[SQUEAKING]

[RUSTLING]

[CLICKING]

ERIK DEMAINE: Good morning.

AUDIENCE: Morning!

ERIK DEMAINE: Welcome to 006, Introduction to Algorithms, lecture two. I am Erik Demaine and I love algorithms. Are you with me?

[APPLAUSE]

Yeah. Today, we're not doing algorithms. No, we're doing data structures. It's OK. There's lots of algorithms in each data structure. It's like multiple algorithms for free. We're going to talk about sequences, and sets, and linked lists, and dynamic arrays. Fairly simple data structures today. This is the beginning of several data structures we'll be talking about in the next few lectures. But before we actually start with one, let me tell you/remind remind you of the difference between an interface-- which you might call an API if you're a programmer, or an ADT if you're an ancient algorithms person like me-- versus a data structure.

These are useful distinctions. The idea is that an interface says what you want to do. A data structure says how you do it. So you might call this a specification. And in the context of data structures, we're trying to store some data. So the interface will specify what data you can store, whereas the data structure will give you an actual representation and tell you how to store it. This is pretty boring. Just storing data is really easy. You just throw it in a file or something. What makes it interesting is having operations on that data. In the interface, you specify what the operations do, what operations are supported, and in some sense, what they mean. And the data structure actually gives you algorithms-- this is where the algorithms come in-- for how to support those operations. All right.

In this class, we're going to focus on two main interfaces and various special of them. The idea is to separate what you want to do versus how to do it. Because-- you can think of this as the problem statement. Yesterday-- or, last class, Jason talked about problems and defined what a problem was versus algorithmic solutions to the problem. And this is the analogous notion for data structures, where we want to maintain some data according to various operations.

The same problem can be solved by many different data structures. And we're going to see that. And different data structures are going to have different advantages. They might support some operations faster than others. And depending on what you actually use those data structures for, you choose the right data structure. But you can maintain the same interface. We're going to think about two data structures. One is called a set and one is called a sequence. These are highly loaded terms. Set means something to a mathematician. It means something else to a Python programmer. Sequence, similarly. I guess there's not a Python sequence data type built in.

The idea is, we want to store n things. The things will be fairly arbitrary. Think of them as integers or strings. And on the one hand, we care about their values. And maybe we want to maintain them in sorted order and be able to search for a given value, which we'll call a key. And on the other hand, we care about representing a particular sequence that we care about. Maybe we want to represent the numbers 5, 2, 9, 7 in that order and store that. In Python, you could store that in a list, for example. And it will keep track of that order. And this is the first item, the second item, the last item.

Today, we're going to be focusing on this sequence data structure, although at the end, I'll mention the interface for sets. But we're going to be actually solving sequences today. And in the next several lectures, we'll be bouncing back and forth between these. They're closely related. Pretty abstract at the moment. On the other hand, we're going to have two main-- let's call them data structure tools or approaches. One is arrays and the other is pointers-- pointer-based or linked data structures. You may have seen these. They're used a lot in programming, of course. But we're going to see both of these today. I'll come back to this sort of highlight in a moment.

Let's jump into the sequence interface, which I conveniently have part of here. There's a few different levels of sequences that we might care about. I'm going to start with the static sequence interface. This is where the number of items doesn't change, though the actual items might. Here, we have n items. I'm going to label them x 0 to x n minus 1, as in Python. So the number of items is n. And the operations I want to support are build, length, iteration, get, and set.

So what do these do? Build is how you get started. To build a data structure in this interface, you call build of x. Exactly how you specify x isn't too important, but the idea is, I give you some items in some order. In Python, this would be an iterable. I'm going to want to also know its length. And I want to make a new data structure of size and a new static sequence of size n that has those items in that order. So that's how you build one of these. Because somehow, we have to specify n to this data structure, because n is not going to be allowed to change.

I'm going to give you a length method. Methods are the object-oriented way of thinking of operations that your interface supports. Length will just return this fixed value n. Iter sequence. This is the sense in which we want to maintain the order. I want to be able to output x 0 through x n minus 1 in the sequence order, in that specified order that they were built in or that it was changed to. This is going to iterate through all of the items. So it's going to take at least linear time to output that.

But more interesting is, we can dynamically access anywhere in the middle of the sequence. We can get the ith item, x i, given the value i, and we can change x i to a given new item. OK. So that's called get_at and set_at. Pretty straightforward. This should remind you very closely of something-- a data structure. So this is an interface. This is something I might want to solve. But what is the obvious data structure that solves this problem? Yeah?

AUDIENCE: A list.

ERIK DEMAINE: A list. In Python, it's called a list. I prefer to call it an array, but to each their own. We're going to use "list." List could mean many things, but the solution to this interface problem-- the natural solution-- is what I'll call a static array. Jason mentioned these in lecture one. It's a little tricky because there are no static arrays in Python. There are only dynamic arrays, which is something we will get to. But I want to talk about, what is a static array, really? And this relates to our notion of-- our model of computation, Jason also talked about, which we call the word RAM, remember?

The idea, in word RAM, is that your memory is an array of w-bit words. This is a bit circular. I'm going to define an array in terms of the word RAM, which is defined in terms of arrays. But I think you know the idea. So we have a big memory which goes off to infinity, maybe. It's divided into words. Each word here is w bits long. This is word 0, word 1, word 2. And you can access this array randomly-- random access memory. So I can give you the number 5 and get 0 1, 2, 3, 4, 5, the fifth word in this RAM. That's how actual memories work. You can access any of them equally quickly. OK, so that's memory.

And so what we want to do is, when we say an array, we want this to be a consecutive chunk of memory. Let me get color. Let's say I have an array of size 4 and it lives here. Jason can't spell, but I can't count. So I think that's four. We've got-- so the array starts here and it ends over here. It's of size 4. And it's consecutive, which means, if I want to access the array at position-- at index i, then this is the same thing as accessing my memory array at position-- wherever the array starts, which I'll call the address of the array-- in Python, this is ID of array-- plus i. OK.

This is just simple offset arithmetic. If I want to know the 0th item of the array, it's right here, where it starts. The first item is one after that. The second item is one after that. So as long as I store my array consecutively in memory, I can access the array in constant time. I can do get_at and set_at as quickly as I can randomly access the memory and get value-- or set a value-- which we're assuming is constant time. My array access is constant time.

This is what allows a static array to actually solve this problem in constant time per get_at and set_at operation. This may seem simple, but we're really going to need this model and really rely on this model increasingly as we get to more interesting data structures. This is the first time we're actually needing it. Let's see. Length is also constant time. We're just going to store that number n, along with its address. And build is going to take linear time. Iteration will take linear time. Pretty straightforward.

I guess one thing here, when defining build, I need to introduce a little bit more of our model of computation, which is, how do you create an array in the beginning? I claim I could do it in linear time, but that's just part of the model. This is called the memory allocation model. There are a few possible choices here, but the cleanest one is just to assume that you can allocate an array of size n in theta n time. So it takes linear time to make an array of size n. You could imagine this being constant. It doesn't really matter much. But it does take work.

And in particular, if you just allocate some chunk of memory, you have no idea whether it's initialized. So initializing that array to 0s will cost linear time. It won't really matter, constant versus linear, but a nice side effect of this model is that space-- if you're just allocating arrays, the amount of space you use is, at most, the amount of time you use. Or, I guess, big O of that. So that's a nice feature. It's pretty weird if you imagine-- it's unrealistic to imagine you can allocate an array that's infinite size and then just use a few items out of it. That won't give you a good data structure. So we'll assume it costs to allocate memory. OK, great. We solved the sequence problem. Very simple, kind of boring. These are optimal running times.

Now, let's make it interesting-- make sure I didn't miss anything-- and talk about-- oh, there is one thing I want to talk about in the word RAM. A side effect of this assumption that array access should take constant time, and that accessing these positions in my memory should take constant time, is that we need to assume w is at least log n or so. w, remember, is the machine word size. In real computers, this is currently 64-- or 256, in some bizarre instructions. But we don't usually think of the machine as getting bigger over time, but you should think of the machine as getting bigger over time.

This is a statement that says, the word size has to grow with n. It might faster than log n, but it has to grow at least as fast as log n. Why do I say that? Because if I have n things that I'm dealing with-- n, here, is the problem size. Maybe it's the array I'm trying to store-- whatever. If I'm having to deal with n things in my memory, at the very least, I need to be able to address them. I should be able to say, give me the ith one and represent that number i in a word. Otherwise-- because the machine is designed to only work with w-bit words in constant time, they'll want to be able to access the ith word in constant time, I need a word size that's at least log n just to address that and n things in my input.

So this is a totally reasonable assumption. It may seem weird because you think of a real machine as having constant size, but a real machine has constant size RAM, also. My machine has 24 gigs of RAM, or whatever. That laptop has 8. But you don't think of that as changing over time. But of course, if you want it to process a larger input, you would buy more RAM. So eventually, when our n's get really, really big, we're going to have to increase w just so we can address that RAM. That's the intuition here.

But this is a way to bridge reality, which are fixed machines, with theory. In. Algorithms, we care about scalability for very large n. We want to know what that growth function is and ignore the lead constant factor. That's what asymptotic notation is all about. And for that, we need a notion of word size also changing in this asymptotic way. All right. That would be more important next week, when we talk about hashing and why hashing is a reasonable thing to do.

But let's move on to dynamic sequences, which is where things get interesting. I have the update here. We start with static sequences. All of these operations are still something we want to support in a dynamic sequence, but we add two dynamic operations-- somewhat controversial operations, very exciting. I want to be able to insert in the middle of my sequence and I want to be able to delete from the middle of my sequence. Here's my sequence, which I'm going to think of in a picture. I'm going to draw it as an array. But it's stored however it's stored. We don't know. This is an interface, not an implementation. So we have x 0, x 1, x 2, x 3.

And let's say I insert at position 2. Position 2 is here. So I come in with my new x, and I would like x to be the new x 2, but I don't want to lose any information. If I did set_at 2, then I would erase this and replace it with x. But I want to do insert_at, which means all of these guys, conceptually, are going to shift over by 1 in terms of their indices. Then, I would get this picture that's one bigger. And now I've got the new x. I've got what was the old x 2, which I don't-- I hesitate to call x 2 because that's its old name, not its new name. I'm going to draw arrows to say, these guys get copied over. These ones are definitely unchanged. Our new x 2, which prime is x This is x3 prime, 4 prime, and so on. I want to be careful here-- and of course, the new n prime is n plus 1.

I want to be careful about the labeling, because the key-- what makes insert_at interesting is that, later, when I call get_at, it's with the new indexing. So previously, if I did get_at at 2, I would get this value. And afterwards, if I did get_at at 2, I would get the new value. If I did get_at at 3 down here, I would get the value that used to be X 2. That's maybe hard to track. But this is a conceptually very useful thing to do, especially when you're inserting or deleting at the ends. So we're going to define, in particular, insert and delete first and last. These are sometimes given-- if you have an insert, it has an x. If you do a delete, it has no argument.

This means insert_at the beginning of the array, which would be like adding it here. And insert_last means adding it on here. insert_last doesn't change the indices of any of the old items. That's a nice feature of insert_last. Insert-first changes all of them. They all get incremented by 1. And we're also interested in the similar things here. We could do get-first or -last or set-first or -last, which are the obvious special cases of get_at and set_at. Now, these special cases are particularly interesting in an algorithms context. If you were a mathematician, you would say, well, why do I even bother? This is just shorthand for a particular call to get or set.

But what makes it interesting from a data structures perspective is that we care about algorithms for supporting these operations. And maybe, the algorithm for supporting get-first or set-first, or in particular, insert-first or insert_last, might be more efficient. Maybe we can solve this problem better than we can solve insert_at. So while, ideally, we could solve the entire dynamic sequence interface constant time preparation, that's not actually possible. You can prove that. But special cases of it-- where we're just inserting and leading from the end, say-- we can do that. That's why it's interesting to introduce special cases that we care about. Cool. That's the definition of the dynamic sequence interface. Now, we're going to actually solve it.

Our first data structure for this is called linked lists. You've taken, probably-- you've probably seen linked lists before at some point. But the main new part here is, we're going to actually analyze them and see how efficiently they implement all of these operations we might care about. First, review. What is a linked list? We store our items in a bunch of nodes. Each node has an item in it and a next field. So you can think of these as class objects with two class variables, the item and the next pointer.

And we assemble those into this kind of structure where we store-- in the item fields, we're going to store the actual values that we want to represent in our sequence, x 0 through x n minus 1, in order. And then we're going to use the next pointers to link these all together in that order. So the next pointers are what actually give us the order. And in addition, we're going to keep track of what's called the head of the list. The data structure is going to be represented by a head. If you wanted to, you could also store length. This could be the data structure itself. And it's pointing to all of these types of data structures.

Notice, we've just seen an array-based data structure, which is just a static array, and we've seen a pointer-based data structure. And we're relying on the fact that pointers can be stored in a single word, which means we can de-reference them-- we can see what's on the other side of the pointer-- in constant time in our word RAM model. In reality, each of these nodes is stored somewhere in the array of the computer. So maybe each one is two words long, so maybe one node is-- the first node is here. Maybe the second node is here. The third node is here. They're in some arbitrary order.

We're using this fact, that we can allocate an array of size n in linear time-- in this case, we're going to have arrays of size 2. We can just say, oh, please give me a new array of size 2. And that will make us one of these nodes. And then we're storing pointers. Pointers are just indices into the giant memory array. They're just, what is the address of this little array? If you've ever wondered how pointers are implemented, they're just numbers that say where, in memory, is this thing over here? And in memory, they're in arbitrary order. This is really nice because it's easy to manipulate the order of a linked list without actually physically moving nodes around, whereas arrays are problematic. Maybe it's worth mentioning.

Let's start analyzing things. So we care about these dynamic sequence operations. And we could try to apply it to the static array data structure, or we could try to implement these operations in a static array. It's possible, just not going to be very good. And we can try to implement it with linked lists. And it's also not going to be that great. Let's go over here. Our goal is the next data structure, which is dynamic arrays. But linked lists and static arrays each have their advantages.

Let's first analyze dynamic sequence operations, first on a static array and then on a linked list. On a static array, I think you all see, if I try to insert at the beginning of the static array-- that's kind of the worst case. If I insert first, then everybody has to shift over. If I'm going to maintain this invariant, that the ith item in the array represents-- I guess I didn't write it anywhere here. Maybe here. Static array. We're going to maintain this invariant that a of i represents x i. If I want to maintain that at all times, when I insert a new thing in the front, because the indices of all the previous items change, I have to spend time to copy those over. You can do it in linear time, but no better.

Static array. Insert and delete anywhere costs theta n time-- actually, for two reasons. Reason number one is that, if we're near the front, then we have to do shifting. What about insert or delete the last element of an array? Is that any easier? Because then, if I insert the very last element, none of the indices change. I'm just adding a new element. So I don't have to do shifting. So can I do insert and delete last in constant time in a static array? Yeah?

AUDIENCE: No, because the size is constant.

ERIK DEMAINE: No, because the size is constant. So our model is that remember allocation model is that we can allocate a static array of size em but it's just a size n I can't just say please make it bigger by 1 I need I need space to store this extra element. And if you think about where things are in memory, when you call to this memory allocator, which is part of your operating system, you say, please give me a chunk of memory. It's going to place them in various places in memory, and some of them might be next to each other.

So if I try to grow this array by 1, there might already be something there. And that's not possible without first shifting. So even though, in the array, I don't have to do any shifting, in memory, I might have to do shifting. And that's outside the model. So we're going to stick to this model of just-- you can allocate memory. You can also de-allocate memory, just to keep space usage small. But the only way to get more space is to ask for a new array. And that new array won't be contiguous to your old one. Question?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: What is the dynamic array will be the next topic, so maybe we'll come back to that. Yeah? In a static array, you're just not allowed to make it bigger. And so you have to allocate a new array, which we say takes linear time. Even if allocating the new array didn't take linear time, you have to copy all the elements over from the old array to the new one. Then you can throw away the old one. Just the copying from an array of size n to an array of size n plus 1, that will take linear time. So static arrays are really bad for dynamic operations-- no surprise. But you could do them. That's static array.

Now, linked lists are going to be almost the opposite-- well, almost. If we store the length, OK, we can compute the length of the array very quickly. We can insert and delete at the front really efficiently. If I want to add a new item as a new first item, then what do? I do I allocate a new node, which I'll call x. This is insert-first of x. I'll allocate a new array of size 2. I'm going to change-- let me do it in red. I'm going to change this head pointer. Maybe I should do that later. I'm going to set the next pointer here to this one, and then I'm going to change this head pointer to point to here. And, boom, now I've got a linked list.

Again, we don't know anything about the order and memory of these lists. We just care about the order that's represented implicitly by following the next pointers repeatedly. Now, I've got a new list that has x in front, and then x 0, and then x 1, and so on. So insert- and delete_first, at least are really efficient. We won't get much more than that, but the linked list, insert- and delete_first are constant time. So that's cool. However, everything else is going to be slow. If I want to get the 10th item in a linked list, I have to follow these pointers 10 times. I go 0, 1, 2, 3, and so on. Follow 10 next pointers and I'll get the 10th item. Accessing the ith item is going to take order i time. Get- and set_at need i time, which, in the worst case, is theta n.

We have sort of complementary data structures here. On the one hand, a static array can do constant time get_at/set_at. So it's very fast at the random access aspect because it's an array. Linked lists are very bad at random access, but they're better at being dynamic. We can insert and delete-- at the beginning, at least-- in constant time. Now, if we want to actually insert and delete at a particular position, that's still hard, because we have to walk to that position. Even inserting and leading at the end of the list is hard, although that's fixable. And maybe I'll leave that for problem session or problem set. But an easy-- here's a small puzzle. Suppose you wanted to solve get-last efficiently in a linked list. How would you solve that in constant time? Yeah?

AUDIENCE: Doubly linked list.

ERIK DEMAINE: Doubly linked list. It's a good idea, but actually not the right answer. That's an answer to the next question I might ask. Yeah?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: [INAUDIBLE] pointer to the last element. That's all we need here. And often, a doubly linked list has this. They usually call this the tail-- head and tail. And if we always just store a pointer to the last list-- this is what we call data structure augmentation, where we add some extra information to the data structure and-- we have to keep it up to date all the time. So if we do an insert_last or something, insert_last also becomes easy because I can just add a new node here and update the pointer here. delete_last is trickier. That's where you get a doubly linked list. But whenever I add something to the end of this list, I have to update the tail pointer also. As long as I maintain this, now, suddenly get-last is fast in constant time.

So linked lists are great if you're working on the ends, even dynamically. Arrays are great if you're doing random access and nothing dynamic-- nothing adding or deleting at the ends or in the middle. Our final goal for today is to get sort of the best of both worlds with dynamic arrays. We're going to try to get all of the good running times of linked lists and all of the good running times of static arrays. We won't get quite all of them, but most of them.

And in some sense, another way to describe what these introductory lectures are about is telling you about how Python is implemented. What we're going to talk about next, dynamic arrays, I've alluded to many times. But these are what Python calls lists. You don't have to implement a dynamic array by hand because it's already built into many fancy new languages for free, because they're so darn useful. This lecture is about how these are actually implemented and why they're efficient. And in recitation nodes, you'll see how to actually implement them if all you had were static arrays. But luckily, we have dynamic arrays, so we don't have to actually implement them. But inside the Python interpreter, this is exactly what's happening.

The idea is to relax the constraint-- or the invariant, whatever-- that the size of the array we use equals n, which is the number of items in the sequence. Remember, in the sequence problem, we're supposed to represent n items. With a static array, we allocated an array of size exactly n. So let's relax that. Let's not make it exactly n. Let's make it roughly n. How roughly, you can think about for a while. But from an algorithms perspective, usually, when we say roughly, we mean throw away constant factors. And that turns out to be the right answer here. It's not always the right answer.

But we're going to enforce that the size of the array is theta n-- probably also greater than or equal to n. 0.5 n would not be very helpful. So it's going to be at least n, and it's going to be at most some constant times n. 2n, 10n, 1.1 times n. Any of these constants will work. I'm going to use 2n here, but there are lots of options. And now, things almost work for free. There's going to be one subtlety here. And I'm going to focus on-- we're still going to maintain that the ith item of the array represents x i. This data structure-- let me draw a picture.

We've got an array of some size. The first few items are used to store the sequence. But then, there's going to be some blank ones at the end. Maybe we'll keep track of this-- so the data structure itself is going to have an array and it's going to have a length. Something like this. We're also going to keep track of the length. So we know that the first length items are where the data is, and the remainder are meaningless. So now, if I want to go and do an insert_last, what do I do? I just go to a of length and set it to x. And then I increment length. Boom. Easy. Constant time. Yeah?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: How do you have enough room. Indeed, I don't. This was an incorrect algorithm. But it's usually correct. As long as I have extra space, this is all I need to do for insert_last. But I am also going to store the size of the array. This is the actual-- this whole thing is size, and this part is length. Length is always going to be less than or equal to size. And so there's a problem. If length equals size, then I don't have any space.

Just add to end unless n equals size. I'm using n length for the same thing. So length here is the same as n. That's our actual number of things we're trying to represent. And size-- this is great. This is the interface size. This is what we're trying to represent. And this is the representation size. This is the size of my array. These are the number of items I'm trying to store in that array. This is the interface versus data structure. Here's the interface. Here's the data structure. OK, cool.

What do I do in the case when n equals size? I'm going to have to make my array bigger. This should sound just like static arrays. For static arrays, we made our array bigger every time we inserted. And that was this linear cost of allocation. We're going to do that sometimes. With static arrays, we had to do it every single time, because size equaled n. Now, we have some flexibility. We're only going to do it sometimes. It's like, cookies are a sometimes food, apparently, according to modern Cookie Monster. I don't understand. But if n equals size, we're going to allocate a new array of size-- any suggestions?

AUDIENCE: Bigger.

ERIK DEMAINE: Bigger. I like it. Greater than size. How much bigger?

AUDIENCE: Twice.

ERIK DEMAINE: Twice.

JASON KU: Five things.

ERIK DEMAINE: Five things. Size plus 5? Come on, Jason. Trolling me. All right. There are a couple of natural choices here. One is a constant factor larger. You could use 1.1, or 1.01, or two, or 5, or 10. They will all work. Or you could use Jason's trolling answer of size plus a constant, like 5. Why is this bad? Yeah?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: You'll have to do it again. You'll have to resize frequently. When? Five steps later. In the original static array, we were reallocating every single time. That's like n plus 1. If we do n plus 5, that really doesn't change things if we ignore constant factors. Now, we'll have to spend linear time every five steps instead of linear time every one step. That's still linear time per operation, just, we're changing the constant factor. Whereas 2 times size, well, now we have to think a little bit harder. Let's just think about the case where we're inserting at the end of an array. Let's say we do n insert_lasts from an empty array.

When do we resize? Well, at the beginning-- I guess I didn't say what we do for an empty array. Let's say size equals 1. We can insert one item for free. As soon as we insert the second item, then we have to resize. That seems bad. Immediately, we have to resize. Then we insert the third item. OK, now let's draw a picture. So we start with one item. We fill it up. Then, we grow to size 2, because that's twice 1. Then we fill it up. Immediately, we have to resize again. But now we start to get some benefit. Now, we have size 4, and so we can insert two items before we have to resize. And now, we're size 8, and we get to insert four items before we refill.

This is going to resize-- and again, resizes are expensive both because we have to pay to allocate the new array-- I drew it as just extending it, but in fact, we're creating a whole new array, and then we have to copy all of the items over. So there's the allocation cost and then the copying costs. It's linear either way. But we're going to resize at n equals 1, 2, 4, 8, 16-- you know this sequence. All the powers of 2, because we're doubling. That is exactly powers of 2. So we pay a linear cost.

This resize cost, the allocation and the copying, is going to be-- it's linear each time. So it's 1 plus 2 plus 4 plus 8 plus 16. Really, I should write this as sum from i equals 1 to roughly log n. Log base 2 of n is lG of 2 to the i. If you want a terminus here, it's roughly n. It's actually the next-- the previous power of 2 of n, or something. But that won't matter. That will just affect things by a constant factor. What is the sum of 2 to the i? This is a geometric series. Anyone know the answer? Yeah?

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: 2 to the top limit plus 1 minus 1. Yeah. So this is the identity. Sum of 2 to the i from i equals 1 to k is 2 to the k plus 1, plus 1 minus 1. So the plus 1 is upstairs. The minus one is downstairs. An easy way to remember this is if you think in binary-- as we all should. We're computer scientists. 2 to the i means you set the ith bit to 1. Here's a bit string. This is the ith bit. This is 2 to the i. 0 is down here. If I sum them all up, what that means is, I'm putting 1s here. And if you think about what this means, this is up to k from 0-- sorry, I should do 0 to be proper. If I write-- that's the left-hand side. The right-hand side is 2 to the k plus 1, which is a 1 here, and the rest 0s.

So if you know your binary arithmetic, you subtract-- if you add 1 to this, you get this. Or if you subtract 1 from this, you get this. This is why this identity holds. Or the higher-level thing is to say, oh, this is a geometric series. So I know-- you should know this. I'm telling you now. Geometric series are dominated by the last term-- the biggest term. If you have any series you can identify as geometric, which means it's growing at least exponentially, then in terms of theta notation, you can just look at the last term and put a theta around it, and you're done. So this is theta of the last term, like 2 to the log n, which is theta n. Cool.

Linear time. Linear time for all of my operations. I'm doing n operations here, and I spent linear total time to do all of the resizing. That's good. That's like constant each, kind of. The "kind of" is an important notion which we call amortization. I want to say an operation takes t of n amortized time if, let's say, any k of those operations take, at most, k times t of n time. This is a little bit sloppy, but be good enough. The idea is here, if this works for n or k, to do n operations from an empty array here takes linear time, which means I would call this constant amortized. Amortized means a particular kind of averaging-- averaging over the sequence of operations.

So while individual operations will be expensive, one near the end, when I have to resize the array, is going to take linear time just for that one operation. But most of the operations are cheap. Most of them are constant. So I can think of charging that high cost to all of the other operations that made it happen. This is averaging over the operation sequence. Every insert_last over there only takes constant time, on average, over the sequence of operations that we do. And so it's almost constant. It's not quite as good as constant, worst case, but it's almost as good. And it's as good as you could hope to do in this dynamic array allocation model.

Let me put this into a table. And you'll find these in the lecture notes, also. We have, on the top, the main operations of sequence interface, which we will revisit in lecture seven. We'll see some other data structures for this. Get_at and set_at in the first column. Insert_ and delete_first, insert_ and delete_last, insert_ and delete_at an arbitrary position. We've seen three data structures now. Arrays were really good at get_at/set_at. They took constant time. That's the blue one. We're omitting the thetas here.

All of the other operations took linear time, no matter where they were. Linked lists were really good at insert- and delete_first. They took constant time, but everything else took linear time, in the worst case. These new dynamic arrays achieve get_at and set_at in constant time because they maintain this invariant here that a of i equals x i. So we can still do get- and set_at quickly. And we also just showed that insert_last is constant amortized.

delete_last, you don't have to resize the array. You could just decrease length and, boom, you've deleted the last item. It's not so satisfying, because if you insert n items and then delete n items, you'll still have an array of size theta n, even though your current value of n is 0. You can get around that with a little bit more trickery, which are described in the lecture notes. But it's beyond the-- we're only going to do very simple amortized analysis in this class-- to prove that that algorithm is also constant amortized, which it is. You'll see in 046, or you can find it in the CLRS book. That's it for today.