PROFESSOR: OK. So who's going lecture? Good. Does everything makes sense?

AUDIENCE: Mostly.

PROFESSOR: That's good, because we're going to do problems.

AUDIENCE: Awesome.

PROFESSOR: So, what I want to talk about is, first, there's a subtlety in the Rubik's cube problem, which I'm guessing nobody has seen yet, but we're going to talk about it anyway, because otherwise you'll trip on it while you're doing the problem. And then we're going to talk about one or maybe two problems. And I'm excited about the problem titled, "there's StarCraft and counter strikes," so, let's hope we get to them.

AUDIENCE: I'm curious, did you write this week's problem set? Because there's some Chinese in there.

PROFESSOR: No.

AUDIENCE: Oh, OK.

PROFESSOR: No, someone else wrote it. I suggested putting that in there, though.

AUDIENCE: You had an awesome internship, right? [INAUDIBLE]

AUDIENCE: Wait, did you make all the AMD [INAUDIBLE] questions?

PROFESSOR: Of course. That's my stuff. OK. So Rubik's cubes. Did everyone see a 2 by 2 by 2 Rubik's cube? You guys are going to lecture, right? You have one. Yeah. Did you guys see Eric's 2 by 2 by 2 Rubik's cube? It looks sort of like this. All right, so big

Rubik's cube has how many small cubelets in it?

AUDIENCE: 80.

AUDIENCE: 80.

AUDIENCE: I like cubelets better than cubies. Cubies is hard to say.

PROFESSOR: Yeah, I would say cubicles. So let's go for-- let's try cubelets and see how this works. OK. So each cubelet has six faces, but three faces are always facing inwards. They're always attached to the center of the cube. So we don't care about them. We only care about the other three faces. So if you have a plastic Rubik's cube. Say this is a plastic-- This is a plastic cube. Then you're going to have eight plastic cubelets. Each cubelet has three colored faces and three faces that we don't care about. So how many total plastic faces in a cubelet, sorry, plastic cube?

AUDIENCE: Twenty-four. [INAUDIBLE]

PROFESSOR: All right. Now let's play with an imaginary cube. And I call this a wireframe cube, because I draw it like this. So, here pretend you have the skeleton of a Rubik's cube, made out of wires or out of wood or something. So it's basically a small sheet of wood, small piece of wood, small sheet of wood, so on and so forth. And so this doesn't have any colors in it, but it's the right shape and sort of the right look. So the way you would build the configuration is, you take 24 plastic faces that you get from a plastic cube, and you paste them one by one onto the faces of a wireframe cube. And they claim that this can build any configuration that you can think about. If you look at a cube, after no matter how many moves you've done to it, it's going to have some-- it's going to have the faces somewhere, right? So you can take the faces and face them where they belong and there you go. You have a real cube built out of a wireframe and plastic faces.

How many wireframe faces there are? Please don't get this wrong.

AUDIENCE: Twenty-four.

PROFESSOR: OK.

AUDIENCE: There are some configurations that aren't possible, right?

PROFESSOR: OK. There are some configurations that aren't possible, and that is very good news. We'll get to that later.

Let's see how the code talks about these faces. So suppose we have a cubelet that has these faces. Front faces yellow, this face is blue, and this face is orange. This facing code is called-- so this yellow face here is called y o b. So y is yellow, that's the yellow face, and is the yellow facing the cubelet that has a yellow face, an orange face, and a blue face. Fortunately, there aren't two cubelets that have the same face color, so this is good enough to distinguish between all of them.

Let's see if you guys are paying attention. How would I call this face?

AUDIENCE: o y b?

PROFESSOR: Yeah. Either that or o b y, whichever one the code happens to use. So the principal is, these are the colors and the first one is the face that you're pointing at. So there are 24 plastic faces, so there are 24 names for these plastic faces. Let's see how we name the wireframe faces.

If I take this face-- so we don't have colors here, they're all the same. So instead we care about their position. A cube has a front and a back, right? This cubelet is on the front somewhere, right? So I'm going to have front somewhere in the name. It has an upper face and a lower face. Is this going to be upper or lower?

AUDIENCE: Upper.

PROFESSOR: And the left or right?

PROFESSOR: Right.

PROFESSOR: OK. So front upper right, which face am I looking at?

AUDIENCE: The front.

PROFESSOR: OK. f u r, fur. How about this space?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So far so good?

Or it could be r u f, right?

PROFESSOR: Or it could be r u f.

AUDIENCE: [INAUDIBLE].

PROFESSOR: I really don't know what it means, I'm behind.

AUDIENCE: Ruff like a dog? It has fur.

PROFESSOR: Oh, ruff ruff ruff. OK. That's not too bad. I was afraid of something worse. OK. That's cool. So a configuration will take plastic faces and face them onto the wireframe faces. So a configuration is going to be an array of 24 elements. And the first element is going to tell me which plastic face ends up in-- Let's say if you are the second one, says which plastic face ends up in r u f, and the third one, let's assume it's the space.

AUDIENCE: Ruff.

PROFESSOR: Ruff. And there are 21 more faces, right? So each face has a number from zero to 23, because we're in Python, so we like zero-based indexing. And 24 elements in the array, 24 plastic faces. Which plastic face goes into f u r? Assuming this cubelet here-- see, I said cubicle anyway. This cubelet here goes here in exactly this configuration.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So this face ends up here. And that's what it means that the first element in the array is y o b. How about r u f? Who ends up in r u f?

AUDIENCE: [INAUDIBLE].

AUDIENCE: B o y.

PROFESSOR: And who ends up in e r f?

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. So does everyone understand configurations? Does this make sense?

AUDIENCE: Yeah. I got lost somewhere in there. Can you explain the order for f u r and [INAUDIBLE]? Like front upright?

PROFESSOR: Oh. So do you mean-- so the order of the letters. So there are two orderings here. One is the order of the letters and there is an order. I think the code uses clockwise order, but what really matters is the first letter tells you which face you're talking about.

AUDIENCE: So this is front face.

PROFESSOR: Face, up face.

AUDIENCE: What's up? What's front?

PROFESSOR: So for this cubelet, this is front, this is the right, this is up. This cubelet wold have a front, an up, and a left. So now there are 24 faces and all of them get numbers from 0 23. And here we assume that f u r is space zero, r u f is space 1, and e r f is face 2, which is not quite [INAUDIBLE], but you'll have the mapping there.

OK. So a configuration is in an array of 24 elements and it maps plastic faces to wireframe faces, which brings us to, how many configurations are there?

AUDIENCE: You mean like mapping 4 factorial?

PROFESSOR: OK. So let's see what it does. We have 24 plastic faces. From 0 to 23. And they have to be mapped to 24 positions.

AUDIENCE: [INTERPOSING VOICES]

AUDIENCE: There's six colors, aren't there?

AUDIENCE: Yeah.

PROFESSOR: There are six colors, but if this is also yellow, but there's a red here. This is a different face, see, this is yellow, red, and there's a green down here. Then this is the yellow face of the yellow, green, red cubelet. So we care about the individual faces. So yeah, your answer was right. I'm trying to explain it to everyone else. But your answer is right.

AUDIENCE: Well, wait. But you can't-- I mean, we talk about some are impossible. Like, you're always gonna have a grouping of y o b. That's never gonna change.

PROFESSOR: Right. So some of these aren't reachable, because you're not allowed to break up the cube.

AUDIENCE: Yeah.

PROFESSOR: And that's OK. This is still how the code represents them. So I'm asking if you had-- if all these would be possible, how many would we get? And the answer is that to map 24 plastic faces to 24 wireframe faces. So this mapping is called-- anyone good with math?

AUDIENCE: 1 to 1.

AUDIENCE: Bijection.

PROFESSOR: OK. So when you're taking-- OK. So when we have two sets and you have a bijection between them or 1 to 1 mapping, that function is also called?

AUDIENCE: [INAUDIBLE].

PROFESSOR: You probably don't know the answer yet. You have to wait until the end of 6042. So I'm trying to hint that right now, if you remember the end of 6042, maybe the answer is somewhere there.

AUDIENCE: Blur?.

AUDIENCE: Perfect.

AUDIENCE: A perfect match

AUDIENCE: Permut--.

AUDIENCE: I totally learned about that in middle school, I think.

PROFESSOR: OK. So this is--

AUDIENCE: Permutations and combinations.

PROFESSOR: So this is a permutation.

AUDIENCE: Yeah.

PROFESSOR: How many permutations do you have out of 24 elements?

AUDIENCE: Does that order [? mean ?] [? matters ?] [INAUDIBLE] 24 [INAUDIBLE].

PROFESSOR: So this is a lot of permutations, right? It's a good thing that we're not going to explore most of these configurations. So we said before, that we're going to build a graph where the vertices are configurations, and the edges are moves that get us from one configuration to another configuration. Can they afford to build this graph first and then run BFS on it? Not going to work, right? We don't have enough RAM for this. So instead we're going to have to operate on an implicit graph for presentation. Well, fortunately, BFS doesn't want the whole graph. The way BFS works is it has a list of nodes that it has to visit, and when it visits a node, it wants to know its neighbors. And that's it. Yes? Everyone happy with it?

So all they have to do is, in order to build this implicit graph representation, is we have to know the start node, So BFS can start somewhere. And we have to be able to give it a node, we have to be able to generate all the neighbor nodes. So given the configuration, we have to be able to generate the configurations that would result by applying most of that configuration. So what are the moves that you can do with this? Suppose you have a cubelet. What are the possible moves? There aren't that many. We're interested in the simplest kind of moves.

AUDIENCE: [INAUDIBLE]

AUDIENCE: Yeah, you can make the right side go right, which is equivalent to the left side going left. And then you can do [INAUDIBLE] 4, left [INAUDIBLE].

AUDIENCE: It's a nice number.

PROFESSOR: OK. I think there are a few more, so here's how I look at it. You can take the front face, so these four, and do a clockwise rotation, 90 degrees, or you can do a counterclockwise rotation, 90 degrees. And you can do that for each face.

AUDIENCE: But doing a front clockwise rotation is the same as doing a back counter clockwise rotation, which is [? it-- ?]

AUDIENCE: Yeah.

PROFESSOR: Doing a front clockwise, doing a back counterclockwise is the same thing after you account for symmetries. Yeah.

AUDIENCE: Exactly.

AUDIENCE: So it's six?

PROFESSOR: Yep. So there's only six after you account for symmetries. Let's play devil's advocate and assume we don't account for symmetries. We're actually doing the code, because I need to teach you something. So let's assume we don't apply. By the way, how would you account for symmetries, easy way? Did anyone read the comments in the code? OK. So would you account for symmetries.

AUDIENCE: Doesn't it just happen? The permutation is going to [INAUDIBLE].

AUDIENCE: Can you just look at it, like, each space is neighboring faces, and [INAUDIBLE]?

PROFESSOR: So the way we do it is we anchor one cubelet. So we have one cubelet that is a plastic cubelet that is always going to go here. So three plastic faces here are always going to go to three wireframed faces here. And if you fix those three faces then you don't have simple symmetries anymore. All right? Three axes of symmetry, so fixing three faces settles it.

AUDIENCE: Wait, so then how many moves are possible? 12?

AUDIENCE: No. But you only do one thing, though. Now you don't do counterclock-- you don't do.

AUDIENCE: If one is anchored, there's six left.

AUDIENCE: Yeah. Yeah. You don't move the anchored one.

AUDIENCE: Oh yeah, because you can only [INAUDIBLE] two of the-- three of the faces now, but.

PROFESSOR: OK. So let's look at the left face. So you have a configuration. So you can move the left face clockwise, left face counterclockwise. You can move the top face clockwise, top face counterclockwise, and a couple more. So what does a move look like? Someone said it before. So when you're doing a move, what's going to happen is, a move doesn't create plastic faces and it doesn't destroy them, right? It just changes their position on the wireframe. So if you have an initial configuration that has plastic faces assigned to wireframe faces, and you have a final configuration, these plastic faces are going to end up somewhere on the wireframe. And that's true for all the plastic faces. So the plastic faces are going to end up somewhere.

Also, what's nice is if we look at the wireframe, if I do a clockwise rotation in the front, whatever plastic face was here is going to end up here. Whatever plastic face was here is going to end up here, so on and so forth.

So it's a function of rotation [INAUDIBLE] inside [INAUDIBLE]. Like the four faces that correspond to each other, rotate.

PROFESSOR: OK. That's some good-- that's going deeper than want to go. I want to account for something simpler.

So my point is that the arrows will always, if I do a right clockwise rotation, the arrows will always look the same. So the only thing that changes is what's inside the array. But if you take one configuration array and you move things according to the way you're supposed to move them for the front clockwise move, you're going to get the right result. So what is this a bunch of arrows called? What does it do? What does it do to the faces? What does it do to the elements of the array?

AUDIENCE: It moves in different spots.

PROFESSOR: OK, so fancy math name?

AUDIENCE: Shuffle.

PROFESSOR: OK. So this is what I got before. This is how you think of it, right? It's a shuffle, so fancy math name for a shuttle.

AUDIENCE: [INAUDIBLE].

PROFESSOR: All right.

AUDIENCE: Something going on here.

PROFESSOR: So then a move is a?

AUDIENCE: Rotation [? permutation?]. [INAUDIBLE]

PROFESSOR: All right, so the thing is-- the difficult part of the code is you have two things that are represented by permutations. Configurations are permutations of plastic faces onto the array, onto the array of wireframes faces. And then a move is a permutation of a configuration into another configuration.

OK, so now if I want to compute neighbors, I take-- we think there are six moves, so I'm going to take six moves. I'm going to type out the permutations, so 6 times 24 numbers, and that's it. Now I can compute neighbors. Does that make sense?

AUDIENCE: I think you have all the permutations.

PROFESSOR: So there are six moves, right?

AUDIENCE: Yeah, but then it's right. Oh yeah--

PROFESSOR: If I take a configuration, and I apply the permutation for this move, I'm going to get a new configuration, and that's its neighbor. So if I apply the permutation for LC 4, left clockwise to this configuration, I get the new configuration. So all I need to do is hard code those configurations, and then I'm done. I can compute neighbors. But that's a lot of typing. I don't want to type as much. I want to reduce my burden a little bit. And we can do that by noticing that some moves are associated with some other moves.

So, if I go left clockwise move, that's the inverse of a left counterclockwise move. So, if given the permutation, I could compute its inverse, then I wouldn't have to type as many permutations. Right? Why do I care about that? Did anyone take 6004? Remember? So at some lab, which is the dreaded beta lab, you have to write the control block for a processor, right? And that's a lot of zeros and 1's that you have to type. And if you get one wrong, good luck to [INAUDIBLE].

[INTERPOSING VOICES]

AUDIENCE: 18 wide and 64 tall.

PROFESSOR: I know.

AUDIENCE: Copy and paste like.

AUDIENCE: Yeah, and copy paste. [INAUDIBLE].

PROFESSOR: OK. Bad memory. Let's move on. Let's move on. So two good news, two pieces of good news. One, some of these are redundant in our case, two, we wrote everything for you, so you don't have to write them, we're just teaching you how we did them, because the inverse of a permutation is cool and I think it's a good concept to know. So let's look at how a permutation looks like and compute its inverse and then move on to other nice problems.

OK so let's take a presentation of five elements, not going to go to 24 on the board here. 3, 5, 1, 2, 4. This is how permutations look like in math mode. If you want to be more explicit, you can add a row numbers above. And this is a bit more explicit about what a permutation does. So permutation takes an input list, let's say a b c d e. And for this is an output list, according to the recipe that you see here . So this pretty much says, the first element of the output is the third element of the input. The second element of the output is the fifth element in the input. Wait, this is the wrong permutation. It doesn't look as pretty, so I'm changing the permutation. The third element of the output is the first element in the input. The fourth element of the output is which letter? So which letter?

AUDIENCE: Oh. [INAUDIBLE]

PROFESSOR: And the fifth element of the output is which letter?

AUDIENCE: Did you mean to get c instead of a? No.

PROFESSOR: Yeah. That would make more sense, wouldn't it? Thank you. OK. So if this is permutation pi, then this is the effect of applying pi to this original list. Now the inverse of a permutation by minus 1 is another bunch of arrows, and what I want to do is, if I take this, which is the output of pi, and I run it through these arrows, I want to get back a b c d e. So I want to undo the effect of pi. OK. How do I compute this inverse? How do I compute pi minus 1 given pi?

AUDIENCE: Wait, you're giving us the initial, when we start with c d a e b?

PROFESSOR: You only start with this. This doesn't matter at all. A different list would get different results. So this is the permutation, and I want pi to the minus 1.

AUDIENCE: So you take the value. So you have 3 4 1 5 2. So the value at index 1 you put that index, that index has a value of index 3, et cetera.

AUDIENCE: So c would go to position 1?

AUDIENCE: Wait.

PROFESSOR: OK.

AUDIENCE: Is it true that you can run pi, so the number of times it will eventually equal pi inverse?

PROFESSOR: Yes, but that's complicated algebra to prove that.

AUDIENCE: Not forget that. Just like make that [INAUDIBLE] thing that you made and sort of flip it.

PROFESSOR: OK. I like that. So 34152, 1 2 3 4 5. Except this doesn't look very pretty, so I need to sort them again, right? So 1 3, 2 5, 3 1, 4 2, 5 4. Right? Did you guys come up with it now? Congrats. So this is good. This is how you compute an inverse. Let's figure out why you compute an inverse that way. Let's look at b. So b starts out at position 2, right? And then pi of 5 equals 2, which means that, after applying 5, b is going to go to position 5. So pi inverse has to take b and put it where? Back to 2, right? Otherwise it's not a good inverse. So pi inverse of 2. So the second element in the output has to be which element in the input? 5, right? Otherwise, this wouldn't undo the effect of pi. So if you write out our permutation and split the index and the value, we get the inverse permutation. So you can compute that with a couple of lines of Python code.

OK. So this lets us now compute some of the permutations there. So we only have to type up two, three, four, or something like that. It's a bit nicer and you guys don't have to type up any, because we gave them to you. Aren't we nice? All right. Any questions about permutations or Rubik's cubes?

AUDIENCE: Wait, so to get high inverse based off those numbers, you just-- well, it's the same thing. Wait, but 5 got flipped. Or you think the maybe if the inverse of that matrix then?

PROFESSOR: So you take these two and you flip them, right?

AUDIENCE: [INAUDIBLE].

PROFESSOR: And then you keep these bound, but you sort them. 2 5, 3 1, 4, no-- 3 5 1 2 4.

AUDIENCE: Oh they're still based off the top row?

AUDIENCE: Yeah.

AUDIENCE: Oh, OK.

AUDIENCE: Wait, how did he get the second one?

AUDIENCE: Based off the top row.

PROFESSOR: So you sort this one, but you keep the bonds, so 3 ends up here and it gives it 1, 4 ends up here and it gives us 2. OK. So this make sense? Good. We're remembering that, that's good.

Now let's talk about a problem. So this is a Rubik's cube, this is how it works. Hopefully we will remember how to solve game. Let's try to solve StarCraft Well, of course we're going to cheat. We're going to solve a simpler version of StarCraft because it turns out you can't really solve StarCraft with the computers that we have now.

So let's make a few simplifying assumptions. Who play StarCraft, by the way? Two? OK. So you guys, please pay attention to the simplifying assumptions, because this isn't real StarCraft. It's a lot easier, so that we can build a problem out of it. All right. So we're going to play with a race called Zerg. And so the idea of StarCraft is you build an army and then you take this army and you destroy your opponent, nice and simple. Good old violent games. So we're going to look at the build order part of the game. So the build order is the opening strategy and it takes care of the building process, so it's the strategy that says, what are you going to build and when, in order to come up with an army as quickly as possible? So the main goal is to amass an army quickly and then take that army and destroy the opponent. And yes, I know, not real life StarCraft, but let's go with it.

OK, so these are the rules in our toy StarCraft And the Zerg is started with a building called the hatchery. And out of a hatchery-- the currency of the game, by the way, is minerals. We'll write that down as dollars. So out of a hatchery you can build a drone by spending 50 minerals. You can build an overlord by spending 100, or you can build a Zergling by spending 50. All right, what do each of these do? A drone harvests minerals for you, so it's a worker. So, for every drone that you have, you get eight minerals a second. So drone gets you eight minerals a second. As Zergling is your attack unit, so this is the guy that you want to build in the end. So this destroys your opponents. I don't know how to make a nice icon for that, so big smiley green. This makes it happy. All right. Overlords help you control your units, so drones and Zerglings are units. You can't build-- for every eight units that you build, you have to have an overlord. Otherwise, you can't build more units until you build overlords. So if you have eight drones, you can't build a ninth on one you build an overlord. If you have eight drones and these eight Zerglings, you have to build a second overlord to be able to build more units. So one overlord can help you control eight units. In order to build a Zergling, which is what you want to do in the end, because these are the attack units, you need to spend $50, and you also need to have one building called a spawning pool. You what?

All right. So how do you build that spawning pool? Drones can work for you, but they can also transform into buildings. So a drone mutates into a building, and after it mutates, it doesn't work for you anymore, by the way, so no more minerals. So a drone can mutate into the spawning pool for 200 minerals, I think. It can mutate into a hatchery. That's the same thing that you have up here for, how much is it? $450.

And it can mutate into an evolution chamber that we'll talk about soon, for $400. OK. Why do we care about having more than one hatchery? One hatchery can build one unit per second, no more than that. So once you have a lot of drones and you're making a lot of money, you need to have a lot of hatcheries so that you can spend the money. Our goal is not to amass money, it's to build units, right? So money doesn't help you, I'm going to spend it as fast as possible.

OK. An evolution chamber lets you build-- let's you upgrade research to technologies. And these technologies aren't like badges. Once you get them, you keep them. So you can research an attack upgrade and you can research a defense upgrade. They're both 1,000 minerals, but you only have to do it once. Why do I care about these? They decide how strong your Zerglings are. So, given a Zergling, at the beginning of the game, you don't have the attack upgrade, and you don't have a defense upgrade. So the Zerglings attack is 1. If you research the attack upgrade, but you haven't researched the defense one, their total power is 133. If you research the defense upgrade, but not the attack upgrade, their total power is 1.2. And after your research both of them, their total power is 2. Again, oversimplifying. So, we have to build an army that's as powerful as possible. So in the end you want to have both of these researched.

All right. Now there's one more limitation, that is, you cannot control more than 200 of these units total, because even if you have a lot of overlords, your brain isn't immensely huge, so you can only control 200 units. So we want to build these 200 units as fast as possible. And out of these 200 units, we want to have at least 150 Zerglings. If you build 200 drones, you're not going to attack your opponent with that, those aren't very useful. But you need some drones in order to make money, so you need the balance. So the goal is to get to 200 units as quickly as possible, and at least 150 Zergs. Yes? Evolution chamber. Easy, that's too much. OK. So how do we solve this?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So how do we solve this?

AUDIENCE: Watch the pros play.

PROFESSOR: Well, they're going to play real StarCraft this is toy StarCraft this is six-level 6 StarCraft so it's not going to work. But there is a simple strategy, because I haven't added one last constraint.

AUDIENCE: Lets me guess.

PROFESSOR: No. It's going to be harassing your enemy. But if you don't have to harass your enemy intuitively and build your economy first, so first build your drones and hatcheries, and then pop out Zerglings, right? So there's a greedy strategy. If you spend about an hour with a sheet of paper, you can realize what is the right order to build drones and hatcheries and overlords, so that you get to a nice big economy, and then, Bam! Build Zerglings. Once you get to 50 drones, you start build Zerglings as fast as possible. So this is a greedy strategy, you don't need to do a lot of work for that. So we want to make things more interesting. So in order to make things more interesting, the game time goes in seconds. Everything that we had here is in seconds. Every two minutes, so two minutes, which is 120 seconds. So every two minutes I want to assemble a small pack of Zerglings and send them to the enemy to harass them. And the Antonio enemy will defend themselves and they will destroy my Zerglings, but they won't be able to focus on the game very well. So I need to do this in order to have a good chance to win. If I don't do this, my strategy is invalid.

AUDIENCE: Does the enemy, does it do that to us as well?

PROFESSOR: No. We're sending them facts. We are going to have a really fast strategy, because we're computing the optimal strategy, so they're not going to have time. They're going to be fighting us off. OK, how many Zerglings do I send? I have to send enough Zerglings so that the attack power at minutes 2m is 6 times log 1 plus m. You knew there has to be some mapping there. So the reason for this is, let's look at the first two minutes. After two minutes, if we haven't researched any upgrades, we need to send this in six Zerglings. If we researched both upgrades, we only need to send in three Zerglings. If we researched one of the upgrades, we need to send in five Zerglings. So this is the total attack power, not the numbers are of Zerglings that we need to send.

AUDIENCE: And is minutes?

PROFESSOR: Actually, it's basically which attack wave you're doing, so every two minutes.

AUDIENCE: Oh, OK. So the first one is m equals zero.

PROFESSOR: And 1. That is not the one attacking right in the beginning when we don't have Zerglings.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Equals 1, n equals 2, n equals 3, so on and so forth.

AUDIENCE: How did you get that formula? OK.

PROFESSOR: Is it making your life hard?

Would that be logged base 2, or is that log base 10?

PROFESSOR: Sure, log base 2. Fine. OK, so how are we going to solve this? Intuitively, how did we solve all the game problems, all the problems recently? What are we going to do?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So the first thing we're going to build a graph, right?

AUDIENCE: Yeah.

PROFESSOR: So build a graph. The vertices are the states of the game, so a vertex shows me what state I'm in and an edge shows possible moves that I do. So what I need to keep track in the vertex, what's my state?

AUDIENCE: How much money you have, and how many units you have? And maybe how fast you're making money?

AUDIENCE: Oh, unless the states, are states-- they're not movable? Oh, I guess they could be moveable [INAUDIBLE] could do nothing, in the next second, right?

PROFESSOR: So you're saying you need to keep track of time, too?

AUDIENCE: I don't know.

PROFESSOR: OK so maybe time. What else?

AUDIENCE: Badge [INAUDIBLE] thing.

PROFESSOR: OK so upgrades. You said units, I'm going to say buildings.

AUDIENCE: Wait, is Zergling is a building?

PROFESSOR: A Zergling is a unit, so-

AUDIENCE: Overlords.

PROFESSOR: Unit, unit, unit. Building, building, building, upgrade, upgrade. So let's see how are we going to keep track of them. That's a good point. We started looking at units, we started looking at units. So how do I keep track of my units? What I need to know?

AUDIENCE: The number of drones, overlords, servlings.

PROFESSOR: Droids, drones, overlords, Zerglings. OK. How about buildings?

AUDIENCE: [INAUDIBLE] PHEC--

PROFESSOR: Almost. So I want to know the number of hatcheries, because this is how fast I can produce. But then how many spawning pools am I going to build? Good. So that was StarCraft player 1. Once you have a spawning pool, you can build Zerglings, there's no reason to build more than one. So a spawning pool is going to be like a badge. Once you build it, you have it, you're done. So it's Boolean.

AUDIENCE: Wait, what's a spawning pool again?

PROFESSOR: So a spawning pool is something that you build off a drone and once you have it, you can build Zerglings from a hatchery. So that's all it does.

AUDIENCE: Why would you want two if you can build four Zerglings?

PROFESSOR: So the spawning pool doesn't build a Zergling. See, the hatchery builds the Zerglings. So the spawning pool is just there. You need to have it to cross the edge. Otherwise you can't.

AUDIENCE: OK. I understand.

PROFESSOR: OK. So what else for buildings? Whether you have it or not, right? OK. How about upgrades?

[INAUDIBLE] a u, d u.

PROFESSOR: So two flags, a u, d u. Cool. So now we have the thorny issues of money and time. Let's say that we're going to have some sort of approximation that's going to allow us to not keep track of money, because money is-- you can have a lot of money, so that will give us an explosion of states. So we're going to assume that we're spending money as fast as possible, because that's what you usually want to do.

AUDIENCE: So should your money always be zero then?

PROFESSOR: Almost. It's very close to that. So you start off with no money. You accumulate money, then you're going to issue some build orders and build something. And your money's going to drop, maybe to zero, maybe to almost zero. Then you wait to accumulate more money, issue another build order. Wait to accumulate more money, hopefully it goes faster now. Issue another build order. We're going to approximate that every time we issue a build order, so every time we tell our things to build something, the money will drop to zero. So that doesn't mean you can only build one thing and then your money drops to zero. If you wait for a few seconds and then you have two hatcheries and tell both of them to build drones, you build two drones and then your money goes to zero. So that means that if we model our states carefully, we don't have to keep track of money.

AUDIENCE: You should probably keep track of time.

AUDIENCE: Actually keep track of the rate you're gathering though, right?

PROFESSOR: So what's the rate that you're gathering?

AUDIENCE: It's the number of--

AUDIENCE: Oh [INAUDIBLE] drones, yeah. [INAUDIBLE], OK.

PROFESSOR: So this multiplied by 8%.

AUDIENCE: So we should keep track of time.

PROFESSOR: OK. So how would we solve it if we keep track of time?

AUDIENCE: You multiply

AUDIENCE: Can you do that layer thing again? Where is that?

PROFESSOR: So you're going to have-- so what's a layer?

AUDIENCE: Actually, never mind.

AUDIENCE: I want to know what a layer is.

PROFESSOR: Sorry?

AUDIENCE: I want to know what a layer is.

PROFESSOR: So last time we-- we solved problems by saying that, so we had the graph of streets. We had a highway graph. And that was a 2D graph. And we made it into a 3D graph by adding time as a third dimension. So we had layers, where each layer was, said that you're at a point in the graph at a certain point in time. So there's a layer 4 times 0, layer 4 time 1, so on and so forth. And you start at time 0 and kept going up the graph.

AUDIENCE: You have a flag that says, I've just purchased something? Because that way then you'll know that you've completely exhausted your-- that maybe that will tell you that you can't buy anything for awhile, right? Because if we're not keeping track of money then we should know [INAUDIBLE].

PROFESSOR: OK. So that's a good point. So if I have-- so suppose here I have to wait for five seconds to get enough money to do a move. If I keep track of time in my state and I have five things here. I also have to keep track of money, right? I have to know how much money I'm getting, so that I know where I'm buying things. On the other hand, in this case, if I keep track of both time and money as states, I can implement a precise strategy. I don't need this approximation. If I do that, then what search algorithm would I use? So if I have time in states, so each vertex in my note says that I get from some states to some other state in one unit of time. What algorithm do I use to search for my strategy? Yeah. Why do I use BFS?

AUDIENCE: Because every edge has the same weight.

PROFESSOR: Every edge has the same weight, right? So if we keep track of time, then we-- we keep track of time, we keep track of money, we use BFS, we have a solution. So. Let's try to not do it, either of these. If I don't want to keep track of time and I don't want to keep track of money, what would an edge be? If I want to say that, hey, this is a state and this is a state. What should be the edge connecting them.

AUDIENCE: What other item you got is.

AUDIENCE: Is that after a second?

PROFESSOR: So the distance between these won't be a second, right? If I have to wait for five seconds to accumulate enough to issue this build order that would get me the state then--

AUDIENCE: The time you have to wait?

PROFESSOR: So that's what I would put on an edge. So this way I don't have to keep track of money, I don't have to keep track of time. So the graph is smaller. So I can look into bigger graphs. If I do this, what algorithm do I use? To find my strategy?

AUDIENCE: [INAUDIBLE].

[INTERPOSING VOICES]

PROFESSOR: So no time, no money. And [? extra ?]. So this looks reasonably easy, right? I mean, if you want to get from one state to another, you see what you'd have to build, you see how much it costs, and you see how much you have to wait to accumulate those resources. Except there's one glitch. How do we deal with this? How do we make sure that we do our attacks on time and how do we know that we're-- how do we keep track of them?

AUDIENCE: What is this again representing?

PROFESSOR: So this is that every two minutes you have to have a number of Zerglings that you're sending to attack the enemy. And the number of Zerglings that you need to have depends on the time and it depends on the upgrades that you got. So how do we keep track of this?

AUDIENCE: If the edges are a function of time, then just know that at radius 120-- at radius 120 over 5, then you'd have to send more Zerglings.

PROFESSOR: So what if not all the edges have the same weight. So let's look at how a graph looks like for a little bit. So we're going to have an initial state, say one hatchery and six drones and one overlord. And we want to build a new drone.

Say we want to build a new drone. A drone costs 50 minerals. We have six drones. Each drone gets us eight minerals a second, so in one second we're going to get 48 minerals. So bummer, we have to wait for two seconds to build one drone. And now we're going to be in the state of one hatchery, seven drones, one overlord.

If you said we want to build a hatchery-- then we're going to need to accumulate 400 or 450. Oh, we have to accumulate 450, using six drones, 48 minerals, so I guess nine seconds. So we're going to have to wait for nine seconds to get that money. And then when we build, we're going to spend the drone building, because it's going to turn into a hatchery. And then we're going to end up with two hatcheries, five drones, and one overlord. So edges have different weights. So I can't look at the radius in terms of number of his edges.

AUDIENCE: Everything builds instantaneously.

PROFESSOR: Yeah, sure.

AUDIENCE: You could keep track of just the total distance from whatever starting build you are?

AUDIENCE: Actually, if the distance from the origin to where you're at is a multiple of 120, then you should have another state that you have to [INAUDIBLE].

AUDIENCE: Assuming you have any drones left, hopefully you do.

PROFESSOR: So that's modeling. One way of doing it is sure, model every round of the attack as one, every round of harassment is one there in the graph, and we can do it that way. Is that what you're saying? So you can't-- once you're at 120 seconds, you have to do an attack and then you get to the next layer in the graph.

AUDIENCE: That makes a lot of sense.

AUDIENCE: Of course that's what we were thinking.

PROFESSOR: How about let's not do that. How about let's represent it without layers? So that's good. You're getting closer to the solution. It is much better than representing every time. Write the number of layers this time over 120, it's a huge improvement. Well, we can do it without any-- we can do it without any time notion whatsoever. Let's see, can I let you think about it? Oh, I can let you think about it for 30 seconds.

So there's a key insight here that--

AUDIENCE: Ooh.

PROFESSOR: Yeah.

AUDIENCE: [INAUDIBLE] but if you figure how much time has elapsed, based on the items we have and our starting state, right? Because if we have two hatcheries and we started with one hatchery, we know that at least nine seconds has elapsed.

PROFESSOR: Well, so I like the first part of what you said, the second part is too complicated. But we can know what time we're at. Why?

AUDIENCE: Because it takes a certain amount of time to achieve everything, each thing.

AUDIENCE: Are we assuming that we take the shortest path to get to each state that we're at?

PROFESSOR: So we're using the extra. How does the extra work? You have a queue, a priority queue. You extract a node, you look at the neighbors. When you extract the node, do you know the distance to that node?

AUDIENCE: Can you explain? You ask?

PROFESSOR: I'm asking.

AUDIENCE: I know, but like the node asks and says, how far away are you?

PROFESSOR: So you need to keep that in the priority queue. So the extra-- the main invariant is that once you pull out the node, you're not going to find any shorter path to that node. So you already know the shortest path. That how you sort the nodes in the priority queue. So when we're at a configuration-- Say we're at some configuration here, two hatches, six drones, and five Zerglings. We already know that, say we're at time 119 seconds.

If what I want to do next is build another hatchery, then I have to wait, what, six drones, same thing as before. I have to wait for nine seconds, right? So the edge would look like this. And then we're going to get to three hatcheries, five drones, five Zerglings. This crosses a one, this crosses a two-minute boundary, right? So when I cross this boundary, I'd better have enough Zerglings to do an attack. Do I have enough Zerglings to do an attack?

No. So is this a valid move? Is this the valid edge in our graph? It's not. So when you're at a node, you already know the time that you need to get to that node, so when you generate the neighbors, you can check for each edge to see it crosses that 120 second game. Now let's assume it does. Let's see how this works and this is the last thing we're doing. So suppose you have two hatcheries, six drones, and eight Zerglings. And you're going to build that same hatchery. So you're going to go past this 120 second mark. What state are you going to end up in? How many hatcheries?

AUDIENCE: Three.

PROFESSOR: OK. How many drones.

AUDIENCE: Five.

PROFESSOR: How many Zerglings.

AUDIENCE: Two.

PROFESSOR: Yes. So key elements, two Zerglings, because this edge crosses the 120-second harassment boundary, so you have to send in six Zerglings and you're going to lose them. So we're taking advantage of how the extra works to sort of generate our edges on the fly. So we wouldn't be able to regenerate this graph, even if we wanted to have to. We have to work with the implicit representation and, as we learn the distances to the nodes, the minimum distances, we also learned the edges that we have coming out of those nodes.

AUDIENCE: So it's kinda of like this graph that's growing and we're like as it gets to a certain point we kill a bunch of paths and then it keeps on going.

PROFESSOR: Yeah. Doesn't this makes sense? So the good news is that this is like we're training you to run a 10-mile marathon, so you can run 50 meters for the test. Let's still try to get it, because I think it's a cool problem.