PROFESSOR: We're going to cover hashing next time. Now we're Going to focus on pset three because I heard pset three is hard so we're going to look at the code and try to understand it.

AUDIENCE: This course is hard.

PROFESSOR: I've tried to make it easy. I really tried. How many people recognize the code? Does this look familiar? One, two, three. Everyone else got stuck on question one? I'm not going to give you the answers to the psets. We are going to--

AUDIENCE: [LAUGHING]

PROFESSOR: Sorry. We're going to look at the code and understand what it does because once you understand what it does you can understand how to modify it. All right, before I do that, any questions? And I might not answer them right away, but I will keep them in mind while I go through the code. Are there any specific pin-points.

AUDIENCE: The types.

PROFESSOR: Yep. Covered in five minutes. Oh well. The first thing that I want to talk about is sweep line algorithms. We're asking you to implement the sweep line algorithm and we're giving you one that is horribly inefficient. Suppose you have some line segments that look like this. Sorry, drawing isn't. They look like this. And suppose you want to find the intersections between them.

Now let's figure out what the algorithm that we gave you does. Did people look at the trace for that? Not the good trace, the trace that's output by the algorithm that we gave you. Does anyone understand what that does.

AUDIENCE: I kind of get it.

PROFESSOR: OK.

AUDIENCE: I think what it does is it just hits through all the horizontal lines and it goes through all the vertical ones. First it goes through all the horizontal lines and what does it do with them?

AUDIENCE: [INAUDIBLE].

PROFESSOR: They get bolded in the visualizer which means they're added to the range index. So one, add all the horizontal lines. And two-- oh you said it goes through all the vertical lines after that. Any idea what it does?

AUDIENCE: It looks for intersections.

PROFESSOR: When he looks at a vertical line segment you'll see some blue square and you'll see some segments that are green. This is a query to the range index and the green segments are the answers and the blue square is the range that's queried. First add all the segments to the range index and then do a query.

In principle sweep line algorithms have some geometric inputs. For example, a bunch of lines. And conceptually they have this vertical line that they sweep across the plane all the way from minus infinity to plus infinity. Now if you try to sweep a line continuously from minus infinity to plus infinity in code it might take forever. We don't really do it that way. The way we do it in code is you know when something interesting is going to happen and you only simulate what happens to the sweep line at that point. In this case the sweep line starts at minus infinity and then all the horizontal segments are added to our range index. And then as the sweep line goes across the plane, when it hits a vertical segment a query happens.

The only times when the sweep line stops is first, at the beginning to add all the horizontal line segments, and then every time it hits a vertical line segment. We know this ahead of time so we can compute a list of all these coordinates of interesting things. And then we can sort them and then go through them and

simulate the behavior of the sweep line that way.

If you look at the code listing at events from there, this is the gist of it. This is what it does. It compiles a list of events. The first key in each event is the x, is that x? Yeah. x like this, y like this. The first case, the x-coordinate of an event and by sorting that list of events it will then process them in order. Events from there is responsible for looking at all the wires and compiling events. How does an event for a horizontal layer look like? Can anyone tell me? Sorry, for a horizontal wire.

AUDIENCE: It will run across [INAUDIBLE].

PROFESSOR: First off it looks like a list, right? It's a regular python list. And I will try to call it vector so that we don't confuse it with a big list of all the events. You said the first thing is the leftmost point. Right? If this segment, for example, starts at minus 100, this one starts at minus 95, this one starts at minus 90, and this one starts at minus 50, by the way not to scale, then the leftmost point is minus 100. Right? This is the first element in the vector. And that's computed on line four where it goes through all the segments, looks at all the X-coordinates and chooses the minimum.

What's the next thing in the vector? We don't have to understand why. We'll go through why in a bit. I'm just interested in what?

AUDIENCE: Zero.

PROFESSOR: Zero. Excellent.

PROFESSOR: After that there is a wire ID. Each wire gets a unique ID, that's a number between zero and the number of wires. Say this is one, two, three. This is going to be one. And then I have the string add. And then I have the actual wire object. What does the vector look like for a query event? Sorry, what does it look like for a vertical wire? I'm giving away some of the answer. Bad, bad, bad. All right, vertical wire, look at the code and.

AUDIENCE: It keeps the left [INAUDIBLE].

PROFESSOR: Of the wire. OK. And?

AUDIENCE: It keeps the x-coordinate [INAUDIBLE].

PROFESSOR: A vertical wire will have the same X-coordinates for all the points on the wire, right? I don't really worry about which coordinate it is. This wire is at minus 50. The first element here is going to be minus 50. What's next?

AUDIENCE: A 1.

PROFESSOR: A 1. OK. And suppose this is wire ID 4. I'm going to have 4.

AUDIENCE: Query.

PROFESSOR: Query.

AUDIENCE: And the wire.

PROFESSOR: And the wire. All right, what happens with these in the init method, which is not shown in the code listing, is after the list is put together sort is called on it. These are all compared and then reordered to be sorted according to some ordering relationship. Does anyone know what the ordering relationship is for Python lists? Yeah.

AUDIENCE: It starts from the first, from zero, and the next it goes to sort the same with as X 1.

PROFESSOR: OK. Say if I have 1, 2, 3 and 1, 3, 2. Sorry, and 2, 1, 2. He's going to look at the first element and if they are different than the one with the smaller element is smaller. 1, 2, 3 as a vector, sorry, as a list, is smaller than 2, 1, 2 as a list. If this guy becomes one then they're equal. Then it has to go to the next element, compare them and see if they're different we have an answer, if not we have to keep going all the way until the end of the list. This is called lexicographic comparison and this is the same ordering that you have for the words in a dictionary. Right, if you think of each letter as an element in a list and the word is the list then this is how you look up words in a dictionary.

The reason events look like this is so that sort would output them in the right order

for the next state. So, yes.

AUDIENCE: Does that mean all the horizontal wires, does that mean that they start at same place even though that they don't?

PROFESSOR: Yup. all the horizontal wires will be sorted. All these add events will be sorted before the query events. Yup, very good observation. This is what I'm trying to achieve. That's why I have this special value here. I go through some extra effort to compute it. I had to write an extra line of code so this is the motivation for it.

The first key in the vector is the X element on the sweep line. I could've also had the very large negative value that something that would behave like minus infinity here and that would work just as well. By computing the left edge I don't have to deal with that. I don't have to worry of whether my negative infinity is small enough, because if it's not I will fail the test and that's bad.

This is the x-coordinate at which the sweep line stops and something happens. So if all the events are different the x-coordinates then I have my ordering, I'm done. Now what if I have two events at the same time? For example, what if I have two vertical wires? The x-coordinates are going to be the same so I need to use something else to break the ties, right?

The first thing that I use to break the ties is all add events have a zero here and all the query events have a one. And the point of that is if I have two events at the same x-coordinate, if I have a query and an add, I want the add to happen before the query. If this is the same this is going to be different for ad versus query. Does that make sense? This is how we order events relatively to each other on the same line.

Now suppose I have these two wires, they're both vertical so they're both going to be queries. They're both at the same index. I need something else, and that other thing that I have is the wire ID which is guaranteed to be unique. I know for sure the comparison will stop here. And the comparison had better stop here because if the comparison gets all the way to here, wires aren't comparable so the code is going to

crash. It stops here because the IDs are unique, everyone is happy. All right, do the events make more sense now?

To complete the picture if you look at compute crossings, lines nine and ten. Line nine goes through the vector-- sorry, goes through the list of events that have been sorted and processes them in order so sort had better do the right job. And then it extracts the x-coordinate, that's here, it extracts the event type, that's here, and then it extracts the wire from here. These guys really are just there to help with the sorting, they're never read afterwards.

AUDIENCE: Where are the events actually sort? I thought they just put into order.

PROFESSOR: The events are sorted in an init method that's not here. It's in the piece of code but it's not here. And that may be a hint that you don't want to change it. As in you don't need to. All right, events look like this. Any questions about events? Everything make sense?

Presumably when you write your own sweep line algorithm you're going to come up with your own events which are going to be different. You're going to change these methods to add your own events to the list. And then it's going to be sorted for you and then you're going to change compute crossings to process your events in the way that they should be processed.

Let's look at the range index. What do we store in a range index?

AUDIENCE: The horizontal wires.

PROFESSOR: Horizontal wires. Very good. What's the point of storing horizontal wires in an index?

AUDIENCE: That's when you want to query, the area has a line through it.

PROFESSOR: For the algorithm that we gave you how does a query look like? Suppose I have this wire here and I'm doing a query for it.

AUDIENCE: Just ask to list horizontal wires that are between that Y.

PROFESSOR: Between this guy's top and this guy's bottom. Right?

AUDIENCE: Yeah.

PROFESSOR: A query would look like this. Basically all the horizontal wires that have their y-coordinates between this guy and this guy. Now if I have a wire that's way up here or way down here I know for sure that it's not going to intersect this wire. Right? I don't care about it. The range index helps me eliminate some wires. Now, will the range index eliminate all the wires that don't intersect with this wire? No, OK. Why not?

AUDIENCE: Well in this code it doesn't.

PROFESSOR: OK, in this code it doesn't. We're only talking about this code, I'm not talking about the solution code that you guys have to implement.

AUDIENCE: It doesn't because it never removes the horizontal wires.

PROFESSOR: OK. What's an example of a query that would give me some wires that don't intersect my wire?

AUDIENCE: Like if they're in between the y-axis, they're, yeah but they're not actually.

PROFESSOR: OK. So you mean.

AUDIENCE: Yeah.

AUDIENCE: Negative infinity and infinity on the x-axis would give you a bunch of wires and not many of them will intersect with the vertical wires.

AUDIENCE: If the end were a vertical wire, [INAUDIBLE].

PROFESSOR: OK. I understood this because it matches what I drew. I didn't understand the minus infinity plus infinity.

AUDIENCE: Well if you think about the x-axis on the bottom, if you go from all the way on the left to all the way on the right you're going to get a whole bunch of wires that are short

that--

PROFESSOR: Are you talking about one wire that's like that, or many?

AUDIENCE: If you're grabbing from your range index.

PROFESSOR: The range index has horizontal wires in it.

AUDIENCE: Right.

AUDIENCE: Are you assuming that they're all short?

AUDIENCE: Yes. But it's not necessarily true. Some of them will be short.

PROFESSOR: You're saying a lot of short horizontal lines-- you mean like this, right?

AUDIENCE: Yeah.

PROFESSOR: OK. That's what you mean.

AUDIENCE: You have one intersection and--

PROFESSOR: A ton of non-intersections.

AUDIENCE: Exactly.

PROFESSOR: OK, cool. Thank you. Doing a range query on this can give me a lot of false positives. That's why after I get them I have to make sure that they intersect and I can't just blindly output them. These are bad false positives, these are bad false positive.

Now let's look at how the range index works. Can anyone remind me of what the range index does? It's a data structure, right? And it has some operations. What are the operations for it?

AUDIENCE: It returns everything between two values.

PROFESSOR: You have a list operation where you give it two values and it will return everything inside the index, in the interval between those two values. What else can I do?

AUDIENCE: It can also tell you how many. It returns those keys and how many there are.

PROFESSOR: I have the count and you said that I have some things in the index. Right? How do I get them in there? These are queries, I need updates.

AUDIENCE: Add and remove.

PROFESSOR: For the code that they gave you, what is the running time for each of these operations?

AUDIENCE: It would be constant had to be removed.

PROFESSOR: OK. Add calls the pend. Right? So that is constant. I will agree there.

AUDIENCE: To remove it you had to shift everything.

PROFESSOR: Yep. Let's start with an example. Say I have these keys in my range index. 5, 8, 11, 13. Suppose they're magically inserted in this order. If I want to remove two, since this is an array I have to shift everything to the left.

AUDIENCE: Yes.

PROFESSOR: So that's.

AUDIENCE: Order and time.

PROFESSOR: OK. How about list and counts? What's the running time for counts? How does count work?

AUDIENCE: It's not just through all of the keys.

PROFESSOR: Count goes through everything and--

AUDIENCE: It has to be order [INAUDIBLE].

PROFESSOR: Cool. Count goes through all the keys, evaluates the predicate that checks whether they're in the range and then it adds up one to the sum for the right place. How

about list?

AUDIENCE: Also [INAUDIBLE].

PROFESSOR: OK, that's a list comprehension. The code is a bit tricky, but it pretty much does the same thing. It goes through the list and it puts all the keys that are in the interval in the list.

Now let's look at another version on the next page of our range index that is slightly better. What's the rapid variance for this? How is it different from the first one?

AUDIENCE: Sorted already.

PROFESSOR: Sorted already. Very good. The keys look exactly like this. Right? What is the running time for-- let's start with add or remove, what's the running time for add and remove?

AUDIENCE: N, order of N.

PROFESSOR: Why is it order of N?

AUDIENCE: No, oder log N.

PROFESSOR: Why is it order log N.

AUDIENCE: Because if it's sorted you can look in the middle and do a binary search.

PROFESSOR: I can do a binary search in order to find out where I insert something. Right? But what if I want to insert zero in this?

AUDIENCE: Is it log N plus order N? Does that make sense? Because you have to shift all of the [INAUDIBLE].

PROFESSOR: Yeah. So worst case if I have to insert zero here, I have to shift everything to the right to make room for it. Right? So your first instinct was good. Order N. And questions? No more? Cool. How about remove?

AUDIENCE: Order N.

PROFESSOR: Order N. Why is it order N?

AUDIENCE: You have to shift left everything to the right.

PROFESSOR: Is this what you were going to say too? Same reason as before, if I want to remove 2 I still have to move everything to the left. I can find the key that I want to remove very fast but then shifting things is still order N. OK how about count? How does count work?

AUDIENCE: It's the same thing, isn't it?

AUDIENCE: Binary search so it's log n.

PROFESSOR: Binary search, so it's log n. If I look at the count there it's--

AUDIENCE: It's not though, that's telling you how many keys between.

AUDIENCE: So it's a simple subtraction to find the number of keys between.

PROFESSOR: Count calls binary search twice and then it does a arithmetic operation, yes.

AUDIENCE: It would be log n because with the count where the range is that's log n because you have to actually sift through it, but binary search you have two other operations and then you just do a surprise.

PROFESSOR: Yep, thank you. Very good. This is log n because of good old binary search. How about lists? I think you were thinking ahead of lists. How about lists?

AUDIENCE: You have to list every single thing that you read.

AUDIENCE: You've got the values you might potentially have to [INAUDIBLE].

AUDIENCE: Is it log n plus the length of the list.

PROFESSOR: Yeah. I was going to do a nice long analysis and there's the answer. Log n plus, suppose you return i values from the list, i. Why does this matter? If I have a range query from minus infinity to plus infinity I have to create a new list, copy all these

keys in and return them. That's order n. I can't just say it's log n. On the other hand, if the answers to my list queries are small, if they're one element, then it's going to take log n time to do a binary search and then order one to produce the list if I have a constant size list. If I just say order n I'm selling myself short. It's a lot better if I want to have short queries.

What would be the best possible running time for lists? If I had a magic algorithm that works as fast as possible would I be able to run in order one time?

AUDIENCE: No.

PROFESSOR: Super fast magic algorithm order one. No, why not?

AUDIENCE: Because you still have to find l and h in your array.

PROFESSOR: Suppose I have some other data structure, a super magical data structure.

AUDIENCE: You can do searches in order one time, is what you're saying.

AUDIENCE: You still have to have the contents of it for what you're listing. It would have to be order i.

PROFESSOR: Even if I have a super magical data structure where I can do everything that I want inside in order one time then when I produce the output I still have to write the output. This thing is going to be omega i and I can't possibly do any better than that. No matter what running time you have for list it has to include this i. Unless you're running time is order i, in which case i is smaller or equal to n.

Does this look right to you? There is a small bug in this code. What if I do count of 4, 11. Binary search returns, if there's a key there it will give me the position of that key. And if there's no key it will tell me where I should insert the key if the key doesn't exist in theory. So if this is my array then the positions are 0, 1, 2, 3, 4, 5. Count of 4, 11 would do a binary search of 11 and see 4 and then it will do a binary search for 4 and return 2. Right? Because if I want to insert 4 I would have to put it here and then shift everything to the right. And 4 minus 2 is 2 therefore it's broken.

AUDIENCE: Off by one block.

PROFESSOR: Off by one block, right? Well the interesting thing about it is that I had this code, we actually shipped this code between Saturday morning and Sunday evening and it passed all the tests. As we go to the next section keep in mind that this is broken but it would still pass the test for our problem and we'll see why in a bit.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Count of 4, 11.

AUDIENCE: So confined to them then it says that when you're down 5, right? Four.

PROFESSOR: Four. Binary search, if it finds the key it returns the position of they key.

AUDIENCE: [INAUDIBLE] 4 divided by x, and 4, 4 would it have had two?

PROFESSOR: Yeah, because it tells me where to insert it.

AUDIENCE: OK, then it is. [INAUDIBLE].

PROFESSOR: Well it says 4 minus 2, right? The last line is high index minus low index?

AUDIENCE: Oh, well that's wrong. [INAUDIBLE].

AUDIENCE: So are you saying that it tells you where to insert it instead of actually inserting it.

PROFESSOR: Yeah. Binary search doesn't insert the element. It just tells me this is where you put it. And then if you look at add, add inserts it right there. That's why binary search is done that way so that that would work.

AUDIENCE: Can I ask a style question?

PROFESSOR: Sure.

AUDIENCE: The underscore before binary search, is that like saying private?

PROFESSOR: Yep.

AUDIENCE: OK.

AUDIENCE: Wait, what does private mean?

PROFESSOR: Private means that it's not part of the public interface of the class. If we're in a team and we're working on something, if I mark a method as private that means if you use my class, you're not supposed to call it because I might change it's name, I might delete it completely, I can do anything I want to it. But if the method is public, so add, remove, list, and count, those are the API or the public interface of the class. I guarantee that as long as the blit range index class exists it's going to have an add to remove a list and the conduct will behave in exactly in that way. Once you have a class with public methods you're not supposed to change them because other people might depend on them and that would make other people unhappy.

We said that this range index holds wires. Right? So if you have these wires here they're not really numbers. I know how to compare numbers, I don't know how to compare wires. That's not something that's readily comparable, right? We have to bridge the gap between numbers or things that are comparable and wires.

The way we do that is the key wire pair class. If you look at key wire pair class it has-- I will write it as KWP here. So it has a key and the wire. Right? That's why it's a key wire pair. And if you look at the comparison methods, lines 11 through 26, but if you look at 11 through 13 pretty much everything else is the same. Then comparisons are done in a certain way. So the first criterion is the key. If you have two key wire pairs that have different keys, then the one with the bigger key wins. Now if they have the same key then this field called wire IDs is used to break ties.

So first key and then wire ID. What's the wire ID set to? And where? Where would you set a field in a nice class?

AUDIENCE: The constructor.

PROFESSOR: In the constructor, very good. Where is it set? Line number?

AUDIENCE: 10.

PROFESSOR: Line 10. So the wire ID uses the same object IDs that we had earlier. So each wire has a unique ID that's between zero and the number of wires. This works when I insert my wires into the index. What's the key for wires, by the way? Where are wires inserted into the index in the algorithm that we have and what's the key? The algorithm that we gave you. Yes.

AUDIENCE: The y-coordinates.

PROFESSOR: The? Which one, I couldn't hear you.

AUDIENCE: The y-coordinate.

PROFESSOR: The y-coordinate. Can you tell me which line? The intuition is very good. You want the wires to be keyed by the y-coordinate so that when you a range query between this guy and this guy you get these wires. That's the intuition. Now what's the code?

AUDIENCE: Line 13.

PROFESSOR: Line? I didn't hear you well. Line?

AUDIENCE: 13.

PROFESSOR: 13. Wires are added to the index when an event of type add is processed. Right? Line 12 says if event type is add so that's probably what we want. And line 13 adds it to the index and builds the key wire repair with the key, the y-coordinate of the wire.

Now let's look back at the keys and see that we have two special key wire repair classes. Key wire pair l and key wire pair h. These don't want the wire. Why would I make a wire key pair that doesn't want the wire? Why is that useful?

AUDIENCE: You have the minimum and maximum wire at ease.

PROFESSOR: Very good observation. The wire ID for the low key pair looks like minus infinity and the one for the high key pair look like plus infinity as long as I don't have more than a billion wires. What do these things not have? They don't have a wire. So if you

look at key wire pair l and h, if you look at the initializers on lines two and eight, they only take a key they don't take a wire. This is useful in which situation? Where do I want to create an index key that's not associated with the real wire?

AUDIENCE: When you want to make the list when you want to do a query.

PROFESSOR: When I do a query. Very good. If I'm querying, if I have this vertical wire, which starts from minus 80 to plus 80 then I would like to make a low key that corresponds to minus 80 and a high key that corresponds to plus 80, but I don't have any wire with a horizontal coordinate of 80. Right? A hackish solution would be to make fake wires with those coordinates but if I make fake wires God knows where my system is going to break elsewhere. Instead the clean solution is to have these key wire pairs so that when I insert wires I have a wire and when I don't the wire is set to none. And the wire IDs are these special values, minus infinity and plus infinity.

What's cool about setting the wire IDs to minus infinity and plus infinity?

AUDIENCE: It's so that if you have a y of 80 this y is at negative infinity so it's going to take that y also because its y value is [INAUDIBLE].

PROFESSOR: All right. Cool. If I have a real key wire pair and it has a wire whose coordinate is 80 this is going to be bigger than a key wire pair l with coordinate 80 and it's going to be smaller than a key wire pair h with coordinate 80. If you do a query using this and this it's going to grab all the wires with coordinate 80. Does this makes sense? So what's cool about this in terms of coding? Do I have equal keys in my range index? Do I have to handle multiple equal keys?

AUDIENCE: No because, actually in this case yes.

PROFESSOR: Do I?

AUDIENCE: Because if you have all the horizontal wires then it's possible that you have two horizontal wires at the same--

PROFESSOR: If I have two horizontal wires, say this guy here and this other guy here, this guy here, what's the key value pair for it? Key wire pair? Let's say the coordinate is

minus 95 and the wire ID is 2. The key wire pair is they key is minus 95 and the wire ID is 2, right?

AUDIENCE: Yeah.

PROFESSOR: And then for this other wire, suppose the wire ID is 10, the coordinate is going to be minus 95 then the wire ID is going to be 10. So how are they going to compare?

AUDIENCE: Then that one is less than that one.

PROFESSOR: Yup. So even though I have two wires with the same y-coordinates of the key as far as the algorithm is concerned is the same, in the implementation the keys are artificially different because I introduce an artificial ordering on the wires using that wire ID. In Which case would I have the same key inserted in my index twice? Yes?

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. If these two wires overlap, say this guy ends here and this guy ends here, and if I put their keys into the index they're still different.

AUDIENCE: Only if they're the same wire ID would they have the same--

PROFESSOR: OK, and when do two wires have the same wire ID?

AUDIENCE: If one of them was negative infinity.

PROFESSOR: Let's assume that the infinities were in the right way. Yes? No? Wire IDs are unique. Every wire has it's own wire ID. Wires will never have the same wire ID. So the only case in which you have two equal keys in the index is if you insert the same wire twice. Would you ever want to do that? Probably not. You're going to the same wire twice from the range query and that's not useful. When implementing a data structure for the range index we don't have to deal with equal keys and that is nice because that's less thinking. Now when we do a query, will the keys in the query be in the range index?

AUDIENCE: Not necessarily.

PROFESSOR: How would I do a query if I want to do a query from minus 80 to plus 80?

AUDIENCE: You grab all the horizontal wires whose y-coordinates are--

PROFESSOR: You're thinking in the range index. I'm thinking as the client of the range index. So you have this range index class, how would you issue a query to the range index?

AUDIENCE: The list on this page.

PROFESSOR: OK. I would call the list method. And-- oh crap, I thought I was going to avoid doing that today. l and h are keys, right? How do I construct my keys?

AUDIENCE: Use key wire pair.

PROFESSOR: Key wire pair left of 80-- sorry low, and key wire pair high of 80. So then I'm going to get to these-- I don't have wires for this 80 coordinate. Sorry, minus 80 to 80 because that's what the query looks like. I don't have wires for these coordinates so I'm going to use these special values and this one is smaller than all the wires with coordinate minus 80 and this one is bigger than all the wires with coordinate 80. Will these keys ever be in the index? Nope, we're only using them for updates. If you put this in the index then when you're getting it back and you try to get the wire associated with it you're going to get none, your code is going to crash. If I have a query will the keys in the query ever being the index?

AUDIENCE: No.

PROFESSOR: No. That means fewer border cases to consider when implementing query. All the keys in the range index are different and they keys in the query will never be in the range index.

AUDIENCE: When you do the count l and h aren't in your range index you have to count that.

PROFESSOR: Yeah. In this case that's why the code didn't fail any test. Because I'm not using numbers I'm actually using this. So for the number analogy this is equivalent to when I'm doing a count of 4,11 that would get rewritten as count of 3, 99 and 11.01 and this works. This is what these two values are doing. Except if you try to do 0.99

and 0.01 that's brittle because you might actually have those keys. All right, does this make sense? Any questions on this part? Nope? Yes? Yes.

AUDIENCE: Is this thing in a count off situation where you have a gap in the wires but you still have a [INAUDIBLE] associated with it as it's going through it.

PROFESSOR: Sorry, say that again.

AUDIENCE: So with that example right there.

PROFESSOR: Yeah.

PROFESSOR: Are you worried about the fact that I'll have two equal keys or are you worried about the fact that these wires are crossing?

AUDIENCE: No that they're not overlapping. You have a wire and you have a wire coming down and you have another wire.

PROFESSOR: OK. This is not going to account for that. You have to write the better sweep line algorithm.

AUDIENCE: OK.

PROFESSOR: What I'm talking about now will make your range index faster. Or it makes your range index simpler to implement.

AUDIENCE: OK.

PROFESSOR: This is magic behind the scenes that makes your code smaller. You still have to implement the sweep line algorithm that makes sure that when you a range query it will either return less false positives or ideally no false positives. And then the code will be faster. But then when you do that, you'll see that it's still slow because the range index that we have here isn't optimal. Why isn't the range index here optimal, by the way? What can I improve about it?

AUDIENCE: Add and remove.

PROFESSOR: Add and remove. OK. We're using the comparison model because it's convenient to use the comparison model. You have complex objects to implement those six methods that you see in key wire pair and bam. You can do comparisons, you can use any data structure or algorithm that assume a comparison model. What's the penalty of the comparison model? What's the problem with it? Yes.

AUDIENCE: It will be slower, or it will never be faster than the [INAUDIBLE].

PROFESSOR: Yup. If I want to do a sort that is n log n. And if I want to do a search that is theta what?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Now what are the best bounds that they have for these if we can go outside the comparison model? Just to see if you guys are paying attention in lecture. So best running time for sort if we don't have to use the comparison model.

AUDIENCE: n.

PROFESSOR: n. OK. Best running time for search if we don't to use the comparison model.

AUDIENCE: It's order one with [INAUDIBLE].

PROFESSOR: Order one, with hashing. Can you do any better than that?

AUDIENCE: You can code it as zero.

[LAUGHING]

PROFESSOR: Sort has to output an array of elements so it can be faster than order n. Search has to output yes or no so it has to output something so it has to be order one. So this is as fast as it gets, we're definitely not going to go faster than that.

We have this extra log n factor that we pay if we use the comparison model if we don't collect more information about our keys. But in return we have the convenience that all we have to do is establish an ordering relationship. Does it make sense? Cool.

Let's try to talk about the list pseudocode, which you might have seen on the pset before. And that's on the last page here. I can't give you a proof of why it works, but we can work together to figure out how it works and get an intuitive understanding.

Let's get the keys here and let's assume we're using a regular BSD for implementing a range index. I'm going to put my keys in a BSD. 8, 3, 2, 5. OK. List those two things. LCA and no list. Right. What's LCA? Suppose I want to list all the keys between one and six. First I'm going to call LCA and get some answer and then I'm going to call node list. What's the answer for LCA? OK.

AUDIENCE: Node at three. Key value three.

PROFESSOR: What is this? What's LCA? It's OK to give out an answer to a dumb question on the pset. So LCA is lowest common ancestor. Very good. It's the lowest common ancestor of what?

AUDIENCE: Subtree of values that should be returned nodeless.

PROFESSOR: So usually you have two nodes in a tree and you want to return the LCA. If I want to find out the LCA of 2 and 5, that is 3. If I want to find out the LCA of 2 and 13, that's 8. Now if the keys are not in the tree what does the LCA give me?

AUDIENCE: It's where they would be.

PROFESSOR: It's where they would be if they were inserted. Right. So 1 would be inserted here and 6 would be inserted here. And then their LCA is indeed 3. What does that tell me? Why is that useful?

AUDIENCE: You can cut off a bunch of the rest of the tree so you don't have to look at that.

PROFESSOR: A bunch of, or the rest of the tree?

AUDIENCE: The rest of the tree.

PROFESSOR: The rest of the tree. So if 1 is here and 6 is here, then I know that all the keys-- sorry, so 3 is to the left of 8 so all the keys to the right of 8 are going to be bigger

than 6, for example. Right. Because 6 is here and it's under 8. I can throw this away. Now if this would be to the left of something else, because 1 is here I know that all the keys to the left side of that thing would be smaller than 1. This subtree has to have all the keys between 1 and 6. Yes.

AUDIENCE: You don't have to move 8, right? Where would you move 8?

PROFESSOR: Yup. So we only look at the subtree rooted at LCA. LCA is passed as an argument to node list, and node list seems to do sort of an-- this is a pre-ordered traversal. You look at the key, you look at the left node, at the left subtree, you look at the right subtree. Except there is some magic there and in some cases it doesn't go and look. Let's see what would a good case where it doesn't go and look like. Let's get the same tree. Suppose I want to do a list between 4 and 11-- sorry between 4 and 10. We'll the LCA of 4 and 10?

AUDIENCE: Eight.

PROFESSOR: OK, so node list would have to start here. Suppose I'm here and I go to 3. I know 3 is smaller than 4. Do I want to go left? Everything to the left of 3 is smaller than 4. This entire subtree can be pruned. Now suppose I'm on the right side and I'm at the 11. 11 is bigger than 10, do I want to go right? So if 11 is bigger than 10 I never want to go right because everything to the right of that is bigger. OK so this is how pruning works.

Now if I'm at 3, can I stop? I can't stop and exit completely because on the right side of 3 I might have keys that are bigger than three and that are in my interval. And in this example that seems to be the case.

Let's look very quickly at the intuitive way of analyzing the running time for this. Suppose I have my LCA up here and suppose I go on a path from LCA to L and then-- let's not worry about the right, about h. h is somewhere here, the case is going to be symmetric. When I go down the path, after I go here would I ever want to go left? If this node is before a right turn then I know that l is going to be bigger than that node. Right? So I'm never going to take a left turn. This subtree be

pruned, this subtree will be pruned, this subtree will be pruned. Now if I go right here, what happens?

AUDIENCE: It's node--

PROFESSOR: All the nodes here are going to be between l and LCA. LCA is smaller or equal to h so all these nodes here are guaranteed to be nodes that come out in my list.

All right, why am I doing this? I was saying before that I absolutely have to have an order i because I have to produce the output list. If we look at the lines three and four, line four outputs a key right? Line four is definitely order i and line three is an if condition so we know for sure that line four is only going to execute four keys that are between l and h. This algorithm is going to visit some nodes in the tree. For example, if I have a list of 4 and 10 it's going to visit 8, it's going to visit 3, 5, and 11. And for some of the nodes that are visited it's going to output them. Right this guy gets output and this guy gets output.

The nodes that are outputs are order i. Those are already covered in the i term. And in order to get the running time what I need to know is, how many nodes do I visit that are not part of the output. Because if I end up visiting the entire tree then that's order n plus i. So order n. But if I only end up visiting a few nodes outside of the output then that might be a lot better.

And here I'm trying to argue that I will only visit a few notes the outside the path. In fact, for every node I'm going to visit at most of the node and the node at its left. And then everything to the right of the path is between l and h, so everything to the right is going to be output. Everything here is in order i and everything here is visited in that output. And potentially everything on the path as well.

So what's the height of a path in a binary search tree? Regular binary search tree? OK, and worst case. So worst case the height is order n. But we usually call it height because on average, at least, it's log n. It's somewhere between n and log n. What is the running time for lists in a binary search tree? If you buy this argument here. What is the running time?

AUDIENCE: [INAUDIBLE].

PROFESSOR: You could say n plus i and you're perfectly correct. But we can make it a bit better. What's the height of this path?

AUDIENCE: h.

PROFESSOR: H. What's the running time?

AUDIENCE: h plus i.

PROFESSOR: OK. Cool. Does this make sense at all? Is this too complicated?

AUDIENCE: That's the running time for what?

PROFESSOR: This is the running time for lists. All right so, questions? Is this too far out? Did I lose you guys completely? Makes perfect sense. I like that.

AUDIENCE: It's oder n plus i because it could have up to any nodes on it, right?

PROFESSOR: Yup.

AUDIENCE: And then it's order i because you have to add that [INAUDIBLE] of nodes anyway. It's order h.

PROFESSOR: I'm saying that it's a binary search tree of height h and if h is better than n then I'm going to do much better than n and that's why I'm putting that h there.

AUDIENCE: OK, that makes sense. I think it's just a ltitle confusing at that part, but the [INAUDIBLE] makes sense because of course that's a smaller node you're not going to look at the left portion you're going to look at the right because it's bigger than your smaller [INAUDIBLE], right? So if you go to the right, that means that--

PROFESSOR: Here I know that l is on the left, right? So l is smaller than this key. Everything to the right of this key is between l and LCA. LCA is smaller or equal to h. Everything here is between l and h, so all the nodes here are going to be output. All the nodes here count as i.

AUDIENCE: OK.

PROFESSOR: Now when I turn left here, this is smaller than l. This is smaller than l because l is on the right. This thing is going to be pruned. So I will visit the parent, this node, and that's it. I'm not going to look down. And this is how the nodes look like. Everything that's to the left of the path is not visited, everything that's to the right it open. Any more questions? I hope you get it right on pset then. Thanks guys.