference for a moment. And in addition, we get the weight of this last edge from y to z. What is the length of this last match? It's w(y,z). That's the length of this edge from y to z. So, the length of this shortest path from s to z, which goes through y - as the penultimate vertex - has length D(i-1,y) for this first i-1 edges, as captured right here, plus the length of this last edge from y to z. And we're going to take that sum, which is the total length of this path from s to z going through y and we have minimized over the choices of y. And the best choice of y is going to give us this value, D(i,z).

This gives us a valid recurrence when the subproblem is defined as the length of the shortest path from s to z using exactly i edges. Notice, in this case when we're looking for dist(z), this is not necessarily the shortest path using i=n-1. We might not necessarily use n-1 edges in the shortest path from s to z. So, this is not stored in D(n-1,z). Instead, to find dist(z), we've got to look at the min over all choices of i. So, to avoid this, we, instead, want to store D(i,z) as the length of the shortest path using at most i-1 edges, instead of exactly i edges.

Let's look. Is there a simple way to modify a recurrence so that it changes the subproblem definition to our original proposal? - using at most i edges instead of exactly i edges? All we have to do is we have to modify it so that we take the best of *this* (min{D(i-1,y)+w(y,z)}) solution, which minimizes over y, and the previous solution, which was D(i-1,z). So, if this stores the length of the shortest path from s to z using at most i-1 edges, this gives the length of the shortest path using exactly i edges. If we take the min of these two values, then that will give us the length using at most i edges.

(Slide 5) Single Source Summary
***



Recurrence

For $0 \le i \le n-1$ & $z \in V$:
let $D(i,z)$ = length of shortest path from s to z using $\le i$ edges

Base case: $D(0,s)=0$ & for all $z \ne s$, $D(0,z)=\infty$

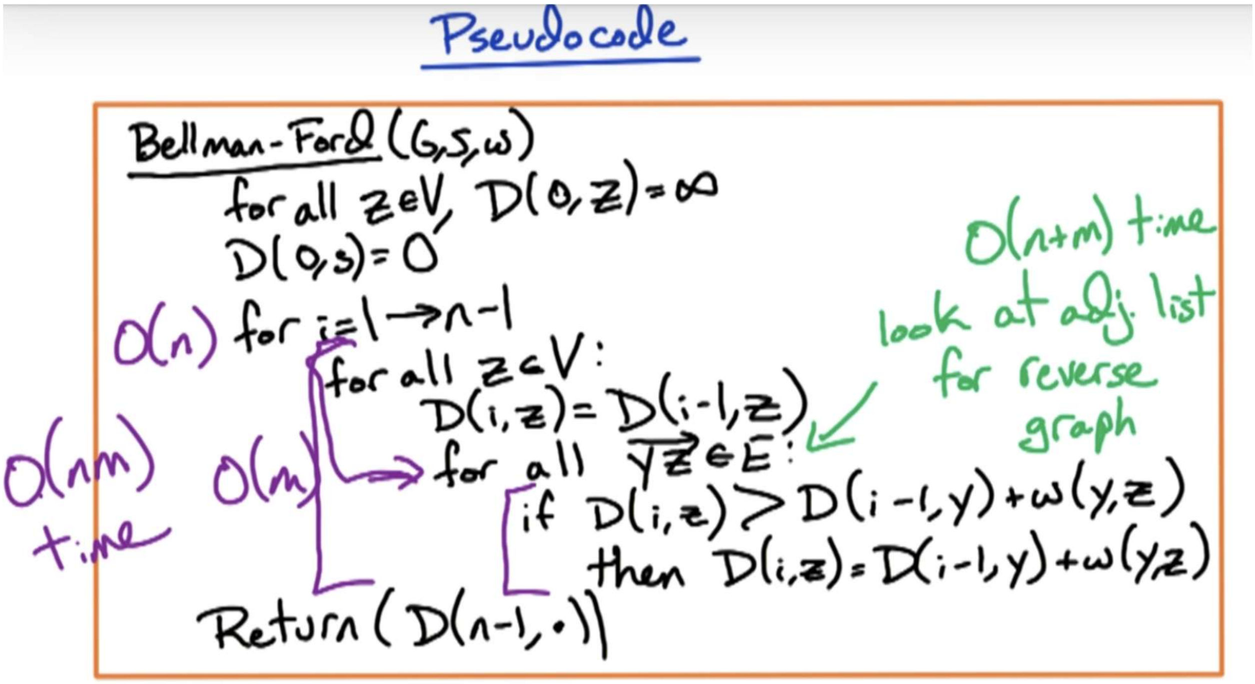
For $i \ge 1$:

$$D(i,z) = \min\left\{ D(i-1,z), \min_{y}\{ D(i-1,y) + w(y,z) \} \right\}$$

Let's recap and summarize our recurrence. We're looking the length of the shortest path from s to z using at most i edges and we're considering the case where i is at least one. When i=0, we have our base case. Now either the shortest path from s to z uses at most i-1 edges or it uses exactly i edges. And we're going to take the best of those two scenarios. If it uses at most i-1 edges, then the solution is in D(i-1,z). If it uses exactly i edges, then we're going to try all choices for the penultimate vertex y and then we're going to take the length of the shortest path from s to y using at most i-1 edges plus the length of the last edge. So we have a double min. - D(i,z) = min {D(i-1,z), min { D(i-1,y) + w(y,z)}} - we take the min of the i-1 case and the min over y, choices for y. And this defines a recurrence for D(i,z).

Notice, in order to get this D(i,z), we're using entries which involve i-1. So if we build up the table from i going from 0, which is the base case, to n-1, we'll have a valid dynamic programming algorithm. So let's go ahead and detail that algorithm.

(Slide 6) Single Source Pseudocode
***



Pseudocode

```
Bellman-Ford$(G,s,w)$
    for all $z \in V$, $D(0,z)=\infty$
    $D(0,s)=0$
    for $i=1 \rightarrow n-1$
        for all $z \in V$:
            $D(i,z)=D(i-1,z)$
            for all $\vec{yz} \in E$:
                if $D(i,z) > D(i-1,y)+w(y,z)$
                    then $D(i,z)=D(i-1,y)+w(y,z)$
    Return$(D(n-1,\cdot))$
```

$O(n+m)$ time
look at adj. list for reverse graph

$O(n)$

$O(m)$

$O(nm)$ time

Now, let's just go ahead and detail the pseudocode for our dynamic programming algorithm for the shortest path, single source shortest path problem. This algorithm is called the Bellman-Ford algorithm.

The input to the algorithm is a directed graph G, a star vertex s, and weights on the edges. This algorithm goes back to the 50s. It was devised by Richard Bellman and Lester Ford. Now one interesting note is that Richard Bellman who devised this algorithm - he's actually the one who developed the dynamic programming approach in the 1940s - long before there was any programming on personal computers or anything like that.

So let's start with the base case, which is the case i=0. And we're going to initialize D(0,s) = 0 so going from S to itself using in most zero edges that's going to be of length 0.

Now, we're going to work up from i=1 up to i = n -1. We're going to go over the vertices of the graph in V. We're going to initialize D(i, z) = D(i-1,z) - this is the same as saying, that if we look at the minimum length path using the most i edges, we're going to first consider using the most i-1 edges and then we'll look at the scenario using i edges. To consider the scenario where we use i edges, we look at all choices for the penultimate vertex with this y. And we're considering those y's which have an edge from y to z. Now we look at the path using y and whether that path using y is better than the current best solution. The current best solution is stored in D(i,z) and the path through y has length D(i,-1,y) plus the length of this last edge W(y,z). If the length of this solution through y (D(i,-1,y) + W(y,z)) is better than the current best solution, then we're going to update the current solution.

Finally, what do we return? We return the case where i = n -1. And we're going to return it for all values of z - D(n-1,·) - I just put a dot to denote this array of size n. So D(n-1) is an array of size n or if you think of it as two-dimensional table we're returning the last row of the table.

Let's take a look at the algorithm a little bit more in detail. There's one slightly non-trivial step. Here, we're looking at the edges into z. Normally, when we have adjacency list we look at the edges out of z. So how do we get the edges into a vertex?

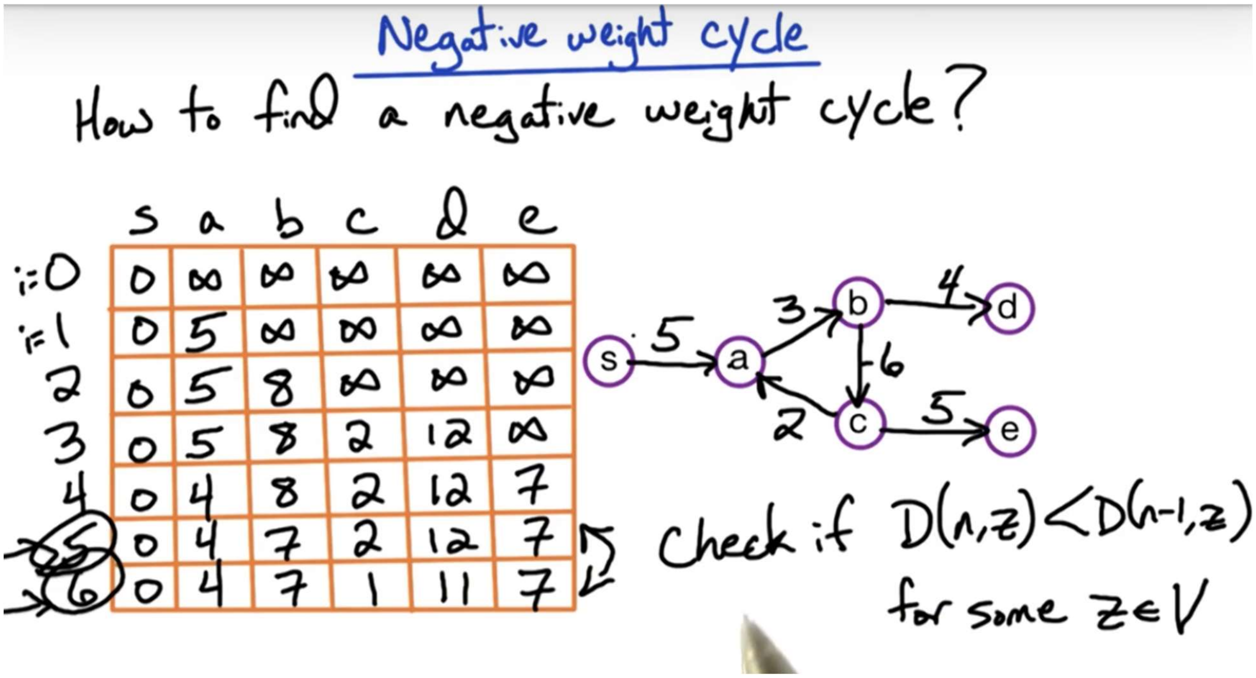
What we do is we look at the adjacency list for the reverse graph, so we flip all the edges. To construct this graph takes linear time O(n+m) time. And then we take the adjacency list for this reverse graph and that's going to give us the edges into z because in the reverse graph these are the edges out of z. The edges in the reverse graph which are out of z are the edges into z in the original graph.

Now, let's look at the running time with this algorithm. We have a for loop over i from one to n -1. This for loop is over O(n) choices. Then within this for loop, we're going over the vertices of the graph and then we go over the edges into each vertex. So, really the key step is right here, this **for loop**. How many choices are we going over here? We're going over all edges of the graph once. So this is O(m) choices here and for each edge of the graph, we do a simple if then statement which is O(1) time. So the total time for this for loop for this nested pair of for loops, we go over the vertices of the graph and then we go over the edges into that vertex – so, the nested for loops combined go over all edges of the graph exactly once. So, these inner four loops take O(m) time and the outer for loop takes O(n) time. So the total time is O(nm).

So this algorithm is actually slower than Dijkstra's algorithm but it allows negative weight edges. Also, in addition, it's going to allow us to find negative weight cycles.

So far, we've been assuming that there's no negative weight cycles in the graph so that the shortest path from s to every other vertex is well defined. Now let's go back and see if we can figure out whether there is a negative weight cycle in the graph or not and detect it. And if there is one output it.

(Slide 7) Finding Negative Weight Cycle
***



Negative weight cycle

How to find a negative weight cycle?

	s	a	b	c	d	e
i=0	0	∞	∞	∞	∞	∞
i=1	0	5	∞	∞	∞	∞
2	0	5	8	∞	∞	∞
3	0	5	8	2	12	∞
4	0	4	8	2	12	7
5	0	4	7	2	12	7
6	0	4	7	1	11	7

Check if $D(n,z) < D(n-1,z)$

for some $z \in V$

How can we find whether a graph has a negative way cycle or not? Let's go back and look at our earlier example. Here's our earlier example on six vertices and it has a negative weight Cycle a b c a, which is of length -1. So, what's going to happen for our algorithm - the Bellman Ford algorithm that we just defined - on this example?

Let's look at the Bellman Ford algorithm on this example. We're going to have a two dimensional table. The columns of the tables are going to be the vertices of the graph. There are six vertices of the graph, s, a, b, c, d, e. The rows of the table are going to correspond to the path lengths we consider.

We're going to start with the base case, i=0. In the base case we have D(0,s) = 0, and the other entries are infinite.

Our algorithm can fill up the table from i=1, i=2, i=3, i=4, i=5. The current algorithm is going to finish at i=5, which is n-1 in this example. But let's do one more row of the table and see what happens. Now, let's go ahead and fill in the table by hand. In this particular example, the column for s is going to stay 0, since there's no edges into S.

- Now let's look at the row i=1. There's a path using one edge from s to a that's of length 5. All of the vertices are inaccessible by at most one edge - so their length stays infinite.

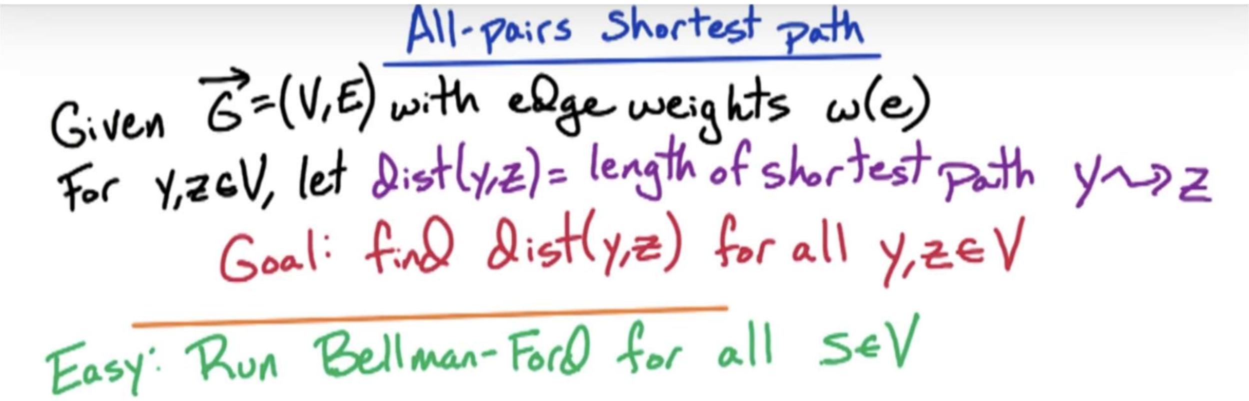
- Now for i=2, we update the length to b to be 8. The other vertices stay infinite and a stays 5.
- For i=3, we can update the path length to c, which is it going to be of length 2 using the path from b (-6). In addition, we can update the path length to d. The path to d is a plus 4, 12. And the other vertices are going to stay the same.
- Now, for i=4, we can update the path length to e going through c which is of length 2 plus 5. That gives us a path of length 7 to e.
  Now the other interesting thing is that there's a path to a itself. So we can go to c of length 2 and then we can go back along this edge from c to a, which is of length 2. And then we get a new path to a going along this cycle, which is of length 4. So that's a shorter path to a. The path length to a went down from 5 to 4 using this cycle as negative weight cycle. And the other vertices stay the same.
- Now, let's continue. Notice, now we can update the path length to b, because a's distance went down from 5 to 4. So now the path length to b using a goes down to 7. And the other vertices stay the same. Now since the path length to b went down, actually, we can update the path length to d and also to c. The path length to c now goes down to 1 and the path length to d goes down to 11.

Notice that when there is a negative weight cycle, then every row is going to be different from the previous row. It's going to be the next step on this cycle. And that next vertex on that cycle is going to have shorter path length than it previously had. So a is going to decrease, and then b is going to decrease, and then c is going to decrease, and then a is going to decrease, then b and c and a and so on. So every row is going to be different. So normally if there's no negative weight cycle, then notice our algorithm was going to stop at the case i=n-1, which is i=5 in this example, because every path used a vertex at most once. So the maximum path length was at most five edges (n-1 edges), but if there's a negative weight cycle then what we're going to have is that i=n is going to be different. It's going to be smaller than i=n-1, because every row is going to shrink further. So if there's no negative weight cycle and we can stop here. But if there is a negative weight cycle then what we're going to notice is that, i=n is going to be different from i=n-1.

So how do we detect a negative weight cycle? We compare these two rows and if i=n is different from i=n-1 then that shows us that there is a negative weight cycle. So how do we check that there's a negative weight cycle? We check if the entry D(n,z) is smaller, strictly smaller, than D(n-1,z), for some z, some vertex z. So the row i=n is different from the row i=n-1 for some entry z. If that is the case then there is a negative weight cycle and it involves z. And, actually, we can backtrack, we can see that that cycle involved is vertex c and then we can see it involves b and a. So we can detect that cycle, c, b, a. But our check is just we take our algorithm from before, our Bellman Ford algorithm, which ran from i=1 to n-1 and instead we run it from i equals 1 to n and we check if the row i=n is different from i=n-1. If it is different, then we found a negative weight cycle. If it's not different, then we output the row i-n-1 or the row i=n, because they're both the same and that gives us the shortest path length from s to every other vertex.

So that completes the dynamic programming algorithm known as Bellman Ford for finding the shortest path from a single source vertex and it allows positive and negative weight edges. And if there is negative weight edges, it can detect whether or not there is a negative weight cycle.

(Slide 8) All Pairs Shortest Path
***



All-pairs Shortest path

Given $\vec{G}=(V,E)$ with edge weights $w(e)$

For $y,z \in V$, let Dist(y,z) = length of shortest path $y \rightsquigarrow z$

Goal: find Dist(y,z) for all $y,z \in V$

Easy: Run Bellman-Ford for all $s \in V$

Let's look at one more variant of shortest path problem and this will give us a chance to look at a slightly different style of dynamic programming solution.

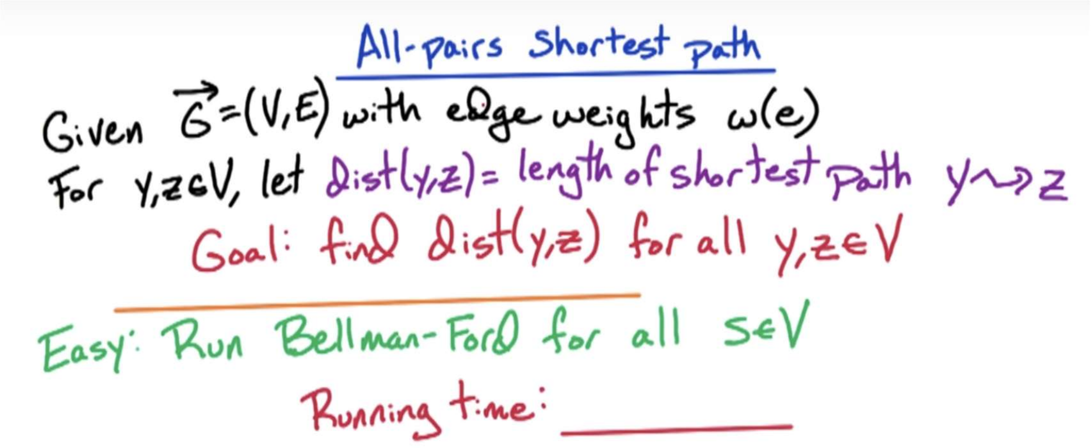
So what we did before with Bellman-Ford was we had a single source and we looked at the shortest path from that single source to all other vertices. Now we're going to look at all pairs shortest path. Once again we're given a directed graph, G, along with edge weights. And these edge weights again can be positive or negative. Now, for a pair of vertices y and z, let's define dist(y, z) to be the length of the shortest path from y to z.

Previously, we are always looking at the shortest path from a single start vertex s to the given vertex. Now we're looking at for all pairs of vertices, we're looking at the shortest path. So we have this n^2 size matrix dist. Our goal in this problem is to find the length of the shortest path between from y to z and we want to do it for all pairs of vertices y and z. So for all n^2 vertices, we want to find the distance from y to z.

Now last lecture, we saw Bellman-Ford algorithm, which did the distance from a single start vertex s to all other vertices. Can we use that as an easy algorithm to solve this all pairs shortest path problem? Yes! - so the easy approach to solve this all pairs shortest path problem is to run Bellman-Ford algorithm which has a single start vertex and we try all n possibilities for the start vertex. So we run Bellman-Ford n times.

(1st Slide 9) Naïve Approach

***



All-pairs Shortest path

Given $\vec{G}=(V,E)$ with edge weights $w(e)$

For $y,z \in V$, let $Dist(y,z)$ = length of shortest path $y \rightsquigarrow z$

Goal: find $dist(y,z)$ for all $y,z \in V$

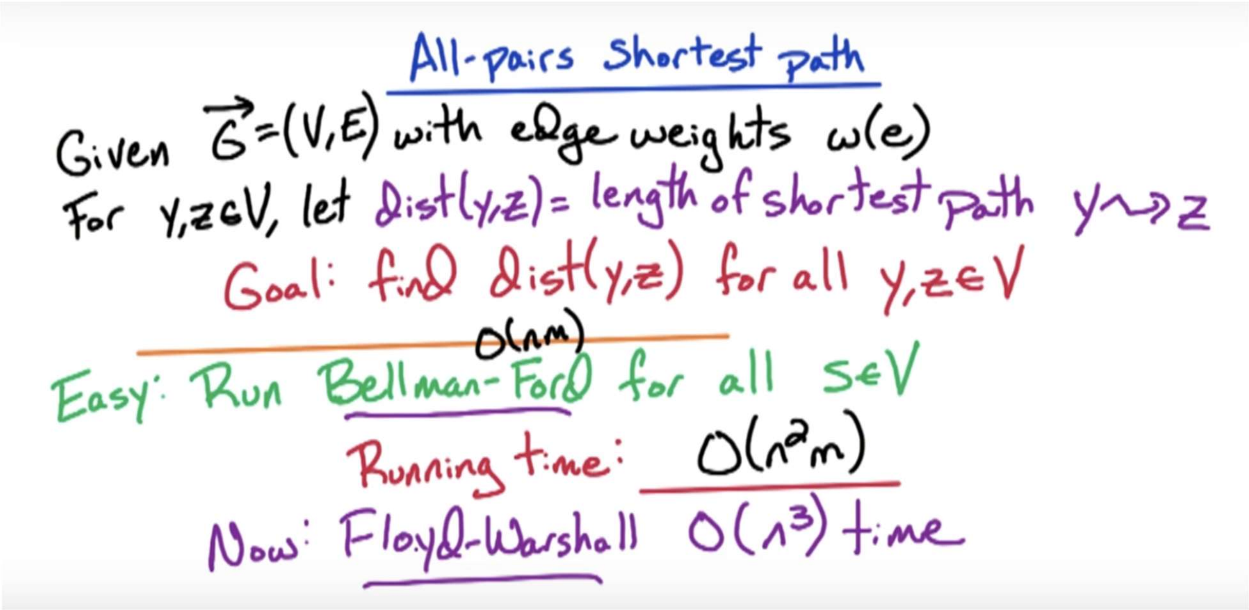
Easy: Run Bellman-Ford for all $s \in V$

Running time: ________

Let's pause for a quick quiz. So what was the running time of the Bellman-Ford algorithm? And now if we run Bellman-Ford algorithm N times, so we're running it for all N possible start vertices, what's the running time of this approach for the all pairs shortest path problem?

(2nd Slide 9) Naïve Approach (Answer)

***



All-pairs Shortest path

Given $\vec{G}=(V,E)$ with edge weights $w(e)$

For $y,z \in V$, let $Dist(y,z)$ = length of shortest path $y \rightsquigarrow z$

Goal: find $dist(y,z)$ for all $y,z \in V$

Easy: Run $\overset{O(nm)}{\text{Bellman-Ford}}$ for all $s \in V$

Running time: $O(n^2m)$

Now: Floyd-Warshall $O(n^3)$ time

For Bellman-Ford, first off, takes O(nm) time for each run.

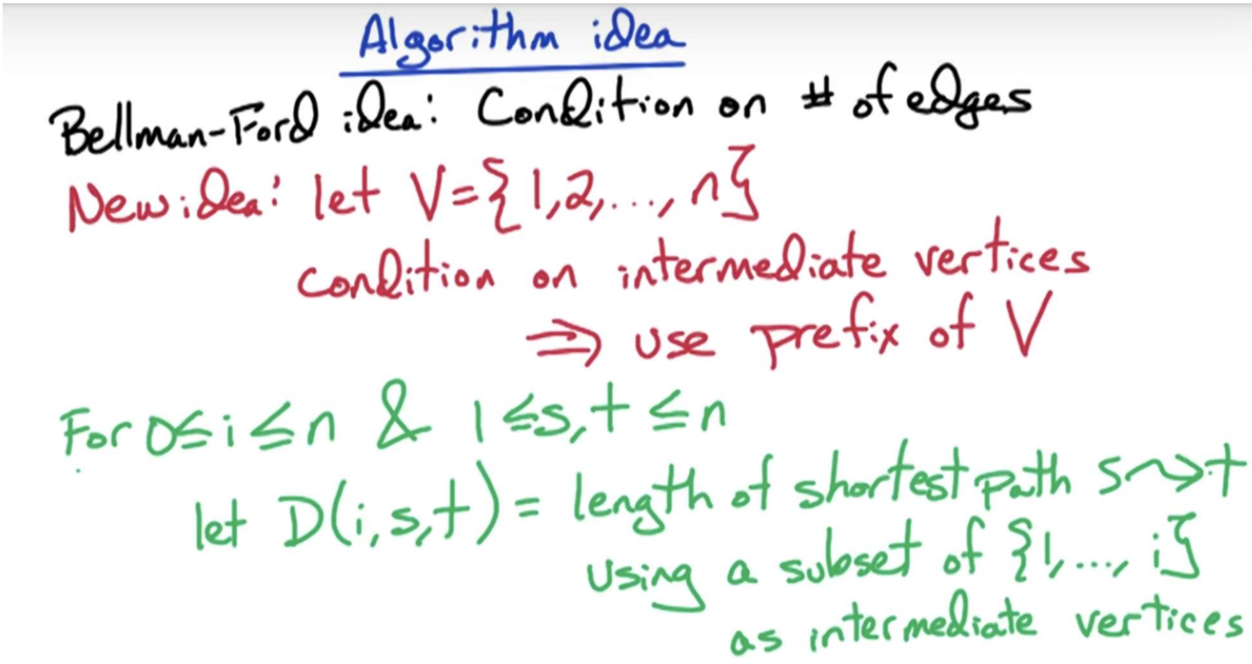
Now, I'm terrible at memorizing these things, so actually, the name itself, Bellman-Ford, is not important. What's important is we had the single source shortest path algorithm using dynamic programming. And if you want to reconstruct the running time in your head, then what you can do is you can try to remember how it worked, the basic idea. So the basic idea was we conditioned on the number of edges in the path. So we went and we had this variable i which went from zero to n-1. So we had a for loop over i, which had n choices. Okay, that's O(n). And then in each step, we conditioned on the last edge in the path. So we had a for loop over the edges of the graph, and that gave us the m factor. So the total run time was O(nm).

And now, we're running this algorithm n times, so the total run time is going to be O(n^2 m).

What we're going to do now is the direct algorithm for this problem, and this algorithm is called Floyd-Warshall algorithm, and the running time is going to be O(n^3). So it's better than *this* (O(n^2 m)) because in this case, m the number of edges of the graph can be up to n^2. And if the graph is connected, it's going to be at least n-1. So, *this* is possibly n^4, whereas *this* is n^3.

Now, just to clarify, I'm not expecting you to memorize the names of these algorithms. What I want you to do is to understand the basic approach in the two algorithms and then using that basic high level intuition, you can reconstruct the running time for these algorithms, and if you need be with enough sufficient time you can reconstruct the actual algorithms, okay? But I myself get confused all the time, which one is Bellman-Ford, which one is Floyd-Warshall, that's not the important aspect for here. I want you to understand the high level idea, and I'm illustrating these algorithms because they both have some nice dynamic programming approaches which are slightly different than the approaches that we've seen in the past. So now let's dive into this Floyd-Warshall algorithm and see how we get this O(n^3).

(Slide 10) All Pairs: Subproblem
***



Algorithm idea

Bellman-Ford idea: Condition on # of edges

New idea: let $V=\{1,2,\dots,n\}$

condition on intermediate vertices

$\Rightarrow$ use prefix of V

For $0\le i\le n$ & $1\le s,t\le n$

let $D(i,s,t)$ = length of shortest path $s\rightsquigarrow t$

using a subset of $\{1,\dots,i\}$

as intermediate vertices

Let's look at the basic idea for this dynamic programming algorithm we're going to construct.

First, let's go back and look at the Bellman-Ford algorithm … the basic idea of this algorithm. This was the algorithm where we had a single start vertex and we're looking at the shortest path from that single start vertex, s, to all other vertices. Now, in that dynamic programming algorithm, what we did is we conditioned on the number of edges in length of the path.

Obviously, we're going to try to do something different here. What else can we condition on instead of the number of edges? Actually, if you think of our basic dynamic programming approach that we always try…we try the prefix of the input. Okay?   What is the input here? The input here is the graph, or, one important aspect is the vertices of the graph. So, can we try a prefix of the vertices of the graph? Let's try to formalize that.

First off, let's order the vertices, 1 through n. So, it’s just assigning numbers to the vertices: 1, 2, …up to n. How we do that doesn't matter. Okay?   Just the point is that now we can index the vertices by numbers 1 through n and we're often doing this anyways … implicitly, because we have a table where we have a single one dimensional array for the vertices and we're indexing them by their numbers. Okay? The important thing now is that now we can look at a prefix of the vertices. So, we want to solve the same all pairs shortest path problem where we only allow a prefix of the vertices to be used. So, we're going to condition on the intermediate vertices that are allowed to be used in the past that we consider. And we're going to go back and use prefixes as we've done them in many problems and we're going to use prefixes of the vertex then, okay?

So, let's formalize this more precisely. So, we're going to have three parameters:

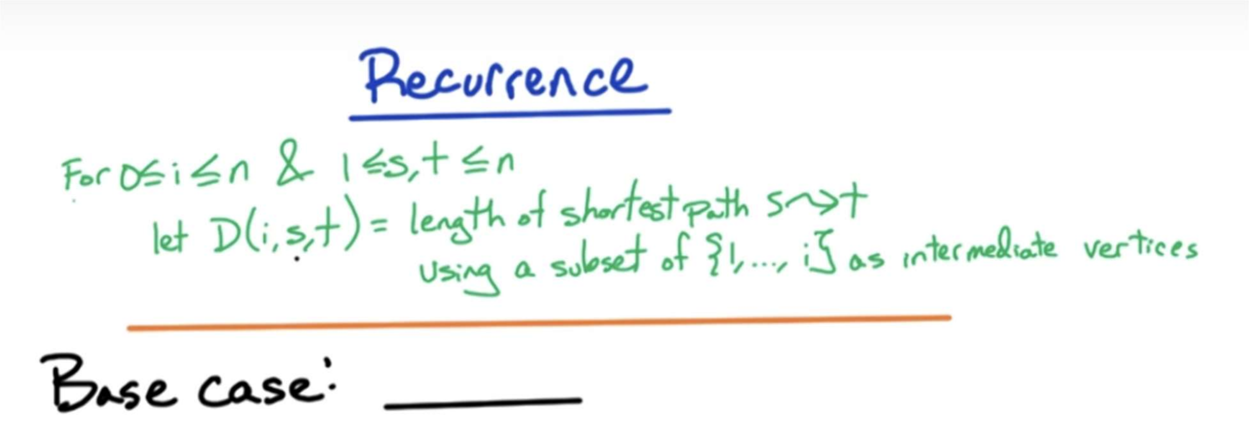
- First parameter is going to be the prefix of the vertex set that we consider. So, we're going to use the variable i for the prefix of the vertex set that we consider. This prefix is going to vary from 0, for the empty set, up to n. And for given i, we're going to consider the set of intermediate vertices 1 through i. That's going to be the set of allowable vertices to be used as intermediate vertices on the paths that we consider.
- Now, the other two parameters that we need are the start vertex and the end vertex. And we want to do all pairs shortest path. So we want to find - try all possible start vertices and all possible end vertices. So let's use s for the start vertex and let's use t for the end vertex. And now, we want to vary s and t over all possible choices. So, we want to try each 1 through n. So s and t both vary between 1 and n and we want to try all n-squared choices for s and t.

Finally, let's define our dynamic programming subproblem in words. It's going to be a 3-dimensional table, now - D(i,s,t) - it's going to be the length of the shortest path from s to t.

And, now, how does i work in? Well, we're going to condition on the set of intermediate vertices that are allowed to be used. Now, the perimeter i tells us the prefix of the vertex set which are allowed to be used as intermediate vertices. So that we only consider paths from s to t, where a subset, possibly empty set, or possibly the whole thing …and we're not saying anything about the ordering of these intermediate vertices … but the only intermediate vertices that can be used on the path from s to t are 1 through i. And we're going to vary i from zero to n. When i=n, that means we allow all vertices to be intermediate vertices on the path, so we're considering all paths.

So then D(n, s, t), tells us the length of the shortest path from s to t. So, now, let's go ahead and try to find a recurrence for this subproblem definition that we just defined.

(Slide 11) All Pairs: Base Case
***



Recurrence

For $0 \le i \le n$ & $1 \le s,t \le n$

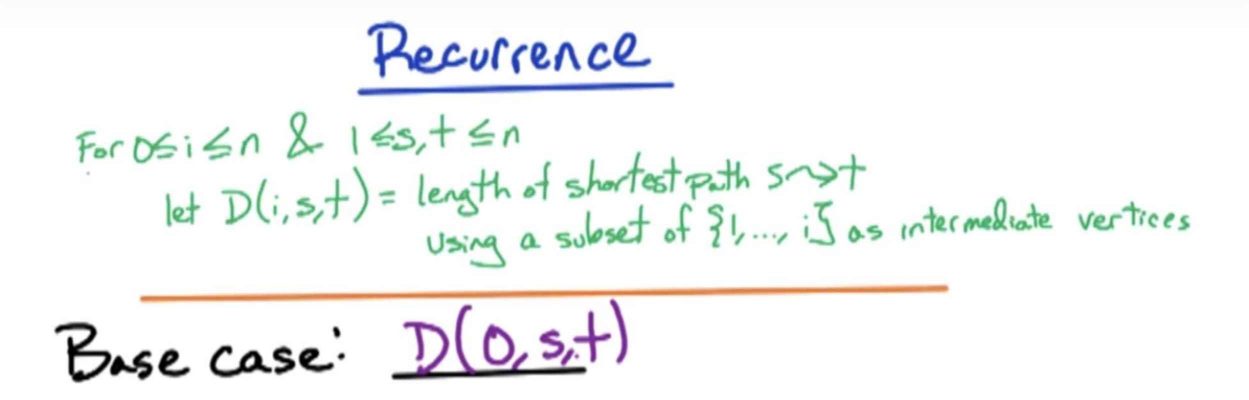
let $D(i,s,t)$ = length of shortest path $s \rightsquigarrow t$ using a subset of $\{1,\ldots,i\}$ as intermediate vertices

Base case: ______

Let's go ahead and try to find the recurrence for the subproblem we just defined. The subproblem we just defined was D(i,s,t). This is the length of the shortest path from vertex s to vertex t, where the set of allowed vertices on the path from s to t are a prefix of the vertex n. These are the first i vertices, 1 through i. These are the vertices that are allowed to be intermediate vertices on the path from s to t.

Let's start with the base case. Why don't you go ahead and write down what is the base case and what is the solution to the base case? For intuition about the solution for the base case and what is the base case itself, think back to the Bellman-Ford algorithm for the single source problem.

(Slide 11) All Pairs: Base Case (Answer)
***



Recurrence

For $0 \le i \le n$ & $1 \le s,t \le n$

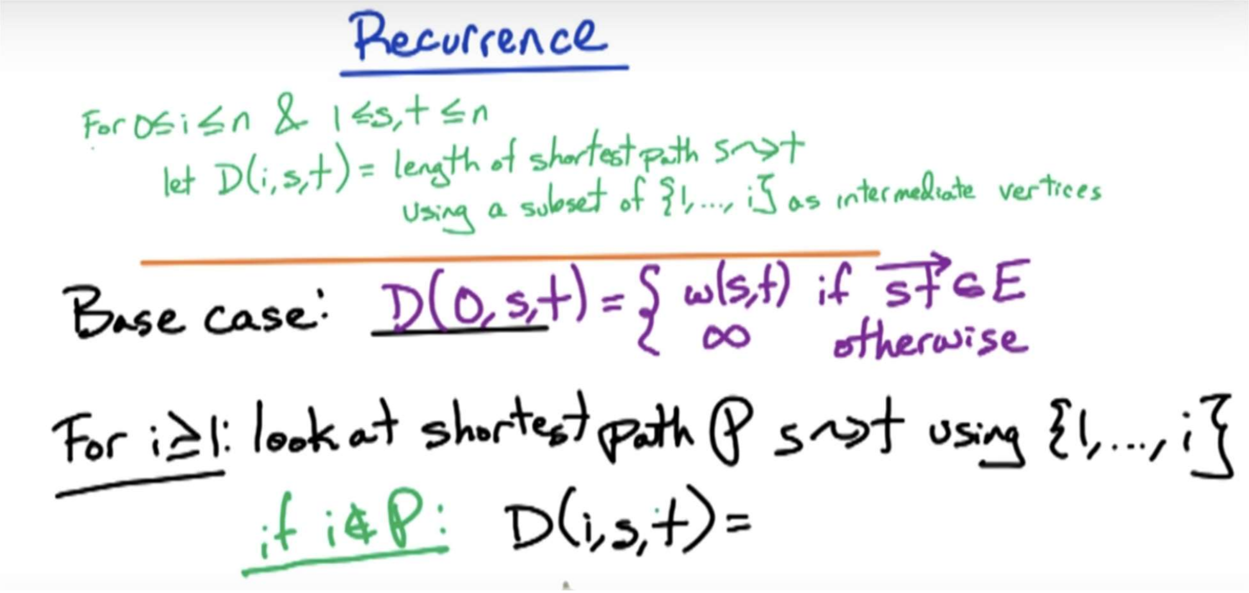
let $D(i,s,t)$ = length of shortest path $s \rightsquigarrow t$ using a subset of $\{1,\ldots,i\}$ as intermediate vertices

Base case: $D(0,s,t)$

As in the Bellman-Ford algorithm, the base case is the situation when i=0. That's when we have the empty set as possible intermediate vertices. So we have to go directly from s to t without any intermediate vertices allowed.

(Slide 12) All Pairs: Recurrence

***



Recurrence

For $0 \le i \le n$ & $1 \le s,t \le n$

let $D(i,s,t)$ = length of shortest path $s \rightsquigarrow t$ using a subset of $\{1,\dots,i\}$ as intermediate vertices

Base case: $D(0,s,t) = \begin{cases} w(s,t) & \text{if } \vec{st} \in E \\ \infty & \text{otherwise} \end{cases}$

For $i \ge 1$: look at shortest path P $s \rightsquigarrow t$ using $\{1,\dots,i\}$

if $i \notin P$: $D(i,s,t) =$

So we're going to have two situations, either s and t are connected by an edge and therefore, we don't need to use any intermediate vertices, or they're not connected by an edge and then there's no path from S to T.

In the first case, if there's a directed edge from s to t, then D(0,s,t) is exactly the length of that edge from s to t which is w(s,t).

In the other scenario - there's no edge from s to t. In this case, then what is the length of the shortest path from S to T which doesn't allow any intermediate vertices?   It's infinite.   Let's look at the shortest path from s to t where we only allow vertices 1 through i as intermediate vertices. So this is the solution. This path P is the solution to this problem. And if there's multiple paths, just choose any path arbitrarily.

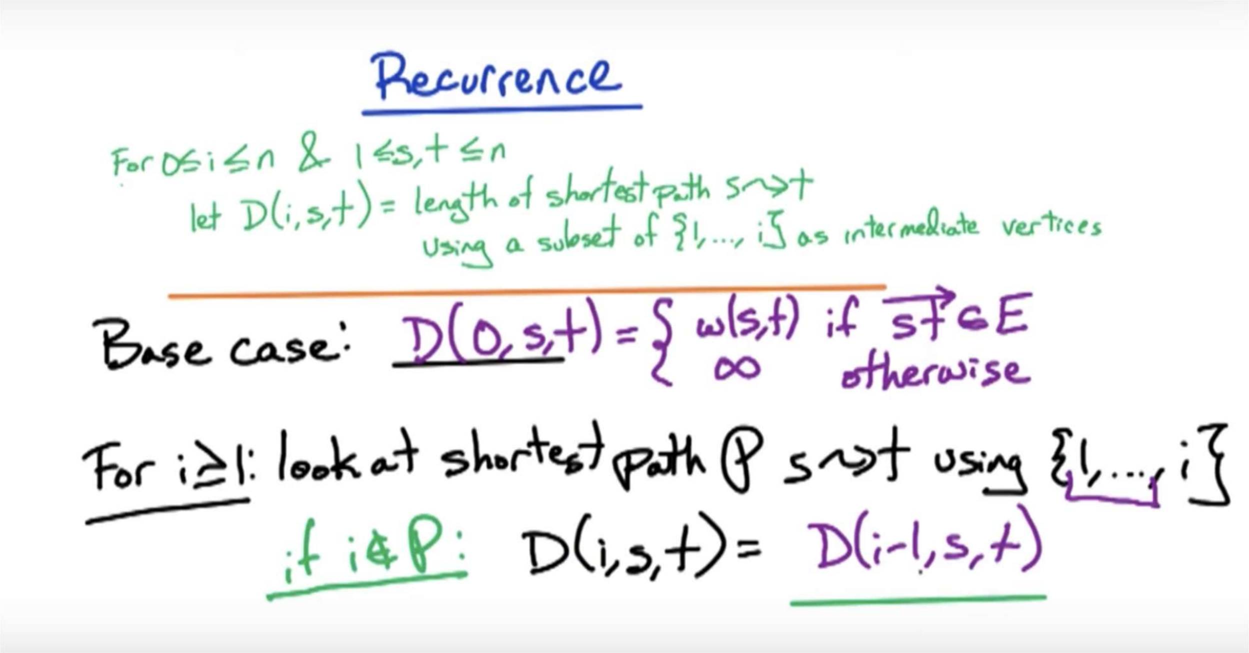
So our goal is to figure out D(i,s,t), and we want to write the recurrence for this so we want to do this for D(i,s,t) in terms of smaller i’s. Okay?   So we're going to have two cases.

We're considering this vertex as last vertex i. So what are the two cases for this path? Either this path uses vertex i or it doesn't use vertex i. Let's try the two cases separately. So what's the easy case? The easy case is when the path doesn't include vertex i and then the slightly harder cases when the path includes vertex i.

Let's start with the case where vertex i is not on the path and in this case, what is D(i,s,t)?

(1st Slide 13) Case: i not on path
***
Let's pause for a moment and you go ahead and think about what is the solution to the recurrence in this case, where vertex i is not on the shortest path from s to t. And so, can we express D(i,s,t) in terms of smaller subproblem?

(2nd Slide 13) Case: I not on path (Answer)
***



Recurrence

For $0 \le i \le n$ & $1 \le s,t \le n$
let $D(i,s,t)$ = length of shortest path $s \rightsquigarrow t$ using a subset of $\{1,\dots,i\}$ as intermediate vertices

Base case: $\underline{D(0,s,t)} = \begin{cases} w(s,t) & \text{if } \overrightarrow{st} \in E \\ \infty & \text{otherwise} \end{cases}$

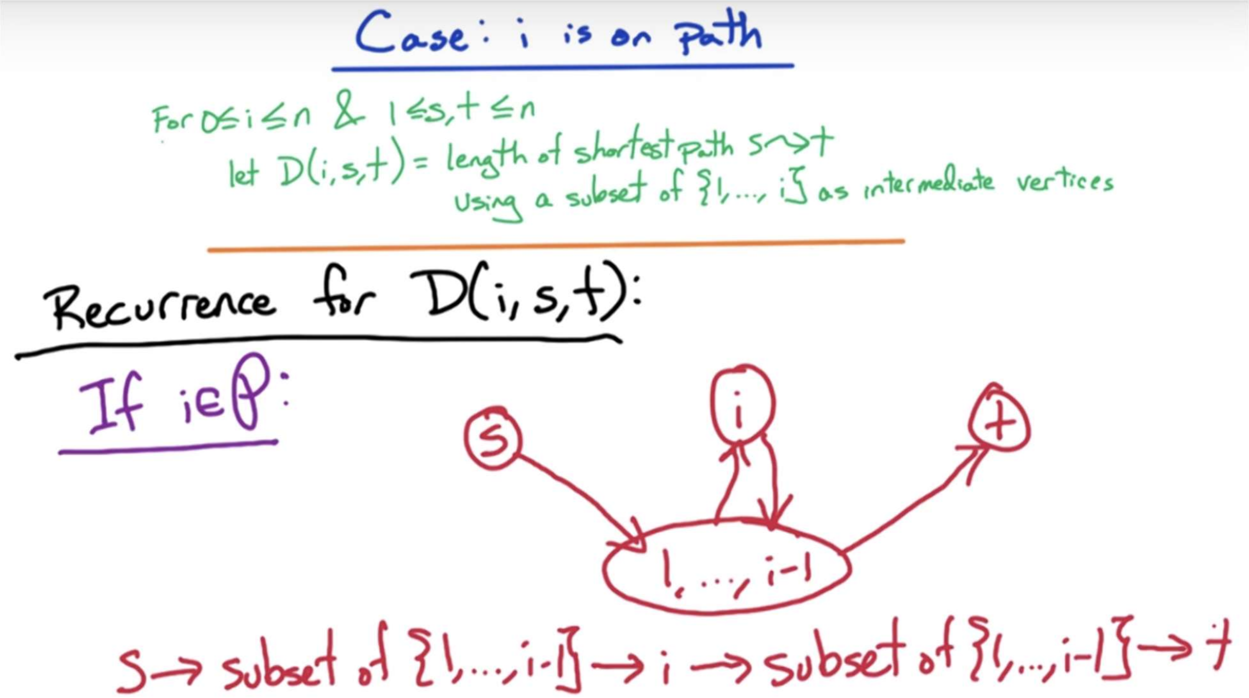
$\underline{\text{For } i \ge 1}$: look at shortest path P $s \rightsquigarrow t$ using $\{\underbrace{1,\dots,}\, i\}$

$\underline{\text{if } i \notin P}$: $D(i,s,t) = \underline{D(i-1,s,t)}$

If vertex i is not on the path P, then this path P only uses a subset of vertices, 1 through i-1, as intermediate vertices. Therefore, D(i,s,t), in this case, is the same as D(i-1,s,t) since we're only using a subset of the first i-1 vertices intermediate vertices. The solution is obtained by this subproblem.

The other case is when i is on this path.   Let's dive into this a little bit more carefully.

(Slide 14) Case: i is on path
***



Case: i is on path

For $0 \le i \le n$ & $1 \le s,t \le n$
let $D(i,s,t)$ = length of shortest path $s \rightsquigarrow t$
using a subset of $\{1,\dots,i\}$ as intermediate vertices

Recurrence for $D(i,s,t)$:

If $i \in P$:

$s \to$ subset of $\{1,\dots,i-1\} \to i \to$ subset of $\{1,\dots,i-1\} \to t$

So we try to write Recurrence for D(i,s,t). We did the case where the vertex i is not on the shortest path from s to t. And now we want to do the case where vertex i is on the path from s to t. And we're talking about shortest path which only used vertices 1 through i as intermediate vertices.

So, let's try to figure out the recurrence for this case where i is on the shortest path from s to t.

Let's look at our path graphically. We're starting at s, we're ending at t. We know, at some point in the middle, that we go through vertex i. What are other vertices that we could possibly use? We use a subset of vertices 1 through i as intermediate vertices. Vertex i is here, so the other vertices that we can possibly use are a subset of vertices 1 through i-1. We don't know anything about that subset, what order or which particular vertices in that subset are used but it's a subset of 1 through i-1.

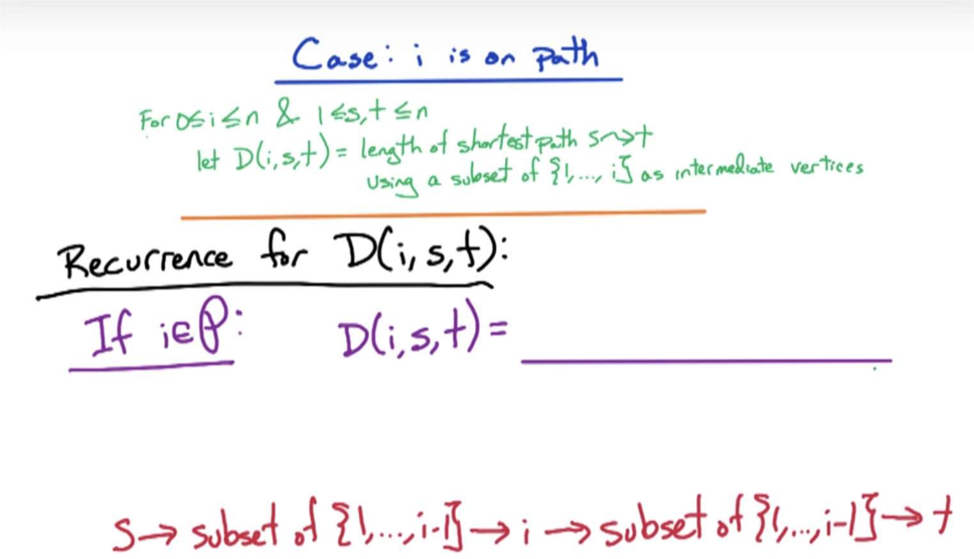
Now, what does this path look like? It starts at s and then it goes through some subset of vertices 1 through i-1. It might be the empty subset of these or might be all of them. We don't know. Just goes from s to some subset here then, at some point, it visits vertex i. After it visits vertex i, what happens? It goes to some subset of vertices 1 through i-1. Might be the empty subset, not sure. It might be the full subset. And then, afterwards what happens? It finishes at vertex t.

So our path can be broken up into these four segments. s to this subset … then it's going, moving around in that subset. It might be an empty subset so in which case it's actually skipping - it's going directly from s to i.   But it goes from s to this subset, towards around the subset, goes to i, back down, and then goes around the subset and possibly empty, visit, and then back to t. So we have these four segments of our path.

Let's rewrite this partition of the path into these four parts in words. So we're starting at vertex s, we visit some subset of vertices 1 through i-1 (possibly empty subset). And we visit vertex i then we go back and visit some subset of vertices 1 through i-1 and then we go to vertex t. And we're trying to express D(i,s,t). Now, given our insight of breaking up this path from to s to t into these four parts, now it will be straightforward to write a recurrence for D(i,s,t).

(1st Slide 15) Recurrence: i is on path

***



Case: i is on path

For $0 \le i \le n$ & $1 \le s,t \le n$

let $D(i,s,t)$ = length of shortest path $s \rightsquigarrow t$ using a subset of $\{1,\dots,i\}$ as intermediate vertices

Recurrence for $D(i,s,t)$:

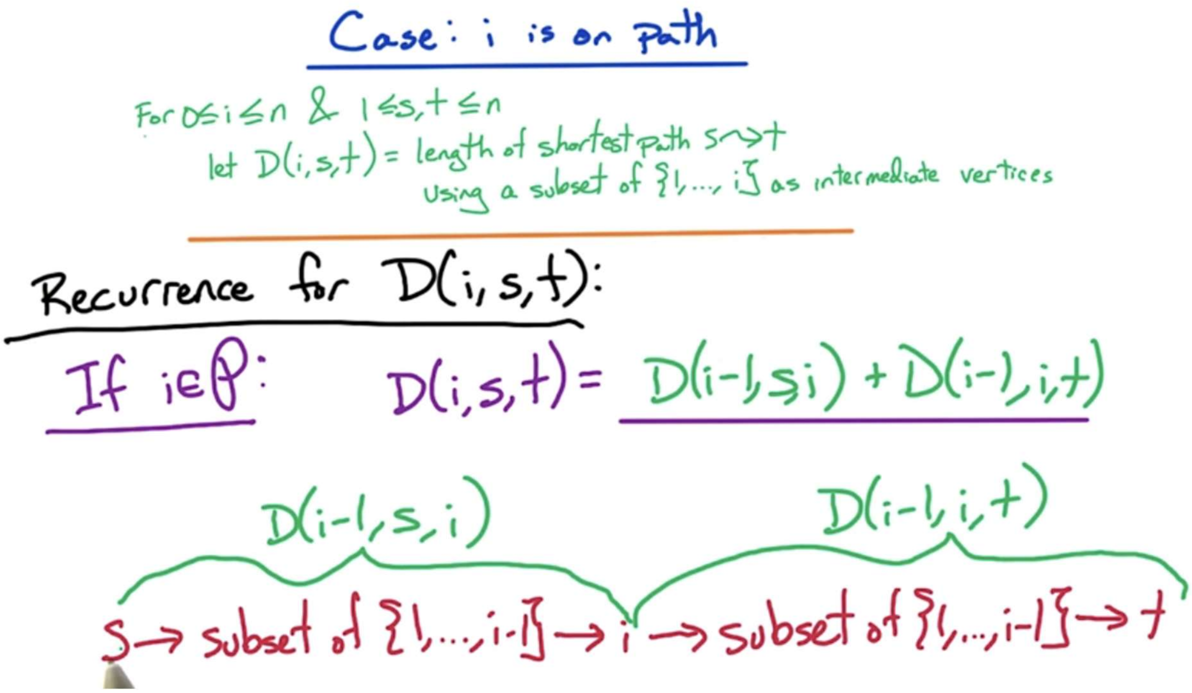
If $i \in P$: $D(i,s,t) =$ ______

$s \to \text{subset of } \{1,\dots,i-1\} \to i \to \text{subset of } \{1,\dots,i-1\} \to t$

Let's go ahead and take a break and see if you can express a recurrence for D(i,s,t) in terms of smaller subproblems.

(2nd Slide 15) Recurrence: i is on path (Answer)

***



Case: i is on path

For $0 \le i \le n$ & $1 \le s,t \le n$

let $D(i,s,t)$ = length of shortest path $s \rightsquigarrow t$ using a subset of $\{1,\dots,i\}$ as intermediate vertices

Recurrence for $D(i,s,t)$:

If $i \in P$: $D(i,s,t) = D(i-1,s,i) + D(i-1,i,t)$

$$\overbrace{s \to \text{subset of } \{1,\dots,i-1\} \to i}^{D(i-1,s,i)} \overbrace{\to \text{subset of } \{1,\dots,i-1\} \to t}^{D(i-1,i,t)}$$

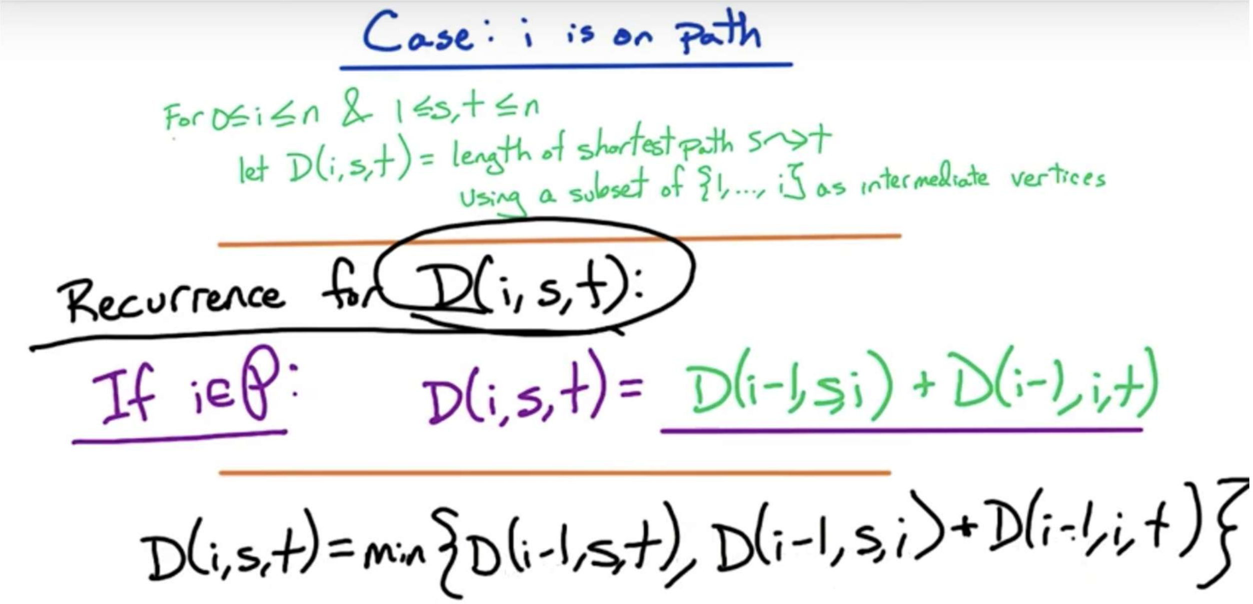
Now, to write a recurrence for D(i,s,t), we want to use D(i-1,s,t) for some pair s and t, which might be different vertices. But the key thing is that we want to change this parameter i to i-1.

- So how do we get that? Look at *this* (s -> subset of {1,…,i-1] -> i} path. This is a path from s to i which only uses a subset of the first i-1 vertices as intermediate vertices. So, that is D(i-1,s,t). That's a smaller subproblem which we've already solved in our dynamic programming algorithm.
- Similarly, *this* portion (i -> subset of {1,…,i-1} -> t) of the path is the shortest path from i to t which uses a subset of the first i-1 vertices as intermediate vertices. This is D(i-1,) because the first i-1 vertices only are intermediate vertices on the path and it's starting at i and it's ending at t.

So the total length of this path is the sum of these two terms. So now we have a recurrence with D(i,s,t). It's D(i-1,s,i) that gives us the shortest path from s to i using a subset of the first i-1 as intermediate vertices plus D(i-1,i,t) that gives us the path from i to t using this subset of 1 through i-1 as intermediate vertices on the path.

So we have a recurrence for D(i,s,t) in the case when i is in the path and we have the recurrence for the case when i is not in the path. What do we do? We take the best of those two scenarios.

(Slide 16) Recurrence: Summary
***



Case: i is on path

For $0 \le i \le n$ & $1 \le s,t \le n$
let $D(i,s,t)$ = length of shortest path $s \rightsquigarrow t$
using a subset of $\{1,\dots,i\}$ as intermediate vertices

Recurrence for $D(i,s,t)$:

If $i \in P$: $D(i,s,t) = D(i-1,s,i) + D(i-1,i,t)$

$$D(i,s,t) = \min\{D(i-1,s,t), D(i-1,s,i) + D(i-1,i,t)\}$$

So, we've handled the case where i is on the path, we've written the recurrence for D(i,s,t), and we've handled the case where i is not on the path on the previous slide.

So, let's go ahead and summarize the recurrence for D(i,s,t). We have two scenarios, i is on the path or its not on the path. We're going to take the best of those two. So, we're going to take the min, because we're trying to find the shortest path.

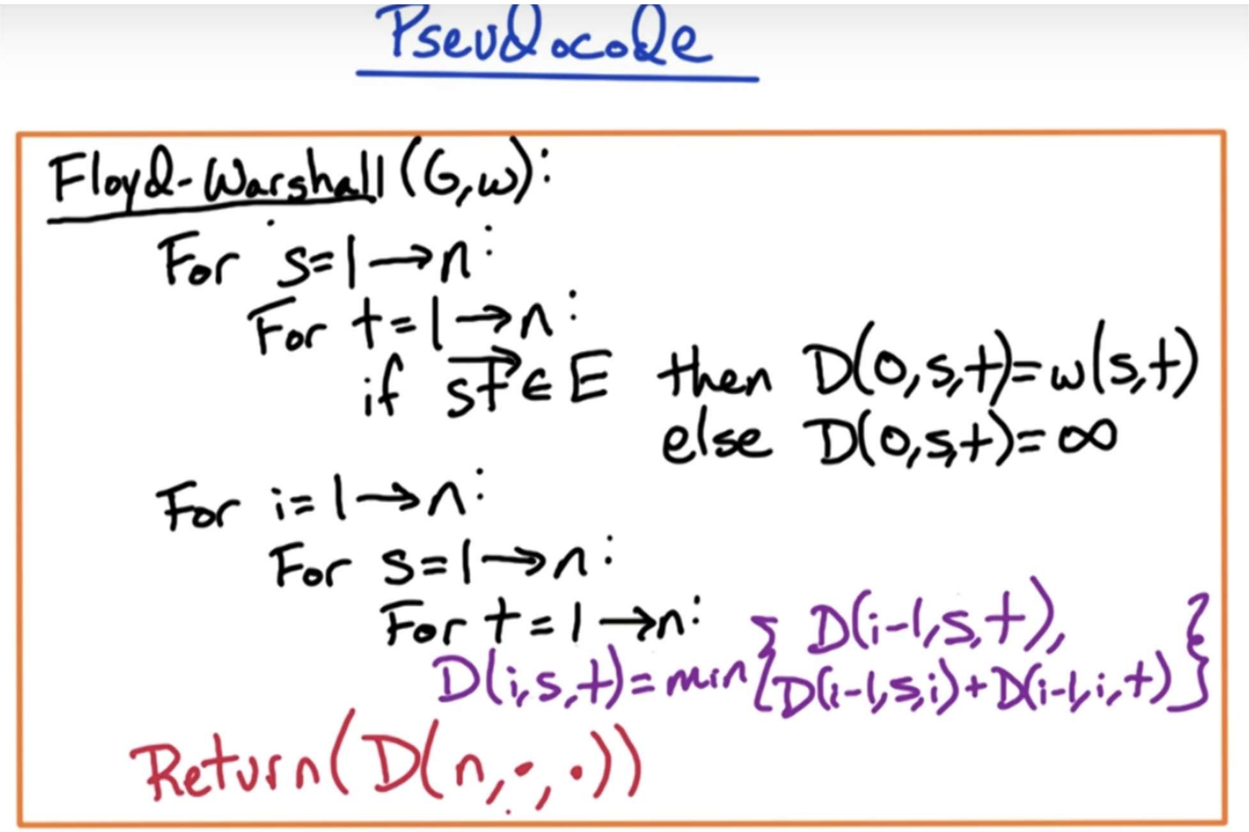
- When i is not on the path, recall it's just D(i-1,s,t).
- When i is on the path we have these two terms (D(i-1,s,i) + D(i-1,i,t)). We get the shortest path from s to i using the first i-1 as our main vertices. Then we go from i to t.

This is our recurrence for D(i,s,t) in terms of smaller subproblems (D(i,s,t) = min{ D(i-1,s,t), D(i-1,s,i) + D(i-1,i,t)}). Notice, its using D(i,s,t); and it is using D(i-1). So, if we go for i going from 0 up to n, then in order to solve D(i,s,t), it will use smaller subproblems which are already solved in our dynamic programming algorithm.

So, let's go ahead and write the pseudo code for this dynamic programming algorithm.

(Slide 17) All Pairs: Pseudocode

***



Pseudocode

```
Floyd-Warshall(G,w):
    For s=1→n:
        For t=1→n:
            if $\vec{st} \in E$ then $D(0,s,t)=w(s,t)$
                           else $D(0,s,t)=\infty$
    For i=1→n:
        For s=1→n:
            For t=1→n:
                $D(i,s,t)=\min\{D(i-1,s,t), D(i-1,s,i)+D(i-1,i,t)\}$
    Return(D(n,·,·))
```

So now we can write our pseudo code for the All Pairs shortest path problem. This algorithm is called the Floyd-Warshall algorithm.

The input to the problem is a directed graph G and a set of weights on the edges. And these edge weights are allowed to be positive or negative.

Now we'll start with the base case. What is the base case? The base case was the case D(0,s,t). So we want to iterate through all s and all t and fill in the entries D(0,s,t). Recall our vertices are numbered 1 through n, so to iterate through all possible choices of s, we just have a **for loop** where s goes from 1 to n.   Similarly, to go through all possible choices for t, we're going to have a **for loop** that varies t from 1 to n.

Now to fill in the entry D(0,s,t), we got to check whether there is an edge from s to t. So if (s, t) is an edge, then this entry D(0,s,t) is exactly the weight of this edge from s to t. In the other case where (s,t) is not an edge, then we're going to set D(0,s,t) to be infinite.

Now we go ahead and do the general case, where i is at least 1. So we're going to vary i from 1 to n. We're going to try to fill the entry D(i,s,t). So once again, we're going to go over all choices for s and all choices for t. So we're going have a **for loop** going over the choices for s and a **for loop** going over the choices for t. Finally we can fill in the entry D(i,s,t).

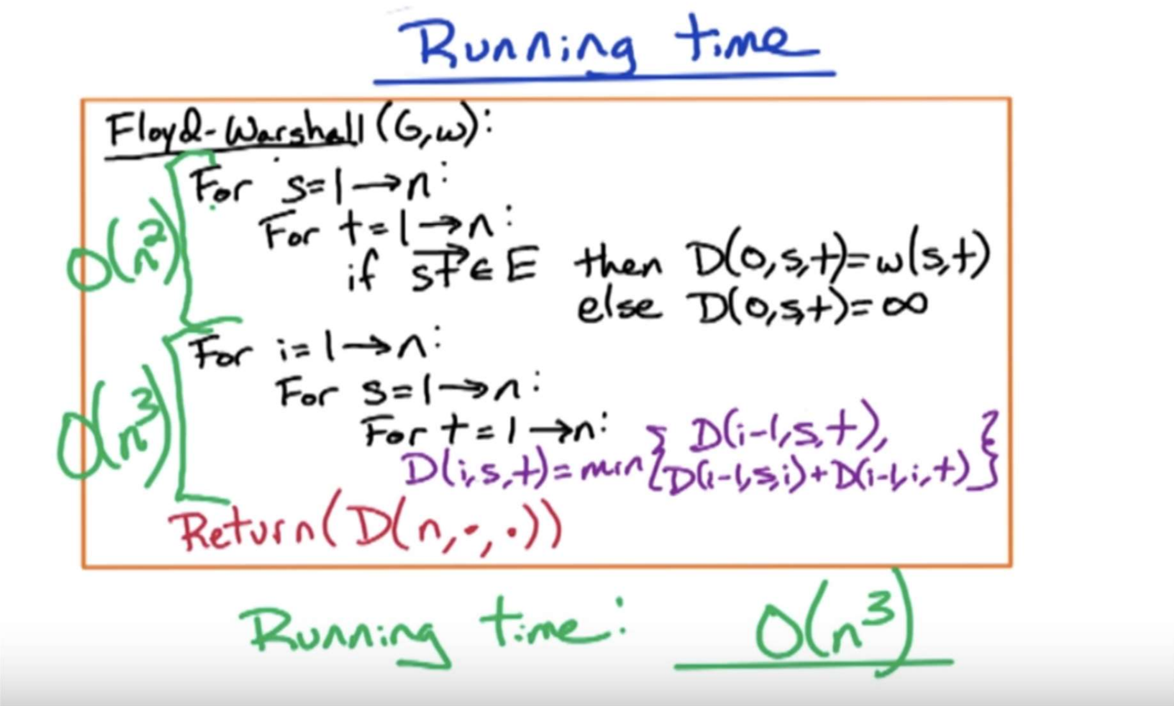
Now let's go ahead and write the recurrence for D(i,s,t). This is the recurrence that we just defined on the previous slide. There are two cases and we're going to take the min or the best of the two cases. The two cases depend on whether vertex i is on the path or it is not on the path. If it's not on the path then D(i,s,t) is exactly D(i-1,s,t). In the other case, when vertex i is on the path, we can break up that path into two parts. The first part goes from s to i and the second part goes from i to t and we take the sum of those two parts.

Finally, what do we return? We return the case where i=n. So we turn this matrix D(n,·, ·) for all possible choices s and all possible choices t. I put these dots that signifies we're varying over all possible choices for that index. So we want all possible choices for s and all possible choices for t. If you think of this as a three dimensional array - then we're returning the slice corresponding to i=n which isn't two dimensional array. It has n-squared entries.

This details the Floyd-Warshall algorithm. Now let's go ahead and analyze the writing time of the algorithm.

(Slide 18) All Pairs: Running Time
***
So, why don't you go ahead and analyze the running time for this algorithm.

(Slide 18) All Pairs: Running Time (Answer)
***



Running time

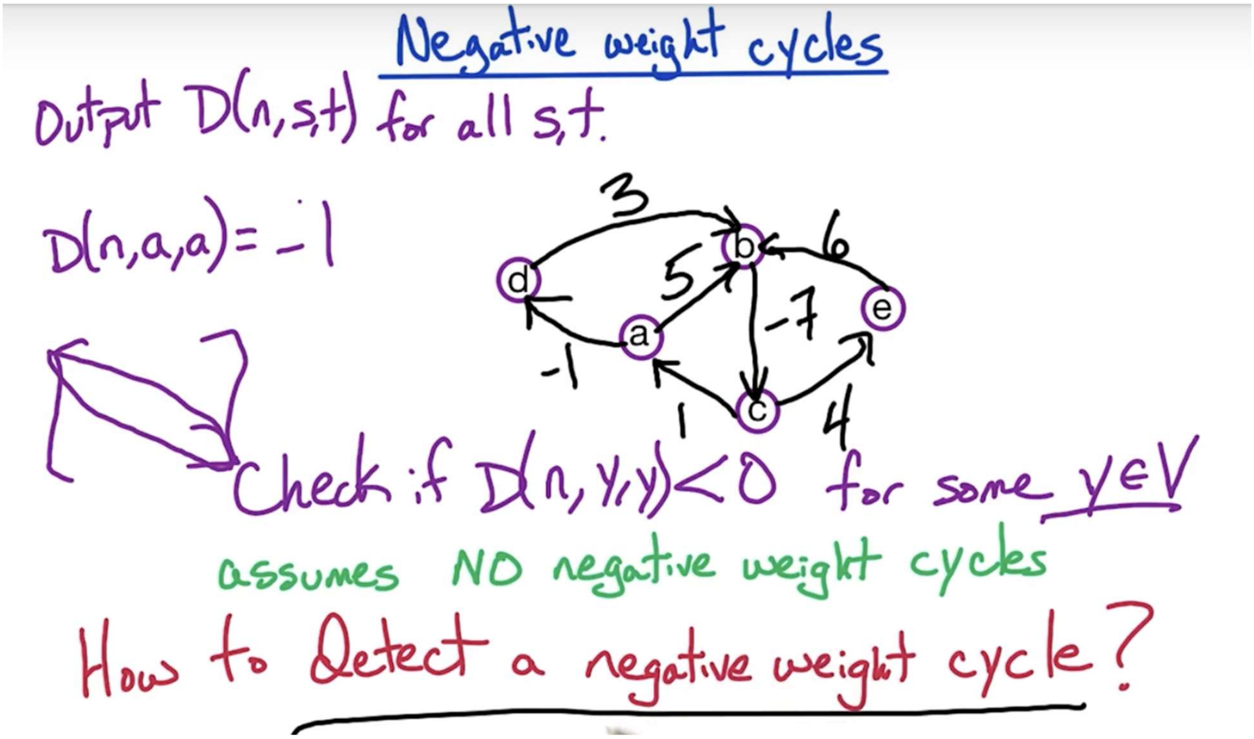
```
Floyd-Warshall$(G,w)$:
    For $s=1\to n$:
        For $t=1\to n$:
            if $\vec{st}\in E$ then $D(0,s,t)=w(s,t)$
                           else $D(0,s,t)=\infty$
    For $i=1\to n$:
        For $s=1\to n$:
            For $t=1\to n$:
                $D(i,s,t)=\min\{D(i-1,s,t), D(i-1,s,i)+D(i-1,i,t)\}$
    Return$(D(n,\cdot,\cdot))$
```

Running time: $O(n^3)$

Now, the running time of this algorithm is fairly straightforward to analyze because there's just **nested for loops**. And then, within the **for loops**, it's just O(1) time. Order one time. For the base cases, we have two nested for loops, each of size O(n). So, the total time to fill in the base cases is O(n^2) - order n-squared time. For the general case, we have three nested for loops, each of size O(n), so the total time is O(n^3) - order n-cubed. O(n^3) dominates, so total run time is O(n^3).

(Slide 19) finding Negative Weight Cycle #2

***



Negative weight cycles

Output $D(n,s,t)$ for all s,t.

$D(n,a,a) = -1$

Check if $D(n,y,y) < 0$ for some $y \in V$

assumes NO negative weight cycles

How to Detect a negative weight cycle?

Now, what happens in our algorithm if we have negative weight cycles in the graph? Well, the algorithm, as written, assumes there's no negative weight cycles. If there's no negative weight cycles in the graph, then this is correct. If there are negative weight cycles, then this is not necessarily correct, and we want to detect these negative weight cycles.

If there is a negative weight cycle, how can we detect a negative weight cycle? How can we find out if the graph has a negative weight cycle and output “YES” (there is a negative weight cycle)? And if there is no negative weight cycles then we can run the algorithm as is.

To get an idea for how to detect negative weight cycles, let's look at a simple example. So this is a cycle of length -1: 5 plus -7 plus 1 is -1. And let's add in two more vertices to make it a non-trivial example. Let's add some connections from d to this cycle and from this cycle back to d. And also, from this cycle to e and from e to this cycle. And let's add in some weights to these edges.

So, how can we find this negative weight cycle which goes from a -> b -> c -> a? Cycle of length three. What's going to happen in our algorithm? That's a more open-ended question. I want you to think about how can we modify the algorithm in some simple way to detect a negative weight cycle … How can we find this cycle of length 3 (a -> b -> c -> a) in this graph?

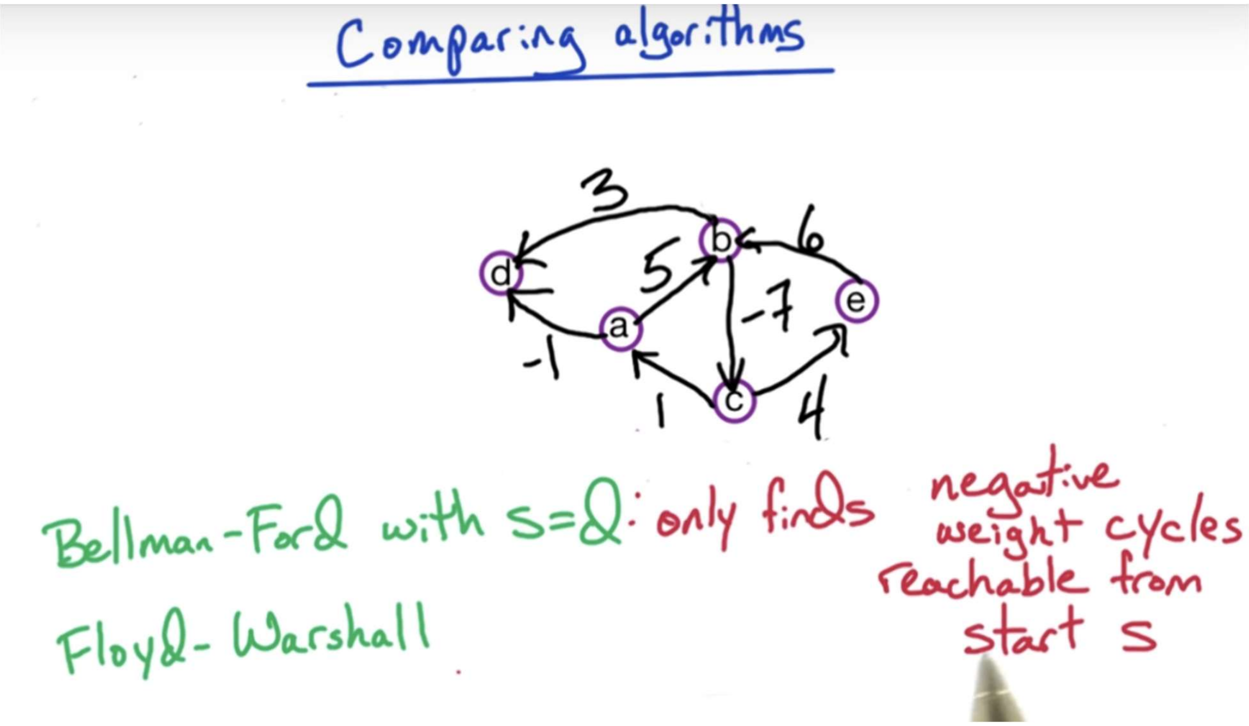
What is the final output of our algorithm? It's D(n,s,t) for all pairs s, t.

So let's look at this output on this example. Take somebody on this negative weight cycle, let's say vertex a. What is D(n,a,a) for this example? What is the length of the shortest path from a to itself? It's allowed to use anybody as intermediate vertices. When this example is going to be of length -1. And notice also D(n,b,b) is also going to be -1 and D(n,c,c) is going to be -1. That is going to signify that there's a negative weight cycle.   If we have any diagonal entries, so these are passed from vertex to itself. So any diagonal entries in the matrix which are negative then that means there's a path from a vertex to itself, which is shorter, which is negative. That means there's a negative weight cycle which includes that vertex.

So to check for negative weight cycle we check if any diagonal entry, any entry D(n,y,y), for any y is less than zero. And that signifies that there is a negative weight cycle which includes this vertex y. So if we think of this three-dimensional table, and we look at this slice for i=n, then we have a two-dimensional table. We look at the diagonal entries and we check if there's any negative entries on that diagonal entry.

Now notice we have two algorithms now for detecting negative weight cycles. We have the Floyd-Warshall algorithm which is all pair shortest path, and that's going to find any negative weight cycle in the graph by checking the diagonal entries. We also have the Bellman-Ford algorithm, which is the single source shortest path algorithm, and that also detects negative weight cycle. So there's some difference though, an important difference, that I want to distinguish between the two algorithms.

(Slide 20) Comparing Algorithms
***



Comparing algorithms

Bellman-Ford with s=d: only finds negative weight cycles reachable from start s

Floyd-Warshall

Let me modify this example slightly. I've taken the example from the last slide and I would just flip *this* one edge from d to b, it used to be, and now goes from b to d. And, I want to compare the two algorithms we have for detecting negative weight cycles.

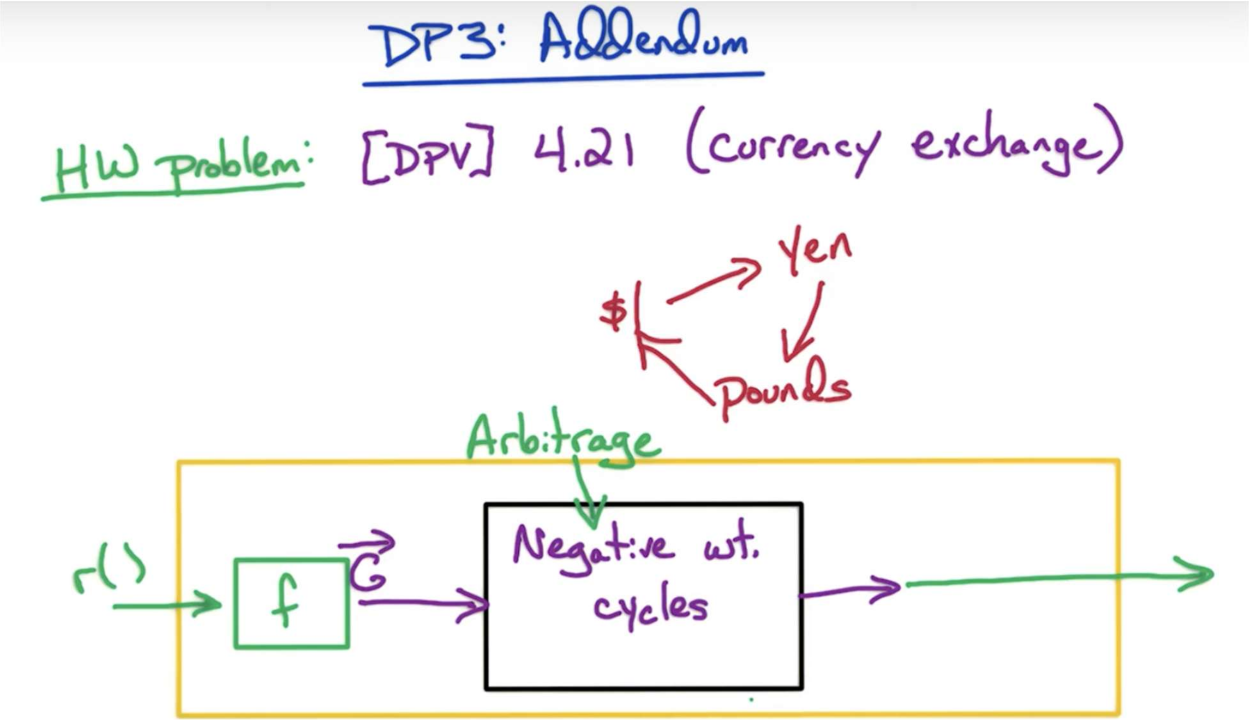
First algorithm is Bellman-Ford. (The second algorithm we have is Floyd-Warshall. Floyd-Warshall does all pair shortest path. Bellman-Ford does single source shortest path.)

- So, we give it a start vertex and it finds the shortest path from that started vertex to all other vertices.
- Now, in this example, suppose we run Bellman-Ford with the start vertex as d. It's not going to do much interesting because from d, you can't even reach any end of the graph.

  But, is it going to find this negative weight cycle, a, b, c? It's not going to find it because that cycle is not reachable from the start vertex. So Bellman-Ford only finds negative weight cycles reachable from the start vertex. It can't find this cycle because it's not reachable from the start vertex d, in this case. If we had run Bellman-Ford from a or b or c or even e, then we would have found this negative weight cycle.

So Bellman-Ford finds negative weight cycles which are reachable from the start vertex, whereas Floyd-Warshall will find a negative weight cycle anywhere in the graph because it's doing all pairs, so it's finding everything in the graph.

(Slide 21) DP3: Practice Problems
***



DP3: Addendum

HW problem: [DPV] 4.21 (currency exchange)

Here's a homework problem I often like to assign after this lecture. It's from Chapter Four, problem 21.

- It's about currency exchange and we're looking for an arbitrage situation. We're going to start, say, with one currency, say dollars. We're going to change it to another currency, let's say yen. And then we exchange it to another currency, let's say pounds. And then we go back to dollars. We're looking for a situation when we do this cycle and we end up with more than one dollar. So after this series of exchanges we end up with more than we started with.
- This is an arbitrage situation and we're looking for such anomalies in the exchange market. Now this is in Chapter Four which is about graph algorithms. And what we want to do, is we want to reduce it to the negative weight cycle problem. So we want to reduce this arbitrage situation to a negative weight cycle.

Now we just saw two algorithms for detecting negative weight cycles. Now we want to use these algorithms as **blackboxes**. So, we don't want to modify these algorithms in any way. We want to think of them as like a library, as a subroutine we got from a library and we can't modify the code. We can simply give it some graph as input and we can get the output, which is a negative weight cycle if one exists in that graph. Now we want to use this blackbox subroutine to construct an algorithm for this currency exchange problem. So we want to build an algorithm for detecting an arbitrage situation.

So we're going to take as input our currency exchange rates and we want to figure out how to convert these currency exchange rates into a graph. So we need to build this function. This function is reducing this currency exchange problem into a graph problem of detecting a negative weight cycle. So we had to build this function which converts these currency exchanges into a graph and then we simply want to run this graph problem for detecting negative weight cycles. We're going to take the output from that and that's going to give us the output for our arbitrage problem. We might have to do some conversion on that output, but we don't have to touch this algorithm at all.

It's called a **reduction** … where we're reducing the arbitrage problem, to this negative weight cycle detection problem. And we denote it this way (Arbitrage -> Negative Wt. Cycle), we're reducing arbitrage to negative weight cycles. So we want to use this as a blackbox to solve this problem.

Reductions are going to be an underlying theme in the course. We're going to use them to design efficient algorithms and later we're going to use them to prove hardness. We're going to use them to show NP-completeness. But this is a general theme, we know how to solve one problem … we want to use that to solve a new problem. So we want to reduce this new problem to some existing problem that we know how to solve.