MIT OpenCourseWare
http://ocw.mit.edu

6.00 Introduction to Computer Science and Programming, Fall 2008
Transcript – Lecture 10

PROFESSOR: Last time we were talking about binary search and I sort of left a promise to you which I need to pick up. I want to remind you, we were talking about search, which is a very fundamental thing that we do in a whole lot of applications. We want to go find things in some data set. And I'll remind you that we sort of a separated out two cases. We said if we had an ordered list, we could use binary search. And we said that was log rhythmic, took log n time where n is the size of the list.

If it was an unordered list, we were basically stuck with linear search. Got to walk through the whole list to see if the thing is there. So that was of order in. And then one of the things that I suggested was that if we could figure out some way to order it, and in particular, if we could order it in n log n time, and we still haven't done that, but if we could do that, then we said the complexity changed a little bit. But it changed in a way that I want to remind you.

And the change was, that in this case, if I'm doing a single search, I've got a choice. I could still do the linear case, which is order n or I could say, look, take the list, let's sort it and then search it. But in that case, we said well to sort it was going to take n log n time, assuming I can do that.

Once I have it sorted I can search it in log n time, but that's still isn't as good as just doing n. And this led to this idea of amortization, which is I need to not only factor in the cost, but how am I going to use it? And typically, I'm not going to just search once in a list, I'm going to search multiple times. So if I have to do k searches, then in the linear case, I got to do order n things k times. It's order k n.

Whereas in the ordered case, I need to get them sorted, which is still n log n, but then the search is only log n. I need to do k of those. And we suggested well this is better than that. This is certainly better than that. m plus k all times log n is in general going to be much better than k times n. It depends on n and k but obviously as n gets big, that one is going to be better.

And that's just a way of reminding you that we want to think carefully, but what are the things we're trying to measure when we talk about complexity here? It's both the size of the thing and how often are we going to use it? And there are some trade offs, but I still haven't said how I'm going to get an n log n sorting algorithm, and that's what I want to do today. One of the two things I want to do today.

To set the stage for this, let's go back just for a second to binary search. At the end of the lecture I said binary search was an example of a divide and conquer algorithm. Sort of an Attila the Hun kind of approach to doing things if you like. So let me say -- boy, I could have made a really bad political joke there, which I will forego, right.

Let's say what this actually means, divide and conquer. Divide and conquer says basically do the following: split the problem into several sub-problems of the same type. I'll come back in a second to help binary searches matches in that, but that's what we're going to do. For each of those sub-problems we're going to solve them independently, and then we're going to combine those solutions.

And it's called divide and conquer for the obvious reason. I'm going to divide it up into sub-problems with the hope that those sub-problems get easier. It's going to be easier to conquer if you like, and then I'm going to merge them back. Now, in the binary search case, in some sense, this is a little bit trivial. What was the divide? The divide was breaking a big search up into half a search. We actually threw half of the list away and we kept dividing it down, until ultimately we got something of size one to search. That's really easy.

The combination was also sort of trivial in this case because the solution to the sub-problem was, in fact, the solution to the larger problem. But there's the idea of divide and conquer. I'm going to use exactly that same ideas to tackle sort. Again, I've got an unordered list of n elements. I want to sort it into a obviously a sorted list. And that particular algorithm is actually a really nice algorithm called merge sort. And it's actually a fairly old algorithm. It was invented in 1945 by John von Neumann one of the pioneers of computer science. And here's the idea behind merge sort, actually I'm going to back into it in a funny way.

Let's assume that I could somehow get to the stage where I've got two sorted lists. How much work do I have to do to actually merge them together? So let me give you an example. Suppose I want to merge two lists, and they're sorted.

Just to give you an example, here's one list, 3121724 Here's another list, 12430. I haven't said how I'm going to get those sorted lists, but imagine I had two sorted lists like that. How hard is it to merge them? Well it's pretty easy, right? I start at the beginning of each list, and I say is one less than three? Sure. So that says one should be the first element in my merge list.

Now, compare the first element in each of these lists. Two is less than three, so two ought to be the next element of the list. And you get the idea. What am I going to do next? I'm going to compare three against four. Three is the smallest one, and I'm going to compare four games twelve, which is going to give me four. And then what do? I have to do twelve against thirty, twelve is smaller, take that out. Seventeen against thirty, twenty-four against thirty And by this stage I've got nothing left in this element, so I just add the rest of that list in.

Wow I can sort two lists, so I can merge two lists. I said it poorly. What's the point? How many operations did it take me to do this? Seven comparisons, right? I've got eight elements. It took me seven comparisons, because I can take advantage of the fact I know I only ever have to look at the first element of each sub-list. Those are the only things I need to compare, and when I run out of one list, I just add the rest of the list in.

What's the order of complexity of merging? I heard it somewhere very quietly.

STUDENT: n.

PROFESSOR: Sorry, and thank you. Linear, absolutely right? And what's n by the way here? What's it measuring?

STUDENT: [UNINTELLIGIBLE]

PROFESSOR: In both lists, right. So this is linear, order n and n is this sum of the element, or sorry, the number of elements in each list. I said I was going to back my way into this. That gives me a way to merge things. So here's what merge sort would do.

Merge sort takes this idea of divide and conquer, and it does the following: it says let's divide the list in half. There's the divide and conquer. And let's keep dividing each of those lists in half until we get down to something that's really easy to sort. What's the simplest thing to sort? A list of size one, right? So continue until we have singleton lists.

Once I got a list of size one they're sorted, and then combine them. Combine them by doing emerge the sub-lists. And again, you see that flavor. I'm going to just keep dividing it up until I get something really easy, and then I'm going to combine. And this is different than binary search now, the combine is going to have to do some work.

So, I'm giving you a piece of code that does this, and I'm going to come back to it in the second, but it's up there. But what I'd like to do is to try you sort sort of a little simulation of how this would work. And I was going to originally make the TAs come up here and do it, but I don't have enough t a's to do a full merge sort. So I'm hoping, so I also have these really high-tech props. I spent tons and tons of department money on them as you can see.

I hope you can see this because I'm going to try and simulate what a merge sort does. I've got eight things I want to sort here, and those initially start out here at top level. The first step is divide them in half. All right? I'm not sure how to mark it here, remember I need to come back there. I'm not yet done. What do I do? Divide them in half again.

You know, if I had like shells and peas here I could make some more money. What do I do? I divide them in half one more time. Let me cluster them because really what I have, sorry, separate them out. I've gone from one problem size eight down to eight problems of size one.

At this stage I'm at my singleton case. So this is easy. What do I do? I merge. And the merge is, put them in order. What do I do next? Obvious thing, I merge these. And that as we saw was a nice linear operation. It's fun to do it upside down, and then one more merge which is I take the smallest elements of each one until I get to where I want.

Wow aren't you impressed. No, don't please don't clap, not for that one. Now let me do it a second time to show you that -- I'm saying this poorly. Let me say it again. That's the general idea. What should you see out of that? I just kept sub-dividing down until I got really easy problems, and then I combine them back.

I actually misled you slightly there or maybe a lot, because I did it in parallel. In fact, let me just shuffle these up a little bit. Really what's going to happen here, because this is a sequential computer, is that we're going to start off up here, at top level, we're going to divide into half, then we're going to do the complete subdivision and merge here before we ever come back and do this one. We're going to do a division here and then a division there. At that stage we can merge these, and then take this down, do the division merge and bring them back up.

Let me show you an example by running that. I've got a little list I've made here called test. Let's run merge sort on it, and then we'll look at the code. OK, what I would like you to see is I've been printing out, as I went along, actually let's back up slightly and look at the code. There's merge sort. Takes in a list. What does it say to do? It says check to see if I'm in that base case. It's the list of length less than two. Is it one basically? In which case, just return a copy the list. That's the simple case.

Otherwise, notice what it says to do. It's says find the mid-point and split the list in half. Copy of the back end, sorry, copy of the left side, copy of the right side. Run merge sort on those. By induction, if it does the right thing, I'm going to get back two lists, and I'm going to then merge them together. Notice what I'm going to do. I'm going to print here the list if we go into it, and print of the when we're done and then just return that.

Merge up here. There's a little more code there. I'll let you just grok it but you can see it's basically doing what I did over there. Setting up two indices for the two sub-list, it's just walking down, finding the smallest element, putting it into a new list. When it gets to the end of one of the lists, it skips to the next part, and only one of these two pieces will get called because only one of them is going to have things leftovers. It's going to add the other pieces in.

OK, if you look at that then, let's look at what happened when we ran this. We started off with a call with that list. Ah ha, split it in half. It's going down the left side of this. That got split in half, and that got split in half until I got to a list of one.

Here's the first list of size one. There's the second list of size one. So I merged them. It's now in the right order, and that's coming from right there. Having done that, it goes back up and picks the second sub-list, which came from there. It's a down to base case, merges it. When these two merges are done, we're basically at a stage in that branch where we can now merge those two together, which gives us that, and it goes through the rest of it.

A really nice algorithm. As I said, an example of divide and conquer. Notice here that it's different than the binary search case. We're certainly dividing down, but the combination now actually takes some work. I'll have to actually figure out how to put them back together. And that's a general thing you want to keep in mind when you're thinking about designing a divide and conquer kind of algorithm.

You really want to get the power of dividing things up, but if you end up doing a ton of work at the combination stage, you may not have gained anything. So you really want to think about that trade off. All right, having said that, what's the complexity here?

Boy, there's a dumb question, because I've been telling you for the last two lectures the complexity is n log n, but let's see if it really is. What's the complexity here? If we think about it, we start off with the problem of size n. What do we do? We split it into two problems of size n over 2. Those get split each into two problems of size n over 4, and we keep doing that until we get down to a level in this tree where we have only singletons left over.

Once we're there, we have to do the merge. Notice what happens here. We said each of the merge operations was of order n. But n is different. Right? Down here, I've just got two things to merge, and then I've got things of size two to merge and then things of size four to merge. But notice a trade off. I have n operations if you like down there of size one. Up here I have n over two operations of size two. Up here I've got n over four operations of size four.

So I always have to do a merge of n elements. How much time does that take? Well, we said it, right? Where did I put it? Right there, order n. So I have order n operations at each level in the tree. And then how many levels deep am I? Well, that's the divide, right? So how many levels do I have? Log n, because at each stage I'm cutting the problem in half. So I start off with n then it's n over two n over four n over eight.

So I have n operations log n times, there we go, n log n. Took us a long time to get there, but it's a nice algorithm to have. Let me generalize this slightly. When we get a problem, a standard tool to try and attack it with is to say, is there some way to break this problem down into simpler, I shouldn't say simpler, smaller versions of the same problem. If I can do that, it's a good candidate for divide and conquer. And then the things I have to ask is how much of a division do I want to do? The obvious one is to divide it in half, but there may be cases where there are different divisions you want to have take place.

The second question I want to ask is what's the base case? When do I get down to a problem that's small enough that it's basically trivial to solve? Here it was lists of size one. I could have stopped at lists of size two right. That's an easy comparison. Do one comparison and return one of two possible orders on it, but I need to decide that.

And the third thing I need to decide is how do I combine? You know, point out to you in the binary search case, combination was trivial. The answer to the final search was just the answer all the way up. Here, a little more work, and that's why I'll come back to that idea.

If I'm basically just squeezing jello, that is, I'm trying to make the problem simpler, but the combination turns out to be really complex, I've not gained anything.

So things that are good candidates for divide and conquer are problems where it's easy to figure out how to divide down, and the combination is of little complexity. It would be nice if it was less than linear, but linear is nice because then I'm going to get that n log in kind of behavior. And if you ask the TAs in recitation tomorrow, they'll tell you that you see a lot of n log n algorithms in computer science. It's a very common class of algorithms, and it's very useful one to have.

Now, one of the questions we could still ask is, right, we've got binary search, which has got this nice log behavior. If we can sort things, you know, we get this n log n behavior, and we got a n log n behavior overall. But can we actually do better in terms of searching.

I'm going to show you one last technique. And in fact, we're going to put quotes around the word better, but it does better than even this kind of binary search, and that's a method called hashing. You've actually seen hashing, you just don't know it. Hashing is the the technique that's used in Python to represent dictionaries. Hashing is used when you actually come in to Logan Airport and Immigration or Homeland Security checks your picture against a database. Hashing is used every time you enter a password into a system.

So what in the world is hashing? Well, let me start with a simple little example. Suppose I want to represent a collection of integers. This is an easy little example. And I promise you that the integers are never going to be anything other than between the range of zero to nine. OK, so it might be the collection of one and five. It might be two, three, four, eight. I mean some collection of integers, but I guarantee you it's between zero and nine.

Here's the trick I can play. I can build -- I can't count -- I could build a list with spots for all of those elements, zero, one, two, three, four, five, six, seven, eight, nine. And then when I want to create my set, I could simply put a one everywhere that that integer falls.

So if I wanted to represent, for example, this is the set two, six and eight, I put a one in those slots. This seems a little weird, but bear with me for second, in fact, I've given you a little piece a code to do it, which is the next piece of code on the hand out. So let's take a look at it for second.

This little set of code here from create insert and number. What's create do? It says, given a low and a high range, in this case it would be zero to nine. I'm going to build a list. Right, you can see that little loop going through there. What am I doing? I'm creating a list with just that special symbol none in it. So I'm building the list. I'm returning that as my set. And then to create the object, I'll simply do a set of inserts. If I want the values two, six and eight in there, I would do an insert of two into that set, an insert of six into that set, and an insert of eight into the set. And what does it do? It marks a one in each of those spots.

Now, what did I want to do? I wanted to check membership. I want to do search. Well that's simple. Given that representation and some value, I just say gee is it there? What's the order complexity here? I know I drive you nuts asking questions? What's the order complexity here? Quadratic, linear, log, constant? Any takers? I know I have the wrong glasses on the see hands up too, but...

STUDENT: [UNINTELLIGIBLE]

PROFESSOR: Who said it?

STUDENT: Constant.

PROFESSOR: Constant, why?

STUDENT: [UNINTELLIGIBLE]

PROFESSOR: Yes, thank you. All right, it is constant. You keep sitting back there where I can't get to you. Thank you very much. It has a constant. Remember we said we design lists so that the access, no matter where it was on the list was of constant time. That is another way of saying that looking up this thing here is constant. So this is constant time, order one.

Come on, you know, representing sets of integers, this is pretty dumb. Suppose I want to have a set of characters. How could I do that? Well the idea of a hash, in fact, what's called a hash function is to have some way of mapping any kind of data into integers.

So let's look at the second example, all right, -- I keep doing that -- this piece of code from here to here gives me a way of now creating a hash table of size 256. Ord as a built in python representation. There is lots of them around that takes any character and gives you back an integer.

In fact, just to show that to you, if I go down here and I type ord, sorry, I did that wrong. Let me try again. We'll get to exceptions in a second. I give it some character. It gives me back an integer representing. It looks weird. Why is three come back to some other thing? That's the internal representation that python uses for this. If I give it some other character, yeah, it would help if I could type, give it some other character. It gives me back a representation.

So now here's the idea. I build a list 256 elements long, and I fill it up with those special characters none. That's what create is going to do right here. And then hash character takes in any string or character, single character, gives me back a number. Notice what I do. If I want to create a set or a sequence representing these things, I simply insert into that list. It goes through and puts ones in the right place.

And then, if I want to find out if something's there, I do the same thing. But notice now, hash is converting the input into an integer. So, what's the idea? If I know what my hash function does, it maps, in this case characters into a range zero to 256, which is zero to 255, I create a list that long, and I simply mark things. And my look up is still constant.

Characters are simple. Suppose you want to represent sets of strings, well you basically just generalize the hash function. I think one of the classic ones for strings is called the Rabin-Karp algorithm. And it's simply the same idea that you have a mapping from your import into a set of integers.

Wow, OK, maybe not so wow, but this is now constant. This is constant time access. So I can do searching in constant time which is great. Where's the penalty? What did I trade off here? Well I'm going to suggest that what I did was I really traded space for time. It makes me sound like an astro physicist somehow right?

What do I mean by that? I have constant time access which is great, but I paid a price, which is I had to use up some space. In the case of integers it was easy. In the case of characters, so I have to give up a list of 256, no big deal.

Imagine now you want to do faces. You've got a picture of somebody's face, it's a million pixels. Each pixel has a range of values from zero to 256. I want to hash a face with some function into an integer. I may not want to do the full range of this, but I may decide I have to use a lot of gigabytes of space in order to do a trade off.

The reason I'm showing you this is it that this is a gain, a common trade off in computer science. That in many cases, I can gain efficiency if I'm willing to give up space. Having said that though, there may still be a problem, or there ought to be a problem that may be bugging you slightly, which is how do I guarantee that my hash function takes any input into exactly one spot in the storage space?

The answer is I can't. OK, in the simple case of integers I can, but in the case of something more complex like faces or fingerprints or passwords for that matter, it's hard to design a hash function that has completely even distribution, meaning that it takes any input into exactly one output spot. So what you typically do and a hash case is you design your code to deal with that.

You try to design -- actually I'm going to come back to that in a second. It's like you're trying to use a hash function that spread things out pretty evenly. But the places you store into in those lists may have to themselves have a small list in there, and when you go to check something, you may have to do a linear search through the elements in that list.

The good news is the elements in any one spot in a hash table are likely to be a small number, three, four, five. So the search is really easy. You're not searching a million things. You're searching three or four things, but nonetheless, you have to do that trade off.

The last thing I want to say about hashes are that they're actually really hard to create. There's been a lot of work done on these over the years, but in fact, it's pretty hard to invent a good hash function. So my advice to you is, if you want to use something was a hash, go to a library. Look up a good hash function.

For strings, there's a classic set of them that work pretty well. For integers, there are some real simple ones. If there's something more complex, find a good hash function, but designing a really good hash function takes a lot of effort because you want it to have that even distribution. You'd like it to have as few duplicates if you like in each spot in the hash table for each one of the things that you use.

Let me pull back for a second then. What have we done over the last three or four lectures? We've started introducing you to classes of algorithms. Things that I'd like you to be able to see are how to do some simple complexity analysis.

Perhaps more importantly, how to recognize a kind of algorithm based on its properties and know what class it belongs to. This is a hint. If you like, leaning towards the next quiz, that you oughta be able to say that looks like a logarithmic algorithm because it's got a particular property. That looks like an n log n algorithm because it has a particular property.

And the third thing we've done is we've given you now a set of sort of standard algorithms if you like. Root force, just walk through every possible case. It works well if the problem sizes are small. We've had, there are a number of variants of guess and check or hypothesize and test, where you try to guess the solution and then check it and use that to refine your search.

Successive approximation, Newton-Raphson was one nice example, but there's a whole class of things that get closer and closer, reducing your errors as you go along.

Divide and conquer and actually I guess in between there bi-section, which is really just a very difficult of successive approximation, but divide and conquer is a class of algorithm. These are tools that you want in your tool box. These are the kinds of algorithms that you should be able to recognize. And what I'd like you to begin to do is to look at a problem and say, gee, which kind of algorithm is most likely to be successful on this problem, and map it into that case.

OK, starting next -- don't worry I'm not going to quit 36 minutes after -- I got one more topic for today. But jumping ahead, I'm going to skip in a second now to talk about one last linguistic thing from Python, but I want to preface Professor Guttag is going to pick up next week, and what we're going to start doing then is taking these classes of algorithms and start looking at much more complex algorithms. Things you're more likely to use in problems. Things like knapsack problems as we move ahead.

But the tools you've seen so far are really the things that were going to see as we build those algorithms. OK, I want to spend the last portion of this lecture doing one last piece of linguistics stuff. One last little thing from Python, and that's to talk about exceptions.

OK, you've actually seen exceptions a lot, you just didn't know that's what they were, because exceptions show up everywhere in Python. Let me give you a couple of examples. I'm going to clear some space here. Before I type in that expression, I get an error, right?

So it's not defined. But in fact, what this did was it threw an exception. An exception is called a name error exception. It says you gave me something I didn't know how to deal. I'm going to throw it, or raise it, to use the right term to somebody in case they can handle it, but it's a particular kind of exception.

I might do something like, remind you I have test. If I do this, try and get the 10th element of a list that's only eight long. I get what looks like an error, but it's actually throwing an exception. The exception is right there. It's an index error, that is it's trying to do something going beyond the range of what this thing could deal with.

OK, you say, come on, I've seen these all the time. Every time I type something into my program, it does one of these things, right? When we're just interacting with idol, with the interactive editor or sorry, interactive environment if you like, that's what you expect. What's happening is that we're typing in something, an expression it doesn't know how to deal. It's raising the exception, but is this simply bubbling up at the top level saying you've got a problem.

Suppose instead you're in the middle of some deep piece of code and you get one of these cases. It's kind of annoying if it throws it all the way back up to top level for you to fix. If it's truly a bug, that's the right thing to do. You want to catch it. But in many cases exceptions or things that, in fact, you as a program designer could have handled.

So I'm going to distinguish in fact between un-handled exceptions, which are the things that we saw there, and handled exceptions. I'm going to show you in a second how to handle them, but let's look at an example. What do I mean by a handled exception?

Well let's look at the next piece of code. OK, it's right here. It's called read float. We'll look at it in a second. Let me sort of set the stage up for this -- suppose I want to input -- I'm sorry I want you as a user to input a floating point number.

We talked about things you could do to try make sure that happens. You could run through a little loop to say keep trying until you get one. But one of the ways I could deal with it is what's shown here. And what's this little loop say to do? This little loop says I'm going to write a function or procedures that takes in two messages.

I'm going to run through a loop, and I'm going to request some input, which I'm going to read in with raw input. I'm going to store that into val. And as you might expect, I'm going to then try and see if I can convert that into a float. Oh wait a minute, that's a little different than what we did last time, right? Last time we checked the type and said if it is a float you're okay. If not, carry on.

In this case what would happen? Well float is going to try and do the cohersion. It's going to try and turn it into a floating point number. If it does, I'm great, right. And I like just to return val. If it doesn't, floats going to throw or raise, to use the right term, an exception. It's going to say something like a type error.

In fact, let's try it over here. I if I go over here, and I say float of three, it's going to do the conversion. But if I say turn this into a float, ah it throws a value error exception. It says it's a wrong kind of value that I've got. So I'm going to write a little piece of code that says if it gives me a float, I'm set, But if not, I'd like to have the code handle the exception. And that's what this funky tri-accept thing does. This is a tri-accept block and here's the flow of control that takes place in there.

When I hit a tri-block. It's going to literally do that. It's going to try and execute the instructions. If it can successfully execute the instructions, it's going to skip past the except block and just carry on with the rest of the code. If, however, it raises an exception, that exception, at least in this case where it's a pure accept with no tags on it, is going to get, be like thrown directly to the except block, and it's going to try and execute that. So notice what's going to happen here, then. If I give it something that can be turned into a float, I come in here, I read the input, if it can be turned into a float, I'm going to just return the value and I'm set.

If not, it's basically going to throw it to this point, in which case I'm going to print out an error message and oh yeah, I'm still in that while loop, so it's going to go around. So in fact, if I go here and, let me un-comment this and run the code. It says enter a float. And if I give it something that can be -- sorry, I've got, yes, never mind the grades crap. Where did I have that? Let me comment that out. Somehow it's appropriate in the middle of my lecture for it to say whoops at me but that wasn't what I intended. And we will try this again.

OK, says it says enter a float. I give it something that can be converted into a float, it says fine. I'm going to go back and run it again though. If I run it again, it says enter a float. Ah ha, it goes into that accept portion, prints out a message, and goes back around the while loop to say try again. And it's going to keep doing this until I give it something that does serve as a float.

Right, so an exception then has this format that I can control as a programmer. Why would I want to use this? Well some things I can actually expect may happen and I want to handle them. The float example is a simple one. I'm going to generalize in a second.

Here's a better example. I'm writing a piece of code that wants to input a file. I can certainly imagine something that says give me the file name, I'm going to do something with it. I can't guarantee that the file may exist under that name, but I know that's something that might occur. So a nice way to handle it is to write it as an exception that says, here's what I want to do if I get the file. But just in case the file name is not there, here's what I want to do in that case to handle it. Let me specify what the exception should do.

In the example I just wrote here, this is pretty trivial, right. OK, I'm trying to input floats. I could generalize this pretty nicely. Imagine the same kind of idea where I want to simply say I want to take input of anything and try and see how to make sure I get the right kind of thing. I want to make it polymorphic.

Well that's pretty easy to do. That is basically the next example, right here. In fact, let me comment this one out. I can do exactly the same kind of thing. Now what I'm going to try and do is read in a set of values, but I'm going to give a type of value as well as the messages. The format is the same. I'm going to ask for some input, and then I am going to use that procedure to check, is this the right type of value. And I'm trying to use that to do the coercion if you like. Same thing if it works, I'm going to skip that, if it not, it's going to throw the exception.

Why is this much nice? Well, that's a handy piece of code. Because imagine I've got that now, and I can now store that away in some file name, input dot p y, and import into every one of my procedure functions, pardon me, my files of procedures, because it's a standard way of now giving me the input.

OK, so far though, I've just shown you what happens inside a peace a code. It raises an exception. It goes to that accept clause. We don't have to use it just inside of one place. We can actually use it more generally.

And that gets me to the last example I wanted to show you. Let me uncomment this. Let's take a look at this code. This looks like a handy piece of code to have given what we just recently did to you. All right, get grades. It's a little function that's going to say give me a file name, and I'm going to go off and open that up and bind it to a local variable. And if it's successful, then I'd just like to go off and do some things like turn it into a list so I can compute average score or distributions or something else. I don't really care what's going on here.

Notice though what I've done. Open, it doesn't succeed is going to raise a particular kind of exception called I O error. And so I've done a little bit different things here which is I put the accept part of the block with I O error. What does that say? It says if in the code up here I get an exception of that sort, I'm going to go to this place to handle it.

On the other hand, if I'm inside this procedure and some other exception is raised, it's not tagged by that one, it's going to raise it up the chain. If that procedure was called by some other procedure it's going to say is there an exception block in there that can handle that. If not, I am going to keep going up the chain until eventually I get to the top level. And you can see that down here. I'm going to run this in a second. This is just a piece of code where I'm going to say, gee, if I can get the grades, do something, if not carry on.

And if I go ahead and run this -- now it's going to say woops, at me. What happened? I'm down here and try, I'm trying do get grades, which is a call to that function, which is not bound in my computer. That says it's in here. It's in this tri-block. It raised an exception, but it wasn't and I O error. So it passes it back, past this exception, up to this level, which gets to that exception.

Let me say this a little bit better then. I can write exceptions inside a piece of code. Try this, if it doesn't work I can have an exception that catches any error at that level. Or I can say catch only these kinds of errors at that level, otherwise pass them up the chain. And that exception will keep getting passed up the chain of calls until it either gets to the top level, in which case it looks like what you see all the time. It looks like an error, but it tells you what the error came from, or it gets an exception , it can deal with it.

OK, so the last thing to say about this is what's the difference between an exception and an assert? We introduced asserts earlier on. You've actually seen them in some pieces of code, so what's the difference between the two of them? Well here's my way of describing it. The goal of an assert, or an assert statement, is basically to say, look, you can make sure that my function is going to give this kind of result if you give me inputs of a particular type. Sorry, wrong way of saying it. If you give me inputs that satisfy some particular constraints.

That was the kind of thing you saw. Asserts said here are some conditions to test. If they're true, I'm going to let the rest of the code run. If not, I'm going to throw an error. So the assertion is basically saying we got some pre-conditions, those are the clauses inside the assert that have to be true, and there's a post condition. and in essence, what the assert is saying is, or rather the programmer is saying using the assert is, if you give me input that satisfies the preconditions, I'm guaranteeing to you that my code is going to give you something that meets the post condition. It's going to do the right thing.

And as a consequence, as you saw with the asserts, if the preconditions aren't true, it throws an error. It goes back up the top level saying stop operation immediately and goes back up the top level. Asserts in fact are nice in the sense that they let you check conditions at debugging time or testing time. So you can actually use them to see where your code is going. An exception, when you use an exception, basically what you're saying is, look, you can do anything you want with my function, and you can be sure that I'm going to tell you if something is going wrong. And in many cases I'm going to handle it myself.

So as much as possible, the exceptions are going to try to handle unexpected things, actually wrong term, you expected them, but not what the user did. It's going to try to handle conditions other than the normal ones itself. So you can use the thing in anyway. If it can't, it's going to try and throw it to somebody else to handle, and only if there is no handler for that unexpected condition, will it come up to top level.

So, summarizing better, assert is something you put in to say to the user, make sure you're giving me input of this type, but I'm going to guarantee you the rest of the code works correctly. Exceptions and exception handlers are saying, here are the odd cases that I might see and here's what I'd like to do in those cases in order to try and be able to deal with them.

Last thing to say is why would you want to have exceptions? Well, let's go back to that case of inputting a simple little floating point. If I'm expecting mostly numbers in, I can certainly try and do the coercion. I could have done that just doing the coercion.

The problem is, I want to know if, in fact, I've got something that's not of the form I expect. I'm much better having an exception get handled at the time of input than to let that prop -- that value rather propagate through a whole bunch of code until eventually it hits an error 17 calls later, and you have no clue where it came from.

So the exceptions are useful when you want to have the ability to say, I expect in general this kind of behavior, but I do know there are some other things that might happen and here's what I'd like to do in each one of those cases. But I do want to make sure that I don't let a value that I'm not expecting pass through.

That goes back to that idea of sort of discipline coding. It's easy to have assumptions about what you think are going to come into the program when you writ it. If you really know what they are use them as search, but if you think there's going to be some flexibility, you want to prevent the user getting trapped in a bad spot, and exceptions as a consequence are a good thing to use.