[SQUEAKING]

[RUSTLING]

[CLICKING]

ERIK DEMAINE: All right. Welcome to practice problem session 3, 006. Today, we are going to go through a bunch of problems, which you should have already. I was thinking of skipping the very first problem, because it's just a mechanical thing. If we have time at the end, we can come back to it. But there's not really any insight I can give you in how to approach this problem. It's just, do you understand hashing?

So I want to go into the more creative problems first. Let's start with problem 3.2, hash sequence. So I'll just read it. And then our first task is to convert the word problem into a concise formal algorithms thing we need to achieve. Then we need to come up with ideas for how to achieve it. And we need to check the details. That'll be our general pattern.

So this problem says, hash tables are not only useful for implementing set operations, they can also be used to implement sequences. Remember from lecture 2, we have a set interface, which is about querying items by their key and sort of intrinsic order that's about the items themselves, versus a sequence interface that we started out with, with linked lists and so on, and arrays, where we're given an order, and we want to maintain that order.

And that order may not have anything to do with the items themselves. So that's what we call an extrinsic order. We're told what the order is by saying, insert this item after this one, or append this one to the end, or prepend it to the beginning. So in lecture last week, we saw hash tables implement sets. And let me just remind you some things that they can do.

So we have, on the one hand, set hashing. So we're going to need this in a moment. This is just a reminder from lecture. We can build one in linear time expected. We can find an item in constant timed expected by key. And we can insert or delete an item in constant expected amortized.

OK, so this is a black box that we're given. And the problem statement says that, imagine you're given a hash table as a black box, which means we're giving a thing that behaves just like a-- thank you, 2. We're given something that is a hash table. But it's black box in the sense we're not allowed to reach in and change the implementation details. We're supposed to use it as is, just by calling its interface.

So in particular, we're giving these three operations. I'll maybe also use iter to iterate through the items. So we're allowed to build something in linear time, find, and insert and delete in constant expected amortized. And what the problem is asking is to build out of this data structure a sequence with particular time balance.

So this is what we call a reduction in that we're going to convert-- I guess technically, we're reducing the sequence problem to the set problem, because we're showing how to solve the sequence problem using the set problem. But the way we'll think about it is in the other direction. We're given a data structure that solves set. And we're going to convert it into a data structure that solves sequence.

So given that we already know how to do this from lecture, we're going to learn how to do this. This is teaching you new stuff in a problem set. So the specific bounds that we're told to achieve are build in constant expected time, get and set_at in constant expected time, insert and delete_at in linear expected time, and insert and delete first and last in-- we're running out of room here-- constant expected amortized.

OK. So this is just what we're told to do. And now we start thinking. So we're given this. We want to build this. And so I'm going to tell you a little bit about my thought process. When I'm presented with a problem like this, the first thing is to read the problem and see, OK, what's the hard part here? What are the challenges?

So clearly, we have to do all four of these types of operations. Build in linear expected time-- that's basically everything we've seen. Get or set_at in constant expected time-- that's fast. And that feels kind of like this find operation. So both of these seem pretty matchy-matchy. So that looks like a good mapping. I'm going to try to build these operations using those operations.

Insert and delete at a specific location in constant expected time-- sorry, linear expected time-- that's big. Linear expected time means I can rebuild the entire data structure every time I do an operation. So this is easy. OK, that's the first thing to realize. This is big.

Great. So I don't really have to worry about these operations. I mean, I do have to implement them. But it's not hard to do it that fast, because I can rebuild. And then here, insert and delete at the beginning and the end of the array, these are the DEQ, Double-Ended Queue operations, insert and delete at either end, in constant expected amortized time. This, I feel like, is a tricky one.

You've seen one way to do this in a problem set. But now we're going to see another way with-- OK, the other thing to notice is these "expected" words. In this case, we're told to use hashing. But with lot of the problems, you're not told how to solve it or what you should be basing your thing on.

And so "expected" is always a good keyword, because it means randomization is involved somehow. If you're told the bound is going to be expected, you probably need to use randomization. And in this class, the only form of randomization you will use is essentially hashing. So that's a good hint. In this case, we know what we're supposed to use hashing.

All right, so this is going to be the challenge. But any ideas on how we might tackle this problem? How can we-- so set, remember, every item has a key. In a sequence, items are just items. And we're told to insert and delete them at particular locations. But they don't have keys.

So one of the challenges is going to be to take our items here, give them keys so that we can store them in a set. Otherwise, we can't use find. If there's no keys there, there's no way to search by key. Ideas?

So let's think about what we want to do. Let's start with-- so build, I think, is fine. If you just want to build a data structure, you don't need to do anything. The hard part are the queries or updates you want to be able to do on your data structure.

Let's start with this operation, get and set_at. So remember, get_at, you're given an index i, and you want to find the item at position i. And set_at, we're given a position, and we want to change the item stored at that position, at that index, i.

Now, over here, what we're given, we can insert and delete. But the main sort of lookup-- let's think about get_at first. A natural mapping, given this arrow, is find. Find will search for an item by key. So here's-- just staring at that, let's look at all the possible pairings you could do.

We have find by key over here. And we need to implement get_at by index. So let's make the indices keys. OK, so this is idea number one-- index-- assign a key to each item equal to index in sequence.

OK, so then, when I do-- to implement get_at, I can just call find of i if i is also a key. And that should give me the thing that I want. Maybe for this to make sense, let me tell you how I'm building. So if I'm given, say, an array A of items, and they're both-- the only name conflict here is build. So let me call this one sequence build. And I'm going to implement it using set build.

And I'll use some shorthand notation here. Let's say I want to make an object that has a key equal to i and a value equal to A of i-- that's my object notation-- for i equals 0, 1, up to size of A minus 1. That's a little bit code-like, but not quite literal code.

So I'm just going to use this to say, let's make an object that has two parts. One is called the key. So we can talk about the object.key, so we can-- which sets want to do. And we're also going to store a value, which is the actual item that we're given.

So I'm just-- because these are given in the sequence, I'm just representing that sequence order by assigning i to be the key. And so now, if I want to find the item at index i, I can do find of i. And technically, I should probably do .value. That will give me the actual item that's stored at that position. When I do find of i, I'm going to get this whole object with the key of i. And then I want to get the value part of it.

So then set_at, I can just use this find operation to get the object and set its value to x. Boom. We've implemented array-like semantics, get_at i and set_at i, using a set. If you've ever programmed in JavaScript, this should feel very familiar, because JavaScript actually implements arrays, at least at the conceptual level, as just general mapping types, which are-- they call them objects, but they are basically sets.

And it's even grosser. They convert the integers into strings and then index everything by the strings, semantically, anyway. Implementation details can be more efficient. But conceptually, that's what's going on. And so that's the idea we're doing here, which seems great.

Any problems? So let's see. There's insert_at and delete_at. As I mentioned, what I'm going to do for those operations is just rebuild the entire structure. I'll just write that briefly. Basically, let's just iterate all the items. Iterate all items, let's say, into an array. Insert, delete one of them. And then rebuild.

OK, and if I was writing a P set answer, I would say a little bit more detail what I mean in this step. I've done it in the notes. Not that hard. But we can afford linear expected time. I can afford to call build again. I guess, technically, I'm calling this build, sequence build.

So I can afford just to extract things into an array, do the linear time operation on the array with the shifting and everything, and then just call build again. Yeah, question?

AUDIENCE: I had a question about get_at.

ERIK DEMAINE: Yeah.

AUDIENCE: [INAUDIBLE] get_at.

ERIK DEMAINE: Sorry, no. These are separate definitions, yeah? Sorry, they got a little close.

AUDIENCE: Oh.

ERIK DEMAINE: So this is the definition of get_at. This is the definition of sequence build.

AUDIENCE: Oh, I see.

ERIK DEMAINE: Yeah. Thanks for asking. OK, all good, yeah?

AUDIENCE: Can you explain the insert and delete again?

ERIK DEMAINE: Explain insert and delete. OK, so maybe I should actually write one of them down, or I'll just draw a picture, maybe. So we have this data structure, which is now a sequence data structure, represents some sequence of items. And my goal is to say, delete the i-th item.

So there's some items in here, x0 up to xn minus 1. I want to remove xi from the sequence. Or I guess I should draw it this way. It's coming out. So what I'm going to do is first extract all the items from the sequence. And I didn't write it, but there's an interface over here called iter, which just gives me all the items in order.

So I'm going to extract this into an array sequence. Let's say I'll just build a static array of size n. I also have a length operation that tells me how many items are in here. And the iter operation will give me all the items in order.

And so I'll put into my array x0 and then x1, and so on, as they come out. Then I go to position i. And I want to delete that item and shift all the others over. This is the boring-- I think we even said how to do delete_at in dynamic arrays in recitation 2, pretty sure.

So I'm just mimicking that. I'm building this just to get the new order of things. And then I'm applying, via the build operation, I'm building a totally new sequence. And that's how I would implement delete_at, one way. There are other ways. Yeah?

AUDIENCE: Do you have [INAUDIBLE] space [INAUDIBLE], or--

ERIK DEMAINE: How much space is this using? Oh, problem with space if you're inserting-- if you're inserting, you probably want to allocate a static array of size n plus 1. You know exactly what's going to happen. So just allocate a little bit bigger. Then you can do the shift.

You could also use dynamic arrays. But then you would get maybe an-- it's not an amortized bound, because you're only doing one insertion. The point is this is really easy. We can spend linear time. So we can rebuild the-- we can rebuild this array three times if we wanted. Question?

AUDIENCE: What if you weren't allowed external non-constant space?

ERIK DEMAINE: Huh. You're going to throw me and open problem. What if you only have constant extra space? Right. Then I think we need to use insert and delete. So we could-- good question. We could conceptually do this shifting, but do it using insert and delete.

So we can-- so let's do the delete case again. So we want to-- here's xi. We want to replace it with xi plus 1 and so on. And so we can start out by deleting the item with key i. That will get rid of this guy. Then we can delete the item with key i plus 1. And that gives us the item. And then we can reassign its key to i instead of i plus 1 and then reinsert it.

So we can take this item out. It has a key, which is-- I'll draw this properly. So we have key i plus 1 and value xi plus 1 stored in this data structure. Then we update the key to i. And then we reinsert it. And it takes the place of this guy.

So you could do that. You could go down this list and-- or not the list, but you could iterate for i equals-- sorry, for j equals i to n minus 1, and for each of those items, delete it, change its key, reinsert it with the new key. And then you don't have to build this intermediate data structure.

So if you're told to minimize space, great. And maybe you think of that as simpler. I like to think of this as simpler, because I-- point is, I have linear time. I can do crazy, silly, very non-data-structures-y things, where I just start from scratch. OK, great. But there's one more set of operations, insert, delete, first and last. Are these easy? Good?

Shall we try? We can insert last. So this is given an item, x, we want to add it to the end of the structure. So that means its index is going to be equal to-- because we start at 0, it's going to be equal to the length, current length of the structure. So let's just insert a new object, which has key equal to the length. And it has value equal to x. We're done. Delete last, similar. Just delete the item with key length minus 1.

OK, what about first? This is supposed to add x to the beginning of my sequence. Well, now I realize I have a problem, because I want this new item to have key 0, because after I do an insert first, get_at of 0 should return this item.

But I already have an item with key 0, and an item with key 1, and an item with key 2, and so on down the way. And so if I wanted to give x a key of 0, I have to shift the keys of all of those items, just like we were doing here. And that's going to take linear time. But we were supposed to do this in constant expected amortized time. So that's no good.

So this idea is not enough. It's not a bad idea. It's still a good idea. But it's no longer what we actually want to do. It's only morally what we want to do. So do you have any thoughts on how we might get around this problem? Seems like inserting at position 0, I need to shift everything down, linear time. That really sucks. Yeah?

AUDIENCE: You could create some sort of link to something else.

ERIK DEMAINE: Link this data structure with another one. So we could build more than one set. That's certainly allowed. I don't know how to do-- oh, I see. You're saying maybe build a whole other structure for the items that come before 0?

AUDIENCE: Yeah.

ERIK DEMAINE: Yeah, actually. That would work, I think, maybe. It's like in the P set. Then you have to deal with when-- if you delete one of them, it becomes empty. Then things get messy. Delete first is also going to be a problem, because I delete beginning of this data structure, then I lose my 0 item.

And I want the new 0 item to be the 1 item. And again, all the indices shift. So delete and inserting at the first is hard. So we could do that trick like in the P set, but-- or like in the last problem session and so on. But there's a much simpler idea.

AUDIENCE: Can you have an extra variable to keep track of where is the beginning?

ERIK DEMAINE: Nice. I can have an extra variable to keep track of where the beginning is. Call this first. This is going to be the key of the first item, index 0.

Another way to say this is, let's just use negative integers, right? Sets work for any keys, any integer keys. OK, actually, we technically said, make sure you use keys 0 to u minus 1. But then, if you have negative numbers, you can easily fold--

AUDIENCE: Wait, doesn't it [INAUDIBLE] to like [INAUDIBLE]?

ERIK DEMAINE: Ah. Python negative numbers mean something else. But we're not using a Python interface. We're using our custom magical set interface, which we show how to implement in recitation notes, which can take an arbitrary key. It hashes that key and finds a place to put that item.

So we're not actually storing things in order here. We're storing things in a hash table. But we're not supposed to get into the implementation details. I think the way we presented hashing with our universal hash functions, we only allowed positive numbers.

So maybe, technically, I should point out, if you have positive and negative numbers, you can fold this in half by mapping 0 to 0, 1 to 2, 2 to 4, spreading it out. And then you can take minus 1 and map it to plus 1, and minus 2 and map it to plus 3. So this is like multiplying each of these guys by 2, and multiplying each of these guys by minus 2 and adding 1. And then you get non-negative integers out of all integers.

This is a typical math trick for showing that the number of integers is equal to the number of non-negative integers, which may seem weird to you. But they're both countably infinite. So you could-- if your structure only supports non-negative keys, you could map negative keys in this way and throw them into the hash table, OK?

So now, I allow negative things for-- like that. And so, great. If I want to insert at the beginning, what I can do is just decrement my first variable, which is keeping track of the index. So initially, first is going to be 0. So I'm going to add into my build. First, I'm going to say first equals 0, because I start with key 0 when I initially build a structure.

And if I want to-- if I need more room before 0, I just set first to minus 1. And if I already have a minus 1 element, I'll decrement it to minus 2. Decrement means decrease by 1-- shows my assembly language programming. This is usually a built-in operation on most computers.

And then I can insert an item with key first and value x. Great. And if I want to delete the first item, I would delete the item with key first and then increment first. And now all of my operations have to change a little bit-- let me use another color-- because I was implicitly assuming here that all my indices started at i. But now they start at first.

The index 0 maps to key first. And so the right thing to do here is plus first and plus first. Basically, add a whole bunch of plus firsts throughout. This one is probably fine. If I'm globally rebuilding, I can reassign all my labels. But this one should be first plus length.

OK, so just by keeping track of where my keys are starting, I can do this shifting and not have to worry about stuff. And this is a lot easier than having to worry about maintaining two structures, and keeping them both non-empty, and stuff like that, because of-- if I assume my mindset has this power of dealing with negative integers, and strings, and whatever else. Cool? Yeah?

AUDIENCE: Is there a reason why you didn't do like the sorting-- like, have [INAUDIBLE]?

ERIK DEMAINE: Oh, why didn't I use a linked list? Because this. Linked lists are very bad at get and set at a given index.

AUDIENCE: Is that the-- the bottom idea, is that a linked list?

ERIK DEMAINE: This is not a linked list. This is just storing a single number as integer in your data structure that says, what is the smallest key in my data structure? That's all it this. It's a counter.

AUDIENCE: Ah.

ERIK DEMAINE: OK, so data structure keeps track of its length. And it keeps track of the minimum key. And so it will always consist-- the invariant is, you will always have keys from first up to first plus length minus 1. And that's what we're exploiting here. We have no idea where first will be. It depends how many operations you've done, how many inserts at the beginning, and so on. But the keys-- keys will always be first to first plus length minus 1.

This is what we call an invariant. Useful to write these things down so you can understand what the heck-- why is your data structure correct? Because of invariants like this, which you can prove by induction, by showing, each time you do an operation, this is maintained, even when I'm changing first in order to maintain this invariant.

Cool. Sometimes you come up with the invariant first. In this case, I came up with it post facto, after the fact. Cool. Let's move on to problem 3, which is called critter sort. And the other key thing I want you to learn about-- question? Sorry.

AUDIENCE: Yeah. So when you do first, first plus 1, is that a rebuilding of the [INAUDIBLE]?

ERIK DEMAINE: This is just a sentence. It is not an algorithm or data structure. This is a mathematical property.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: This is not an assignment. This is a mathematically is equal to.

AUDIENCE: But you are re-indexing it though, because you're doing first plus 1.

ERIK DEMAINE: So are you asking about one of these operations, like this one?

AUDIENCE: Oh, OK. Never mind. I get it. OK.

ERIK DEMAINE: Yeah. OK. So the other important takeaway I want you to get about reading our problem sets is that they have hidden humor inside. I don't know if you've noticed. But here's an example of a problem called critter sort. Ashley Gettem collects and trains pocket critters to fight other pocket critters in battle. What is this a reference to?

AUDIENCE: Digimon.

ERIK DEMAINE: Digimon. Wow, you guys are so young. Pokemon, the ancient form. Pokemon is short for pocket monsters. And in fact, in the original--

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: --anime--

AUDIENCE: Actually, there's [INAUDIBLE].

ERIK DEMAINE: I don't know. This is all after my time. We can debate after. So pocket critters is a reference to pocket monsters, which is Pokemon. Who's Ashley Gettem?

AUDIENCE: Ash.

ERIK DEMAINE: Ash Ketchum is his full name in the English version. Totally different name in the Japanese version. But they're both puns on collect them all, right? All right, so that's the important stuff. We'll see more jokes later.

So there's this setup. But basically, we have n critters. And we want to sort them by four different things. And so I'm just going to abstract this problem to sort n objects by the following types of keys. And for each one, we want to know what the best sorting algorithm is.

And there's this footnote that's very important that says, faster correct algorithms will receive more points than slower correct algorithms. Also, correct algorithms will receive more points than incorrect algorithms. But that's implicit. Incorrect generally gets zero.

OK, so part a, it says, species ID. But basically, we have integers and the range minus n to n. So if I want to sort n integers in the range minus n to n, what should I do? This is a reference to yesterday's lecture.

Yeah? Radix sort, yeah. Always a good answer. Or almost always a good answer when you have integers. It's a good answer whenever you have small integers. Now, radix sort, the way we phrased it-- let me maybe put it down here. Radix sort sorts n integers in the range 0 to u minus 1 in n plus n log base n of u time.

And in particular, this is linear time if u is n to some constant power. OK, so can I just apply this as is to these integers? No, because they're negative. So what should I do? Maybe I should do my folding trick. We just saw how to take negative numbers and fold them in, interspersed with positive numbers.

If I sort that, will that work? No, because that does not preserve order. It would intersperse. We want all the negative numbers to come before all the positive numbers. Yeah?

AUDIENCE: Can you just add n to all the integers?

ERIK DEMAINE: Just add n, yep. Boom. Plus n. Now we have integers in the range-- let's be careful-- 0 to 2n. Cool. Now we can apply this. Now u equals, technically, 2n plus 1, because we're only supposed to go to u minus 1. But that's fine. That's linear. And so we can sort in linear time, easy. This is a super easy problem.

But in each one, we might need to do some transformation. Part b is a little more interesting. So we have strings over 26 letters of length at most 10 ceiling log n. OK, this is a little trickier.

What could I do? Again, I'd like to see whether radix sort applies. I should say radix sort sorts. I'd like to see if radix sort applies. To do that, I have to map these strings into integers somehow. Any way to do that? This is easy if you understand radix sort. Yeah?

AUDIENCE: Can you just index the letters?

ERIK DEMAINE: Index, the letters. Yeah. Yeah, we can map-- right. So we can map A to 0, B to 1. Then what?

AUDIENCE: Oh, wait. Length--

ERIK DEMAINE: So that's for each letter. But we have a lot of letters. There are only 26 letters. But then we have 10 log n letters in a string. That is, together, a single key that we need to sort. Yeah?

AUDIENCE: Can't we just sort by the first letter first, then--

ERIK DEMAINE: Sort by the first letter, then the second letter. That is exactly the opposite of radix sort. Remember, radix sort, we want to start by the last letter, and then the next to last letter, and finally, the first letter.

AUDIENCE: But you want to sort by the first one in order to alphabetize things.

ERIK DEMAINE: No, to alphabetize-- we do want to, in the end, sort by the first letter. But that's at the end. So at the end-- remember, radix sort always goes backwards from the least significant to the most significant. And so indeed, that is what we want to do. You're just saying, use radix sort. But what am I radix sort on? What am I radix sorting on?

AUDIENCE: Yeah, on the last letters, not the first letters.

ERIK DEMAINE: So technically, that would be using counting sort on the last letter, counting sort of the next to last letter, dot, dot, dot, counting sort on the first letter. But that is, together, radix sort on something, or Jason likes to call this tuple sorting.

Tuple sort is the thing-- is the algorithm that says, sort by the last thing, then sort by the previous thing, and so on. You can also think of this as radix sorting on a number written in base 26. They're the same thing. But in the end, we can sort in linear time.

AUDIENCE: How do you ensure that the letters are sorted in order, though? Like, how do you ensure that-- how do you tell the algorithm that you want A to come-- just like not-- 0 is less than 1, A is less than B.

ERIK DEMAINE: Right. So I mean, technically, when you call something like tuple sort-- or maybe it's even clearer when you call radix sort. Radix sort, you're giving it a bunch of numbers. So you're taking these strings and mapping them to numbers. And when you do that, you get to decide which letter is the most significant, which is the least significant, right?

So you will choose to always map the first letter in your string to position-- to value, or the-- what do you call it? Position in positional notation. Position 26 to the power 10 log n as the most significant. So it's always the most significant. Even if your string is of length 1, you want to put that in the most significant digit. And you'll pad with zeros at the end if you run out of letters in your string.

AUDIENCE: How many times are you running counting sort here?

ERIK DEMAINE: How many times am I running counting sort? Oh, 10 log n times. Whoops. Yeah, good question. Good point. Yeah, I computed this wrong. So right. There are log n digits in the string. So that is bad. I mean, it's OK. We'll end up with n log n running time by the tuple sort.

However-- so that's the tuple sort. So I should really make this not equivalent. If I run tuple short letter by letter, I'm going to do-- I'm running counting sort log n times. And so I get n log n, because each one takes linear time. If I map my strings into numbers first, radix sort doesn't use base 26. It uses base n. And then it will only run 10 times, because 2 to the 10 log n is n to the 10.

And so the numbers that we're sorting are between 0 and n to the 10. And so u is n to the 10. And so that's the case when radix sort runs in linear time. So if you run tuple short letter by letter, it's slow. If you run radix sort, it's doing a whole bunch of letters at once.

Effectively, it's doing log n letters at a time in a single call to counting sort. And so the radix sort will actually win and get linear. There's a subtlety here, which is, I'm assuming that we can actually take these strings and convert them into integers in constant time each. And this problem set was ambiguous. And both answers were accepted.

If you assume these letters are nice and compactly stored, and they fit in 10 words, because a word is at least log n bits long, then you can actually do this. If you store each letter in a separate word, then just reading the entire input will take n log n time. So that's a subtlety which we don't need to worry too much about in this class. Yeah?

AUDIENCE: So [INAUDIBLE] bounding the letters to numbers. And like, how would that help? Because we still have to do 26--

ERIK DEMAINE: Yeah, there are 26 possible letters, numbering them 0 to 25. And then when we take a string, like AA, we map this into 00 in base 26. That's a number. If we do BB, for example, this maps to 11 in base 26, which means 1 times 26 plus 1, which is 27. OK, so that's the mapping that I mean.

AUDIENCE: You're mapping the whole string [INAUDIBLE]?

ERIK DEMAINE: The whole string to a single number, yeah. And there's a subtlety, because I want lexicographic. I need to pad things with spaces at the end or pad them with As at the end in case they're shorter than 10 log n. OK, cool. That was b.

c is not very interesting. It's integers in the range 0 to n squared. This, I can just solve with radix sort, because my radix sort, at this point, we've done it a third time. Radix sort, we can sort as long as the integers are bounded by a polynomial. Here, it's a fixed polynomial with a constant exponent. So this will-- and this is radix sort, like we saw, that just calls counting sort twice, linear time.

d is where things get more interesting. Let me get this phrasing the same. So in d, we have rational numbers of the form w over f. This is some win ratio. Always in the range 0 to 1. So we saw w is at most f.

And 0 is less than w, is less than f, is less than n squared, because the-- that is really confusing-- is less than n squared-- those are separate statements-- because the f actually comes from part c. And c is really a setup for this one.

Doesn't really matter what this means. It's just that we have numbers w and f, where w is always less than f. And they're between 0 and n squared. So you should think, this is a good range for me, right? That I'm representing this rational in terms of two numbers between 0 and n squared. So there's like n to the 4th possible choices for what w and f are.

So the range of my values is n to the 4th. That's the setting where radix sort should run fast. Unfortunately, these numbers-- what I want to sort by is not an integer. It's a rational. And that's annoying. So there are a couple of ways to solve this problem.

In general, a good way to solve sorting is to use merge sort. Merge sort is always a good answer. It's not the best answer. In these cases, we shaved off a log. We got to linear time. But n log n is pretty good. It's pretty close to n.

So first goal might be, can we even achieve n log n via merge sort? What would I need to do in order to actually apply merge sort to this instance? What does merge sort do to its keys? Sorry?

AUDIENCE: Isolate and compare them.

ERIK DEMAINE: It isolates and compares them, yeah, right. So there's an array data structure. And it indexes into the array. That's the isolation. But then the thing it actually does with the items themselves is always a comparison. And this is why we introduced the comparison model and proved an n log n lower bound in the comparison model, because merge sort, and insertion sort, and selection sort are all comparison algorithms.

Radix sort is not. But this one is. But to apply merge sort, I need to say, how do I compare wi over fi versus wj over fj? My computer only deals with integers. We can't actually represent wi over fi explicitly in binary, because it has infinitely many bits. But I can represent it implicitly by storing wi and fi. Yeah?

AUDIENCE: Multiply by fi and fj.

ERIK DEMAINE: Multiply by fi and fj, yeah. When I went-- I didn't go to school, but then we learned cross multiplication, which is the same as multiplying both sides by fi and multiplying both sides by fj, as you said. So then we get fi fj less than question mark f-- whatever-- fi wj.

When we do that, we better make sure that the things we're multiplying by are non-negative. Otherwise, the sign flips. But here, we assume they're all non-negative. So this is good. And now we're just multiplying two integers here, multiplying two integers here, and comparing. Those are all things I can do in a word RAM.

So this was actually the intended solution when this problem was posed. Here's a way to do comparison sort. We get n log n. But in fact, you can achieve linear time. Yeah?

AUDIENCE: [INAUDIBLE] that solution, how would you quickly say which one's bigger? Because wi times f of j, one of them belongs to one of the Pokemons, and the other one is [INAUDIBLE].

ERIK DEMAINE: I feel like there's a joke here, like--

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Pikachu is superior. That's always the answer. So how do I tell whether one Pokemon is superior to the other? If I multiply my-- I multiply i's f value with j's w value. And I see whether that's greater than i's w value times j's f value.

And if it is-- so these are equivalent. If this one is greater than this one, I know that this is greater than this. These are equivalent sentences by mathematics, by algebra. And so this is what I want to know. This would say j is superior to i. And so I determine that by actually doing this. So then I don't have to divide and deal with real numbers, because I don't know how, because I'm a computer. We're all computers in the end.

OK. So it would be great if my numbers all had the same denominator. If they all had the same f, then I could just compare the w's. So that's one intuition for why we can actually do this in linear time.

But the way I like to think about it-- so let's just draw the real interval from 0 to 1. And there are various spots all over here that represent-- I can't actually compute this. But conceptually, each of these wi fi's falls somewhere in that interval from 0 to 1. And I want to sort them somehow.

So one thing that would be great is if I could take these real numbers and somehow map them to integers, which are uniformly spaced, maybe a few more of them. But these go from 0 to u minus 1. And if I could get u relatively small, and I could map each of these-- so I want that mapping to be order preserving. And I want two very close, but distinct items to map to-- distinct keys here. I want them to map to distinct integers down here.

If I could do that, then I just sort by the integers. And that's the same as sorting by the real numbers. And so at this point, I wonder, how close can two of these numbers be? So how close can two keys be? So I want to consider wi over fi minus wj over fj in absolute value.

Now I do algebra. So this is-- I'd like to bring this into one ratio. So this is-- I can do that by multiplying 1 by fi, 1 by fj. Now that's wi fj minus wj fi, which should look a lot like something here. But never mind. I'm sure there's a deep connection here. I can probably use this to prove that and vice versa.

Cool. So with some absolute values, same thing. Maybe these are non-negative, so I can actually just put absolute values on the top part. And OK, wi is an integer, fj is an integer, wj is an integer, fi is an integer, all greater than or equal to 0. So this thing is an integer.

So it could be equal to 0. It's a non-negative integer, because all the things are non-negative. It could be equal to 0. But if they're equal to 0, that's actually identical ratios, right? If this is 0, the whole thing is 0. And so these two values were the same.

OK, but let's suppose it's not 0. If it's not 0, it's actually at least 1, the absolute value, because it's an integer. What about the bottom? fi-- so now we want this-- I want to know how small this ratio can be. It's going to be small when this is small and this is big. How big could fi fj be? Well, we're told that all the f's are less than n squared.

So this thing is at most n squared, n to the 4th, less than n the 4th-- n squared minus 1 squared, less than n to the 4th.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: fi is at most n squared. fj is at most n squared. So it's n squared squared. So this is at least 1 over n to the 4th. So the closest the two points can get here is 1 over n to the 4th. So what can I do to scale that up to make them kind of like integers? Multiply by n to the 4th.

So just multiply by n to the 4th and then floor. So we're going to take each fi over-- I'd like to compute this ratio. But I don't know how. So instead, I'm going to take fi, multiply-- OK. Conceptually, what I want to do is multiply by n to the 4th and take the floor.

How do I actually do this in a machine that doesn't have real numbers like this? So I don't have a floor operation. I just have integer operations. Then I can take fi, multiply it by n to the 4th, and integer divide by wj.

That is the same-- that computes exactly this, because I can do the multiplication and the division in either order in real space. And then this does the floor at the appropriate time. But this is just operations on integers. And now these are integers representing how good my Pokemon are.

They have the property that any two distinct ones-- before I take the floor, any two distinct ones are at least 1 apart. So after I take the floor, they will remain 1 apart. They will remain distinct integers. And so I have successfully mapped my real numbers to integers where distinct real numbers match distinct integers. Yeah?

AUDIENCE: Wait. So why is fi now in the numerator, and wi in the denominator?

ERIK DEMAINE: Did I flip them? Yeah, sorry. Please invert everything-- just here. This is w and fi. That was just a typo. That's all of them. OK.

AUDIENCE: Are they both i's or j's?

ERIK DEMAINE: These are supposed to both be i's, yeah. Thank you. This was for each Pokemon, i, we're going to compute this as our key. And then we're going to sort by those integer keys. And that will sort the Pokemon by their ratios. Let's write mon for monster. Yeah?

AUDIENCE: [INAUDIBLE] u minus 1 [INAUDIBLE]?

ERIK DEMAINE: So u was just a-- sorry, this is-- a label on this thing might help you.

AUDIENCE: Oh.

ERIK DEMAINE: Yeah. So now my u-- oh, right. What is my u? What is my largest key? It occurs to me, I really would like fi to be bigger than 0. But let's not worry about it. How big can u be?

Well, the biggest this can be is if fi is small, and this is big. Let's say fi can only go down to 1. Otherwise, we'll get a division by 0. We have to deal with infinity especially. Probably, the problem isn't even well defined then. How big could this be? Well, I know the wi's--

AUDIENCE: f's are defined as positive.

ERIK DEMAINE: Oh, good. Thank you. So there's also a positive constraint here. Just I failed to preserve that constraint in my mapping from the word problem into the formal problem. So f is the least 1. Good. But worst case is when it's 1.

And when wi-- how big could it be? Well, n squared minus 1. So this could be, basically, n squared times n to the 4th divided by 1, which is n to the 6th. So w-- or sorry, u, the largest key I can have plus 1, is n to the 6th. But that's OK, because radix sort can handle any fixed polynomial in n. So it's going to end up doing six counting sort passes.

OK, that's problem 3. Let's move on to problem 4.

So problem 4, MIT has employed Gank Frehry. Who's that? Frank Gehry, yeah. This is a common encoding that Jason really likes. I've grown to like it. This is called spoonerism, where you replace some part of the beginning of your thing. OK, that's one joke.

There's another joke in this problem. Anyway, they're building a new wing of the Stata Center, as one does. We have a bunch of cubes. If you read long enough, you realize that's a red herring. Cubes do not play a role in this problem. In the end, what we have is a bunch of integers, which happen to be the side length of the cubes.

But we just care about the side lengths, not their volume or anything-- s n minus 1. And we want two numbers in s summing to h.

AUDIENCE: This is dumb, but how can cubes have more than six sides?

ERIK DEMAINE: This is a side length, not the number of sides. So a cube--

AUDIENCE: Oh, OK.

ERIK DEMAINE: Cool. I didn't know we'd be doing 3D geometry today. That's si. OK, so you got little cubes. You've got big cubes. This is a small si. This is a big si. Doesn't matter, though. They're just numbers. We're not using them at all.

In the problem, you're trying to like stack one cube on the other. But all we really care about is two numbers whose sum, regular old sum, is exactly h, ideally. There's going to be two versions of this problem. And so the first goal is to solve this exactly in linear expected time. That's what the problem says.

So what do we know? Well, linear time, that's-- can't get much faster than that, because we need that just to read the input. Expected time-- hashing, right? We're told, basically, we should use hashing. Now, if we're really annoying, maybe we throw that in even when you don't need it. But that's pretty rare.

So when we see expected, we should, in a problem set setting like this-- in real life, you never know what you should use. But in our-- with your learning in this class, we're going to tell you basically what tricks you're allowed to use. Here, you're allowed to use randomization. So probably, we need it. Indeed, you need it to achieve this bound.

Cool. Hashing. Not obvious how to approach this problem with hashing. So I'm going to give you the way I-- it's hard for me to not know this algorithm. But to me, the first thing you should think about is if I have linear time and n things, and I'm going to use hashing, the obvious thing to do is to take those n things and put them in a hash table. Build. Why not?

So let's just build a hash table on all the keys in s. That's idea one. Seems like the first thing to try. So what does that let me do? It lets me-- I just erased the interface for hash tables. But I can build a sequence out of it.

But normally, it gives me a set interface. So I can call find now in constant time. It lets me, given the number, determine immediately whether that number is in s. Well, that sounds interesting, because I'm looking for two numbers in s. So it lets me find one of them. So I call it twice? No. Calling it twice and only spending constant time on this beautiful data structure will not give you anything useful.

But we have linear time, right? So in addition to building a table, we could call find on that table a linear number of times, because each find only takes constant expected amortized time. So if I do n of them, that will take linear expected time. The amortization disappears, because I'm using it n times.

AUDIENCE: [INAUDIBLE]

ERIK DEMAINE: Oh, right. Find never has amortization. So it doesn't disappear, because it was never there. Never mind. I can afford n calls, or 5n calls, to find, because each one costs constant expected. And the total for that will be linear time.

So the next idea is let's just somehow call find a linear number of times, OK? So I want to find two numbers summing to a given value, h. That wasn't maybe clear, but h is given.

AUDIENCE: Sorry. How long does it take to build the hash table?

ERIK DEMAINE: How long does it take to build a hash table? It was previously on this board-- linear expected time. See previous lecture. No, two years ago. OK. Well, if we're going to do this a linear number of times, I guess we should have a for loop. Let's do a for loop over the numbers. That's the next idea. Loop over s. And at this point, we're done, almost. I want space.

So I want to loop over the numbers. And each one, I want to do a find. That's kind of all I have time to do. So seems like a natural thing to try. This is by no means easy. Don't get me wrong. Having these ideas is-- while I'm explaining them as the obvious ideas, they're not obvious. But they are easy, at least, just not obvious to come up with the easy ideas.

So let's loop over s, somehow call find, using our hash table. So the order is actually, we're going to build the hash table, then loop. And inside the loop, we're going to call find once per loop iteration. So let's do it. Let's say, for si in S-- so I want to find two numbers.

Here, I have exhaustively looped over one number. I just need to find the second number that can possibly add up, right? I want to find whether there's an sj in S such that si plus sj equals h. Can I do that query with find? How?

So what does find do? Find says, if I give you a key, it will tell me whether-- like, if I knew what sj was, it would tell me whether it's in S. Yeah?

AUDIENCE: Can't you just subtract h from si and then see if [INAUDIBLE]?

ERIK DEMAINE: Subtract h from si and see whether that exists. Did get it right?

AUDIENCE: h minus si.

ERIK DEMAINE: h minus si. I always get it wrong. Don't feel bad that you also got it wrong. It makes me feel better, because I always get it wrong. So the claim is this. Why? Because what we want to do is find-- well, OK. Let's see what it says over here.

So if we do h minus si equals sj-- so these are equivalent statements, just by moving the si over. And this is a query we can do. So let's remember, these are things we know. And s.j is something we don't know. All that we know is that it's an s.

OK, so we know these two things. So if we bring them over to the same side, we're searching for an unknown thing, which is equal to exactly this thing that we can compute. So we just compute h minus si. We call find. That will tell us whether there is an sj equal to this.

OK, so this is like a comment. And this is what we actually do. And if there is a pair of numbers summing to h, this will find it. How much time did it take? Well, we're doing n iterations of this loop. Each one, we're calling a single find operation. And find costs constant expected time. And so the total is linear expected time. Great. Part A done.

Then they throw part b at us, make it harder. Those pesky instructors. So we read part b. Part b says two things to make it harder. So first of all, we want linear worst-case time. And furthermore-- so we can't use hashing anymore.

Furthermore-- so here, we just needed to solve the exact problem to find whether the two numbers sum exactly to h. Now we would like to find the best solution smaller than or equal to h. So find biggest pairwise sum that's less than or equal to h if there's no perfect pair.

But we're given a little bit of extra information, which is, we can assume h equals 600 n to the 6th. That's a weird polynomial. Took me a while to even notice that that was a joke in here-- 6006, hiding in a polynomial.

All right, so polynomial. Hm. That should make you think radix sort. It is radix sort week. So that is a natural thing to try. But in general, even later in the semester, when you see a nice polynomial with a fixed constant like this, and it's somehow related to the integers we're dealing with, you should think radix sort. Especially because now, we want constant worst-case time, radix sort seems like a good thing to do.

Don't know what to do with it yet. In fact, I can't even apply radix sort. But idea one is radix sort. Just because I see that polynomial, I think maybe I should try it. Now, there's a problem here, because we're given some numbers, some integers, si's. We're also given h.

We're told now that is a nice, small polynomial. But we have no idea how big these numbers are. So the problem with this idea is that-- but si could be bigger than h. We have no idea how big the si's are. What can I say about si's that are bigger than h for this problem? Summing to h.

Oh. I didn't say it, but all these numbers are non-negative. That's important. That looks like [INAUDIBLE]. Greater than or equal to 0. Yeah?

AUDIENCE: [INAUDIBLE] solution [INAUDIBLE].

ERIK DEMAINE: Right. If I'm finding a sum that's less than or equal to h, they're non-negative. And any number that's greater than h, I can just throw away. They'll never be in a solution. So they already-- a sum of one number is bigger than h. So two is only going to get bigger if they're non-negative.

So idea number two is let's just throw out all the big si's, anything bigger than h. Now, that won't change the answer, because those can never be in a solution. And now I have all the si's having the property that they're less than or equal to h. And so they are small, bounded by a fixed polynomial. And now I can apply radix sort. So after this idea, I can apply this idea.

OK, this gives you a flavor of how I like to think about problems. I see clues, like a polynomial. I think radix sort doesn't work. But with some more ideas, I can get it to work. OK. What goes with the-- so now I've sorted si. OK, great. S is sorted.

I guess we can try to do the same algorithm, except I don't have a hash table anymore. So let's just try doing a for loop over the S. Why not? So let's do for si in S. But now it's sorted. So presumably, I should exploit the sorted order. So let's do them in order. So i equals 0, 1, up to n minus 1.

Let's say that s0 is the smallest. s1 is the next smallest. sn minus 1 is the biggest. So I want to do something with-- so I have si. And I want to figure out whether h minus si is in there. Hard to do that better than-- actually, I could do this with binary search. I'm looking for this value. And I have a sorted array now.

So I could binary search for h minus si. And in log n time, I will find whether that guy is in there. And if not, keep looping. I can keep track of the best thing that I found. And so in n log n time, I can definitely solve this. But I'd like to get linear time. Do you have a question?

AUDIENCE: Well, I'm just wondering, how would you [INAUDIBLE]? Like, why would you [INAUDIBLE] whether that is in there [INAUDIBLE]?

ERIK DEMAINE: I'm not looking for si. I'm going to compute h minus si. So this is-- maybe I shouldn't even write this down, but--

AUDIENCE: She's asking about the [INAUDIBLE] constraint of, we're not looking for h. We're looking for something smaller than h.

ERIK DEMAINE: This one? Or--

AUDIENCE: Something larger.

ERIK DEMAINE: Oh, this thing.

AUDIENCE: A large thing less than h.

ERIK DEMAINE: Right. So in particular, if there are two items that sum to h, I want to find it. So let's start with that. So I'm binary searching for h minus si in S. So I could certainly do that. And if I find it, great. I found a pair that sum to exactly h.

If I don't find it, binary search tells me not only that it's not there, but it tells me what the previous and next value are. So even though h minus si isn't there, I can get the next largest thing and the next smallest thing. What I want is the next smallest thing.

And that will be the largest sum I can get using si. And so then that's one candidate for a sum less than or equal to h. I want to find the largest one. So I do a for loop. I always keep track of-- I take a list of all the candidates I got. Each time I do an iteration of this loop, I get one candidate. Then I take the largest one. OK, so return largest candidate.

So this gives me a candidate, just the previous item. This is what we called find previous, or find prev, probably, in our set interface. And if you have a sorted set, you can do that in log n time. So this is an n log n solution, because we do n iterations through the loop. Each binary search takes log n. I want to get linear.

This is not obvious. The best intuition I can think of for this next idea is, well, I start with the very smallest item in S. And I want to sum up to something that's kind of big, right? I threw away all the items bigger than h. If s0 is like tiny, like close to 0, because it's the smallest one, then maybe I should look at the end of the array, because I want to compare, or I want to add the smallest thing probably with the biggest thing. That's as close as I can imagine.

So then-- so here's my sorted S. It's the smallest item, biggest item. So I'm going to loop over these items one by one. So let's start by comparing the first one with the last one. The two-finger algorithm, OK?

This is the big idea. You're doing it all the time in this class. It's super useful. We saw it in merge sorts, for example, and merging two lists. We have fingers in two lists that advance. And because they only advance, it takes linear total time.

So we're going to do this kind of folded in backwards here. We're going to start here. This seems like a good candidate to start with. Now, what else could this add with? Well, maybe smaller items. And maybe I have to go all the way through here. And then I've got to advance my left finger. Yeah, OK. So here's the idea.

So let's look at-- so I'm going to call this finger i and this finger j. So we want to sum two things. So I guess one other inspiration here is, we want to add two things up. And we have one algorithm that has the word "two" in it. And it's the two-finger algorithm. So let's try that.

So we're going to start at i equals 0 and j equals n minus 1. We're going to look at si plus sj and see, how good is it? How close to summing to h is it? Well, in particular, it's either less than or equal to h or bigger than h.

If it's bigger than h-- so this sum is too big. I can't even use it as a candidate. Well, that means I really don't need this guy, right? It's too big overall. I'm adding the smallest item to this item. And it's too big. Well, then I should go to the left. I should move my right finger to the left.

So in this case, we decrement j. Move the right finger to the left. So I'm guessing, in this case, I increment i. Why? If I add these two items up, and this is too small, it's smaller than h, then this item was probably too small. It might actually-- it's an OK solution. It's less than or equal to h. So I should keep it as a candidate. Let's say add candidate.

So I'm just going to keep a list of candidates that I see. So this is a possible solution. It might not be the best one. But it's one to add to my list. And then I'm going to increase i and now work on this sub-array, because that will be a little bit bigger.

I can't go this way to make it bigger, because I'm at the last item. And it's not obvious that this works. I think there's a nice invariant that will help somewhere. Where'd I put my piece of paper? Yeah. So here's an invariant.

Oh, yes.

It's really clear this is the right thing to do in the first step. And the tricky part is to argue that it works in all steps, because when I really have the smallest item and the largest item, it's clear that I should advance one or the other if I'm too small or too big. But the way to prove it in general by induction is to show this invariant that-- so at some point through this execution, i and j are somewhere.

And I want to say that if I take any j from the right-- any j prime to the right of j and any i prime to the left of i, unstrictly, then all of those pairs, all those pairwise sums, are either too big-- and that's when we decrease j-- or they're less than or equal to the largest candidate that we've seen so far. That's because we added these candidates in there. So that invariant will hold by induction, because whenever there's a possible thing that's good, I add it to my candidate list.

And then, at the end of the algorithm, I just loop through my candidate list, compute the max, return that pair. OK, so that is two-finger algorithm, which solves the non-exact problem in linear worst-case time. Yeah?

AUDIENCE: i cannot equal j, right?

ERIK DEMAINE: Oh, i cannot-- right. So what are the termination conditions? When i equals j, that's probably when you want to stop. It depends. You could say, if i is greater than j, stop. Return max candidate. There are two ways to interpret this problem. One is that the two values you choose in S need to be different values, or you allow them to be the same value, like they can both be h over 2.

And either way is easier to solve. If you want to allow s over 2, then I would put greater than here. If you don't want to allow h over 2, then I would put greater than or equal to-- either way. Both of these problems, you can solve both ways. Or both algorithms can handle both situations.

OK, one more problem. All right. Yeah, I'm all out of time. But I'm getting faster and faster. Of course, on the hardest problem, I can do it the fastest. All right, so Meff Ja-- this is a reference to Jeff Ma of the MIT Blackjack Team, who I got to speak here at LSC a bunch of years ago. But he's featured in the movie *21* and so on-- fictionalized.

So I was playing this game. It's a great setup. You should definitely read this problem-- Po- k -er. And he has a deck of cards, where each card has a letter of the alphabet on it. I guess this is the right way up. So I, of course, have such a deck. Doesn't everyone?

You can buy these. I have several, actually. And so we can do a quick magic trick, like pick a card, any card-- here, pick a card. [INAUDIBLE] OK, good choice. I can't force, so it doesn't really matter. OK, and so this is your card, right? And your card is an s, right? OK, good. No, not all the cards are s's.

[LAUGHTER]

But he has mirrors in his glasses. No. I can reveal later how that's done. OK, so a deck of cards. Each card has 26 possible letters on it. And there's this weird dealing process. Even just defining this problem is going to take a little while. Oh, here's my piece of paper.

So we have this dealing process. Here's an example that's in the program, abcdbc. So you know the order of the cards. This is the top card. This is the bottom card. And now, randomly, you do a cut. Cut is this. So I take some chunk off the top, move it to the bottom, once, randomly.

So for example, I could take this cut. And then what I would get is cdbc for this part that's copied here, and ab as the-- so this is-- so the first thing we do is cut at i. This is position i. In this example, i equals 2.

OK, then we deal the top k cards. So let's say we deal the top four cards, k equals 4. So this is deal k. So we get cdbc, in that order. But the order doesn't matter, because the last operation we do in the problem is sort them, which is bccd, like you do when you get a hand of cards. You tend to sort them.

OK, so this is a process. Given a deck-- so the deck here is fixed. We call this process, I think, P of D comma i comma k. We're told what D is. We're told what k is. i is chosen randomly. And we'd like to know what happens with different i's.

So if you stare at this problem enough, it begins to simplify. So this is a complicated setup. But what's really going on is we're starting at position i. And we're taking the next k cards from there cyclically. So here, we just took those four. If i equaled 3, we would deal d, then b, then c, then a. But then we sort them.

OK, so we're getting different substrings of length k, cyclically. But then we're sorting those letters. Sorting is really crucial for this problem to at all be feasible. It took me a while even to see how to solve this problem. But the key is sorting, that they get sorted, because that means-- because we sort, it doesn't matter whether you have aaba, or baaa, or abaa. These are all the same.

If you take these cards dealt, you sort them to the same thing, which is the one I didn't write, aaab. All of these get sorted to the same thing. So we lost some information when we sort, lost the order. The first question to get you thinking in this direction, part a, says, build a data structure given D and k that lets me know, given two indices, i and j, do I end up with the exact same hand? This thing is called a hand. And it's exactly this P D, i, k.

So I want to do P D, i, k and P, d, j, k. And I want to know-- JK. And I want to know whether those two things are equal in constant time. That's what this says-- constant time. Doesn't say worst case, but worst case is possible.

And that sounds hard, because, I mean, there's k symbols for one of them, another k symbols for the other guy. But we don't have to compare the symbols. We just need to compare the sorting of those strings. And this, we can compress. So this is a subtlety.

But all I really need to know is that there are three a's here, and one b, and zero c's, and zero d's, and zero e's, and so on. But because there's only 26 letters in this deck-- and indeed, in this deck, it happens upper and lowercase a through z.

But we might have n cards. But there are only 26 possible labels. So in fact, a lot of them are going to be equal if n is large. So this is a good compression scheme, because to represent the things I get after sorting, I just need to give you 26 numbers. And for us, 26 is small, because 26 is a constant.

Independent of the number of cards, I just need to say, how many a's are there? It could be anywhere between 0 and n. How many b's are there? Between 0 and n. How many c's are there? Between 0 and n. So 26 numbers in the range 0 to n-- I like to think of this as a 26-digit number base n plus 1.

We can map this into base n plus 1. And we get 26 digits In that base. Another way to say it is that the number of possible combinations here-- how many a's, how many b's, how many c's-- is not even theta. It is n plus 1, anything between 0 and n, to the power of 26. This is a good polynomial. So I can do stuff like radix sort.

Cool. So let me summarize a little bit how we solve part a. So I want to build a data structure, which is, for each value i, I know I'm going to end up serving these four cards, or in general, k cards. So for those cards, I would like to compute how many a's, how many b's, how many c's are there? And then just write down this number.

This is a number which I can write down in at most 26 words, because we can represent numbers between 0 and n in a single word. That's the w is at least log n assumption. So it's constant size. In a constant number of numbers, I can represent all I need to know about a thing of size-- of length k here.

So I don't need to know which letters is where. I just need to know the sorted order. So I just need to know-- this is called a frequency table-- how many a's, how many b's? And so if I can compute those, then given that representation for starting at i, and given that representation for starting at j, say, which would be these two and these two, I can compare them by just comparing those 26 numbers.

If they're all equal, then they're the same string after sorting. And if there's any difference, then they're different. So that's how I could do it in constant time if I can compute these representations. And it's not hard to do that. It's called a sliding window technique, where you compute it for the first k guys. And then you remove this item and add this item.

And just by incrementing the counter for b, decrementing the counter for a, now I know the representation for these guys. Make a copy of that, which is a copy of those 26 numbers, constant. Then I add on c, remove b. Then I add on a, remove c, add on d, remove d, add on c, remove b, and add on d, and remove c. Well, I got back to the beginning. So now I have representation of those.

OK, so by sliding this window, I'm only changing at the two ends. I add one guy on. I increment one of these counters. I decrement one of these counters. So in constant time, given the representation of one of these substrings, I can compute the representation of the next one. And that's how I, in linear time, can build such a data structure that lets me tell whether any two hands are equal.

The next problem, part b, is, given all these representations, can you find which one is the most common? Because we were choosing i uniformly at random, I want to know what the most likely hand that you get is. And I think the easiest way to say this is you can do that by radix sorting.

You take all these representations. They are nice numbers in the range 0 to n plus 1 to the 26th power. So I can just run radix sort and sort them all. And then with a single scan through the array, I can see which one is the most common. Or rather, I can-- in a single scan, I can compute, OK, how many of the same things are at the front?

If they're sorted, then all the equal ones will be together. So how many are there? Then how many equal ones next? And how many equal ones next? Each time, comparing each item to the previous one. Then I get frequency counts for all of these hands. And then I do another scan to find the most common one. And I can do another scan to find the lexically best one, lexically last one. And that's how you solve problem 5.