MIT OpenCourseWare
http://ocw.mit.edu

6.00 Introduction to Computer Science and Programming, Fall 2008

Please use the following citation format:

> Eric Grimson and John Guttag, *6.00 Introduction to Computer Science and Programming, Fall 2008*. (Massachusetts Institute of Technology: MIT OpenCourseWare). http://ocw.mit.edu (accessed MM DD, YYYY). License: Creative Commons Attribution-Noncommercial-Share Alike.

Note: Please use the actual date you accessed this material in your citation.

For more information about citing these materials or our Terms of Use, visit:
http://ocw.mit.edu/terms

MIT OpenCourseWare
http://ocw.mit.edu

6.00 Introduction to Computer Science and Programming, Fall 2008
Transcript – Lecture 7

ANNOUNCER: Open content is provided under a creative commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation, or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu .

PROFESSOR JOHN GUTTAG: OK. I finished up last time talking about lists. And I pointed out that lists are mutable, showed you some examples of mutation. We can look at it here; we looked at append, which added things to lists, we looked at delete, deleting things from a list. You can also assign to a list, or to an element of a list. So ivy sub 1, for example, could be assigned minus 15, and that will actually mutate the list.

So heretofore, when we wrote assignment, what we always meant, was changing the binding of a variable to a different object. Here, we are overloading the notation to say, no, no, ivys is still bound to the same object, but an element of ivys is bound to a different object.

If you think about it, that makes sense, because when we have a list, what a list is, is a sequence of objects. And what this says is, is the object named by the expression ivys sub 1, is now bound to the object, if you will, named by the constant minus 15. So we can watch this run here. Idle can-- that's exciting. I hadn't expected that answer. All right, your question.

STUDENT: [INAUDIBLE] four elements to ivys, and you tell it to change the fifth element of ivys to negative 15, will it add it or [INAUDIBLE]

PROFESSOR JOHN GUTTAG: Well, I'll tell you how ol-- let's answer that the easy way. We'll start up a shell and we'll try it. All right, we'll just get out of what we were doing here.

And so, we now have some things, so we, for example, have ivys, I can print ivys , and it's only got three elements but your question probably is just as good for adding the fourth as adding the fifth, so what would happen if we say ivys sub 3-- because that of course is the fourth element, right? Let's find out.

OK. Because what that does is, it's changing the binding of the name ivys, in this case, sub 1. What it looked at here, with the name ivys sub 3, and said that name doesn't-- isn't bound, right? That isn't there. So it couldn't do it, so instead that's what append is for, is to stick things on to the end of the list. But a very good question.

So we can see what we did here, and, but of course I can now, if I choose, say something like, ivys sub 1 is assigned minus 15, and now if I print ivys, there it is.

And again, this points out something I wanted to me-- I mentioned last time, list can be heterogeneous, in the sense that the elements can be multiple different types. As you see here, some of the elements are strings and some of the elements are integers.

Let's look at another example. Let's suppose, we'll start with the list, I'll call it l 1. This, by the way, is a really bad thing I just did. What was-- what's really bad about calling a list l 1?

STUDENT: [INAUDIBLE]

PROFESSOR JOHN GUTTAG: Is it l 1, or is it 11, or is it l l? It's a bad habit to get into when you write programs, so I never use lowercase L except when I'm spelling the word where it's obvious, because otherwise I get all sorts of crazy things going on.

All right, so let's make it the list 123. All right? Now, I'll say L 2 equals L 1. Now I'll print L 2. Kind of what you'd guess, but here's the interesting question: if I say L 1 is assigned 0, L 1 sub is assigned 4, I'll print L 1.

That's what you expect, but what's going to happen if I print L 2? 423 as well, and that's because what happened is I had this model, which we looked at last time, where I had the list L 1, which was bound to an object, and then the assignment L 2 gets L 1, bound the name L 2 to the same object, so when I mutated this object, which I reached through the name L 1 to make that 4, since this name was bound to the same object, when I print it, I got 423.

So that's the key thing to-- to realize; that what the assignment did was have two separate paths to the same object. So I could get to that object either through this path or through that path, it didn't matter which path I use to modify it, I would see it when I looked at the other. Yes.

STUDENT: [INAUDIBLE]

PROFESSOR JOHN GUTTAG: So the question, if I said a is assigned 2, b is assigned a, and then a is assigned 3. Is that your question?

So the question is, a is assigned 1, b is assigned a, a is assigned 2, and then if I print b, I'll get 1. Because these are not mutable, this is going to be assigned to an object in the store, so we'll draw the picture over here, that we had initially a is bound to an object with 1 in it, and then b got bound to the same object, but then when I did the assignment, what that did was it broke this connection, and now had a assigned to a different object, with the number, in this case, 2 in it. Whereas the list assignment you see here did not rebind the object l 1, it changed this. OK?

Now formally I could have had this pointing off to another object containing 4, but that just seemed excessive, right? But you see the difference. Great question, and a very important thing to understand, and that's why I'm belaboring this point, since this is where people tend to get pretty confused, and this is why mutation is very important to understand. Yeah.

STUDENT: [UNINTELLIGIBLE]

PROFESSOR JOHN GUTTAG: I'm just assuming it'll be a great question.

STUDENT: [INAUDIBLE]

PROFESSOR JOHN GUTTAG: Exactly. So if-- very good question-- so, for example, we can just do it here.

The question was, suppose I now type L 1 equals the empty list. I can print L 1, and I can print L 2, because again, that's analogous to this example, where I just swung the binding of the identifier. So this is important, it's a little bit subtle, but if you don't really understand this deeply, you'll find yourself getting confused a lot. All right?

OK. Let me move on, and I want to talk about one more type. By the way, if you look at the handout from last time, you'll see that there's some other examples of mutation, including a function that does mutation. It's kind of interesting, but I don't think we need-- think we've probably done enough here that I hope it now make sense.

That one type I want to talk about still is dictionaries. Like lists, dictionaries are mutable, like lists, they can be heterogeneous, but unlike lists, they're not ordered. The elements in them don't have an order, and furthermore, we have generalized the indexing. So lists and strings, we can only get at elements by numbers, by integers, really.

Here what we use is, think of every element of the dictionary as a key value pair, where the keys are used as the indices. So we can have an example, let's look at it. So, if you look at the function show dics here, you'll see I've declared a variable called e to f, ah, think of that as English to French, and I've defined a dictionary to do translations. And so, we see that the string one corresponds the-- the key one corresponds to the value un the key soccer corresponds to the French word football, et cetera. It's kind of bizarre, but the French call soccer football. And then I can index in it. So if I print e to f of soccer, it will print the string football.

So you can imagine that this is a very powerful mechanism. So let's look what happens when I run-- start to run this.

All right. So, it says not defined-- and why did it say not defined, there's an interesting question. Let's just make sure we get this right, and we start the show up again-- All right, so, I run it, and sure enough, it shows football. What happens if I go e to f of 0? I get a key error. Because, remember, these things are not ordered. There is no 0th element. is not a key of this particular object.

Now I could have made a key, keys don't have to be strings, but as it happened, I didn't. So let's comment that out, so we don't get stuck again. Where we were before, I've printed it here, you might be a little surprised of the order. Why is soccer first? Because the order of this doesn't matter. That's why it's using set braces, so don't worry about that.

The next thing I'm doing is-- so that's that, and then-- I'm now going to create another one, n to s, for numbers to strings, where my keys are numbers, in this case the number 1 corresponds to the word one, and interestingly enough, I'm also going to have the word one corresponding to the number 1. I can use anything I want for

keys, I can use anything I want for values. And now if we look at this, we see, I can get this.

All right. So these are extremely valuable. I can do lots of things with these, and you'll see that as we get to future assignments, we'll make heavy use of dictionaries. Yeah. Question.

STUDENT: [INAUDIBLE]

PROFESSOR JOHN GUTTAG: You can, but you don't know what order you'll get them in. What you can do is you can iterate keys, which gives you the keys in the dictionary, and then you can choose them, but there's no guarantee in the order in which you get keys.

Now you might wonder, why do we have dictionaries? It would be pretty easy to implement them with lists, because you could have a list where each element of the list was a key value pair, and if I wanted to find the value corresponding to a key, I could say for e in the list, if the first element of e is the key, then I get the value, otherwise I look at the next element in the list.

So adding dictionaries, as Professor Grimson said with so many other things, doesn't give you any more computational power. It gives you a lot of expressive convenience, you can write the programs much more cleanly, but most importantly, it's fast.

Because if you did what I suggested with the list, the time to look up the key would be linear in the length of the list. You'd have to look at each element until you found the key.

Dictionaries are implemented using a magic technique called hashing, which we'll look at a little bit later in the term, which allows us to retrieve keys in constant time. So it doesn't matter how big the dictionary is, you can instantaneously retrieve the value associated with the key. Extremely powerful.

Not in the next problems set but in the problem set after that, we'll be exploiting that facility of dictionaries. All right. Any questions about this? If not, I will turn the podium over to Professor Grimson.

PROFESSOR ERIC GRIMSON: I've stolen it. This is like tag team wrestling, right? Professor Guttag has you on the ropes, I get to finish you off. Try this again.

OK. We wanted to finish up that section, we're now going to start on a new section, and I want to try and do one and a half things in the remaining time. I'm going to introduce one topic that we're going to deal with fairly quickly, and then we tackle the second topic, it's going to start today, and we're going to carry on. So let me tell the two things I want to do.

I want to talk a little bit about how you use the things we've been building in terms of functions to help you structure and organize your code. It's a valuable tool that you want to have as a programmer.

And then we're going to turn to the question of efficiency. How do we measure efficiency of algorithms? Which is going to be a really important thing that we want

to deal with, and we'll start it today, it's undoubtedly going to take us a couple more lectures to finish it off.

Right, so how do you use the idea of functions to organize code? We've been doing it implicitly, ever since we introduced functions. I want to make it a little more explicit, and I want to show you a tool for doing that. And I think the easy way to do is-- is to do it with an example.

So let's take a really simple example. I want to compute the length of the hypotenuse of a right triangle. And yeah, I know you know how to do it, but let's think about what might happen if I wanted to do that.

And in particular, if I think about that problem-- actually I want to do this-- if I think about that problem, I'm going to write a little piece of pseudo code. Just to think about how I would break that problem up.

Pseudo code. Now, you're all linguistic majors, pseudo means false, this sounds like code that ain't going to run, and that's not the intent of the term. When I say pseudo code, what I mean is, I'm going to write a description of the steps, but not in a particular programming language. I'm going to simply write a description of what do I want to do.

So if I were to solve this problem, here's the way I would do it. I would say, first thing I want to do, is I want to input a value for the base as a float. Need to get the base in. Second thing I want to do, I need to get the height, so I'm going to input a value for the height, also as a float, a floating point.

OK. I get the two values in, what do I need to do, well, you sort of know that, right? I want to then do, I need to find the square root-- b squared plus h squared, right? The base plus the height, that's the thing I want for the hypotenuse-- and I'm going to save that as a float in hyp, for hypotenuse. And then finally I need to print something out, using the value in hyp.

OK. Whoop-dee-doo, right? Come on. We know how to do this. But notice what I did.

First of all, I've used the notion of modularity. I've listed a sequence of modules, the things that I want to do.

Second thing to notice, is that little piece of pseudo code is telling me things about values. I need to have a float. I need to have another float here, it's giving me some information.

Third thing to notice is, there's a flow of control. The order which these things are going to happen.

And the fourth thing to notice is, I've used abstraction. I've said nothing about how I'm going to make square root. I'm using it as an abstraction, saying I'm going to have square root from somewhere, maybe I'll build it myself, maybe somebody gives it to me as part of a library, so I'm burying the details inside of it.

I know this is a simple example, but when you mature as a programmer, one of the first things you should do when you sit down to tackle some problem is write something like this pseudo code. I know Professor Guttag does it all the time.

I know, for a lot of you, it's like, OK, I got a heavy problem. Let's see, def Foobar open paren, a bunch of parameters.

Wrong way to start. Start by thinking about what are the sequences.

This also, by the way, in some sense, gives me the beginnings of my comments for what the structure of my code is going to be.

OK. If we do that, if you look at the handout then, I can now start implementing this. I wanted to show you that, so, first thing I'm going to do is say, all right, I know I'm going to need square root in here, so I'm going to, in fact, import math.

That's a little different from other import statements. This says I'm going to get the entire math library and bring it in so I can use it. And then, what's the first thing I wanted to do? I need to get a value for base as a float. Well OK, and that sounds like I'm going to need to do input of something, you can see that statement there, it's-- got the wrong glasses on but right there-- I'm going to do an input with a little message, and I'm going to store it in base.

But here's where I'm going to practice a little bit of defensive programming. I can't rely on Professor Guttag if I give this-- if this code to him, I can't rely on him to type in a float. Actually I can, because he's a smart guy, but in general, I can't rely on the user--

PROFESSOR JOHN GUTTAG: I wouldn't do it right to see if you did.

PROFESSOR ERIC GRIMSON: Actually, he's right, you know. He would not do it, just to see if I'm doing it right. I can't rely on the user. I want to make sure I get a float in it, so how do I do that? Well, here's one nice little trick.

First of all, having read in that value, I can check to see, is it of the right type? Now, this is not the nicest way to do it but it'll work. I can look at the type of the value of base and compare it to the type of an actual float and see, are they the same? Is this a real or a float?

If it is, I'm done. How do I go back if it isn't? Well, I'm going to create a little infinite loop. Not normally a good idea. I set up a variable here, called input OK. Initially it's false, because I have no input. And then I run a loop in which I read something in, I check to see if it's the right type, if it is, I change that variable to say it's now the correct type, which means the next time through the loop, I'm going to say I'm all set and I'm going to bounce out.

But if it is not, it's going to print out a message here saying, you screwed up, somewhat politely, and it's going to go back around. So it'll just cycle until I get something of the right type. Nice way of doing it.

Right, what's the second thing I do? Well, I get the same sort of thing to read in the height, once I have that I'm going to take base squared plus height squared, and there's a form that we've just seen once before, and it's going to repeat it, that is math.SQRT and it says the following: it says, take from the math library the function called sqrt.

OK. We're going to come back to this when we get to objects, it's basically picking up that object and it's applying that, putting that value into hype, and then just printing something out.

And again, if I just run this, just to show that it's going to do the right thing, it says enter base, I'm obnoxious, it says oops, wasn't a float, so we'll be nice about it, and I enter a height, and it prints out what I expected. I just concatenated those strings together, by the way, at the end.

All right. Notice what I did. OK. I went from this description, it gives me [UNINTELLIGIBLE] some information. I need to have a particular type. I made sure I had the particular type. I've used some abstraction to suppress some details here.

Now if you look at that list, there is actually something I didn't seem to check, which is, I said I wanted a float stored in hyp.

How do I know I've got a float in hyp? Well I'm relying on the contract, if you like, that the manufacturer of square root put together, which is, if I know I'm giving it two floats, which I do because I make sure they're floats, the contract, if you like, of square root says I'll give you back a float. So I can guarantee I've got something of the right type.

OK. I know this is boring as whatever. But there's an important point here. Having now used this pseudo code to line things up, I can start putting some additional structure on this. And in particular, I'm sure you're looking at this going-- will look at it if we look at the right piece-- going, wait a minute. This chunk of code and this chunk of code, they're really doing the same thing.

And this is something I want to use. If I look at those two pieces of computation, I can see a pattern there. It's an obvious pattern of what I'm doing. And in particular, I can then ask the following question, which is, what's different between those two pieces of code?

And I suggest two things, right? One is, what's the thing I print out when I ask for the input? The second thing is, what do I print out if I actually don't get the right input in? And so the only two differences are, right there, and there versus here and here.

So this is a good place to think about, OK, let me capture that. Let me write a function, in fact the literal thing I would do is to say, identify the things that change, give each of them a variable name because I want to refer to them, and then write a function that captures the rest of that computation just with those variable names inside. And in fact, if you look down-- and I'm just going to highlight this portion, I'm not going to run it-- but if you look down here, that's exactly what that does.

I happen to have it commented out, right? What does it do? It has height, it says, I've got two names of things: the request message and the error message. The body of that function looks exactly like the computation up above, except I'm simply using those in place of the specific message I had before. And then the only other difference is obviously, it's a function I need to return a value. So when I'm done, I'm going to give the value back out. All right?

And that then let's me get to, basically, this code. Having done that, I simply call base with get float, I call height with get float, and do the rest of the work.

All right. What's the point of doing this? Well, notice again. What have I done? I've captured a module inside of a function.

And even though it's a simple little thing here, there's some a couple of really nice advantages to this. All right? First one is there's less code to read. It's easier to debug. I don't have as much to deal with. But the more important thing is, I've now separated out implementation from functionality, or implementation from use.

What does that mean? It means anybody using that little function get float doesn't have to worry about what's inside of it. So for example, I decide I want to change the message I print out, I don't have to change the function, I just pass in a different parameter.

Well if I-- you know, with [UNINTELLIGIBLE PHRASE sorry, let me say it differently. I don't need to worry about how checking is done, it's handled inside of that function. If I decide there's a better way to get input, and there is, then I can make it to change what I don't have to change the code that uses the input.

So, if you like, I've built a separation between the user and the implementer. And that's exactly one of the reasons why I want to have the functions, because I've separated those out.

Another way of saying it is, anything that uses get float doesn't care what the details are inside or shouldn't, and if I change that definition, I don't have to change anything elsewhere in my code, whereas if I just have the raw code in there, I have to go off and do it.

Right, so the things we want you to take away from this are, get into the habit of using pseudo code when you sit down to start a problem, write out what are the steps. I will tell you that a good programmer, at least in my mind, may actually go back and modify the pseudo code as they realize they're missing things, but it's easier to do that when you're looking at a simple set of steps, than when you're in the middle of a pile of code.

And get into the habit of using it to help you define what is the flow of control. What are the basic modules, what information needs to be passed between those modules in order to make the code work.

OK. That was the short topic. I will come back to this some more and you're going to get lots of practice with this. What I want to do is to start talking about a different topic. Which is efficiency.

And this is going to sound like a weird topic, we're going to see why it's of value in a second. I want to talk about efficiency, and we're going to, or at least I'm going to, at times also refer to this as orders of growth, for reasons that you'll see over the next few minutes.

Now, efficiency is obviously an important consideration when you're designing code, although I have to admit, at least for me, I usually want to at least start initially with code that works, and then worry about how I might go back and come up with more

efficient implementation. I like to have something I can rely on, but it is an important issue.

And our goal over the next couple of lectures, is basically to give you a sense of this. So we're not going to turn you into an expert on computational efficiency. That's, there are whole courses on that, there's some great courses here on that, it takes some mathematical sophistication, we're going to push that off a little bit.

But what we-- what we do want to do, is to give you some intuition about how to approach questions of efficiency. We want you to have a sense of why some programs complete almost before you're done typing it. Some programs run overnight. Some programs won't stop until I'm old and gray. Some programs won't stop until you're old and gray. And these are really different efficiencies, and we want to give you a sense of how do you reason about those different kinds of programs.

And part of it is we want you to learn how to have a catalog, if you like, of different classes of algorithms, so that when you get a problem, you try and map it into an appropriate class, and use the leverage, if you like, of that class of algorithms.

Now. It's a quick sidebar, I've got to say, I'm sure talking about efficiency to folks like you probably seems really strange. I mean, you grew up in an age when computers were blazingly fast, and have tons of memory, so why in the world do you care about efficiency?

Some of us were not so lucky. So I'll admit, my first computer I program was a PDP6, only Professor Guttag even knows what PDP stands for, it was made by Digital Equipment Company, which does not exist anymore, is now long gone. It had, I know, this is old guy stories, but it had 160k of memory. Yeah. 160k. 160 kilobits of memory. I mean, your flash cards have more than that, right? It had a processor speed of one megahertz. It did a million operations per second.

So let's think about it. This sucker, what's it got in there? That Air Mac, it's, see, it's got, its go-- my Air Mac, I don't know about John's, his is probably better, mine has 1.8 gigahertz speed. That's 1800 times faster. But the real one that blows me away is, it has 2 gig of memory inside of it. That's 12 thousand times more memory. Oh, and by the way? The PDP6, it was in a rack about this tall. From the floor, not from the table.

All right, so you didn't grow up in the late 1800s like I did, you don't have to worry about this sort of stuff, right? But a point I'm trying to make is, it sounds like anymore computers have gotten so blazingly fast, why should you worry about it?

Let me give you one other anecdote that I can't resist. This is the kind of thing you can use at cocktail parties to impress your friends from Harvard. OK. Imagine I have a little lamp, a little goose-- one of those little gooseneck lamps, I'd put it on the table here, I'd put the height about a f-- about a foot off the table. And if I was really good, I could hit, or time it so that when I hurt-- yeah, try again. When I turn this on switch on in the lamp, at exactly the same time, I'm going to hit a key on my computer and start it running.

OK. In the length of time it takes for the light to get from that bulb to the table, this machine processes two operations. Oh come on, that's amazing. Two operations. You know, you can do the simple numbers, right? [UNINTELLIGIBLE PHRASE]

Light travels basically a foot in a nanosecond. Simple rule of thumb. Now, the nanosecond is what, 10 to the minus 9 seconds. This thing does 2 gig worth of operations. A gig is 10 to the 9, so it does two operations in the length of time it takes light to get from one foot off the table down to the table. That's amazing.

So why in the world do you care about efficiency? Well the problem is that the problems grow faster than the computers speed up. I'll give you two examples.

I happen to work in medical imaging. Actually, so does Professor Guttag. In my in my area of research, it's common for us to want to process about 100 images a second in order to get real time display. Each image has about a million elements in it. I've got to process about a half a gig of data a second in order to get anything out of it.

Second example. Maybe one that'll hit a little more home to you. I'm sure you all use Google, I'm sure it's a verb in your vocabulary, right? Now, Google processes-- ten million? Ten billion pages? John? I think ten billion was the last number I heard. Does that sound about right?

PROFESSOR JOHN GUTTAG: I think it might actually be more by now.

PROFESSOR ERIC GRIMSON: Maybe more by now. But let's, for the sake of argument, ten billion pages. Imagine you want to search through Google to find a particular page. You want to do it in a second. And you're going to just do it the brute force way, assuming you could even reach all of those pages in that time.

Well, if you're going to do that, you've got to be able to find what you're looking for in a page in two steps. Where a step is a comparison or an arithmetic operation. Ain't going to happen, right? You just can't do it.

So again, part of the point here is that things grow-- or to rephrase it, interesting things grow at an incredible rate. And as a consequence, brute force methods are typically not going to work.

OK. So that then leads to the question about what should we do about this? And probably the obvious thing you'll think about is, we'll come up with a clever algorithm. And I want to disabuse you of that notion. It's a great idea if you can do it,

The guy who-- I think I'm going to say this right, John, right? Sanjay? Ghemawat?-- with a guy who was a graduate of our department, who is the heart and soul behind Google's really fast search, is an incredibly smart guy, and he did come up with a really clever algorithm about how you structure that search, in order to make it happen. And he probably made a lot of money along the way. So if you have a great idea, you know, talk to a good patent attorney and get it locked away.

But in general, it's hard to come up with the really clever algorithm. What you're much better at doing is saying how do I take the problem I've got and map it into a

class of algorithms about which I know and use the efficiencies of those to try and figure out how to make it work.

So what we want to do, is, I guess another way of saying it is, efficiency is really about choice of algorithm. And we want to help you learn how to map a problem into a class of algorithms of some efficiency. That's our goal.

OK. So to do this, we need a little more abstract way of talking about efficiency, and so, the question is, how do we think about efficiency? Typically there's two things we want to measure. Space and time. Sounds like an astrophysics course, right?

Now, space usually we-- ach, try it again. When we talk about space, what we usually refer to is, how much computer memory does it take to complete a computation of a particular size?

So let me write that down, it's how much memory do I need to complete a computation. And by that, I mean, not how much memory do I need to store the size of the input, it's really how much internal memory do I use up as I go through the computation? I've got some internal variables I have to store, what kinds of things do I have to keep track of?

You're going to see the arguments about space if you take some of the courses that follow on, and again, some nice courses about that. For this course, we're not going to worry about space that much. What we're really going to focus on is time.

OK. So we're going to focus here. And the obvious question I could start with is, and suppose I ask you, how long does the algorithm implemented by this program take to run? How might I answer that question? Any thoughts? Yeah.

STUDENT: [INAUDIBLE]

PROFESSOR ERIC GRIMSON: Ah, you're jumping ahead of me, great. The answer was, find a mathematical expression depending on the number of inputs. It was exactly where I want to go. Thank you.

I was hoping for a simpler answer, which is, just run it. Which is, yeah I know, seems like a dumb thing to say, right?

One of the things you could imagine is just try it on and input, see how long it takes. You're all cleverer than that, but I want to point out why that's not a great idea.

First of all, that depends on which input I've picked. All right? Obviously the algorithm is likely to depend on the size of the input, so this is not a great idea.

Second one is, it depends on which machine I'm running on. If I'm using a PDP6, it's going to take a whole lot longer than if I'm using an Air Mac. All right?

Third one is, it may depend on which version of Python I'm running. Depends on how clever the implementer of Python was.

Fourth one is, it may depend on which programming language I'm doing it in.

So I could do it empirically, but I don't want to do that typically, it's just not a great way to get at it. And so in fact, what we want is exactly what the young lady said.

I'm going to ask the following question, which is-- let me write it down-- what is the number of the basic steps needed as a function of the input size?

That's the question we're going to try and address. If we can do this, this is good, because first of all, it removes any questions about what machine I'm running on, it's talking about fundamentally, how hard is this problem, and the second thing is, it is going to do it specifically in terms of the input. Which is one of the things that I was worried about.

OK. So to do this, we're going to have to do a couple of things. All right, the first one is, what do we mean by input size?

And unfortunately, this depends on the problem. It could be what's the size of the integer I pass in as an argument, if that's what I'm passing in. It could be, how long is the list, if I'm processing a list or a tuple It could be, how many bits are there in something. So it-- that is something where we have to simply be clear about specifying what we're using as input size. And we want to characterize it mathematically as some number, or some variable rather, the length of the list, the size of the integer, would be the thing we'd want to do.

Second thing we've got to worry about is, what's a basic step? All right, if I bury a whole lot of computation inside of something, I can say, wow, this program, you know, runs in one step. Unfortunately, that one step calls the Oracle at Delphi and gets an answer back. Maybe not quite what you want.

We're typically going to use as basic steps the built-in primitives that a machine comes with. Or another way of saying it is, we're going to use as the basic steps, those operations that run in constant time, so arithmetic operations. Comparisons. Memory access, and in fact one of the things we're going to do here, is we're going to assume a particular model, called a random access model, which basically says, we're going to assume that the length of time it takes me to get to any location in memory is constant.

It's not true, by the way, of all programming languages. In fact, Professor Guttag already talked about that, in some languages lists take a time linear with the length to get to it. So we're to assume we can get to any piece of data, any instruction in constant time, and the second assumption we're going to make is that the basic primitive steps take constant time, same amount of time to compute. Again, not completely true, but it's a good model, so arithmetic operations, comparisons, things of that sort, we're all going to assume are basically in that in that particular model.

OK. Having done that, then, there are three things that we're going to look at. As I said, what we want to do is, we want to count the number of basic steps it takes to compute a computation as a function of input size.

And the question is, what do we want to count? Now, one possibility is to do best case. Over all possible inputs to this function, what's the fastest it runs? The fewest, so the minimum, if you like. It's nice, but not particularly helpful.

The other obvious one to do would be worst case. Again, over all possible inputs to this function, what's the most number of steps it takes to do the computation?

And the third possibility, is to do the expected case. The average. I'm going to think of it that way.

In general, people focus on worst case. For a couple of reasons. In some ways, this would be nicer, do expected cases, it's going to tell you on average how much you expect to take, but it tends to be hard to compute, because to compute that, you have to know a distribution on input. How likely are all the inputs, are they all equally likely, or are they going to depend on other things? And that may depend on the user, so you can't kind of get at that.

We're, as a consequence, going to focus on worst case. This is handy for a couple of reasons. One, it means there are no surprises. All right? If you run it, you have a sense of the upper bound, about how much time it's going to take to do this computation, so you're not going to get surprised by something showing up.

The second one is, a lot of the time, the worst case is the one that happens. Professor Guttag used an example of looking in the dictionary for something. Now, imagine that dictionary actually has something that's a linear search to go through it, as opposed to the hashing he did, so it's a list, for example. If it's in there, you'll find it perhaps very quickly. If it's not there, you've got to go through everything to say it's not there. And so the worst case often is the one that shows up, especially in things like search.

So, as a consequence, we're going to stick with the worst case analysis.

Now, I've got two minutes left. I was going to start showing you some examples, but I think, rather than doing that, I'm going to stop here, I'm going to give you two minutes back of time, but I want to just point out to you that we are going to have fun next week, because I'm going to show you what in the world that has to do with efficiency.

So with that, we'll see you next time.