PROFESSOR: One of the things that you should probably have noticed is as we're moving in the terms, the problem sets are getting less well defined. line And I've seen a lot of email traffic of the nature of what should we do with this, what should we do with that? For example, suppose the computer player runs out of time. Or the person runs out of time playing the game. Should it stop right away? Should it just give them zero score? That's left out of the problem set. In part because one of the things were trying to accomplish is to have you folks start noticing ambiguities in problem statements. Because that's life in computing. And so this is not like a math problem set or a physics problem set. Or, like a high school physics lab, where we all know what the answer should be, and you could fake your lab results anyway. These are things where you're going to have to kind of figure it out. And for most of these things, all we ask is that you do something reasonable, and that you describe what it is you're doing. So I don't much care, for example, whether you give the human players zero points for playing after the time runs out. Or you say you're done when the time runs out. Any of that -- thank you, Sheila -- is ok with me. Whatever. What I don't want your program to do is crash when that happens. Or run forever. Figure out something reasonable and do it. And again, we'll see this as an increasing trend as we work our way through the term.

The exception will be the next problem set, which will come out Friday. Because that's not a programming problem. It's a problem, as you'll see, designed to give you some practice at dealing with some of the, dare I say, more theoretical concepts we've covered in class. Like algorithmic complexity. That are not readily dealt with in a prograing problem. It also deals with issues of some of the subtleties of things like aliasing. So there's no programming. And in fact, we're not even going to ask you to hand it in. It's a problem set where we've worked pretty hard to write some problems that we think will provide you with a good learning experience. And you should just do it to learn the material. We'll help you, if you need -- to see the TAs because you can't do them, by all means make sure you get some help. So I'm not suggesting that it's an optional problem set that you shouldn't do. Because you will come to regret it if you don't do it. But we're not going to grade it. And since we're not going to grade it, it seems kind of unfair to ask you to hand it in. So it's a short problem set, but make sure you know how to do those problems.

OK. Today, for the rest of the lecture, we're going to take a break from the topic of algorithms, and computation, and things of the sort. And do something pretty pragmatic. And we're going to talk briefly about testing. And at considerable length about debugging. I have tried to give this lecture at the beginning of the term, at the end of the term. Now I'm trying it kind of a third of the way, or middle of the term. I never know the right time to give it. These are sort of pragmatic hints that are useful. I suspect all of you have found that debugging can be a frustrating activity. My hope is that at this point, you've experienced enough frustration that the kind of pragmatic hints I'm going to talk about will not be, "yeah sure, of course." But they'll actually make sense to you. We'll see.

OK. In a perfect world, the weather would always be like it's been this week. The M in MIT would stand for Maui, instead of Massachusetts. Quantum physics would be easier to understand. All the supreme court justices would share our social values. And most importantly, our programs would work the first time we typed them. By now you may have noticed that we do not live in an ideal world. At least one of those things I mentioned is not true. I'm only going to address the last one. Why our programs don't work. And I will leave the supreme court up to the rest of you. There is an election coming up.

Alright, First a few definitions. Things I want to make sure we all understand what they mean. Validation is a process. And I want to emphasize the word process. Designed to uncover problems and increase confidence that our program does what we think it's intended to do. I want to emphasize that it will increase our confidence, but we can never really be sure we've got it nailed. And so it's a process that goes on and on. And I also want to emphasize that a big piece of it is to uncover problems. So we need to have a method not designed to give us unwarranted confidence. But in fact warranted confidence in our programs. It's typically a combination of two things. Testing and reasoning. Testing, we run our program on some set of inputs. And check the answers, and say yeah, that's what we expected. But it also involves reasoning. About why that's an appropriate set of inputs to test it on it. Have we tested it on enough inputs? Maybe just reading the code and studying it and convincing ourselves that works. So we do both of those as part of the validation process. And we'll talk about all of this as we go along.

Debugging is a different process. And that's basically the process of ascertaining why the program is not working. Why it's failing to work properly. So validation says whoops, it's not working. And now we try and figure out why not. And then of course, once we figure out why not, we try and fix it. but today I'm going to emphasize not how do you fix it, but how do you find out what's wrong. Usually when you know why it's not working, it's obvious what you have to do to make it work.

There are two aspects of it. Thus far, the problem sets have mostly focused on function. Does it exhibit the functional behavior? Does it give you the answer that you expected it to give? Often, in practical problems, you'll spend just as much time doing performance debugging. Why is it slow? Why is it not getting the answer as fast as I want it to? And in fact, in a lot of industry -- for example, if you're working on building a computer game, you'll discover that in fact the people working the game will spend more time on performance debugging than on getting it to do the right thing. Trying to make it do it fast enough. Or get to run on the right processor.

Some other terms we've talked about is defensive programming. And we've been weaving that pretty consistently throughout the term. And that's basically writing your programs in such a way that it will facilitate both validation and debugging. And we've talked about a lot of ways we do that. One of the most important things we do is we use assert statements so that we catch problems early. We write specifications of our functions. We modularize things. And we'll come back to this. As every time we introduce a new programming concept, we'll relate it back, as we have been doing consistently, to defensive programming.

So one of the things I want you to notice here is that testing and debugging are not the same thing. When we test, we compare an input output pair to a specification. When we debug, we study the events that led to an error. I'll return to testing later in the term. But I do want to make a couple of quick remarks with very broad strokes.

There are basically two classes of testing. There's unit testing, where we validate each piece of the program independently. Thus far, for us it's been testing individual functions. Later in the term, we'll talk about unit testing of classes. The other kind of testing is integration testing. Where we put our whole program together, and we say does the whole thing work? People tend to want to rush in and do this right away. That's usually a big mistake. Because usually it doesn't work. And so one of the things that I think is very important is to always begin by testing each unit. So before I try and run my program, I test each part of it independently. And that's because it's easier to test small things than big things. And it's easier to debug small things than big things. Eventually, it's a big program, I run it. It never works the first time if it's a big program. And I end up going back and doing unit testing anyway, to try and figure out why it doesn't work. So over the years, I've just convinced myself I might as well start where I'm going to end up.

What's so hard about testing? Why is testing always a challenge? Well, you could just try it and see if it works, right? That's what testing is all about. So we could look at something small. Just write a program to find the max of x and y. Where x and y are floats. However many quotes I need. Well, just see if it works. Let's test it in all possible combinations of x and y and see if we get the right answer. Well, as Carl Sagan would have said, there are billions and billions of tests we would have to do. Or maybe it's billions and billions and billions. Pretty impractical. And it's hard to imagine a simpler program than this. So we very quickly realize that exhaustive testing is just never feasible for an interesting program.

So as we look at testing, what we have to find is what's called a test suite. A test suite is small enough so that we can test it in a reasonable amount of time. But also large enough to give us some confidence. Later in the term, we'll spend part of a lecture talking about, how do we find such a test suite? A test suite that will make us feel good about things. For now, I just want you to be aware that you're always doing this balancing act.

So let's assume we've run our test suite. And, sad to say, at least one of our tests produced an output that we were unhappy with. It took it too long to generate the output. Or more likely, it was just the wrong output. That gets us to debugging.

So a word about debugging. Where did the name come from? Well here's a fun story, at least. This was one of the very first recorded bugs in the history of computation. Recorded September 9th, 1947, in case you're interested. This was the lab book of Grace Murray Hopper. Later Admiral Grace Murray Hopper. The first female admiral in the U.S. navy. Who was also one of the word's first programmers. So she was trying to write this program, and it didn't work. It was a complicated program. It was computing the arctan. So you can imagine, right? You had a whole team of people trying to figure out how to do arctans. Times were different in those days. And they tried to run it, and it ran a long time. Then it basically stopped. Then they started the cosine tape. That didn't work. Well they couldn't figure out what was wrong. And they spent a long time trying to debug the program. They didn't apparently call it debugging. And then they found the problem. In relay number 70, a moth had been trapped. And the relay had closed on the poor creature, crushing it to death. The defense department didn't care about the loss of a moth. But they did care about the fact that the relay was now stuck. It didn't work. They removed the moth, and the program worked. And you'll see at the bottom, it says the first actual case of a bug being found. And they were very proud of themselves. Now it's a wonderful story, and it is true. After all, Grace wouldn't have lied. But it's not the first use of the term "bug." And as you'll see by your handout, I've attempted tend to trace it. And the first one I could find was in 1896. In a handbook on electricity.

Alright. Now debugging is a learned skill. Nobody does it well instinctively. And a large part of being a good programmer, or learning to be a good programmer, is learning how to debug. And it's one of these things where it's harder. It's slow, slow, and you suddenly have an epiphany. And you now get the hang of it. And I'm hoping that today's lecture will help you learn faster. The nice thing, is once you learn to debug programs, you will discover it's a transferable skill. And you can use it to debug other complex systems. So for example, a laboratory experience. Why isn't this experiment working? There's a lecture I've given several times at hospitals, to doctors, on doing diagnosis of complex multi illnesses. And I go through it, and almost the same kind of stuff I'm going to talk to you about, about debugging. Explaining that it's really a process of engineering.

So I want to start by disabusing you of a couple of myths about bugs. So myth one is that bugs crawl into programs. Well it may have been true in the old days, when bugs flew or crawled into relays. It's not true now. If there is a bug in the program, it's there for only one reason. You put it there. i.e. you made a mistake. So we like to call them bugs, because it doesn't make us feel stupid. But in fact, a better word would be mistake.

Another myth is that the bugs breed. They do not. If there are multiple bugs in the program, it's because you made multiple mistakes. Not because you made one or two and they mated and produced many more bugs. It doesn't work that way. That's a good thing. Typically, even though they don't breed, there are many bugs. And keep in mind that the goal of debugging is not to eliminate one bug. The goal is to move towards a bug free program. I emphasize this because it often leads to a different debugging strategy. People can get hung up on sort of hunting these things down, and stamping them out, one at a time. And it's a little bit like playing Whack-a-Mole. Right? They keep jumping up at you. So the goal is to figure out a way to stamp them all out.

Now, should you be proud when you find a bug? I've had graduate students come to me and say I found a bug in my program. And they're really proud of themselves. And depending on the mood I'm in, I either congratulate them, or I say ah, you screwed up, huh? Then you had to fix it. If you find a bug, it probably means there are more of them. So you ought to be a little bit careful. The story I've heard told is you're at somebody's house for dinner, and you're sitting at the dining room table, then you hear a [BANG]. And then your hostess walks in with the turkey in a tray, and says, "I killed the last cockroach." Well it wouldn't increase my appetite, at least. So be worried about it.

For at least four decades, people have been building tools called debuggers. Things to help you find bugs. And there are some built into Idol. My personal view is most of them are not worth the trouble. The two best debugging tools are the same now that they have almost always been. And they are the print statement, and reading. There is no substitute for reading your code. Getting good at this is probably the single most important skill for debugging. And people are often resistant to that. They'd rather single step it through using Idol or something, than just read it and try and figure things out. The most important thing to remember when you're doing all of this is to be systematic. That's what distinguishes good debuggers from bad debuggers. Good debuggers have evolved a way of systematically hunting for the bugs. And what they're doing as they hunt, is they're reducing the search space. And they do that to localize the source of the problem. We've already spent a fair amount of time this semester talking about searches. Algorithms for searching. Debugging is simply a search process. When you are searching a list to see whether it has an element, you don't randomly probe the list, hoping to find whether or not it's there. You find some way of systematically going through the list. Yet, I often see people, when they're debugging, proceeding at what, to me, looks almost like a random fashion of looking for the bug. That is a problem that may not terminate. So you need to be careful.

So let's talk about how we go about being systematic, as we do this. So debugging starts when we find out that there exists a problem. So the first thing to do is to study the program text, and ask how could it have produced this result? So there's something subtle about the way I've worded this. I didn't ask, why didn't it produce the result I wanted it to produce? Which is sort of the question we'd immediately like to ask. Instead, I asked why did it produce the result it did. So I'm not asking myself what's wrong? Or how could I make it right? I'm asking how could have done this? I didn't expect it to do this. If you understand why it did what it did, you're half way there.

The next big question you ask, is it part of a family? This gets back to the question of trying to get the program to be bug free. So for example, oh, it did this because it was aliasing, where I hadn't expected it. Or some side effect of some mutation with lists. And then I say, oh you know I've used lists all over this program. I'll bet this isn't the only place where I've made this mistake. So you say well, rather than rushing off and fixing this one bug, let me pull back and ask, is this a systematic mistake that I've made throughout the program? And if so, let's fix them all at once, rather than one at a time.

And that gets me to the final question. How to fix it. When I think about debugging, I think about it in terms of what you learned in high school as the scientific method. Actually, I should ask the question. Maybe I'm dating myself. Do they still teach the scientific method in high school? Yes, alright good. All is not lost with the American educational system. So what does the scientific method tell us to do? Well it says you first start by studying the available data. In this case, the available data are the test results. And by the way, I mean all the test results. Not just the one where it didn't work, but also the ones where it did. Because maybe the program worked on some inputs and not on others. And maybe by understanding why it worked on a and not on b, you'll get a lot of insight that you won't if you just focus on the bug. You'll also feel a little bit better knowing your program works on at least something. The other big piece of available data we have is, of course, the program text. As you the study the program text, keep in mind that you don't understand it. Because if you really did, you wouldn't have the bug. So read it with sort of a skeptical eye. You then form a hypothesis consistent with all the data. Not just some of the data, but all of the data. And then you design and run a repeatable experiment.

Now what is the thing we learned in high school about how to design these experiments? What must this experiment have the potential to do, to be a valid scientific experiment? Somebody? What's the key thing? It must have the potential to refute the hypothesis. It's not a valid experiment if it has no chance of showing that my hypothesis is flawed. Otherwise why bother running it? So it has to have that. Typically it's nice if it can have useful intermediate results. Not just one answer at the end. So we can sort of check the progress of the code. And we must know what the result is supposed to be. Typically when you run an experiment, you say, and I think the answer will be x. If it's not x, you've refuted the hypothesis. This is the place where people typically slip up in debugging. They don't think in advance what they expect the result to be. And therefore, they are not systematic about interpreting the results. So when someone comes to me, and they're about to do a test, I ask them, what do you expect your program to do? And if they can't answer that question, I say well, before you even run it, have an answer to that.

Why might repeatability be an issue? Well as we'll see later in the term, we're going to use a lot of randomness in a lot of our programs. Where we essentially do the equivalent of flipping coins or rolling dice. And so the program may do different things on different runs. We'll see a lot of that, because it's used a lot in modern computing. And so you have to figure out how to take that randomness out of the experiment. And yet get a valid test. Sometimes it can be timing. If you're running multiple processes. That's why your operating systems and your personal computers often crash for no apparent reason. Just because two things happen to, once in a while, occur at the same time. And often there's human input. And people have to type things out of it. So you want to get rid of that. And we'll talk more about this later. Particularly when we get to using randomness. About how to debug programs where random choices are being made.

Now let's think about designing the experiment itself. The goal here, there are two goals. Or more than two. One is to find the simplest input that will provoke the bug. So it's often the case that a program will run a long time, and then suddenly a bug will show up. But you don't want to have to run it a long time, every time you have a hypothesis. So you try and find a smaller input that will produce the problem. So if your word game doesn't work when the words are 12 letters long, instead of continuing to debug 12 letter hands, see if you can make it fail on a three letter hand. If you can figure out why fails on three letters instead of 12, you'll be more than half way to solving the problem. What I typically do is I start with the input that provoked the problem, and I keep making it smaller and smaller. And see if I can't get it to show up.

The other thing you want to do is find the part of the program that is most likely at fault. In both of these cases, I strongly recommend binary search. We've talked about this binary search a lot already. Again, the trick is, if you can get rid of half of the data at each shot, or half of the code at each shot., you'll quickly converge on where the problem is.

So I now want to work through an example where we can see this happening. So this is the example on the handout. I've got a little program called Silly. And it's called Silly because it's really a rather ugly program. It's certainly not the right way to write a program to do this. But it will let us illustrate a few points. So the trick, what we're going to go through here, is this whole scientific process. And see what's going on. So let's try running Silly. So this is to test whether a list is a palindrome. So we'll put one as the first element, maybe a is the second element. And one is the third element. And just return, it's done. It is a palindrome. That make sense. The list one a one reads the same from the front or from the back. So that's good. Making some progress. Let's try it again. And now let's do one, a, two. Whoops. It tells me it is a palindrome. Well, it isn't really. I have a bug. Alright. Now what do I do? Well I'm going to use binary search to see if I can't find this bug. As I go through, I'm going to try and eliminate half of the code at each step. And the way I'm going to do that is by printing intermediate values, as I go part way through the code. I'm going to try and predict what the value is going to be. And then see if, indeed, I get what I predicted. Now, as I do this, I'm going to use binary search. I'm going to start somewhere near the middle of the code. Again, a lot of times, people don't do that. And they'll test an intermediate value near the end or near the beginning. Kind of in the hope of getting there in one shot. And that's like kind of hoping that the element you're searching for is the first in the list and the last in the list. Maybe. But part of the process of being systematic is not assuming that I'm going to get a lucky guess. But not even thinking really hard at this point. But just pruning the search space. Getting rid of half at each step.

Alright. So let's start with the bisection. So we're going to choose a point about in the middle of my program. That's close to the middle. It might even be the middle. And we're going to see, well all right. The only thing I've done in this part of the program, now I'm going to go and read the code, is I've gotten the user to input a bunch of data. And built up the list corresponding to the three items that the user entered. So the only intermediate value I have here is really res. So I'm going to, just so when I'm finished I know what it is that I think I've printed. But in fact maybe I'll do even more than that here. Let me say what I think it should be. And then we'll see if it is. So I think I put in one a two, right? Or one a two? So it should be something like one, a, two. So I predicted what answer I'm expecting to get. And I've put it in my debugging code. And now I'll run it and see what we get. We'll save it. Well all right, a syntax error. This happens. And there's a syntax error. I see. Because I've got a quote in a quote. Alright I'm just going to do that.

What I expected. So what have I learned? I've learned that with high probability, the error is not in the first part of the program. So I can now ignore that. So now I have these six lines. So we'll try and go in the middle of that. See if we can find it here. And notice, by the way, that I commented out the previous debugging line, rather than got rid of it. Since I'm not sure I won't need to go back to it. So what should I look at here? Well there are a couple of interesting intermediate values here, right? There's tmp. And there's res. Never type kneeling. Right? I find something to tmp. And I need to make sure maybe I haven't messed up res. Now it would be easy to assume, don't bother looking at [UNINTELLIGIBLE]. Because the code doesn't change res. Well remember, that I started this with a bug. That means it was something I didn't understand. So I'm going to be cautious and systematic. And say let's just print them both. And see whether they're okay.

Now, let's do this. So it says tmp is two a one, and res is two a one. Well let's think it. Is this what we wanted, here? What's the basic idea behind this program? How is it attempting to work? Well what it's attempting to do, and now is when I have to stand back and form a hypothesis and think about what's going on, is it gets in the list, it reverses the list, and then sees whether the list and the reverse were identical. If so it was a palindrome, otherwise it wasn't. So I've now done this, and what do you think? Is this good or bad? Is this what I should be getting? No. What's wrong? Somebody? yeah.

STUDENT: [UNINTELLIGIBLE]

PROFESSOR: Yeah. Somehow I wanted to -- Got to work on those hands. I didn't want to change res. So, I now know that the bug has got to be between these two print statements. I'm narrowing it down. It's getting a little silly, but you know I'm going to really be persistent and just follow the rules here of binary search, rather than jumping to conclusions. Well clearly what I probably want to do here is what? Print these same two things. See what I get. Whoops. I have to, of course, do that. Otherwise it just tells me that Silly happens to be a function. Alright. How do I feel about this result? I feel pretty good here. Right? The idea was to make a copy of res and temp. And sure enough, they're both the same. What I expected them to be. So I know the bug is not above. Now I'm really honing in. I now know it's got to be between these two statements. So let's put it there. Aha. It's gone wrong. So now I've narrowed the bug down to one place. I know exactly which statement it's in. So something has happened there that wasn't what I expected. Who wants to tell me what that bug is? Yeah?

STUDENT: [UNINTELLIGIBLE].

PROFESSOR: Right. Bad throw, good catch. So this is a classic error. I've not made a copy of the list. I've got an alias of the list. This was the thing that tripped up many of you on the quiz. And really what I should have done is this. Now we'll try it. Ha. It's not a palindrome. So small silly little exercise, but I'm hoping that you've sort of seen how by being patient. Patience is an important part of the debugging process. I have not rushed. I've calmly and slowly narrowed the search. Found where the statement is, and then fixed it. And now I'm going to go hunt through the rest of my code to look for places where I used assignment, when I should have use cloning as part of the assignment. The bug, the family here, is failure to clone when I should have cloned. Thursday we'll talk a little bit more about what to do once we've found the bug, and then back to algorithms.