MITOCW | watch?v=pOKy3RZbSws

The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

ERIK DEMAINE: All right, today we continue our theme of integer data structures. And today we're going to look at priority queues instead of the predecessor problem, so something a little different. Insert, delete, and find min.

It turns out priority queues are equivalent to sorting in the sense that if you have, normally they say f of n, but here we're going to depend on both n and the word size, priority queue. Well then, of course you get a sorting algorithm. Just insert everything into the priority queue and then find min delete, find min delete, and so on. So that's the trivial direction of the equivalence.

But it turns out, both of these are true. If you have a sorting algorithm with n times f time, then you can convert it into a priority queue that in the worst case every operation takes only f of n, w. So that's the equivalence as the result of Thorpe from a few years ago, 2007. Kind of a cool breakthrough. There were earlier versions that weren't quite as efficient but this gives you perfect efficiency, turning any sorting algorithm into priority queue.

Rough sketch of the data structure is you have a bunch of sorting algorithms which you run now and then to get little parts of the input in order to build an overall priority queue. But it's very complicated. There was an entire class project on assimilating it and understanding it. It's a bit beyond what we can cover in the class.

So I'm going to take this as given and tell you instead about sorting algorithms because there's a major open problem here which is, can you sort n integers, n w bit integers in a word ram's, word size w in linear time. We still don't know but we have a bunch of results. In particular, we're going to talk about a couple interesting cases where we can get linear time sorting. If you've seen radix sort you know another interesting case where linear time sorting is possible.

Interesting open problem here. Can you get constant time decrease key? And ideally, also insertion? But decrease key would be handy because then this would be useful.

In this reduction if you have a sorting algorithm, like to get a priority queue with constant time decrease key, that would have applications to Dijkstra's algorithm. Of course, if you can sort in linear time then everything can be done in constant time. So this is not the most interesting open problem but it would be a step in that direction.

Cool. So let me tell you some bounds about sorting. So we have n w bit integers. And you take any of these bounds divide by n you get the priority queue bound, the best that we know.

So of course, there's comparison sorting, which takes n log n. So that's an upper bound. Trying to beat that. Trying to get rid of the log.

There's a counting sort from undergraduate algorithms. That takes n plus u time because you need to build a table of size u. You've got to zero out all the initial counts so that requires order u time. This is interesting when the word size equals log n.

So that's a case of potential interest, because then 2 to the w is just n. So this is linear time. But even if w is 2 log n, radix or 1.1 times log n, then this is bigger n log n. So, not very good.

But we have radix sort, which is a bit better. And this is n times w divided by log n. If you do it right you have some word size w, you're using counting sort for each digit in the radix. So you can afford that the word size of doing counting sort to be exactly log n. Which means you have your overall word you're dividing into chunks of size log n. Each of them you can do in linear time.

So the number of chunks you have to do is w divided by log n. And this is linear if w equals order log n. Right, but only then. So this buys you a little bit, but still not great.

OK, there's some other sorting algorithms we've implicitly seen because we've seen Van Emde Boas. I mean, any predecessor data structure gives you a priority queue data structure. Priority queues-- I didn't write it, but I mentioned-- insert, delete, find min. Find min is a weaker operation than predecessor or successor. So we can take Van Emde Boas's predecessor data structure and convert it into a sorting algorithm and we get order n times log w.

Now this is a little bit tricky to compare. Of course, log w sounds better than w. But there's just divided by log n, which makes it a little hard to compare. It turns out there's another version. You can take this sorting algorithm and achieve a bound of m log w over log n.

OK, so this is a slight improvement in here. It's similar to your problem set this week. This specific problem was a problem set in 2005 of this class. So you can check it out if you want to see how to do this slight improvement.

But now this is clearly better than this because here we have w over log n. Here I've got a log

in front. So this is a win over radex sort. So Van Emde Boas, here's another way in which is interesting. It gives us a strict improvement over radex sort.

And in particular, actually both of these are order n log log n if w is not too big, poly log n. This doesn't actually matter but it's clearly better than this one. Again, if w is order log n you get this to be linear. But in particular, it says overall you get n log log n up to w being poly log.

OK, why do we care? Because there's this other nice algorithm, signature sort, which covers all large word sizes. And this is an algorithm we will cover today.

It only works when w is large, and large means log to the 2 plus epsilon. And this is no matter what epsilon is. In reality, it's probably log squared times log log n squared. But be a little sloppy and just say, a little bit bigger than log squared. And then it runs in linear time.

OK, so if you were graphing this in your mind you've got time and there's a nice bound here which is order n. Then we have everything's above n log n. Maybe here we have-- is hard to draw-- n log log n. This is asymptotic space, which doesn't really exist, but you know what I mean. And then this is the word size versus log n.

Wow, this is hard to-- this is a really weird picture. I've never drawn this before. So you know, initially in a radex sort, OK log n. We could get linear time or order log n, great.

What this is saying is once you get out to log the 2 plus epsilon n you can also do linear. And from here over we have linear time. And here over we have linear time. So the question is what about this middle range between log n and log to the 2 plus epsilon?

Well, one thing we could do is use Van Emde Boas sort, get an n log log n bound. And so that's one way to fill in this range in the middle. And that means overall you can always sort n integers in n log log n time. If w is big you use signature sort, if w small use Van Emde Boas sort. So this implies n log log n for all w.

So that's something. It's not the best known. Let's see, first result in this direction was by Han in 2001. And it achieved this bound, deterministic and in the ac 0 ram, so without using multiplication. Which, I guess Van Emde Boas doesn't either, unless I'm missing something. But it got rid of the randomisation which is necessary to get linear space in Van Emde Boas.

The current best result on sorting is by Han and Thorpe in 2002, so a while ago. They achieved n square root log w over log n, which is similar to this bound, just has an extra

square root here. So it's a little bit better than Van Emde Boas sort. It uses some clever tricks. It is randomized. So still the best deterministic bound is n log log n,

And let's see, we can plug that in for this range here. And it will imply that you get n square root log log n for all w. Because in particular, if w is poly log n, then this is just log log n. So you get this bound. Of course, for all w because large w are already covered.

So this sorting algorithm is a little bit messy, and possibly not the right answer. I would conjecture you can sort n integers in linear time for all w. But that is a big open problem. This is the current state of the art. So this open question is really about for small w when w is poly log n, or actually when it's between log and poly log.

So to give you a piece of the complete picture I want to tell you about signature sort. Because I think you have to use different techniques when w is small versus large. We know that from fusion trees and Van Emde Boas. So I'll tell you about the slightly large case when w is at least log to the 2 plus epsilon, how to get linear time.

And this is fun. It uses bit tricks, it uses a lot of cool ideas, uses hashing, lots of things.

It also uses another sorting algorithm called packed sort. This doesn't fit in the regular table because it's for a situation when more than one input number fits in a word. So we're going to let b denote the number of bits in the inputs. So we have b bit integers. Then when w is at least b times log n log log n, so you can fit log n log log n integers in a single word, then you can sort them in linear time.

So it's like your machine is just a little bit bigger than what you'd need to represent the integers, then all sorting can be done in a linear time. And as you might imagine, this log to the 2 plus epsilon buys you a little bit of room with some tricks, namely hashing.

Right, that's where we're going. And we'll use yet another sorting algorithm, bitonic sort, which you may have seen. It's a sorting network.

It's in CLRS up to second edition, but it got removed from third edition. I guess they thought people didn't use sorting networks, but we use sorting networks here.

Because it gives us parallel sorting and the word ram is all about using the parallelism in your word. So, we'll get to those.

I'm going to start at the top level, which is signature sort. And we're going to leave things to be done and then pack sorting will be one of them. And bitonic sorting will be used in pack sorting. So it's a chain like this. Cool.

So I'm going to assume my signature sort that w is at least log to 2 pulse epsilon n times log log n. This is really just a change in what epsilon means. If you're given some particular epsilon make it a little smaller, then you can afford to add that log log factor. This is just for convenience.

Now here's the idea. It's a pretty cool idea I have to say. It's by Anderson and others in 1998.

OK, word size is big, which is both good and bad. It means the numbers we have to sort are big. They have a lot of bits in them, way more than log n bits. But also, we have this powerful machine that can handle words of very large size and do things and constant time on them.

I'd like to make some slop. I want my words to be bigger than my items, so first few steps are about doing that. The first idea is, well, you've got this giant word log to the 2 plus epsilon. So let's split it into chunks. And the number of chunks is log to the epsilon.

So that means each chunk is-- well, it could be even bigger. So it's at least log squared n. But it could be bigger than that because w is at least log to the 2 plus epsilon. So we don't know. These things are of size w divided by log to the epsilon n. So it's at least log squared.

OK, here's the crazy idea. We're thinking of our number being divided into these chunks, you know, it's being represented in some base, 2 to the w divided by log to the epsilon n. It's a bit of a mouthful.

But think about these digits. I claim if you look at a digit, most values don't appear. Because how many values are there for a digit? Well, it's at least 2 to the log squared n. That's a big number. That's way bigger than polynomial and n.

But how many different digits are there? Well, there's n numbers, each of them has log to the epsilon n digits in it appearing. And so there's roughly a linear number of digit values total. But the number of possible digit values is 2 to the log log squared n, which is huge, super polynomial. So most digit values don't appear in our numbers.

So what we're going to do is hash the digit values to reduce their space. Because they're currently living in a universe of 2 to the log squared n, but there's only n times log to the

epsilon n of them. So we should be able to reduce to have a much smaller universe using hashing. Maybe perfect hashing even. I mean, we don't have to be too intelligent here. Tell you what we get.

What I'd like is to reduce to a polynomial universe. And another way to say that is that the hash values are order log n bits. So this lets us go to n to the 10, or whatever, which is plenty of room. We take any universal hash function, as soon as you get to roughly n squared size, I mean the square of the number of things, by a birthday paradox, constant probability none of them will hit each other.

That was the second level of perfect hashing. We don't need two levels here, just say OK, user order log n bits. You can set this to 100, whatever. Then with high probability you will not get collisions, so none of the chunks will hit each other.

So I'm omitting a step here which is how to-- OK, I'm omitting a couple of things here. One of them is I want to do this in linear time. Everything has to be in linear time. So one front part is I have to compute the hash value of all of these digits in a single word in constant time.

So a brief aside, how would I compute a universal hash function of these digits individually in constant time? Well, my personal favorite universal hashing scheme is the multiplication method. You just need to multiply by a single value m and then take it modular something. So if I do multiplication, I mean if you think of there being something here, something here, something here, something here, something here, m does multiply each of these individually. Wow, that's a rough sketch.

I want to hash each of these guys multiply, them by the multiplication method. This would work fine, except you get overflows, which is a bit messy. So in fact, I'm going to have to do it for the odd guys first, multiply them out, mask out the bits I care about, then do it for the even guys separately. So it's a bit messy. But hand-wavy proof you can compute the hash function of each of these guys individually, but in constant time. So that lets me do this.

You have to then check for collisions, which is kind of annoying. Let's not worry about that. Set this really high, very high probability, there will be no collisions. Cool.

So at this point we have mucked up all the digits. The annoying thing about this hashing is we do not preserve order. It's a little weird for a sorting algorithm not preserve order. The whole point is to put things in order.

But it's going to be useful nonetheless. And so each digit has been mangled. Each digit is out of order. But of course, the sequence of digits has been preserved, in some sense

So, what? Well, at this point we can use packed sorting to sort what remains. In general packed sorting can sort n b bit integers provided b is small, so w is at least b log n log log n. That's the theorem of packed sorting. We're going to delay the coverage of this result until later.

And I claim that that applies here because each chunk is now order log n bits. And there's log to the epsilon n bits. So b is theta log to the 1 plus epsilon n. And w is at least log to the 2 plus epsilon times log log n. OK, so the ratio between these is as needed. It's at least log n times log log n.

OK, so that's why this is good. There were these things were huge before. We've compressed them a lot. They're at least log square, now they're only log. And so now we can apply packed sorting.

You can fit log times log log items into one word. And that's going to make sorting easy, somehow. That's to be done.

But now the question is, suppose we had packed sorting, how does this help you? Because this does not sort the original integers. It sorts these integers that have all of their digits permuted in some random way. We have no idea how the hash function re-orders things. So we're not done with signature sort.

It's called signature sort, of course, because of the hashing. You're taking signatures of each of these digits. Another word for hashes.

All right, next step is to build a compressed trie. This is a term we haven't used yet. So I'm going to call these things that we sorted signatures.

And how many people know what a compressed trie is? OK, no one. Wow, just a few. So you haven't seen suffix trees yet. We're going to cover where compressed tries come from in a couple of lectures but we get to see them a little early because they're cool and needed in this algorithm.

So first notion is the notion of a trie. The word trie comes from retrieval, from old days when information retrieval was what's now called web searching, or whatever. So it's now

pronounced trie. It's just another word for tree, but where the children of the tree come from a fixed universe.

We've been working with tries all the time in pretty much every lecture for the last n lectures, last w lectures, I don't know, a lot of them. We said oh, OK, well let's just consider the space of all possible keys. You know, I've drawn this picture zillion times. Even when we weren't doing Van Emde Boas, which actually stores all of these things, we were thinking about fusion trees like, oh yeah, so we're storing this value, and we're storing this value, and maybe this value.

And we understand this corresponds to a 0-bit. These are 0 and 1 bits. This is a 1, 0, 1, whatever. So we understand how to convert a bit string into this picture. This is what we call a binary trie.

I want to do the same thing over here, except it's not a binary trie. Now each digit is giant so we're going to have some huge branching factor. So it's going to be more like a b-tree. The first node has a whole bunch of children.

I'm not going to even-- I mean, it's 2 to the w divided by log to the epsilon n. That's the branching factor. OK, it's big. That's the number of possible values in the first digit.

OK. So then maybe this child has some node there. And maybe this one. Actually, sorry, I didn't want to draw it that way. I have a specific trie in mind, or part of it.

OK, I mean, trie is going to look something like this. It's harder to draw, but it's the larger than binary equivalent of this picture. This would be a trie.

So let's think about a particular situation. Which is, suppose we just have two items in this structure. Then it's easier to draw. So let's suppose there's some item over here, x1. There's some item over here, x2.

Now, generic picture is they share some common prefix. This is the common prefix. It's just to understand some particular picture here. Then they diverge, and you know, then they have their own digit values. In some sense, this is the digit value we care about because it distinguishes them.

So there's this idea of a compressed trie, which cleans up this picture a little bit. The idea with the compress trie is very simple. You draw this trie. So here, imagine I'm erasing all the things that do not appear. Maybe all this is gone. So I'm just keeping this part of the structure.

Let's do it for the binary case first. The idea in a compress trie is very simple. I'm going to contract every non-branching node into its parent.

So what that means. So here I'm storing three values: 0, 0, 1, 1; 0, 1, 0, 0; and 0, 1, 1, 1. So some nodes are non-branching, that's this node, this node, and this node. And the point is well, why store those nodes? I mean, there was nothing being distinguished at that node, so contract it.

So the new picture if I redrew this would be 0, 0, 1-- sorry, also this is non-branching. So in fact, I'm going to contract this into its parent, this into its parent. So this whole path, this non-branching path will be left with a single edge which I will label 0, 1, 1.

But it's a single edge in this structure. Over here we have an edge 1, and now we have 0, 0 here, and 1, 1 here. That is the compressed trie representation of these three values.

The nice thing about this structure is every node is branching except the leaves, which means the number of nodes here is linear in the number of leaves. That's like twice the number of leaves minus 1, roughly. I guess the root might not be branching. It has no parent to contract into. Whatever.

But more and more nodes, most 2n nodes. That's the binary case. Now over here what we do is in the corresponding contracted try a picture of this thing. Let's see, it's a second edge. I'm going to put just a label of 2 on this edge to say, well, I'm skipping this node. This guy is non-branching.

So I want to get rid of it and contract it into the parent. So now this corresponds to two edges here. There are two digits, which I could write the digits if I knew what they were. But particularly I want to keep track of the fact that there exactly two of them.

And then there's a branching node. That one stays. But then I actually we'll go straight to the leaves here. This will be x1 and x2.

And this will have a lot of digits. OK, so some large number written here. So that's the idea with a compress trie.

Now the nice thing about a compress trie is they have linear size, number of edges and vertices is linear in the number of leaves, which is the number of values we have here.

Another nice thing about compress tries is they preserve order. So if you look at the order of the leaves down here it's the same as the order of the leaves over here. We're not really changing order, we're just making things a little shorter to write down.

Same thing over here. I mean, x1 appears the left of x2. Say the order of the leaves is preserved.

The other cool thing about compress tries is you can compute them in linear time. But let me tell you why we're drawing this picture. We've messed up all these chunks, all these digits. What does that correspond to in this picture?

We've permuted all of the children in this node. We've permuted all the children in this node, permuted all the children in this node. But otherwise, the structure of the tree is correct. What we wanted to do was compute-- if we could compute the compress trie representation of the original values just having broken the integers into these pieces, then if we had that trie we could just do an in order traversal, find the order of the leaves, then we'd have the sorted order of the integers.

Instead what we've computed is we didn't compute the correct compress trie. But we computed a different compress trie where every node has its children permuted. So all we need to do is put the children within each node in the correct order, then we would have the correct compress trie. OK, that's why we're doing this.

But first, how do we compute a compress trie in linear time? This is a simple amortization argument. Remember what we're given. At this point we are given the leaves in sorted order. We've just sorted our things so all I'm trying to do is compute the trie of the stuff above the things.

OK, we're given the leaves of the trie in the correct order. I just need to build the compress trie. The reason I'm building a compress trie is so I can do it in linear time. Constructing the uncompressed trie would take too long. OK, so how do we do this?

We have our items xn in order. I want to just insert it into the compress trie. So I start with an empty compress trie. First thing I do is add x1, so it's just a single leaf. Then I add x2 to the trie, x3 to the trie.

In general, I'll have computed-- I'm going to draw a picture. How to draw a partial trie, something like this. I've computed the left part of trie already. All these x1's have been done.

And now I need to add on the next, say xi over here somewhere.

So think of how that would look in the trie. It has to come off of the right side, I guess what would you call this? The right spine of the tree. Some right most node over here. We want xi to be a new to the right child.

So one possible picture is there was some node which had some other children, maybe this one had xi minus 1. Maybe xi belongs over here. Maybe this leads to xi in the uncompressed trie. This could be a potential picture.

In the compress trie this node either exists or got contracted. If it exists maybe there was a previous xi minus 2 belongs over here. So we already had this node because it distinguished these two xj's. To add the xi is really easy. I just add a node here and then which points to a leaf of xi. So that could be one situation.

The other situation is only slightly more complicated. So the other possibility is this node didn't exist in the compressed trie, which means we jumped over it in the compress trie. Some number bigger than 1 here, maybe you skipped three nodes, who knows.

So now this node matters. It didn't matter before. Before it was non-branching, but now we add xi, it's branching. All we do is update, add this new guy, add the new pointer to xi. OK, so constant work.

Great, constant work per xi inserted, linear time overall. Not really. I've cheated a little bit here, which is how do we know what we're going to do this operation. So the changes we make is constant, that's fine. But how did I find that this was the edge where I need to put xi in? You can't do it in constant time, but we can do it in constant amortized time.

So what we're going to do is start at the right most-- that is not very red. Not a red chalk. Red chalk.

We start at this leaf, the rightmost leaf which you go going right most as possible. We're going to walk up the tree until we find the right point. We're going to spend linear time this path walk. OK, how could this possibly be OK?

I think you believe that I can do it in linear time, right. Each point I see-- what is my common prefix? Actually an easy way is right up front. This thing right here is xi minus 1. These are single integers. So I can just compute their x or, find the most significant one bit, which we did

with the fusion tree lecture.

And boom, I know where they differ in bit position. By some rounding I can figure out which chunk they differ in, just by dividing, taking the ceiling. So I know the correct depth. And then these numbers tell me how much depth I'm losing.

As I walk up at some point I'll discover, oops, I just jumped over the depth I care about where they differ. And so at that point I know to do that operation. So if there are k edges here on the compress trie, I can do this in order k time, k plus 1, let's say.

Why is that OK? Because my potential function, amortization is going to be the length of the rightmost path. OK, before my rightmost path was this thing.

Afterwards, my rightmost path is going to be this thing. So it got roughly k shorter, k minus 1 shorter. OK, so I charged this k plus 1 cost to a potential decrease of k minus 1. And so the amortize cost is only constant. Easy amortization.

Another way to see this is, overall we're basically doing it in order traversal of the tree. So whenever we walk up we'll never visit these nodes again, because then we go down at a different place. OK, yeah, this guy we're going to walk up again because in an in order traversal we would have gone left and then come back to it and then gone down. So overall we're doing an in order traversal of the compress trie, so it's linear time total. OK.

Still not done though. What we have so far-- what have we done? We've taken our numbers, we've split them into digits. We randomly permuted the digits via hash function. That magically let us sort them in linear time. Now we've built a compress trie of that representation.

The last thing to do is to fix the order of the children from each node. The tree is correct except for the order of the children in each node. So we've kind of sorted a little bit, but haven't fully sorted. So that's the last step.

And we're going to do that with recursion. We're only going to recurse a constant number of times. Recursively sort.

OK, I'm going to walk through this compress try trie we've built. And I guess it looks more like this. And for each edge I'm going to write down this key.

First thing you're going to write down is the ID of the node. Let's say the ID is the in order

traversal index of the node. OK, so boom. As we traverse the tree we can just increment this counter.

Then we also want-- what I'm going to put in here is the actual chunk value that's at the top of the node. So something like this, or I guess this one. This one consists of two chunk values. So it's called c1, for chunk values, c1 and c2.

It's the first chunk value and the second chunk value, first digit and the second digit. Just write down the first one. That's really the only one we care about. The second one wasn't branching so it's no big deal.

OK, so there's some chunk value. Now, I don't want to write down the signature chunk value. I don't want to write down the hash value. I want the actual chunk, original chunk not the hash. That's why I write actual over here and then also, I'm going to write down the index of the edge. So this is the second edge, for example.

OK, now this is in some sense the wrong index. And what I want to compute is where it's supposed to be. The edges have been permuted. So if I sort this thing, I claim that I'll find that all the desired permutations. OK, why?

Because if I sort by this triple, first thing I sort by is by node ID, then I'm sorting by actual chunk value, then edge index. I don't care that I'm sorting by it, that's just coming along for the ride. So what this tells me is for each node it gives me the sorted value of the actual chunks, not the hash chunks. And then it tells me the order they used to be in.

So this is a way to figure out the sorted permutation of these things. This gives you the old index. And so once you've sorted them you know that new correct order is in order by chunk. And so this gives you the inverse of the permutation over here, of where each edge should be. So once I've done this the last step is for each node permute the edges as given by this inverse permutation. Questions?

AUDIENCE: Can you explain again that? You were basically sorting three times.

ERIK DEMAINE: No, I'm sorting these single key values. The key value, the most significant part of the key is this. The next most significant part of the key is this. And the least significant part of the key is this. So imagine concatenating these things together into a single key value. Yeah, good question.

What I'll talk about in a moment, how big this key is. But if I sort by the concatenation of those keys then what happens is I'm sorting by this and then sorting by this. But really, I'm doing one sort operation. This is one recursion. Other questions? OK.

Do you see why this gives us what we need? For every node we want to sort all those chunks. And so this information is just so we can keep track of the permutation when we're just sorting by value and not keeping the permutation. This let's us keep track of it.

But our real goal is to sort all the chunk values. And we add in the node ID so that we just learn for every node what we're supposed to do instead of globally what we need to do. Maybe you don't have to do this but it's easier to keep the node ID in there.

OK, how many bits are in this key? So our whole point was to make chunks small so that the keys were small. But I claim this will be a little bit smaller than what we started with.

So the node ID, well, there's only n nodes, so this is log n bits. Chunk value is, I guess, w divided by log to the epsilon n bits. Because there are a log to the epsilon n chunks, so it's w divided by log to the epsilon bits. The edge index, how many different things were there?

Is it also w divided by log to the epsilon? I guess so. Depends how you count. Let me think for a second how these edge indices should be stored. I think I'd like to store the edge indices-- I don't want to store the absent down pointers.

The way that I built this tree, this compress trie, I was always adding things to the right. So it's like OK, I'm storing here. Here's the first child, here's the next child, here's the next child, here's the next child. I just keep appending on. It's like an array that you can grow to the right, which you should know how to do in constant time, even though linked list is probably fine.

If there are a bunch of values here that are just null pointers I don't want to store those items in the array. So in fact, the maximum number of children of a node is n. I mean, at the worst case you have branching everywhere. So this is only log n bits to store the index in that array. So it's an extra level of compression.

I guess I didn't mention don't store null pointers, just skip them. And that's easy to do because you're always just appending. And in the end what do I need to do here?

Step seven is in order traversal of the compress trie. And I output all the leaves I get in order. And the point is we computed the compress trie which has the rough correct topological

structure. Then we fix the order within each node, now we have the correctly sorted compress trie. And so you're just doing in order traversal, output the leave you get in order.

Boom, we've sorted all the items. A little crazy. So if all we need to do in the end is in order traversal then it's fine we don't need to store the null pointers. So we don't need to be able to search in this trie, so this is enough. Clear?

OK, the remaining question is how expensive is this recursion? So I computed the number of bits basically w divided by log to the epsilon n, plus order log n. That order log n won't hurt us. Now, we started with integers of size w, the total size here was w. Now we have integers that their size is w divided by log to the epsilon n. So we've made progress. Cool.

How much progress? Over here maybe. After, let's say, 1 over epsilon plus 1 recursions we will have reached b will have decreased to order log n. I guess, like 1 over epsilon log n plus w over log to the 1 plus epsilon n.

OK. Right, if we take w and every time we divide by log to the epsilon n, then after 1 over epsilon times we have divided by log n, get rid of epsilon. Do it one more time, we get w divided by log to the 1 plus epsilon n. And at this point we're in good shape.

If we get the size of our integers to be only w divided by log to the 1 plus epsilon n, then we can use packed sorting. Because packed sorting, conveniently it's the black box at the moment, says well, as long as w divided by b is a little bigger than log n. So here we've made it quite a bit bigger than log n. I mean, log to the 1 plus epsilon n bigger, then we can use packed sorting. Reach base case, and then use packed sorting.

OK, the only thing is there's this log n. Now, epsilon is a constant here, so 1 over epsilon is constant. So this is just order log n. Order log n doesn't hurt you. I mean, if this happened to dominate this then you're saying your values are order log n bits long, and so you use radex sort. So that's not going to dominate.

In fact, you can prove in this case it's not going to dominate. w is at least log to the 2 plus epsilon. So you take w divided by log to 1 plus epsilon, that's still at least log n. So this will always dominate.

OK, so we did all this work and all we did was reduce our word size by a log to the epsilon factor. But that's enough, because after we do that 1 over epsilon plus 1 times the words are small enough, or the items we're sorting are small enough, that we can use packed sorting.

And boom, we're done. OK, that is the beauty of signature sort.

Any questions about that? Definitely not easy. But, it works.

The crazy idea is, well, decompose into these digits. And when we decompose into log to the epsilon digits because that would be enough. If we can get things down to a single digit life would be easy. Which I guess is kind of like Van Emde Boas, right.

It's like our lower bounds from last time. I don't know. If we can make it to only one digit mattered then we could afford to do it because each digit is quite a bit smaller. It's smaller by a factor of log to the epsilon n.

So we did that splitting and then there was this observation that most digit values aren't used. So if we hash then things got a lot smaller. Then we can afford to pack sort. Except that hash didn't preserve the order. How didn't it preserve the order?

Well, when you look at the compress trie you see how it got messed up. Each node got messed up individually, but overall the structure of the trie was correct. So we did all this work to get the structure of the trie. Then we had to fix the node order. But that's really just sorting a single digit.

And yeah, we have to do it for all nodes but you add them up. It's still only n items that we need to sort. So I wrote that b went down. n stays the same. The number of things we have to sort of stays the same, which is fine. We're not trying to reduce n, we're just trying to reduce the size of our items to be a little bit smaller than w so then we can use packed sorting.

OK, I think I've gone over this algorithm enough times. The next step is packed sorting. If we have integers that are much smaller than our word, how do we sort in linear time? And this is going to be essentially a fancy merge sort.

This is done in the same paper. Before we said omega, I'm going to essentially make omega a little bit bigger than 2. So I want to assume that w is at least twice b plus 1 log n log log n. That's just to make this convenient, but of course it doesn't really matter what the constants are. This will make everything fit in one word though, instead of a constant number of words.

OK, step zero of this algorithm is the packing part. We can fit more than one element into each word. So put them in. Now, we have linear time overall. This is easy to do.

You have some word already. Take those items, shift them left or in the next item, shift them left or in the next item, shift them left or in the next item. OK, linear time. So in the end what it's going to look like-- oh, I also want to leave a 0 bit in between each of them. So we're going to have b. This is going to be x1 here, x2, xn.

Each of these is b bits long. And then we have one more bit. And in fact, the way we set things up this is only half the word. So there's another half over here, which is going to be all 0 for now.

So we have one word contains-- sorry, this is not n items. This is going to be log n log log n. Can't fit them all in one word. That would make life really easy.

You can fit log n log log n items in the first word. Then the next word is the next log n log log n items and so on. Linear time to do that.

Next step, this is an algorithm. It's not really a step. I should say these are things that we're able to do, and we're going to combine them in different ways. So the next thing to observe, this is sort of a bottom up perspective.

So first thing we can do is pack things into word. Next thing we can do is merge two sorted words. Let's say they each have k items and k is at most log n log log n. And I want to merge them into one sorted word with two k elements. So I have two of these things and I'll merge them in sorted order.

And when I did this packing, nothing's sorted. But don't worry about that at the moment. Just if I had two these words and suppose they were already sorted, then I want to merge them and make them still sorted. OK, I want x1 to be less than x2 and so on.

How much time? It'd be cool if you could do this in constant time on a transdichotomous rem. You could do anything on a constant number of words in constant time. We're not going to achieve constant time, but we don't need to. We're going to do order log k time.

ERIK DEMAINE: This is the hardest step of packed sorting, and I'm not going to tell you how to do it until later. I will tell you, don't worry. But this is going to require bitonic sorting. So I have to delay it a little bit.

Suppose you had that for now. Let me tell you how the rest is easy. So many black boxes, but they're all filled in in this lecture.

OK, next thing you might ask is, well, how do I get a sorted word? I see how to get a single word that has log n log log n items. But how would I sort those items? Use merge sort. Here's a merger. You have a way to merge two sorted lists, so use merge sort, that will give you a way to sort the whole thing.

So I'm going to merge sort k equals log n log log n items into one sorted word. So I start out with a word that's unsorted. I do merge sort. And I sort them. So this is not the whole sorting problem but it's the one word sorting problem.

So how long does this take? Well, it's usual merge sort recurrence, but then it's plus how long it takes me to split the word. Well, splitting the word is easy, you just mask out the low end or the high end and then maybe shift over. So in constant time I can get the left half and the right half of the array, which is the word.

Then the hard part is merging them. And merging them is 1, and that takes log k. So this is step one. That's our merger.

And so what does this solve to? Well, probably easiest to draw the recursion tree. So at the root we pay log k. Then we have 2 times log k over 2, log k over 2. So at this level we're paying log k. At this level we're paying 2 log k minus 1, I guess.

I'm just going to call this maybe-- yeah, I do need to do that. I mean everything is constant size so you pay a constant. How many leaves are there at the bottom level? k of them.

OK, so at the root we're paying log k, at the leaves we're paying k. This is roughly geometric. There's this minus 1, but it's geometric enough and when you add up all these levels it's dominated by k. So this ends up being order k time. The minus 1 here turns out not to matter much. Cool.

So this linear time sorting on log n log log n items. Everything it fits in a word. Cool, but we're not done yet. We need another level. The rest is pretty easy. It's just scaling up.

OK, let's see. At this point we're going to assume that all of our words are completely full. So this merging and stuff was in order to fill up words and to make those words sorted. But once we've done that for each cluster of log n log log n items, now we can assume each word is sorted and completely full. It has exactly k elements in it. So each of these r sorted words has k elements.

OK, so now the issue is things don't fit in a word. And so now suppose they fit in r sorted words. So supposed we've already made a sorted list of r times k items which fit in r words, each with k items in it. Then I want to take two of these lists and merge them together and get one sorted list of sorted words. So it's 2 rk items that are distributed into 2r words, each of size k.

How do I do that merge in linear time? And I only need to do it in r log k time, only. Yeah, question?

AUDIENCE: I'm assuming k is the number of items per word?

ERIK DEMAINE: Yeah, k is this from now on. And there are exactly k elements per word. We've already filled up the word so at this point we can assume that.

AUDIENCE: But it's exactly equal, not less than.

ERIK DEMAINE: Right.

AUDIENCE: Because in one--

ERIK DEMAINE: Yeah, over here it's less than or equal to. I know it's a little confusing. I use k for different-- call this k prime. OK, and at this point we've filled up to this value of k. Now everything has exactly log n log log n items. Thanks. OK, let's see.

A regular merging algorithm would take r times k time because they're r times k items. But we have this merger over here, which will merge two words in log k time. So that's why it's going to be r log k. What's the picture? Well, let me draw it.

We have a list of words. A list of words. How do we merge? Well, we take the first two words, merge them. When we merge them and we know how to do this n log k time, that's step 1. When we merge them we get two words, they just are now sorted.

So we've got these two guys sorted and then there's the rest. I'd like to say output these and repeat, but that wouldn't be quite correct. Well, these guys I can output because I've got k items here, k items here. These are the k smallest among all of them. Those must be the overall smallest k items. Because I compared k with k.

These higher k items, I don't know. They might be good or bad. So I have to put this high part

back into one of these lists. You have to think a little bit about which list to put it into.

Turns out if you look at the max item, say where did this come from, list one or list two? You put it into that list. Let's say this came from list two. And you put it back into list two and then repeat.

OK, so in log k time here we have output k items. And so we end up with an r log k running time overall. OK, so it's like regular merge algorithm, except we use step one to merge words, and then there's a little bit of work.

This is like old merging algorithm, right? You take the min and you put the max back in. Except now the max is a whole word. OK that's how we merge.

Now, why did we do a merger? Because next thing we do is merge sort. And this is the overall algorithm. It's one merge sort. We're going to use this is the merger and we're going to use this as the base case.

So it's actually two recursive levels of merge sort. Merge sort is recursive within step four for a while, until we get down to the level of a single word. Then we use this thing to deal with things in a single word. So we end up with t of n is 2 times t of n over 2 plus n over k log k. And base case of t of k is order k.

Why is it all this? This thing is r. So we have r log k to do a merge. r here is n over k.

Why is it n over k? Because n is the total number of items. We had r times k items, so n equals rk. So r equals n over k. That's that.

So this is our merge cost. This is the cost of 3, this is the cost of 2. If we have only log n log log n items, so k items, then we can sort in linear time. So we just need to solve this recurrence. Well, how's it going to work? It's going to be kind of like merge sort, but not quite. Question?

AUDIENCE: When you say-- so you have-- it's definitely merging two lists of sorted words into one sorted list of two words. [INAUDIBLE]? You may have to do that, right?

ERIK DEMAINE: Say that again.

AUDIENCE: So you have two sorted-- two sorted lists of R sorted words.

ERIK DEMAINE: The words are going to change, which elements are in which words are changing. But that's

thanks to step one, for example. I mean, step one can merge any two sorted words and make one sorted word. Or actually, one sorted word. In our case, we're going to get two sorted words. That was the picture over here.

I merge the two guys. I could represent it as one word that's kind of double full, and then I can split that into two words. So we again need the split operation, but split is just a mask these guys out and you get the high part. This is all 0. Then you shift it over to the left, now you've got the high word over the single word.

So they're shuffling within those two words, or within one word, essentially, via this procedure, which we haven't covered. So everything is going to mix around. But then there's the low items and the high items. And so all the items in here are less than all of the items in here. Other questions? Cool.

So I need to solve this recurrence. So let's draw another recursion tree. That's the east way to solve recurrences. Root is n over k log k.

Next level down, see n got divided by 2. So it's going to be 1/2 n over k log k, 1/2 n over k log k, because the n gets divided by 2. So if I add up everything on this level, I get n over k log k. And in general as I go down, all levels are going to sum to this same value, n over k log k. So up here we are going to have a cost of n over k log k times the number of levels.

Now, this is where it's a little tricky. It's not log n levels like regular merge sort. We stop at level k. So it's going to be log of n over k. That's the number of levels we get to.

Now at the leaf level finally we have things of size order k, and we only pay order k. Great. And then how many leaves are there? There's n over k leaves. OK. So, that's an order n cost. This is order k, that's n over k, order n.

What's this thing? Well, what's k? k was log n log log n. So this is n divided by log n log log n. This, therefore, is approximately log log n.

So that log log n cancels this one. This is approximately log n, so that cancels with this one. So this is order n. That's why we chose these values.

So this n over k was not actually that significant. We could have called that n. What is important is that this recursion doesn't work when you get down to a single word. Can't do the same strategy. You need to use a different merger for single words because here we have to

do bit tricks.

For this merge strategy we didn't have to do big tricks because everything was bigger than a word. The reason why we have these two levels is not to make the time fast, it's just to make the algorithm work. I mean, to define the algorithm there's within a word merging and over many words merging. The hard part is this one, within a word merging.

So that's our next goal. But if we could do that we get linear time sorting for packed sorting when you can fit log log log n things in a word. Any questions about packed sorting? Good.

So, last thing is how do we merge two sorted words into one sorted word. And this is where we're going to use bitonic sorting. See, it says right here, bitonic sort for merging sorted words. Here we know what to do. Much easier in hindsight.

So I'm going to take a little diversion tell you about bitonic sorting, briefly. Because it's NCLRS I don't want to spend too much time on it, but it is something we need. So you need to know how it works.

So first thing is the idea of a bitonic sequence. This is going to be a cyclic shift of a uni-modal sequence. So a non-decreasing plus a non-increasing sequence. So I want non-decreasing and then non-increasing. So this is a bitonic sequence. I think you know what this means.

Or it could be a cyclic shift thereof. So I could, for example, take this part here and shift it over to this side. So then I'd end up with this, this, and this. This is also bitonic. or I could shift this whole thing and put it over there so I end up with that. That's also bitonic.

So it basically has one max, one min if you're reading it cyclically. One local max, one local min. Except, it's non-decreasing so it's a little messier, but same thing. OK, let's go over here.

So if you have a bitonic sequence there's something called a bitonic sorting network, which you can think of as a parallel sorting algorithm, to sort such a thing. So it's almost sorted, but not quite. And there's this way of sorting them. So I'm going to draw a picture to make it easier for me. It's a little more annoying to write down the general algorithm.

But so what I want to do, suppose the number of elements is a power of 2. Here's what I'm going to do. First I compare the first and the midway element. And then the next one with the next one, and then the next one with the next one, the next one with the next one. For each of these comparisons I put those items in the correct order.

So if I do that 3 versus 6, 3 is less than 6, so 3 stays here, 6 stays there. 5 is bigger than 4 though, so 4 comes over here, 5 goes over here. Next is 7 versus 2, so 2 comes over here, 7 comes over here. Nine versus 0, so 0 is over here, 9 is over there. OK, that's an easy set comparisons to do. The nice thing is you can do all those in parallel.

OK, just to check here. This was a monotone increasing sub-sequence and then a monotone decreasing. Now, we have increasing then decreasing. And then we have decreasing then increasing.

But if you look at the left half or the right half, they are still bitonic. And that's an invariant. Bitonic will be preserved here. Also notice, all the elements in the left are smaller than all the elements in the right. That's not true for an arbitrary sequence, but it will be true for bitonic sequence.

Essentially the max is somewhere here and so it's going to be spanned by some of these intervals, that max. And by spanning those intervals, and in particular comparing that guy with somebody on the right, the max will get over to the right and then the rest of the increasing and decreasing sub-sequences will follow. That's a very hand-wavy argument. You can see CLRS for a way to prove that OK, just take it for granted now.

Now, we recurse on the two sides. So I compare 3 with 2, 4 with 0, I compare 6 with 7, and 6 with 9. So what do I get? 2 is less than 3. 0 is less than 4. Then I compare 6 with 7, so 6 less than 7. 5 is less than 9.

So now I've got four chunks, this one, this one, this one, and this one. I recurse on each of them. So 0 is less than 2, 3 is less than 4, 5 is less than 6, 7 is less than 9. Now I have all my items in sorted order, magically.

OK, but if you believe that invariant that the smallest n over two items get to the left and the larger n over two items get to the right, then by induction this will sort them. Kind of like quick sort in that sense. OK. The cool thing is I can do all these in parallel, all these in parallel, all these in parallel.

So the total time is log k, if I'm a parallel sorting algorithm or a sorting network. And log k is exactly what I can afford. So the last thing to do is to implement an operation like this, or like this-- this is kind of the generic picture here-- in constant time on a word ram.

One other thing, because I don't want to sort a bitonic sequence. That wasn't my goal. My goal was to merge two sorted sequences. I have two sorted sequences, I want to merge them. How do I make that bitonic?

I just flip, like that. I take this sequence and I reverse it. If I take a sorted sequence and reverse it then it looks like this. And so then the result is bitonic. Then I can use bitonic sorting to sort it. So this is the merge problem and I've just reduced it to bitonic sorting. Cool.

Except, how do I reverse a sequence in log k time on a word ram? Cute trick. If I have a word and it has many items in it-- that's our picture, that's our set up here-- and I want to reverse it over all, what I'll do is cut it in half, take all these guys put them over here, take all these guys put them over here. So I've got the right side-- sorry, it's only four-- cells over here. I've got the left side over here.

Then I recursively reverse the sequence and recursively reverse this sequence. And then I have the reverse sequence. Right, this is a standard. One way to reverse items is to do the big part and then recursively reverse. Then I will get, I mean this is like ab reverse is equal to b reverse a reverse.

Just a fun fact. So we're implementing that.

Now how do I do this? Well, this is the thing I've been talking about over here. You want to take the high part you just mask that part out, shift it to the right. You want to take the low part, just mask that part out. And so I can take each of these parts individually, shift this one to the left, shift this one to the right, order them together, and I've reversed.

So this takes constant time. This is recursion but the total number of steps will be order log k, which is exactly what I can afford. So that's how I do the reverse part. So if I want to merge, now I have a bitonic sequence.

Last thing to do is, how do I do this bitonic comparison in constant time. So, that's next page. Here's a quick sketch. I'm going to try to draw the generic picture. So the generic picture is I have this thing which I call a, this thing which I call b, and we basically want to compare corresponding items in a and b.

Though in fact, there's another a over here and another b over here. In general, there are many a's, many b's, but they appear in this nice periodic pattern. And so I'm just going to look at one ab pair, but it's going to simultaneously work for all of them with the same operations.

So that's the dot, dot, dot here. There's another a and then another b, and another a and another b.

So what do I do? First thing I do is make 1aa, 1aa, 1aa. In other words, I add in these 1 bits. Remember, we had zeros there hanging out. That was when we were going to use them.

So I set them all to ones in the a list. I take the b list, I just shift it over and leave them as zeros. Time for bit tricks.

Then I take this number, I subtract it from this number. What do I get? I get zeros or ones for whether this bit got borrowed, which tells me whether little a is bigger than little b.

Then I get some junk. And then I get 0, 1. Junk. So on. 0, 1, junk. So 0 corresponds to a being smaller than b, and 1 corresponds to the reverse. OK, whatever.

Then I do my usual masking, just the 0, 1 bits, then there's some zeros. OK, here's a new trick which we haven't seen yet. I'm going to shift this to the right by two, or in general by the size of those items. So now I have 0, 1 here, 0, 1 here, 0, 1 here. Each of these bits might be different.

So I take this, shift to the right. Now I subtract. What this gives me-- so I'm taking 1, 0, 0, 0, minus 1. What I get is 0, 1, 1, 1.

OK, in this case, there's just two bits. Except it's not 1, 1. It's either 0, 0 or 1, 1. And then we have here 0, and then 0, 0 or 1, 1, and so on.

So now these are masks. The zeros correspond to when the a's were smaller, the ones correspond to when the b's are smaller. So if I take this [? added ?] with this, I get the large b's. If I take this [? added ?] with this, I get the small a's. Or one of some of those combinations.

The point is I take-- I think here I get the small a's. And then if I take this thing and negate it and then mask it with this I get the small b's. And then I take this thing-- I ran out of room. So I've got small a's, small b's. I take this thing I shift it over so I have small b's here. And then I OR these together, I get all the smalls.

OK, these are designed to never conflict because I took the inversion here. So I get the smaller of the a's, smaller of the b's. So now I put the smaller things in the right order. I do

exactly the same procedure negated, and get the larges. And then I OR these together and I get sorted.

Well, not sorted. I get whatever bitonic sort would have done. It's doing these pairwise comparisons by doing this shift and it's putting things in the right order. The small ones always end up on the left, the large ones always end up on the right. That's the definition of the operation we want to do. Bitonic operated.

And because we are just doing shifts, and subtractions, and all these things, it works even though there's not just one ab pair, but there's a whole bunch of ab pairs. They will all be shifted, and computed, and sorted together. It's a little hard to visualize but this will do one bitonic operation and a constant number of word ram operations. And so you pop out all these stacks this gives us a merger, that gives us packed sorting, and with packed sorting we get signature sorting. Easy, right?

All right, that's probably one of the most complicated algorithms we'll see in this class. But from that you get a priority queue that runs in constant time provided w is a little bigger than log squared n. Boom.