MITOCW | Recitation 9b: DNA Sequence Matching

PROFESSOR: The things we can talk about today, we can talk about this code. We can talk a little bit more about the hash functions. And we can talk a little bit more about amortization. What to do guys want to hear?

AUDIENCE: Amoritizaiton.

PROFESSOR: OK, so one vote for amortization. So who wants to look at the PSET code? Who wants to talk about hashes? Who wants to talk about amortization? Two, three, four, five, OK. So then let's try this. Let's look at the PSET code then talk about amortization a bit at the end.

I do have to talk a little bit about hashes though, because I owe someone a question from last time. And the question was, we have rolling hashes, so the hashes look like this. K where K is a big number, modulo p. And we argue that it's really nice if p is a prime.

And then the question was, what if instead p is 2 to the w, and is not prime, as long as the base that we're using is co-prime with p? Does this work? And the answer is-- I didn't want to say yes without making sure that I don't say something stupid-- but the answer is yes, this works just fine, because the way a compute multiplicative inverse is is you use so something called the extended Euclid's method.

And if we have b and p, then if we compute their GCD, the that's the greatest common divisor-- so GCD is greatest--

If you use extended Euclid you get something like xb plus yp equals GCD of b and p. So if this is 1, then you have xb plus yp equals 1. And if you're working modulo p, whatever that is, then you have that xb is 1 mod p.

So there's your multiplicative inverse. Well so now that's nice math, right? But that doesn't tell me why are we not using this. So with the multiplicative inverse would work, but there's something else that's wrong with using 2 to the w.

Will this give me a good hash function? OK, the fact that it's p might be confusing. So let's say h equals K mod 2 to the w. And remember that the K is some digits in base b, right? It's a big number made out of digits in base b. So K is d1, d2, d3, all the way up until d length in base b.

And I'll make things easier and say that b is 2 to the 8, because we're working with ASCII characters, or colors, or something that fits nicely in a bit.

So what could go wrong with using this?

AUDIENCE: Well if your series of-- if your K is bigger than 2-- if it's K is bigger than 2 to the w--

PROFESSOR: It will be, for sure.

AUDIENCE: Yes, that's the problem, because then you'll loop. You'll get the same hashes for--

PROFESSOR: Yeah, yeah. So you will get-- So hashing takes a lot of possible inputs and maps them to a relatively small set of outputs. Inputs hash output. And we argued last time that we're going to have collisions no matter what, because we have a ton of inputs and not that many outputs.

For example, if we're hashing strings that are a million characters then this is going to be 2 to the 8 to the 1 million possible strings. And then the number of possible values is, if we're using the word size, 2 to the 32. There is no way we can design a function that will take this many inputs, map them to this many outputs, and not do collisions.

But instead, what do we want? What makes a good hash function? Say my hash function is 0 for all the K's. Is that a good hash function?

AUDIENCE: It's an excellent hash function.

PROFESSOR: What's wrong with it?

AUDIENCE: You would put everything in one so that it's searching, or it would take a long time?

PROFESSOR: Yeah, searching takes a long time. And we've don't do sorting with this yet. Searching takes a long time, string sub-matching will take a long time, it's horrible.

AUDIENCE: So what would that distribute-- like [INAUDIBLE] over all--

PROFESSOR: All right, so we want something that looks sort of random. The ideals hash function takes an input then gives it a random output, and then stays consistent. So when it sees an input, returns the same output.

So I think distribute is the keyword here. What's wrong with this hash function? If it takes random data, it's going to distribute it randomly. That's true, so that's all good. But what data that we might see in real life will make it behave badly?

AUDIENCE: The K is a series of characters, right?

PROFESSOR: Maybe.

AUDIENCE: It just could be anything. But we know for sure that L will be larger than w.

PROFESSOR: Say L is a million.

AUDIENCE: OK, well that sucks.

PROFESSOR: Oh, no. That in itself, that doesn't suck. That's what let's us do sub-string matching really fast, even if we have large strings.

AUDIENCE: --say for 2 to the w, though, because then it will be much larger, like the number of--

PROFESSOR: Yeah, but that's OK. So I'm OK with doing this as long as all the values here are distributed sort of uniformly here. So that's fine.

AUDIENCE: OK.

PROFESSOR: But there's-- I'm arguing that there are some values which will make this hash function behave badly. And that those values are so simple that we might see them in real life.

OK, what if all these numbers are-- what if all the digits are even? So d is 0 mod 2. What happens to K?

AUDIENCE: Well, you're saying that instead of 2 to the w, we're just using 2.

PROFESSOR: So no, the modulo is 2 to the w. Say it's 2 to the 32. So d are the digits that make up my K. So what if the base is 2 to the 8? So I have digits from 0 to 255, 256 of them. And all the digits are 0 modulo 2.

For my sub-string matching example, what if all the characters in the sub-string are even?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Not the same thing. But there's a problem. They will hash to-- so if all the digits are 0 modulo 2 then what about the K?

AUDIENCE: [INAUDIBLE] 0 modulo 2--

PROFESSOR: Yep. So it's just like when you have numbers in base 10. 10 happens to be divisible by 2. So if your last digit is even, then the entire number is even. That makes sense, right? That's math. Please nod, tell me that I'm making sense.

OK, so here, the base is 256. And it's also divisible by 2. So if your last digit is divisible by 2, then the whole number is divisible by 2. So then if I take this K modulo 2 to the 32 then the hash is also going to be divisible by 2.

AUDIENCE: Why does it matter if the hash is divisible by 2?

PROFESSOR: So it matters because this is supposed to be my universe, right? These are supposed to be all the outputs. And I'm saying that if my inputs look like this, then the hash function will not distribute them uniformly.

Instead, if this is my possible set of outputs, the hash function will always put outputs in this half. So the outputs will always be here. And these are the numbers that are divisible by 2. So these are even, and these are odd.

And this area gets no love. Absolutely no number will hash here. So--

AUDIENCE: Wait, what about something with all odds?

AUDIENCE: Something with all odd digits?

AUDIENCE: Because you're asking--

AUDIENCE: You have all A's rather than all B's in your sub-string or in your string.

AUDIENCE: Or because your last digit was odd.

PROFESSOR: If all of our digits are odd then the last digit is odd. And then you'd also get something odd, right?

AUDIENCE: Yeah.

AUDIENCE: So there's a pattern. But there's an even distribution.

PROFESSOR: Well if your hash function is always odd, then it's not an even distribution. It's--

AUDIENCE: Wait, our hash function? I thought we were talking about--

AUDIENCE: Isn't it even if your K is even? And if it's odd [INAUDIBLE]?

PROFESSOR: Yeah, so that's bad. Because if all your K's happens to be even-- say if you're doing the nucleotides, and the nucleotides are A, C, G, T. If they happen to be encoded as, say, 0, 2, 4, 6, then these are all even. So the hash function will always be even and I'm wasting the last bit.

So if I'm building a hash table, half the entries will be wasted. They'll never get anything in there. I'm just wasting memory.

AUDIENCE: So if you could guarantee that your inputs would be evenly distributed--

PROFESSOR: So if our inputs are random then the hash function-- most hash functions will do a good job of producing a random output. The problem is real life inputs are not random. For example, if you get-- asides from this-- if you get data from a camera, so if you get your color pixels from a camera, then because of noise those might have the last few bits, always be the same thing.

Also it seems like in real life-- [INAUDIBLE], in his book, argues about this. It seems like in real life there are a lot of sequences that look like that, that would make your hash function behave poorly.

So again, the keyword is distribute. If some non-random property in the input is reflected in the output, then that's a bad hash function.

AUDIENCE: Would you gain a lot of time from your mod operation? Because in mod 2 to the n you just truncate any bits to the left of the n.

PROFESSOR: Yeah, so that's why we would do this, right? That's why we're even considering this case.

AUDIENCE: Because that'd be really nice to be able to not--

PROFESSOR: So modulo is faster, but in return my hash function is crap here. So usually we prefer-- it turns out that in practice nicer hash functions give better speed improvements overall. So if you think of how a hash is laid out in memory, you'll see that because of caching. And everything gets better to take more time on the mod function and use up all your memory for the hash table.

So this is why we don't use the and we use this. Not because of this argument. So a good question required a lot of talking and remembering what's a good hash function, what's a bad hash function. Thank you.

OK, let's look at the code a little bit. Everyone looked at it, right? So this time we have modules. We don't have everything in one big file. Can someone tell me what are the modules we care about, and why?

AUDIENCE: The problem with the one's we have to code ourselves.

PROFESSOR: OK, let's start with that.

AUDIENCE: Sub-sequence hashes-- interval sub-sequence hashes.

PROFESSOR: OK, so these are all in DNA seq, right? So the module is-- so yeah, the PSET hopefully says that you need to upload this file because it's the only file you'll need to modify. So everything that we need to write is here.

Now pretty much everything that's in that file needs to be modified. So I'm not going to list them out. What else do we want to read in that PSET?

AUDIENCE: Rolling [INAUDIBLE]

PROFESSOR: OK, where is rolling hash?

AUDIENCE: In the [INAUDIBLE]

PROFESSOR: So what's different between the API in rolling hash and the API that we talked about last time? Yes?

AUDIENCE: Them having the [INAUDIBLE] pop, or it would skip. And that's something else [INAUDIBLE] just has a slide, it puts everything in one operation.

PROFESSOR: All right, so we have append and skip. And we built some beautiful code with that. And we looked at some fancy math because of it. But it turns out that for this PSET we can get away with slide.

And we started from slide and built these two methods last time. So I'm not going to explain slide again. It's exactly what we had in the code before we started breaking them up.

OK so this is the rolling hash. It is good. Do we care about anything else?

AUDIENCE: I guess you can look at the rest of the code, if you feel like it.

PROFESSOR: You can look at the rest of the code if you feel like it, yep. So I highlighted one file that might be useful, and that's Kfasta.py. That file has a FASTA sequence class, and that's reads from a file and returns something. And the important thing is it doesn't return a list.

If you remember the doc dists, doc dist 1 thorugh doc dist 8 dot PI, fun times. What we had there was we took the input file, and we read it all a list.

This time we're not doing that. We're writing, what, 20 lines of code instead of what could be five lines of code to read the input. Why is that?

AUDIENCE: Less memory?

PROFESSOR: Less memory, OK. So if we're doing it this way, chances are that if we tried to shove the whole input into memory, it wouldn't fit. And it would crash and you would get 0 on the test because of that. So that's not good.

So what do we use instead? Does anyone know what this thing is called? What this class is called?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Iterator, very good.

AUDIENCE: Why do they call it FASTA? Because it goes faster?

PROFESSOR: I think the letters are a bio acronym.

AUDIENCE: Oh, OK.

PROFESSOR: Does anyone, does anyone do bio here? I've seen that before. So it's a bio thing. Let's not worry about it.

AUDIENCE: OK. Or, your can use that for any type of file. Like, you don't have to use it just for bio files.

PROFESSOR: Well, presumably it's reads, it takes advantage of the format that they're stored in, and gives you a list instead of something else.

So how does an iterator work? Suppose you're building your own iterator. What do you have to implement?

AUDIENCE: Iterator [INAUDIBLE]

PROFESSOR: OK, let's start with next, that's the fun one. What does next do?

AUDIENCE: It's like pop.

PROFESSOR: OK, so it's like pop in what way?

AUDIENCE: It gives you the next character.

PROFESSOR: OK. And what happens when you're at the end of the list?

AUDIENCE: It stops.

PROFESSOR: How do you stop?

AUDIENCE: It raises an exception?

PROFESSOR: So next will either return an element, that's the next element in the sequence that you're iterating over. Or it will raise a stop iteration exception error to stop iteration, cool. So what's the other method? Someone said it before, say it again.

AUDIENCE: Iter.

PROFESSOR: Iter. What does this do in an iterator?

AUDIENCE: It returns itself.

PROFESSOR: All right, very good. In an iterator this is how you will implement it all the time. Does anyone know what's the point of iter?

AUDIENCE: So you can return an iterator? Because that's what it told us to do in the PSET.

PROFESSOR: OK, so iter returns and iterator. But it doesn't-- you don't have to start from an iterator. You can start from any object. And if it has a method iter, then it should give you an iterator that iterates over that object.

So if you have something like a list-- 1, 2, 3, 4-- then if you call iter on this, you'll get an iterator for it, hopefully, right? And this is what Python uses when you say for i in.

So behind the scenes, whatever object you give it here, gets an iter call. And then that produces an iterator. And then Python calls next until stop iteration happens.

So you can write an iterator that almost behaves like a list. You can use it in these [INAUDIBLE] instructions, and it works as if it was a list, except it uses a lot less memory, because it computes the elements. Hopefully every time next is called, you're computing the next element that you're returning. If you're storing everything in a list then returning the elements that way, that's not the very smart iterator.

OK let's look at the last page. So the last page has an iterator on top. And the iterator computes-- given a list, it computes the reverse of that list. And you can see that it doesn't reverse the list and then keep the reversed list in memory. Instead, every time you call next, it does some magic with the indexes-- I think the magic is called math-- and then it return something for as long as it can.

So this is how you implement reverse without producing a new list. If the original list was order, say had n elements, then if you'd produce a new list, you'd consume order and memory. This think consumes order 1 memory, and the running time is the same, asymptotically.

OK, any question on iterators?

AUDIENCE: So it's going from the very end, oh, to the very beginning, and then it's stepping back.

PROFESSOR: So reverse, if I give it the list 1, 2, 3, 4, I want reverse to give it back 4, 3, 2, 1. Except it's not going to return a list, it's going to return something that I can use here.

AUDIENCE: Mm hm, ah, OK.

PROFESSOR: OK, yes.

AUDIENCE: Is it ever possible to, sort of, rewind the iterator to like, sort of, reset it?

PROFESSOR: OK, is it?

AUDIENCE: No.

PROFESSOR: Nope. So Python iterators are simple. All you can do is go forward.

AUDIENCE: OK.

PROFESSOR: The reason that is good is because you can use them for streams. So if you get data from a file, or if you can get data from the network, you can wrap it in an iterator. If you wanted to support resume on data that you get from the network, you'd have to buffer all the data.

AUDIENCE: So you would have to call the iter about that again and--

PROFESSOR: Yeah. Yeah, if you want to rewind, get another iterator. OK, that's a good question, thank you.

So these are iterators. Now we're going to go over some Python magic, which is called generators. So look at the iterator code, and then look at the equivalent code right below it.

So 12 lines of Python turned into three lines of Python that do exactly the same thing. So the reverse method will return an object that is an iterator, and that you can use just like the iterator in the reverse class.

Do people understand what that code does? If you do I'm so out of here, we're done.

AUDIENCE: What does yield do?

PROFESSOR: What does yield do? All right, that's the hard question, what does yield do? I will probably spend the rest of the session on the answer to that question. You're asking all the had questions today, man.

So yield, does anyone know conceptually what yield does? Not in detail, just what's it supposed to do so that the rest of the code works? Yes.

AUDIENCE: If you're driving someplace and there's a yield sign, you pause.

PROFESSOR: OK, Python yield. So I like the word pause in there. The word pause is useful. So say, instead of implementing this, say we're implementing sub-sequence hashes.

AUDIENCE: It kind of spit something out, but keeps going.

PROFESSOR: Yep.

AUDIENCE: Returns [INAUDIBLE]

PROFESSOR: OK, so suppose you're implementing sub-sequence hashes. What's the worst, worst possible way you could implement this?

AUDIENCE: Return a list.

PROFESSOR: OK, so the worst, worst way is to go all the way, brute force, don't use the rolling hashes, don't use anything. The next best way is to make a list, right? So you're going to start with an empty list. Then you're going to use the rolling hash in some way. And in some loop you're going to say list.append e. And then you're going to return the list.

Does this makes sense? OK, what's the problem with this code?

AUDIENCE: You're going to have a huge list.

PROFESSOR: Going to have a huge list. So the way we fix it with iterators is we remove this, we replace this with yield e, and we remove this. And now it's a generator. And now this consumes a constant amount of memory, instead of building a list.

And as long as you only want an iterator out of this method, you'll get the right thing. Your code will still work in exactly the same way.

OK, so the big question is what does this guy do, right? This is where the magic is.

So I already said, as a first hint, that this guy will return an iterator. So can someone try to imagine their Python, and see this? So suppose it's your Python, you see this. What do you do?

AUDIENCE: You wait for some sort of command of some sort, right?

PROFESSOR: No, let's try something else.

AUDIENCE: OK.

PROFESSOR: So the execution of this pauses. What happens? So we're looping somewhere, we got a yield. We stop, what's the first thing we do?

AUDIENCE: Spit out e.

PROFESSOR: So you're saying you return e from this guy?

AUDIENCE: [INAUDIBLE] out e [INAUDIBLE]

PROFESSOR: So I want to return something-- I want to return something else from this. So I want to use this as if it was a list, yes?

AUDIENCE: We store e somewhere.

PROFESSOR: OK, store e somewhere.

AUDIENCE: Do you return the pointer of e?

PROFESSOR: Almost, so there's a word for the object that I'm returning. So I want to use it as if it was a list. So I want to pretend that I had returned list in this method, right? So what's the closest thing to a list that I can return.

AUDIENCE: An iterator.

PROFESSOR: An iterator, thank you, all right. So we will grab some information from here. We'll put it in a nice box. And that box will behave like an iterator.

OK, so the first thing, someone said put e away, so that's when we call next we're going to spit that out. What else do I need to put away?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yep, so this is a lot of magic. This tiny box actually has a lot of magic in it. Because when I call next, I want to get e. But I want to come back here and keep going, right?

So I have my code that's using the iterator. And there's this code here, that's sort of in a frozen state. Did you guys see any movies where people are frozen up and then, in the future, they're unfrozen and they start moving again?

AUDIENCE: [INAUDIBLE] movies.

PROFESSOR: All right, cool. So this is like that, this takes up the whole function, freezes it up and puts it in a box here. And it returns an iterator that can use the box in the future.

So when you call next, it gives e, which is the guy that you put in here. And then it take the function out of the box, unfreezes it, and lets it run again until it hits yield again. Then what happens the next time it hits yield?

So, you're looping, and you're yielding again. And say this time you're yielding.

AUDIENCE: Just do the same thing?

AUDIENCE: Do you put it in that iterator? Or do you make another iterator?

PROFESSOR: Same iterator. So while this is looping, the code outside should get the values that it's yielding. So this has to behave as one iterator. So the code is unfrozen, it's allowed to execute until it says yield again. And then it says yield with a new element.

I put this guy in the box. Then I return the old guy as the return value for next.

AUDIENCE: Oh.

PROFESSOR: And then it's frozen again. So this guy's still in a frozen state. In the movies, I think you're only unfrozen once. And then you keep going, right? And there's a happy ending. Where here, every time you call yield you're frozen again, until someone calls next.

Does this make sense?

AUDIENCE: It's kind of like *Groundhog Day*.

PROFESSOR: Yes, except you're allowed to go forward. So this keeps going forward.

AUDIENCE: --up, thought. So it's looping. It's the same day, really. It's doing different things, though.

PROFESSOR: Yeah. But all your state is saved. So there, some of the state is rolled back. Here all the state is saved.

AUDIENCE: OK.

PROFESSOR: OK, but if that analogy helps, keep it.

AUDIENCE: When you call next, are you computing e or e prime to be returned?

PROFESSOR: So when you're calling next, you're computing e prime and returning e.

AUDIENCE: So the value you get from next is pre-computed?

PROFESSOR: So the value you get form next is what you yielded before.

AUDIENCE: Wait, so you would just take some sequence hashes instance of that, and then just by putting in yield, now it's magically become an iterator and you can call that next on it?

PROFESSOR: Yep. And inside, you don't have to know that it's an iterator. So you don't have a method next here, right? I don't implement next or iter here. I write this as if it's printing stuff to the output.

You can think of yield is a print. If you wanted an iterator, then pretend you're printing what you want to iterate over. And instead of saying print you say yield. And then you use that.

OK, now what happens when we're done? What happens when this loop is done and you return from this method? We said there's no return value.

AUDIENCE: It raises a stop?

PROFESSOR: So when we return, it's going to keep in-- have to remember that it's done, right? And the first time, it has some element here that it has to return. So every time you call yield we put a new element in the box, and return the old one.

So now we would return the old one. We've returned e prime, take it out, and put done in the box. So in the future, if next is called again, raise stop iteration. No more freezing, unfreezing, because we're done. We're returned.

AUDIENCE: So if you called next it would just give you nothing?

PROFESSOR: It has to raise this exception.

AUDIENCE: So you mean, like-- oh, so it-- oh, I see. It would give you red text then?

PROFESSOR: If you called it directly, yes, it would give you red text. Yes?

AUDIENCE: So this takes a sequence or a list, not another iterator, ever?

PROFESSOR: This? What's this? This other code here?

AUDIENCE: Yeah.

PROFESSOR: Not necessarily.

AUDIENCE: Or you could give it a procedure.

PROFESSOR: I can give it an iterator if I'm iterating over it using for-in.

AUDIENCE: Like, for something in one iterator, yield that something, and then [INAUDIBLE]

AUDIENCE: Oh, OK.

PROFESSOR: Yeah, that's a good point. I'll get to that later, when we talk about how we're going to solve the PSET. No, we're not solving the PSET for you. But we'll talk about it a little bit.

But yeah, that's a good point. So there's no reason why you can't have an argument here that, either a list or an iterator, and then you're iterating over it. And then you have nested generators. So you have generators returned in other generators, and you have a whole chain of things happening when you say next.

AUDIENCE: Wait, so this is a generator then, because it produces-- well it is an iterator though?

PROFESSOR: So a generator returns an iterator from this method. So a generator acts like an iterator, except when you call next, it unfreezes this code here, and it let's it run.

AUDIENCE: But I mean, it's basically an iterator then?

PROFESSOR: Yeah.

AUDIENCE: But we're just calling it a generator because--

PROFESSOR: Because there's a lot more magic.

AUDIENCE: OK.

PROFESSOR: So an iterator just says next and iter. This is all that an iterator is, nothing more. Any object that has these two methods is an iterator.

AUDIENCE: Oh, OK.

PROFESSOR: Now a generator is a piece of Python magic that let's you write shorter iterators. So three lines, as opposed to 13 lines. And we came up with a way to turn in a code that would build a list, and easily turn it into a code that uses a generator, and that uses constant memory instead of building that list.

AUDIENCE: OK, now I know how an iterator functions.

PROFESSOR: Exactly. OK, do generators make sense now? Yes.

AUDIENCE: If you wanted to loop through all of the values in a generator, do you just wait until the exception's raised? Or should you, like, keep track of how many things are going to be in that generator?

PROFESSOR: So, when you have a generator, you'd have no idea how many things there are. That's a good point. So you're wondering if I have an iterator, say any iterator, not necessarily a generator, how do I know how many things it's going to return, right? Do I have ln? I do not have ln.

So an iterator does not have ln. So you have to iterate through it. And most importantly, some iterators can never return.

So you can have an iterator that streams data for you across the network. Or you can have an iterator that iterates over the Fibonacci numbers. That's an infinite sequence, right? It's never going to end. So ln would not even be defined then.

Good question, I like it.

AUDIENCE: Is there an is-next method for either iterators or generators?

PROFESSOR: Nope. This is what you get, if there is no in.

AUDIENCE: If that is mature then--

PROFESSOR: Yeah. So in Java you have this belief that you shouldn't get exceptions. You should be able to check for them, right? So maybe that's why you're asking.

So if people coming from Java know that any time a method raises an exception, there should be another method that tells you whether this first method is going to raise an exception or not. In Python the exception is just raised.

So exceptions are not a lot more expensive than regular instructions, because we're using an interpreted language, and it's already reasonably slow. So it can do exceptions for free, yay.

So this is how it works. This is how for-in works. Every time you do a for-in, an exception is raised.

AUDIENCE: We don't have to catch that, then?

PROFESSOR: Nope, the for-in catches it for you.

AUDIENCE: That's tricky stuff.

PROFESSOR: But it's nice because then you can build any iterator that acts like a list. And then you can do even more fancy stuff, and build a generator. And you're using constant memory instead of order and memory for producing an order and size list. Yes?

AUDIENCE: So if we get passed in an iterator and then just yielded what we passed in, yielded the iterator, would that just, essentially, delay everything by one?

PROFESSOR: So you're yielding the iterator as next, right?

AUDIENCE: What? Yeah.

PROFESSOR: You want to yield the iterator as next. Because if you yield the iterator object, you're going to return that object every time. So you're thinking of something that--

AUDIENCE: So you need to increase--

PROFESSOR: You'll yield up next, right?

AUDIENCE: Right.

PROFESSOR: You can have a method that says this is the method. And then you take in an iterator. And then you yield it up next. But then you'll, basically, get the same thing.

AUDIENCE: The same thing. But is it delayed by one or no?

PROFESSOR: Nope. No, so you have to work through this to convince yourself that it's not delayed. So if it would be delayed by one, what's the first thing that you're yielding.

AUDIENCE: I don't know.

PROFESSOR: Yeah, so no delay.

AUDIENCE: OK.

PROFESSOR: OK, cool. So let's see, what do we have to implement in DNA seq, sub-sequence hashes. Do people have an idea of how to implement that now? Yes? Does it make sense for everyone?

So you build it as if you were building a list, and then you use yield to make it fast. And by fast I mean less memory.

OK, how about interval sub-sequence hashes? The one below.

AUDIENCE: Is that just like rolling hash, except you, like, have a step in your range?

PROFESSOR: OK, so it's like having a step in your range. So how can you do that? What's one way of doing it?

AUDIENCE: [INAUDIBLE] hashes?

PROFESSOR: Did anyone solve the PSET yet? Yes, OK how did you guys do it? Wait, no. That's a bad question because you guys can answer too much.

So interval sub-sequence hashes versus sub-sequence hashes. Did you copy paste the code?

AUDIENCE: Absolutely.

PROFESSOR: OK, so one way of doing it is copy and pasting the code. The problem if you copy paste the code is then you're not DRY. There's this engineering thing-- DRY means do not repeat yourself. So if you're not DRY, if you copy paste, then suppose you find the bug later. Suppose you run the big test and it crashes somewhere. And you fix a bug in sub-sequence hashes.

AUDIENCE: Oh, we're supposed to, like, call sub-sequence hashes from interval sub-sequence hashes, right?

PROFESSOR: That's another way of doing it that is DRY. So this way you're not copy pasting the code.

AUDIENCE: We're inlining the code.

PROFESSOR: You're inlining it manually, right? All right. So the problem, if you do this on a large scale, like when you go work somewhere, is that you end up with 20 copies of the same code. And then five of them have bug fixes and the other 15 don't, because people forgot where they are. So ideally, try to keep your code DRY.

AUDIENCE: So, basically, a list of tuples, right?

PROFESSOR: OK, so a list of tuples. What does a tuple have?

AUDIENCE: The index at which the sub-sequence operates?

PROFESSOR: So two indexes, right? The index in the first sub-sequence, say--

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK, say i1 and then the index in a second sequence, for the same sub-sequence, r right? And then i1, i2 prime, i1, i2 second, so on and so forth. So you have the same sub-sequence in the first sequence matches more things in the second one. This is how you're supposed to return them.

AUDIENCE: Does the order matter?

PROFESSOR: I hope not. OK, any questions on this? We went through generators fast. You guys are smart. Yes?

AUDIENCE: Can you explain how the imaging works? Like, how they create the [INAUDIBLE] on tuples.

PROFESSOR: No.

[LAUGHTER]

PROFESSOR: Sorry, I do not know.

AUDIENCE: Wait, which part?

AUDIENCE: So we yield the tuples. But I don't really get how they come up with the image from it.

AUDIENCE: From the tuples? Oh, I mean, I guess they're probably values.

AUDIENCE: Yeah, because I thought if you compared two strings of DNA that had the exact same, I thought it would be like a diagonal line down, not just a small black box.

PROFESSOR: OK.

AUDIENCE: So I don't think I'm understanding how they, like, image it.

PROFESSOR: So you're supposed to get-- your image has some things here, and a match is going to give you a big diagonal line that's stronger than everything else, right?

AUDIENCE: It's really fanned out.

PROFESSOR: Well I don't have thin chalk.

AUDIENCE: No, no, there's like one really dark black box, that's like really black. So I thought that meant that all the tuples are there, and everything else is just kind of gray.

PROFESSOR: Good question. I will have to think about that--

AUDIENCE: --supposed to be there. Is it like a notation thing, or--

PROFESSOR: I think that black box is supposed to be there. Did anyone try comparing two things that shouldn't match, like the dog and the monkey?

AUDIENCE: Yeah. And the entire thing was like dark.

PROFESSOR: Yeah.

AUDIENCE: --against, like, two same DNAs everything was very light. And there was like a very, very light gray line. But I thought that would be like black.

PROFESSOR: So I think how black it is means relative to all the sub-sequences, how long it is--

how long the sub-sequence you're recording is. Either that or how many. There is a function somewhere in there that computes the intensity of a pixel, that's square root of order 4 of something.

OK, and I can look at that now and tell you.

AUDIENCE: It's OK. It's not super important.

PROFESSOR: Or we can talk about amortized analysis for a bit. Yay! Let's talk about amortized analysis.

So this is what you're supposed to get, that's what matters.

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK, so amortized analysis, what's the example that we talked about in class?

AUDIENCE: It's like list expansion?

PROFESSOR: OK, so you have-- you have a list. And we know that the list is stored as an array, right? So this means that you can do indexing in constant time. So if you want to get the first element, order 1. If you want to get the millionth element, order 1.

This is not true if you had a link list instead. The millionth element would be order a million.

So this is an array. What do we implement? What's the operation that we implement on this list?

AUDIENCE: Insert--

PROFESSOR: Insert, append, push. Let's go for append, because that's what Python calls it. OK, so append puts an element at the end of the list, right? So how does append work?

AUDIENCE: The array is not full.

PROFESSOR: OK. So say I have some count variable here. So if the length of the array is bigger than count then what do I do?

AUDIENCE: Then we can directly insert. And because we're looking up in an array and we're doing constant time.

PROFESSOR: OK.

AUDIENCE: And so an order amount of information in x [INAUDIBLE]?

PROFESSOR: Sorry?

AUDIENCE: Order amount of information of x [INAUDIBLE]? Or do we just--

PROFESSOR: Let's say this is our reference, so it's constant time.

AUDIENCE: Otherwise we don't have enough room in our array. So we need to make it bigger.

PROFESSOR: OK. So we have array 2 becomes new array of size 2 times count, right? Copy everything from--

AUDIENCE: --length of the array. I guess they're the same.

PROFESSOR: I hope they're the same.

AUDIENCE: It is.

PROFESSOR: Yeah, I'd say that. So copy from array to-- let's do this-- to array 2. And then array 2 becomes array. And then this code here goes here, right? So there's a better way to write this if statement so the code isn't duplicated.

OK, so if the length is bigger than how many elements I have, if I still have room in the array, what's the cost? What's the running time? Constant. Oh, let's put it on the left.

OK, if I have to resize the array, what's the cost?

AUDIENCE: [INAUDIBLE]

PROFESSOR: So, if I did an operations, what then, right? N is the size of the array. If the only operation I have is append, then I can say n operations will cause the array of grow to size n. So n where n is the number of operations.

AUDIENCE: You mean, like, re-adding to the--

PROFESSOR: So an operation is a data structure operation, like a query or an update. This is my update and this is my query.

AUDIENCE: Wait, but like, it's order n though, because--

PROFESSOR: Yeah.

AUDIENCE: I know, it's order n. But because we have like an array, and then you have to make a new one, and you have to move all those old items over, right?

PROFESSOR: Yep.

AUDIENCE: OK. But, I mean, sometimes like, if your actual array, if you expand it before-- like, let's say you notice you're getting full and you decide to like make it bigger at that point, is it still order n, as in the number of elements that are--

PROFESSOR: It depends on how you decide. There's a problem on the PSET that asks you about that. So, depends on when you make the decision and how you make the decision, the answer is either yes, you're still constant time, or no.

So if you understand the amortized analysis then you can argue of whether it still holds or not. If this breaks down at any point, not going to be constant time. Yes?

AUDIENCE: So the only cost is really copying everything from the old array to the new array?

PROFESSOR: Yes.

AUDIENCE: Actually allocating that space is--

PROFESSOR: We assume that allocating the space is constant time. Good question, because you can't take that for granted, right? So we assume that this is order 1, copying is order n. And then the insertion is order 1, just like before.

So allocating may not be constant. In real life, allocating is actually logarithmic either of the size that you're asking for or logarithmic of how many buffers you've allocated. And you can make a constant time allocator. But that's lower than a logarithmic allocator, because the constant factor behind it is so big.

But even if this allocation would be order n, which would be terrible, it would still get absorbed here. So the overall model works no matter what the allocation is. It's reasonable, from a theoretical standpoint, to say that allocation is order 1, from a theoretical standpoint.

So this is the real cost copying the elements. And this makes an append order n worst case. So if you look at this data structure then suppose we want to compute the cost of an append. So say we have code like this, 4, 1, 2, n. First we have L be an empty list. Then we want to compute the cost of this.

So if we do it without amortized analysis, line by line analysis, just like we learned in the first lecture, what's the cost of this, making a new list constant? What's the cost of one append?

AUDIENCE: Constant.

PROFESSOR: One append. So an append can either branch here or branch here. So what's the cost of one append?

AUDIENCE: It would be showing with an empty list?

AUDIENCE: Depends.

PROFESSOR: It depends. So worst case. We have to look at a worst case. So this is line by line analysis. We're going to get one number for this.

AUDIENCE: N.

AUDIENCE: An n.

PROFESSOR: Yep. So in the worst case, the list will be full. And you'll have to make a new one.

And then you're going on this branch of the if, so the cost is order n.

So order n, worst case. So the cost of one call is order n, worst case. How many calls do we make?

So what is the total cost of this thing?

AUDIENCE: It's not actually n squared.

PROFESSOR: Yes, it's not actually n squared. But if we do line by line analysis, before we learn amortized analysis, all we can say it's order of n squared. And this is correct, it's not bigger than n squared, right?

So O is correct. But it's not the tight bound. So if we had a multiple choice, and you selected this, you wouldn't get the score because we usually ask you what the tightest bound that you can get.

OK, so line by line analysis. We worked through that a lot in doc dist. Doesn't work all the time. When it doesn't work, we tell you to use amortized analysis instead.

So what's the goal of amortized analysis? What do we want? You guys are yelling at me that this is not n squared, why? I mean not why, what? What is it instead? What do we want from amortized analysis?

AUDIENCE: [INAUDIBLE]

AUDIENCE: It's a [INAUDIBLE] that's an n.

PROFESSOR: So we want amortized analysis to say that this is order 1 amortized, and this is--

[ALARM SOUNDING]

PROFESSOR: Am I out of time? Yeah. OK, so there's a difference between the worst case and amortized, right? We can argue that this is order 1 amortized. And if this is order 1 amortized, then this is order n amortized.

So does the difference between worst case and amortized make sense now? So this is what I want, the rest is fancy math. If you forget the fancy math after you're done with this class, that's OK. If you remember that this is order 1 amortized, and that's order n amortized, that's good. That's all you need to know to write code if you don't design algorithms.

So this is an important piece of knowledge on its own. OK, so questions about the difference between worst case and amortized? OK, what does amortized mean?

AUDIENCE: Average.

PROFESSOR: Yep, averaged out over multiple operations. So instead of doing line by line analysis, we have to look at what happens over multiple operations, right? So there are two methods that I think are useful in CLRS. There are three in total, but the last one is horribly complicated.

So there's something called aggregate analysis. And there's something called the cost based accounting. So last time when we looked at the costs for append, we argued that, hey, it's order 1 for a lot of times. And then it's only order n for an operation that's a power of 2.

So if we're looking at the K-ith append, then this is order K for K equals 2 to the i. And it's order 1 otherwise. Right?

So if we sum up all these costs, we get-- plus sum over log n of O of 2 to the i. And this is clearly order n. And if you do the math here, this is also order n.

So this is aggregate analysis. This is what we taught you in lecture. Does this make sense?

So the key here is that whenever we are increasing the array, we're increasing it to 2 times. And we start with a size of 1, count is 1. We start with an array with 1 element. So the size of the array will first be 1, then 2, then 4, then 8, then 16, 32, 64, 128, so on so forth. It increases exponentially.

So on the first append I'll have to do a resize. On the second one, resize. Fourth one, resize. Eighth, resize, so on and so forth.

So if I'm adding up the cost for n operations, each operation is order 1 because I'm inserting everywhere. And then all these operations are all order n. But there's few of them. They're few and far out.

So if you write the sum this way, and you do the math, you get that it's order n. So aggregate analysis says, look at n operations and add the costs up together. And last time we had that good example of walking over a tree, and in order traversal where we drew arrows across edges. So that's aggregate analysis.

And then you should look at the cost method in CLRS because that's also useful sometimes. Does this help? Any questions? No, everyone wants to go home.

AUDIENCE: Wait--

PROFESSOR: Almost.

AUDIENCE: For log n, so you're starting from log n going to--

PROFESSOR: So I'm starting from 1 going to log n.

AUDIENCE: Oh, oh, so [INAUDIBLE] after you're buffering.

PROFESSOR: So this is fancy math for saying only add up powers of two. So that's what I'm trying to say, add these guys up.

AUDIENCE: Well that's your step [INAUDIBLE].

PROFESSOR: Yeah.

AUDIENCE: Oh, OK. Oh, I like that. OK.

PROFESSOR: OK.