SRINI DEVADAS: So, so far we haven't really done much with the data structures outside of Python lists. We looked at two dimensional lists but, you know, they are still lists. And so this particular puzzle that we're going to do today, I-- has a graph structure associated with it.

So for the first time, at least in this class, we'll be looking at traversing graphs, and we're going to do this in a recursive way. And, also, we'll have a graph representation using dictionaries. So most of you have probably seen dictionaries. If you haven't, I'm going to explain them. They're essentially ways that you can index into lists, not just using nonzero integers, indices, zero through n, or what have you.

But things like strings or tuples, and so they're just more general and-- they're more general data structure than Python lists. And so they're very convenient when it comes to representing graphs and traversing graphs.

So the puzzle we can look at is, as always, is a little bit contrived. It's about weekend dinner scheduling. So you have a bunch of friends. And let me draw your social network here, or someone's social network. So we won't name names, but each node in this graph represents a friend, and a couple more.

And so that's the set of friends, a through h, and that won't be particularly interesting as a graph. If it just had nodes, you want to add edges to it, what do these edges represent?

Well, you like all of your friends, but your friends don't necessarily like each other. And so an edge between nodes-- so nodes are friends, and an edge between a pair of nodes implies a dislike relationship.

All right. So b dislikes c, and it's symmetric. I mean, you could argue that dislikes is not necessarily symmetric, but we'll just call it that. That implies that c dislikes b as well. So this is not necessarily a directed graph.

Some of the time, especially when you're doing group planning, going from point A to point B, you have to take into account one-way streets, and so you have directed graphs.

But many a time, you can get away with undirected graphs. So you just-- the existence of an edge between B and C implies that in a traversal, you can get from B to C or C to B, and there's no direction there.

And so I'm going to fill in the rest of this, and then get to the question that we want to answer.

And so that's your-- your friend graph. And it also has these six or seven dislike relationships. And your job is to, well, keep all your friends happy. And the way you're going to do that is by-- is by having a couple of parties on the weekend.

So this is a regular weekend. Let's call it Friday and Saturday-- Friday and Saturday night parties. And the only constraint-- actually, there's two constraints. There's-- the two constraints that you need to follow are satisfy when you have these two parties. So you have your Friday party and your Saturday party, is you can't leave anybody out, right?

So you-- effectively, they're constrained as each friend--

--comes to--

--exactly one party. No one gets left out, no one comes to both. So that's fair.

And then the second constraint as you can probably guess is no pair--

--of friends who dislike each other, they're not friendly, they're your friends, can be invited on the same day.

And so you want an algorithm that is going to give you a schedule. And, obviously, the schedule is not necessarily unique. I mean, if you had a bunch of happy-go-lucky low overhead friends, you know, they all like each other and you could just randomly break them up into two groups. You can think of it as a partition. A partition is a grouping where-- then you have anything in a set of nodes, for example, and you partition it, you're essentially saying you're breaking it up into two groups, such that each node is in exactly one group, and each node-- and then we have another property associated with the partition that corresponds to the edges between these nodes. But we'll get to that in just-- just a minute.

Yeah, go ahead, Fadi.

AUDIENCE: How are we going to deal with the-- with FGH, because we only have two parties and--

SRINI DEVADAS: That's exactly right. So-- so the purpose of this puzzle is to determine whether you can actually do this or not, and so you are certainly welcome to-- when you write a computer program-- or when we write a computer program here, it might say, no can do, you know, find

a different set of friends or maybe identify a problem case. You know, there's one person who dislikes all of your other friends. Maybe you should drop that person, right?

And so you can imagine that-- that you would not necessarily be able to satisfy these two constraints for all graphs, right? And so that's really the essence of the puzzle. For what kind of graphs? And this, it turns out that a fairly famous kind of graph, does-- does this puzzle have a solution, right?

And so you-- you have to look at this and you kind of go, and the first thing that occurred to you was, hey, if I had F, G, and H, if I invite F on Friday, I have to invite G and H on Saturday. But then G and H don't like each other, right? So that's a problem, right?

And so in this particular case, you've identified this cycle that indicates that this problem is not solvable. But, let's say, that somehow you've placated G and you patch-- they're patched up.

All right. So-- so you end up having a fairly complex graph structure that looks like this, right? And now what happens? Now, is there-- is there a solution to this problem, to this particular puzzle?

I mean, A can come on any day so let's ignore A. All right. A is easy. Suppose you invite-- you can try and figure this out by saying let's go ahead and invite B on Friday. So what does that imply? That immediately implies that? C is on Saturday, right? What does that imply? D is on Friday, right? So you see-- you see how this is going to go. What does that imply? E and F are on-- are on Saturday. So I have E and F on Saturday, OK? And then what does that imply? That F being on Saturday implies that G, H, and I have to be on Friday, right? And then I-- don't forget A. So let's just stick A in here to even things out.

All right. So-- so we kind of did an algorithm here, we executed an algorithm, right? And it turns out that graphs that can be partitioned in this manner are-- I have a special name. They're called bipartite graphs. Bipartite graphs. I'll write it out.

And that-- the reason they're called bipartite graphs is if you have a bipartite graph, you can always draw it in this fashion.

Where you have A, C, E, F, and then you have B, D, G, H, I, such that you never see edges like this or like that. All the edges go from left to right-- or right to left. They're undirected. OK. So that's really a bipartite graph. And if I draw this out-- oop, don't need that.

C goes to B and D. A goes nowhere. E goes to D. And F goes to a bunch of different places. Yeah.

So you're allowed to have-- this will still be a bipartite-- I won't-- this is not the graph that we had, but it's not that-- the edges can be crossing edges. OK. They just all have to go from left to right. You just can't have an edge between A and C, or C and A. You can't have an edge between D and H, et cetera.

All right. So you want to-- you have to be able to separate out the nodes such that there are no edges in this set of nodes, and no edges between this set of nodes, and no edges between that set of nodes. And this is just a topological property that corresponds to bipartite graphs. And there's another property that essentially is equivalent to our puzzle, which says, bipartite graphs are two colorable.

And it's obvious as to why a bipar-- bipartite graph is two colorable, and what does it mean to be two colorable. Well, you need to color each node with two colors, in this case, Friday and Saturday, such that no pair of nodes with an edge between them have the same color. So exactly equivalent to what we've talked about here.

So our goal now is to discover whether a bipartite graph-- or a given graph, excuse me-- is bipartite or not.

Once you know that a graph is bipartite, you know it's two colorable, you know it's a solution to our problem, to our puzzle problem. OK. And, of course, in order to do this, and if you had a graph with-- you had thousands of friends and, perhaps, tens of thousands of edges, and you need to discover it's not your dinner problem, maybe. But you need to discover whether the graph is two colorable or not, then you need to write a computer program for this. You need a graph representation. You need to do this traversal, et cetera, et cetera.

Go ahead, Josh.

AUDIENCE: If a graph is two colorable, is it always bipartite?

SRINI DEVADAS: Yes. They're equivalent definitions. Absolutely right. Because what would happen is if you can color it, then it means that you can take all of the red colors, you-- red could be equivalent to Friday, and-- and then blue could be equivalent to Saturday, and all of the red colors could be placed on the left, and-- and the blue on the right. Fadi.

AUDIENCE: But then a graph with no edges is one colorable but also, by definition, considered bipartite?

SRINI DEVADAS: Yes. That's it. That-- there's always pathological cases, and they're usually things that get swept under the rug. But one colorable is-- it is considered two colorable. You can color it the two colors, that's really what it is.

Let's talk about cycles to get a sense for this property. So immediately I think you recognize that when you have something like, I forget what it was exactly, FGI or FGH, whatever, that if you had a cycle like this, that a graph is not bipartite, right? A three cycle is not bipartite.

Is any cycle a problem? If I have a cycle in the graph, which means that I can go from one node back to the same node through a sequence of edges and, obviously, traversing nodes, do I have a problem? Is that can be immediately declared that the graph is not bipartite?

Yeah. Go ahead, Julia, why not?

AUDIENCE: Well, like an even cycle if it was, for example, A, B, C & D--

SRINI DEVADAS: Ah, beautiful. That's exactly what I wanted to bring up. So if I had something like A, B, C, D, right? That's clearly a cyclic graph, and I could put on A and C on here, and B and D up here, and clearly there's no edge between A and C, there's no edge between B and D. There's an edge between A and B, there's an edge between C and D, and there's also an edge between A and D and C and B. So as I mentioned, you're allowed crossing edges. They all have to go from left to right or right to left, whichever way you're looking at it.

So interestingly enough, I mean, this is two colorable. But when you get to-- when you get to odd cycles-- so I'll draw one out here-- in the middle, and let's say I want to mark these nodes--

--the names, but let's say that you have a five cycle now. All right. If you have a five cycle, think about it. You-- you are going to go-- color this red. So let me just go ahead and say this is red, which means this needs to be blue, that needs to be blue. This blue implies this needs to be red, which now implies that this needs to be blue. And it's like, whoops, that's a problem, right?

So, interestingly enough, the odd evenness of the parity of a cycle determines whether a graph is bipartite or not, it doesn't mean that a graph that only has four cycles in it is

necessarily bipartite, because there could be other reasons for it not to be-- not to be bipartite. Because you don't know about the other cycles in the graph. I mean, they could be-- you say it's only up to four cycles. Well, there could be a three cycle in it, and things get pretty complicated when you have things like that.

There's a lot of cycles in here. This is a cycle like that, there's a cycle like this, and the moment you put an edge in here, now you're saying this goes over here, goes like this. There's a three cycle here. But even if this didn't exist, there would be a five cycle, et cetera, et cetera.

So there's a lot of cycles and graphs, right? And your algorithm, essentially, needs to ensure that for-- that every cycle is an even cycle. OK. And that is exactly equivalent to coloring it with two colors. OK.

So we, obviously, need to have a way of stopping here in the sense that when you have a cycle, you are going to come back and the cycle may be fine because it's an even cycle. But you, obviously, can't keep going around the cycle without terminating in a computer program.

So we're going to have to do a recursive search. Which has interesting termination conditions, you know, as opposed to the divide and conquer, or even the iterative enumeration that we did where it was pretty clear what the termination condition was. That the base case was fairly straightforward because you came down to a list of length 1, or you had something where you broke things up to the point where you had really small problems and that corresponded to your base case, right?

But cycle detection, especially when you're doing recursive traversal, can be pretty tricky. And we also, of course, as I said initially, we have to be able to represent this graph structure. Right. I mean, you could represent it as a matrix. Right. This-- or you could say, well, I hate dictionaries, I love lists, and that's all I'm going to use for the rest of my life, right? Not a good idea.

But if you did that, you could certainly represent the graphs as a matrix or two dimensional list. It's called an adjacency matrix representation-- and I won't write it out-- but, effectively, your rows are all the nodes, and your columns are all the nodes. And you have a 1 or a zero in the appropriate location based on whether there's an edge between the row node and the column node. And you can certainly do that.

And it's-- for most cases-- in most cases, it's a painful representation to deal with. Graphs are

much easier to deal with, especially when you want to do traversal. So we'll stick with a graph structure. All right.

So let's talk algorithms. I want to take a slightly different example. It's similar to what we have here, but-- and I want to talk about exactly how an algorithm would work. And we kind of did that.

So, actually, let's run it on this one, and then I'll write out what the pseudocode for the algorithm would be.

I'm going to ignore A here. You can imagine that this particular algorithm would be run on this degenerate case that corresponds to A, and you'd end up coloring this red or blue. So I'm just going to go ahead and color this red. And I'm not done with my problem, but I'm done with the particular graph that corresponds to A, because A can't reach any other node in the graph.

So you ought to be a little bit careful. And the code I'm going to show you would need to be generalized to take into account this forest of graphs. OK. Because there's really a bunch of different graphs that are disconnected that are part of the overall problem.

But if I start with B, then I get connectivity to all of the remaining edges, and so I'm good there. So let me go ahead and color this red. And so the way we described it, we say, I'm going to look for the neighbors of B. And so B is connected to just C in this case. And when I traverse an edge, what do I do with respect to the color? I flip the color. Right. I change the color. There's only two colors so I can think of it as flipping the color.

So I go and I traverse an edge. I need to flip the color to-- to blue, right? And then I get to C, and when I get to C, I then start traversing edges. But when I traverse an edge, I need to flip the color. So this goes to R. In the case of D, I can go to two different places, and so I need to go ahead and traverse to E. And just for the heck of it, I'm going to add another edge there. Right. Because now you're going to see how-- the code is going to get a little more complicated with respect to the termination condition that I described to you, and the checks that we need to perform with respect to the fact that we wanted this graph to be two colorable.

OK. So when I-- the way-- the way this code is going to work is I'm going to get to D, and with that color red, and I'm going to now see for the first time-- this is an interesting case. For the first time, I'm going to see that D has two neighbors. OK.

Now, I need to traverse both of these edges that correspond to these two neighbors in

sequence. I'm going to pick one and go with that. And then I'm going to pick the other one, right?

In general, there's two strategies that you can follow. And one of which is called depth first search, and the other is called breadth first search. OK. So I'm not going to-- you're going to see effectively a depth first algorithm. And I'm happy to tell you about breadth first in detail off line, but I'll mention what the difference is here so you get a sense of what the differences are. And they're two fundamental search techniques, traversal techniques.

And the depth first technique says I'm going to take D, and I'm going to pick one of its neighbors, and I'm going to go ahead and say that, effectively, that that's the only neighbor, and I'm going to explore everything that I can get to from that neighbor. And then once I'm all done and there's some termination condition that happens, I'm going to come back and I'm going to look at this other neighbor corresponding to F.

So-- so what would happen here in depth first search is the following. D would go to E, and when I go through an edge, I need to flip the color. Right. So I get a B here. At this point, unlike what we did early on when we did this particular example, or something similar, I-- what I did was when I saw R here, I colored F and E both blue. That's effectively what I did. Remember that? Right?

But now I'm actually doing something different. Right. This is a different execution corresponding to depth first search. Right. When I looked at all of my neighbors, including the F case, I remember we had some color for this, and then we immediately took G, H and I and we colored them, right? That is breadth first search.

OK. That is looking at taking one step in all directions that you can take and, essentially, it's a frontier based approach where I'm taking the frontier of my neighbors and pushing it outward one step at a time, right?

Contrast that with what we're doing here. You are going to take this-- the neighbor E corresponding to D, and then you're not going to go to F, you're, in fact, going to go ahead and pick one of the neighbors of E, right. And you might even have in your representation E's neighbor being D. Because, you know, D has E as a neighbor and so does E having-- has D as a neighbor.

But you'll say, oh, look, look, I don't need to go look at D now in my depth first search, because

D's already been colored. Right. So you check that. And then you say, oh, what has not been colored? I has not been colored. And when I traverse an edge, I need to flip the color, and I'm going to go ahead and put R in here. All right.

And it's not that you even go back to F now from D. You don't go back to D at this point. You now look at I and you do another recursion corresponding to I, and you say, OK, what about I's neighbors. Well, E may be first, but you already done with E. I'm going to go ahead and I'm going to get to F. And you do the flip and you get B over here, right?

You still don't go-- go back to D. You look at F and you say I'm going to now look at the neighbors of F, and maybe you get H. And you say this is R. Then you go back and, now, for the first time, you're going to look at exhausting the neighbors of a node that you've actually reached and you've gone to one neighbor for.

And so-- so at this point you go and you make this R. So now you come back and you say, am-- have I exhausted F's neighbors? Yes, in the sense that all-- the neighbors that I needed to look at, because they didn't have colors, have been exhausted. And, of course, F had D and I as neighbors. And they already had colors so I'm cool with that, right? I don't need to-- I don't need to touch them.

Now, at this point, I'm popping up because I've exhausted F's neighbors. I'm going to go back to where I came from. How did I get to F? I did not get to F from D. I actually got to F from I, right? So I'm going to go to I. But the fact of the matter is that I is done because I didn't have to deal with E and I've already-- I'm already done with that.

Then I go back to E, same thing, right? Then, finally, I get back to D, and I say, oh, now I need to go look at F, and I go and I say, oh, F is already colored so, therefore, I'm done. This is the simplest representation that I can think of for a graph that's also very efficient.

So I've messed with this graph quite a bit, so I want to claim that that dictionary representation that says graph equals open curly brackets, and then eventually close curly brackets, corresponds to that, but it's kind of roughly corresponds to that. And I can certainly write out any graph representation in Python for any topological representation that you give me.

But if you look at what's up there, the important thing is I want to show you-- or tell you about the constituents of this graph. So graph is a dictionary, OK. A dictionary is a set of key value pairs.

You represent sets in Python using curly brackets. Right. And each of the key value pairs is represented using some key colon val. And you see a set of key value pairs, because you see these commas between them, and so what you see here is the key. And the keys could be strings, they could be tuples, you can have many possibilities for keys. And the key comes first.

And in this case, the key is the string B. OK. And so that represents the name of the node B, and the value is usually a list. And in this case, you have a list that has a single entry in it because B only has one edge that goes to C. Right.

And so a more interesting case, as you can see up there, is D. And D connects to C, F and E. And so you see D-- see E and F. Right. So that's exactly what the point here is. Right.

This algorithm would run differently if you had a representation that was equivalent from a topological standpoint, because there's no order in the topology. But I chose C, E, F. I-- you know, it got typed somewhere, C, E, F, and so the order of neighbors is C followed by E followed by F.

Now, that is if you took this list of neighbors and you went from 0, 1, 2, in terms of indices of that list. There's nothing that's stopping you from going 2, 1, zero. In which case, you would go F, E, C. Right. Or you could do some crazy, random ordering for the list itself. Right. But it's all entirely up to you. Right. It is deterministic. It's entirely up to you.

But, now, you see how the order of the values that corresponds to these lists that give you the neighbors for every node affects the execution of the algorithm. Right. Now, you do not want the result of the algorithm yes or no, this graph is bipartite or not, to be affected by the order that you see there. Right. Because when I draw a graph like this, I mean, this graph is effectively something that doesn't have order.

You can't really say anything about the order of G, H and I in relation to F. I mean, they're-- well, that one is to the left of it, they're all-- two of them are at the bottom. But that's not what this is about. And so, obviously, that graph, regardless of whether I-- I drew it like this or I-- you know, I drew it like that, should be bipartite. Both of those graphs that I-- that correspond to H and G with their names flipped, are either bipartite or not bipartite. Right. So you absolutely cannot have the algorithm being affected by the order in which you see C, E, F or what have you. That make sense?

So you can do a lot of operations on dictionaries. It's probably something that you want to look at by looking at the code that we have here and looking at the code from graph analysis algorithms, and I'm happy to point you to other literature.

But the way you access a dictionary is-- looks a lot like accessing a list. So you can say something like graph A, and you say something like graph A, if A is not a key in the graph, it would give you an error.

But you can certainly get values back, and you can also assign values. So graph A, B would give-- it would give you this list C, and you can go off and you can say something like graph A, B dot append--

--and you can say D. And if you did that, if you did graph A, B dot append D, then that's effectively taking this and adding D to it. So you can mutate the graph if you wanted. We don't have to do that in our code for bipartite, graph checking, because we are not mutating the graph, we're just traversing the graph. But you could certainly do that.

So the dictionaries are a pretty cool representation. Encourage you to learn about them. And they're useful in many instances. Clearly, they replace lists because you can certainly have keys as integers, right? You can also have negative numbers as keys. There's nothing that's stopping you from doing that. And you don't have to represent the negative number as a string. I mean, you could just have keys that are-- that go from minus 22 to 27, what have you.

So, hopefully, you have some sense of what this code is going to look like. Both from a standpoint of the algorithm, which we executed a few times, and from the standpoint of the structure, the representation of the graph.

So let's take a look at about 10 lines of code that corresponds to bipartite graph coloring. So it fits on a screen. OK. And so this is a pretty tight code. It does exactly what we want it to do in the sense of it's going to determine whether a graph is bipartite or not. And it has a bunch of arguments. A graph is what you imagine it to be. It's the input graph.

We have to start from somewhere. This graph has a list of key value pairs. The keys happen to be the nodes. You should never depend on the keys being stored in a particular order. There's ways of saying-- you know, you can say things like graph dot keys.

And this is going to give you back a list of keys. And it's possible that when you say graph dot

keys, at one point in the program, you get the keys in a particular order. And when you say it at a different part of the program, you get the keys in a different order. OK. But all-- the keys won't change. If there's no bugs in your program, you'll get the same set of keys, but they may be in different order.

So you obviously want to start with some node in the graph, and you're going to color it with a particular color, color it red. I think I had shaded and hatched here, so sha stands for shaded, and hat stands for hatched. But you can use what you want.

So if you see the invocation here, bipartite graph color-- graph three starts with a node. This is an-- its coloring is a dictionary, and so we are also representing not only the graph as a dictionary, but we're also representing the mapping that we get. Namely, the node B was colored with red. So B is a key in this coloring dictionary. And R red is-- is the color that's a value. Right.

So you can represent mappings. Key values or mappings. Right. So the coloring of a node is also a dictionary. And so that's essentially what we have. So I'm going to start with-- none of the nodes are colored, so that's why this is empty, and I'm going to start with A, and I'm going to start with the color shaded, which means that A is going to be colored shaded. Or A is going to be shaded. OK.

And this is going to return true or false depending on whether the graph is bipartite or not. And if it's true, then I'm going to get something interesting with respect to the mapping of nodes to colors. That's my dictionary coloring. And, otherwise, I get false back, which an empty dictionary. Because when I have something that's false, I mean, you could imagine saying, well, there's a problem case here. You can color all of these different nodes, but I'm kind of stuck here because of that five cycle. But that's kind of indeterminate in terms of what the colors need to be.

And so-- unless you were actually going to go off and mutate the graph and remove a node and-- and say this is the friend you want to dump today, and then the rest of them are going to be invited for dinner. In that case, you could obviously return a coloring dictionary, even in this case of the graph-- original graph not being bipartite. But most of the time you're probably in a situation where if it's not bipartite, you don't return anything other than false.

All right. So let's take a look at what each of these things do, and then you'll also get a sense of how we manipulate dictionary structures. So I'm just going to explain very quickly each line

of code here.

So this pair of lines of-- this pair of lines of code, essentially, say something like, well, if you've given me a starting node, it's just a check on the input. The starting node is not in the graph, it's not a key in the graph, then I'm throwing up my hands here. This is not a graph. It's not a bipartite graph. I can give you a coloring, right. So easy check. Not much there.

This is an interesting check. This, essentially, is going to check to see whether the particular node that you have, this is going to be done recursively. So anytime you arrive at a node, you have to check whether there's already a color there or not. If there's no color, then you move forward, and you color it, and you also move forward in the sense that you'll probably go ahead and if it has neighbors, you will traverse the neighbors. Right.

But if it has a color, if it's already in coloring, that means it already has a color. Right. Because the keys that correspond to the coloring dictionary are exactly the same in the sense that they're graph nodes and they're also in the graph dictionary. Right. The mappings are, of course, completely different, and the coloring dictionary is being grown as we speak, or as we execute. But the graph dictionary is actually not mutated. It's static.

OK. And so this check here says that I'm going to go ahead, and if it's not in coloring, oh, that's great. I'm going to go ahead and whatever color I have, I'm going to go ahead and color it with that. That's what coloring start equals color does. It just colors the node. OK.

And then that effectively is adding the key value pair start comma color to the coloring dictionary. That's what it does. Otherwise-- well, I got two cases here. Right. This thing has already been colored, but I have two cases. I want to color it with something different from what it's already been colored. That's a problem. That's a problem. That's exactly when we get a five cycle and we throw up our hands.

So we're done here. We just say doesn't matter what we've colored so far, it's all garbage. You know, this thing can't be colored with two colors given the constraints I have, so I'm just going to return false and an empty dictionary. So anything that coloring had in it, I mean, is gone. Right. Because it's bogus.

Otherwise, this recursion would say, oh, well, I'm going to return true for this thing here, and I'm going to-- it has been colored, which means that, you know, I haven't-- I don't want to do any more work on it. I mean, I got this node here. It's already-- it was already colored and was

consistently colored. So if I don't want to get into infinite loops, I better not, you know, start traversing nodes from this node because I've already seen it before.

All right. So the fact that it's already colored correctly is a termination condition. This is important. That says that I don't want to keep repeating myself, that is like going through the cycle over and over. Right. If you didn't have that, right? That make sense? Yep. That's important.

So, now-- well, this is a flip. So there's no recursion, there's no procedure calls here, this is just flipping the color. That make sense? So, now, the last part, which is the part where I'm actually doing depth first search is the part which says, OK, I got to the point now where I'm-- essentially, haven't done any returns here. Right. I've colored this with color, and I've flipped the color. Right.

So these two lines are executed. No returns. These lines are executed. And, now, I have clearly already colored start and I flipped the color and I'm going to go ahead and make the recursive call corresponding to the neighbors of start, and all I have to do to look at the neighbors of start is to return the list corresponding to the value of graph start. Remember, I said, graph A or graph start is going to give you this-- this list, and I can go ahead and append to it and-- I'm not going to mutate, but I could have done that.

And so that is going to give me my list and I'm just going to go through, enumerate that list in whatever order that it came to me, and start with-- I'm going to call the first element vertex, because that's also a vertex or a node synonymous. And I'm going to send in vertex here, graph stays the same, vertex is the same. I've modified colorings, so I want to pass that over. And, obviously, I'm going to pass new color here because I've done the flip. Right.

And if I ever-- I get something that's false, that means I've found an odd cycle and I return false. And then if everything works out, I return true in the coloring. OK.

So if I go ahead and run that, if my machine hasn't gone to sleep, doesn't do very much. You know, the first graph, it was not colorable. You can check these results by yourself. But this is more-- the reason I'm putting this up is to give you a sense of what the coloring dictionary looks like. It's just so you get some more exposure to these things.

So in this case you get a non-empty coloring dictionary for the second run. So I ran it four times on four different graphs. You get a non-empty coloring dictionary. It's true. The graph is

bipartite. And notice that the order in which this coloring dictionary was-- what was created was certainly a function of the execution of the algorithm.

But this order might actually change if I keep running this algorithm, and if my machine had different loads, and so on and so forth. But you would always get through for any graph representation that is bipartite or too colorable. But you have no control over whether F hat comes first, or E hat comes first. Those could be flipped.

All right. So never depend on the order of keys in a dictionary. OK. But, certainly, the existence of keys in a dictionary you can-- or the nonexistence you can depend on.

All right. Good. So that's really all I had to say.

Any questions?

All right. Excellent. I want credit for finishing one minute early.