MITOCW | watch?v=ABX-Hvn8ymE

ERIK DEMAINE: All right, welcome back to Succinct Data Structures, part two of two. Today we're going to take all the stuff we know about tries and apply them to the main motivating application, which is suffix trees.

And as we know, suffix trees and suffix arrays are more or less equivalent. But if you build one, you can build the other. But what we're going to show today is they're equivalent also from a space perspective. That will be the last topic. If you can succinctly represent a suffix array, then you can transform-- with a little o of n extra space, you can make a suffix tree as well and do searches in roughly the time we're used to, which is p plus size of output.

It's not going to be exactly that. We're going to lose like log to the epsilons and such. But that's mostly caused-- this transformation only occurs like an additive log, log, log, log, log, log n. You could have as many logs as you want.

Take any arbitrarily, slowly growing function, that it will-- your space bound gets closer and closer to linear. Anyway, that's what we'll get to at the end. The bulk of the lecture will be on building suffix arrays.

Here we're going to lose a log to the epsilon time in the query. And we're going to start out improving down to T log log T space. Our bottom line is T log T space. That's a normal-- if you just stored a suffix array as a bunch of numbers.

First we'll add another log, then we'll get down to linear. That gives us a compact suffix tree, or sorry, suffix array. That's also knowing how to do succinct suffix arrays.

But there are dozens of papers on this topic, it's kind of a big field all to itself. And a lot of the techniques are pretty complicated. So I'm going try to keep it to the bare minimum we can do, that will give us linear-- linear number, bits of space give us a compact data structure.

But before we go to those data structures, I want to give you a little survey of what's known. So compact suffix arrays, and trees, start out with-- I'm going to start out with the original results. And then I'll jump to sort of the latest results, which are getting things to be succinct.

So the first result on this topic that got a compact suffix array, is by Grossi and Vitter. This was in 2000, spring of 2000. And let me tell you the bounds that they achieve. This is actually the

solution that we're going to look at.

So this is the first space bound. I guess the big term here is T log sigma. That's how many bits it takes just to write down the text. So this is what you might call optimal, in this world. I mean, if you have random text, you need that many bits to write it down.

So there's 1 times that. We're also going to have this 1 over epsilon times that. And this is actually the data structure. It's going to store the text, and then it's going to add on a data structure of 1 over epsilon times that.

So it's order-- order [? ops. ?] There's some lower-order terms. We won't actually have this lower-order term, because I'm going to focus on binary alphabet here. Keep it simple. But if you have a non-binary alphabet, they have another order T bits, and so on.

But you get to control this constant. This will work for any epsilon between 0 and 1. And why are you interested in a small epsilon? Because if epsilon is small, this space bound goes up. Well, that happens in the query bound.

So in the query bound, there's this multiplicative log to the epsilon of T. So if you really want queries to go fast, you don't want to pay a big polylog here, then you're going to have to pay for it in space. So those are the same epsilons.

In the Grossi-Vitter paper, they only multiply this by the size of the output. So if you want to just output one guy, you only pay an additive log to the epsilon. If you want to output all the matches you have to pay a number of matches times log to the epsilon.

They achieve the P bound. In fact, they do a little bit better than order P query. On a RAM, you can hope to do-- save a log factor by reading log base sigma of T, of the letters in one word operation.

So I'm not going to go into how to do this-- I'm going to cover this paper today, or a simplification of this paper. You might say, throw away. I'm going to get a slightly worse bounds than this.

Space bound will be the same, but I'm not going to-- I'm not going to worry about this log factor. And in fact, both P and output are going to be multiplied by log to the epsilon. So I won't achieve quite the best query bound, but same space bound, just to give you an idea of how it works.

The next result-- yeah, I'll go to another board. These bounds are a bit big, as you see. The next result, which was done later in the same year. So these are probably discovered, basically at the same time. Because writing a paper takes probably a year or so.

So they were being done in parallel, and then this was published in the spring of 2000. This was published in the fall of 2000. It's called the FM-index. And it achieves this bound, which is going to take a little while to explain.

OK. Think of this right now, as this is T log sigma. Ignore this H. This is entropy stuff. But if you think of this as T log sigma, we're getting 5 times T log sigma, plus some lower-order term. So it's a little less flexible over here.

We kind of got to control the constant. Anything greater or equal to 2 would be all right. Over here, it's always at least 5. This has since been improved. I'm just telling you the historical-- these days people can get down to at least 4 or so. Actually, get down to 1. We'll talk about it in a moment.

Before I get to the Hk part, I want to talk about the lower-order term. There's some scary parts like this. If sigma is that at all large, this is big trouble. Or even sigma log n-- this is a super polynomial. So this cannot handle very large sigma, whereas this solution can. And other structures can, but this is an early result.

This also gets bad when sigma's very large. Even bigger-- when sigma's bigger than log log T, then this starts to dominate. OK, but for sigma small, think binary alphabets, whatever. This is good, and in many ways is actually better than T log sigma.

So let me tell you about this Hk of T thing. This is what's called k-th order empirical entropy. Maybe I should start with an aside of 0-th order entropy, because we haven't talked about-- I guess we talked about entropy in the context of binary search trees.

We said, oh, if you've got-- if you access item i with probability P i, then there's this entropy bound, which is sum of P log 1/P. So I don't know, let's call this character x.

So if you-- let's see, you have H0 substring s. You sum over all characters in the alphabet, of the probability-- this is not really a probability. This is going to be the number of x's in s, divided by the length of s. This is what's called empirical probability. It's what you observe from this string.

There's this many occurrences in the string. You divide by the length of the string. That's kind of like a probability. It's scaled to be like a probability. It's between 0 and 1. And if you take sum of P log 1/P, that gives you a bound.

And this is the bound achieved by say, Huffman coding, or the optimal code. If all you're allowed to do is give a code word for each letter of the alphabet, and then you write down a binary code word for each letter of the alphabet. And you write that down for each letter in s, then you achieve-- I guess Huffman codes achieve ceiling of this.

If you want to achieve exactly that bound, you can use arithmetic coding, but we're not going to get into those kinds of details. So if you used what's called a 0-th order code, where you just have a code for each character of the alphabet, then the space bound you would achieve is H0 of s, times the number of characters in s.

So that would be if you substituted k equals 0 here. So that's kind of neat. This is a compressed representation of the string. Over here, we just wrote down the string. And if the string is incompressible, yeah, T log sigma is optimal.

But if the string is compressible, like many strings we want to store-- you're storing English, whatever-- you should save somewhere between a factor of 2 and 10. This will try to save it.

Of course, factor between 2 and 10 is not-- is a little scary, when there's this factor 5 out here. That might dominate whatever savings you get. But in theory, this could be a lot better. And this is just the first result in this series. Now we can get 1 times Hk of T, and then it's a lot more interesting.

OK, so that was 0-th order entropy. What's this k-th order entropy business? Essentially, it's about taking-- instead of writing a code word for a single letter, you can write a code word for a letter that depends on the previous k characters.

So I'm going to write down a definition. Hk of T is going to be the sum over all words of length k. This is going to be our context of the probability, or empirical probability of w occurring times the 0-th order entropy of the string of successor characters of w.

So again, the empirical probability of w occurring is the number of occurrences of w, divided by T, basically. So the idea is, now you get to encode a character depending on the context of the last k characters. So we're summing over all possible contexts of k characters, and we're

taking the expectation over all possible context w.

That's the sum of the probabilities times something. And then condition on w being the context, the last k characters. We want to measure what characters follow that. And there, we can use a 0-th order encoding. I mean, we've already conditioned on w being right there.

So for all occurrences of w, you look at the next character right after it, and you take 0-th order entropy of that, that's called k-th order entropy. OK, you have to think about it for a while, too. But this essentially means the best, you can prove this is the best encoding you can do, if the codeword of a letter can depend on the previous k characters.

Of course, if you have such a code it's easy to decompress, because as you're decompressing, you know what the previous k characters were. OK, interesting thing about this index or this data structure, is it's independent of k. The data structure doesn't know what k is. This works for all k.

For any fixed k-- k has to be constant here. There are other data structures like [? KB, ?] logarithmic, or so. But here, we'll think of k as a constant.

And so this is really a neat thing about compression. There's a technique called the Burrows-Wheeler transform. And Lempel-Ziv does similar things. You may have heard of those compression schemes. They're used in bzip, and things like-- bzip is named after Burrows-Wheeler, I believe.

And those compression schemes achieve Hk of T bits per character-- so Hk of T times T-- for all k. So if your text is really good, given the context of the last five letters, or three letters. In some sense, the compression scheme adapts to that.

So this is what we call a self index, in that this also stores the string. You can read the data of the string. And so whereas over here, we just stored the string uncompressed. Here we're effectively storing the string in a compressed form, and the data structure is similarly compressed. So if your string is compressible by more than a factor of 5, this will be really good. And that's the FM-index bound.

Now that you have that Hk stuff, it's a lot easier to state all other results. So we have-- oh, I didn't give a query bound. That was the space. Query is P plus size of output times log to the epsilon T. So, similar to this one, but we don't have this trick over here.

Another early result is by Sadakane. I think also, maybe 2001, I have the journal referenced as 2003. This is in some ways better, some ways worse, it's kind of incomparable to the other results.

This is bits, and then the query has an extra large factor. This is again, another early result that I want to highlight. Now I'm going to start skipping results.

The main innovation here, is that it works good for large alphabets. This is a very small dependence on sigma, whereas-- as I mentioned, this structure really doesn't work well for large alphabets.

Here we're getting-- not getting k-th order entropy, we're getting 0-th order entropy. It's a somewhat weaker result. The dependence on epsilon is more like this one. But if you just want a log factor here, then this is a 1 plus epsilon times H0. So in that sense, we're doing better-- only a 1 plus epsilon, which is better.

This thing was always at least 2. This thing was always at least 5, the complete constant. Here the lead constant can be 1 plus epsilon. This is almost succinct, but not quite.

It doesn't quite compress as well-- it only uses 0-th order entropy-- but that's still not bad. And then the other big innovation is the dependence on sigma small. The query is a little bit worse. OK, now fast forward a little bit. I want to talk about succinct data structures for suffix-tree-like queries.

So there's two succinct data structures out there, with more or less the same authors as the first two results I talked about. So Grossi and Vitter, together with Gupta, can get Hk of T times T, which is optimal even with compression, with k-th order compression. And a good dependence on sigma.

Yeah, I guess-- T log sigma is the uncompressed bound. So you have to worry about-- when you're talking about compression, so here we have the optimal bound using k-th order entropy with a lead constant of 1, so that's great. That's what makes it succinct.

As long as this is little o of that. This is going to be a little o of that, as long as Hk of T is not too small. If it's like 1 over log T, then actually this term dominates. But as long as it's bigger than log log T over log T, this thing, then you're fine. Just as long as you're not compressing a huge amount, then this will be lower-order.

Sorry, query time. Query's a little bit worse, though. We have a log term with a P, only a log sigma, but then we also have this log squared over log log. Times log sigma, and here I haven't-- there isn't a clear dependence on the size of the output.

So this is-- let's say size of output is 1. You just want to find one match. I won't write this dependence on the size of the output. My guess is this is multiplied by the size of the output, but it's not stated explicitly in the paper, so I want to be careful.

So we have a polylog additive slowdown here. So it's a little bit worse in time, but this space is obviously, a lot better. We've improved our constant factor from 5, over here, to 1. OK, and then there's one more paper I want to mention, by Ferragina, Manzini, Makinen, and Navarro.

This is from just five years ago now, 2007. They also achieved 1 times Hk of T times T as the lead term. And they get T divided by log to the epsilon n, so this is-- yes, it's slight, there's probably a log sigma here, too. I'm not sure, it might just be T. Probably just T, actually.

So we get rid of the log sigma, but this log log over log gets slightly smaller. It's only a log to the epsilon now. But the query bound is a little bit better. So the P plus-- as the output times log to the 1 plus epsilon T query. So instead of basically log squared, we have log to 1 plus epsilon, slightly better. They also have an order P counting query. So if you just want to know how many matches are there, they can do that really fast in kind of regular time order P.

And this is obviously very small. So this is probably the best result so far, still obviously, lots of open problems in this world. Still an active area of research. There are papers since these, but they don't achieve-- the space bounds they achieve are not quite as good.

There may be like 2 times Hk, and then they can get better query bounds. A lot of papers that I'm not talking about, there's just a few too many. But if you just care about space, this is the best so far. Or I use these two, depending on exactly how big sigma is.

Just to mention, there's some other cool things you can do. So these are small space static data structures. Some of them can be made dynamic. But in particular, there's work on, how do you actually build these data structures with low space?

Because you don't really want to build a huge suffix tree and then compress it. Because the whole point is you have a hard time storing this data structure. So in fact, there's some papers-- I think more along the lines of these original results-- the Grossi-Vitter, Ferragina, Manzini, and Sadakane-- building those data structures. And while you're building the amount

of working space is at least proportional to the size of the final data structure. So that can be done. We're not going to go into it here.

There are other papers about-- all of these papers are focused on how do I do a search, how do I search for a pattern, find all the matches. There's other things you can do with suffix trees like, given two suffixes, you can find the longest common prefix of them.

So there's papers on how to do that kind of stuff in the compressed regime. There's papers on-- or there is a paper on how to do document retrieval, which is a problem we looked at two lectures ago, in the string lecture. You want to find-- not all the matches, you want to find all the documents that have this substring in them.

So that can be-- that reduces the size of the output in these bounds. That can also be done, Sadakane wrote a paper about that. Some work on dynamic-- there's actually a lot of work in implementing these data structures, definitely FM-index, and I believe, maybe the Sadakane one. And maybe this-- versions of this one.

I don't think the succinct ones have been implemented, although I don't know for sure. But there's a lot of work in implementing this, because people care, and indeed they're small and reasonably fast. So if you need a text index, there's freely available implementations of at least some of these.

So this is one of-- I mean this is practical stuff, too. Cool. But as I said, I'm going to focus on the simplest I know, which is Grossi and Vitter. If you look at the paper, there are sort of successive improvements. And we're going to cover up to the point where we get a good space bound, and the query won't be quite as good.

So that's going to be the bulk of the lecture. It's how to get that space bound. And as I mentioned, we're going to start out with a weaker bound, which is getting T log log T bits, and then we'll see how to improve that to T. And then we'll see how to improve it to 1 over epsilon times T.

So it will be a series of improvements. And we're going to start just with thinking about suffix arrays. So what is the compressed suffix array problem? Well, it's just that I have-- I want to be able to do queries of the form SA of k. If I imagine the suffixes in sorted order, what is the k-th suffix? Where does it begin?

So I want to be able to represent that array. And using that, you could do searches, and later we'll see how to use that to make a suffix tree. But for now, that's just our goal, is to compute SA of k. OK, well, the idea is actually going to be very familiar. We saw it two lectures ago, when we did this divide and conquer for building a suffix array.

We did this-- we divided the letters in our string by 0, 1, and 2, mod 3. We won't need mod 3. We'll just do mod 2 here. It won't actually matter what constant we use. But we're going to follow that recursion and use it to represent the suffix array, instead of using it to build it.

So the base case, and set up some notation. T0 is going to represent T. The length of that string I'm going to call n0 or n. And we have a suffix array, which I'm going to call SA 0, which is the suffix array of that text. So that's just notation. We're not actually storing all of those things.

Now, the recursion is T k plus 1. That's going to be the next level, which is, we write-- we combine two letters, Tk-- sorry, square bracket-- 2i comma Tk square bracket 2i plus 1.

Combine two adjacent letters into one letter, and we do that for i equals 0, 1, up to n/2. That's our new string. I'm not going to sort these letters and remap the letters to compress the alphabet. I'm just going to leave those letters alone, as an ordered pair.

In general, at level Tk, a single letter is actually 2 to the k letters. But still, this is a useful way to think about it, because it lets me think about fewer suffixes. Here, we only have the even suffixes, suffixes that begin at even positions relative to Tk.

The size of this string, in terms of number of letters, is 1/2 of the original. So in general, this is going to be n over 2 to the k. And then we're interested in the suffix array SA k plus 1.

This is going to be just looking at the even values. So if we extract even entries from sorry, SA k. So if we already have SA k, we just take the even values that are in there. Those are the ones that are existing suffixes. Extract those, divide by 2. That will be the suffix array of this text.

This is kind of backwards from how you would construct the thing. You would construct it bottom up. Here, we're imagining-- we already know the suffix arrays are just about representation. So this is a top-down kind of definition of what we're trying to store.

OK, so this is what we want to do. Now we are going to build things bottom up. We're going to

imagine we've already represented SA k plus 1. And now we need to represent SA k. If we can represent SA k in terms of SA k plus 1 with not too many bits, then you add up all of the levels of recursion. We'll have to talk about how many levels of this recursion we need to do.

We're not going to go down to constant size. We'll just go log log n levels. But we just add up all those costs, and we'll get the overall size of our data structure. So how do we do this representation?

I need to define two kind of weird things, and then we'll see why they're interesting. OK, the first thing is called even successor sub k of i. So let me define it. It's going to be i if the i-th suffix starts in an even position.

So it doesn't do anything for the even guys. The interesting thing is when the suffix starts in an odd position. Then we're going to write down a different number j. This is going to look kind of weird, but it's actually-- it's simple after you think about it for 10 minutes.

This one is odd. OK, so the other situation is that SA k-- the i-th suffix starts at an even position. So let me draw a little picture. So here is SA of i. OK, if this happens to be odd, this position in the text-- this is Tk.

Then I want to go here. OK? Because that's an even position, it's a suffix, it's right next to the suffix I care about. It is what we call the even successor suffix. But I don't want to know the index of that. The index of that would just be SA k of i plus 1.

I want to map backwards through SA inverse. I want to know, what is the rank of that suffix? Which suffix j starts right there? I want to know that the j-th suffix starts right after the i-th suffix, and I want to write down j. We'll see why this is the right thing in a moment.

We're just mapping through SA, adding 1, and then mapping backwards through SA. So that's a function. We're going to store that function in a particular-- in a very weird way, which we'll get to in a moment.

OK, next thing we need is called even rank. This is going to be like our rank function. We've had it before. This is going to be the number of even suffixes-- even suffixes are suffixes starting at even positions-- preceding the i-th suffix.

i-th suffix meaning the i-th one in sorted order. So the suffix SA of i. Yes, so this is-- let me be more precise. This is the number of even values in SA k up to i. So we're looking-- so this was

the text. Now we're looking at the suffix array, which has the suffixes in sorted order.

We're looking at position i here, and we want to know, of all of these values, which ones are even? Or how many are even-- that's the even rank. Again, a weird thing, we'll see why it's the right thing in a moment.

Right now, in fact. So here is observation 3, putting these together. This is a rather long equation. Ultimately, I want to know-- I want to represent Sk of i. I'm trying to represent that.

And I want the right-hand side to only refer to SA k plus 1. So here's the claim. Take 2 times SA k plus 1 of-- I'm going to need another board. Not of i. Even rank of even successor of i, minus 1 minus is even suffix of i.

OK, so that's the equation. Let me unpack this a little bit. The idea is, we want to know about a suffix i. If i happens to be even-- sorry, not if i happens to be even-- if SA of i happens to be even, we're golden. Because that suffix is represented by SA k plus 1, but it might not be even.

So we want to round it to an even suffix. Knowing about this odd suffix is just about as good as knowing about the suffix that starts right after it. So that's what even successor does. This is rounding to an even suffix, meaning a suffix starting at an even position.

Now there's this issue that over here, we have this relation between SA k and SA k plus 1, but it extracts the even entries. So if you think about the suffix array, which now I'm going to draw a vertical, because that's more normal.

Some of these values are going to be even, but you don't really know which ones are going to be even. It's arbitrary subset of-- in SA k, our even values. And those are the ones that you extract and form SA k plus 1. But it's an arbitrary subset, that's kind of a-- you can't just divide by 2 or something. It's not the right thing.

If I'm given an index into here, even if it's an even one, I need to know what the corresponding index is over here. And that, I claim, is exactly even rank. Because what position does this cell become over here?

Well, however many even numbers there are above it. So you take-- that's what this definition was, a number of even values in that prefix. That is the position you will be in, in SA k plus 1.

So this is what I would call the name-- we've now rounded to an even suffix but now we need

to find the name of that even suffix-- in SA k plus 1. So that's exactly what even rank does. So now we can dereference SA k plus 1 of that thing.

That will give us a position into the text T k plus 1, where that suffix begins. Now that's an index into this divided by 2 string, we need to uncompress that to an index into the actual string.

And there are two parts. One is we need to multiply by 2, because every letter in T k plus 1 is two letters in Tk. So multiply by 2. And sometimes we need to subtract 1. We basically need to subtract 1 if if even successor did anything. If even successor essentially moved us to the right by 1, now we need to move back to the left by 1, if this moved us at all.

So I have one more function here, which is is even suffix. Was SA of i an even-- SA sub k of i, an even number already. Which means that even successor did nothing.

If it did nothing, then 1 minus 1 is 0., and so nothing happens. If it did something, then its even suffix will be 0, because it was odd. And then we're subtracting 1. So this just means subtract 1, if it was odd.

You might say minus is odd suffix, instead of 1 minus is even suffix. But it turns out, this is the thing I want to store, so I wrote it in a weird way. Why did I write it that way? Because is even suffix is related to even rank.

Even rank is just rank sub 1 of is even suffix. And we already saw how to do rank sub 1, and so that's why I wanted to reuse it. I think you see now why this equation holds. What remains is how to store is even suffix, even rank, even successor.

One other thing that remains, is to say when to stop this recursion. So I claim it's enough to just do this recursion for log log n levels. And then I'll call log log n l, the number of levels in this recursion.

Because at that point, n sub l equals n over-- it's n over 2 to the l, so that's going to be n over log n. Once I have a string of length n over log n, I can afford the regular boring representation of a suffix tree. I can afford T log T bits, when T is only n over log n.

If you want to be a little extra clever, you can put a factor 2 here, and then there's a square here. And so then you're really paying little o of T in order to store that thing. So once you get down to here, you can afford a simple representation.

Now let's think about how to compute SA, like the original SA, sub 0, of an index. Well I apply this formula at all times, I do all these computations. And now I've reduced the problem to SA 1, and then I do these computations. I reduce it to SA 2, and so on.

After l steps, I'll have reduced it to an SA query in a boring old suffix array, which I've just stored as an array. So then I can answer it, and then I pop up the recursion, log log n times, doing these adjustments as appropriate. In the end, I get the correct index into the original text T.

How much time did it take? Order log log n time. So I can do a log log n time query to SA. This is, of course, assuming that even rank, even successor, and is even suffix are all constant time operations.

So what remains is to do each of these in small space and constant time. Then my overall query time will only go up by log log factor. This is actually going to be pretty good, we're not going to-- we're going to achieve log log query when we have T log log T bits.

That'll be our first encoding of these things. Later on, we're going have to go up to log to the epsilon, which is worse than log log n. Clear, so far? Everything is pretty easy at this point now. It's going to remain easy, it's just there's a lot of pieces to the puzzle.

This is the first-- this is the big idea. Next thing is some fancy encoding schemes to make these things quite small. Question?

AUDIENCE: [INAUDIBLE] Did you say what the space [INAUDIBLE] was?

ERIK DEMAINE: We haven't analyzed space yet, because I haven't said how we're actually storing these functions. If you stored these functions explicitly, you'd have bad space, probably still T log T. But it turns out, these functions can be encoded in a clever way, that small-- smaller, it's going to be T log log T.

And still has constant time query.

AUDIENCE: Without the functions, how much space are we using?

ERIK DEMAINE: Without the functions, we're using, essentially, no space. I guess, at the end where we're using-- the only thing we've said so far, is at the end we use an explicit suffix array. And if you set this to 2 log log T, then this would be like n over log n bits of space.

Because it's going to be this times-- I mean, the space at the bottom is going to be nl log nl. That's to store an explicit suffix array, so it's going to be this times log of this, which is going to be n over log n, if we put the 2 in.

So that part's really cheap, and that's little o of n. Of course, we probably also have to store the text. So that's n bits. I didn't mention-- I'm going to assume, I don't think we need it yet. At some point I will assume that the alphabets binary. So I'm going to leave off-- when I say n bits, really it's n log sigma bits, or n characters, or whatever. But I'm not going to worry about that here. Are there questions?

So now, it's an encoding problem. How do we encode these guys? Actually, even successor is the only thing that's non-trivial. We're going to do the obvious thing for the rest. So let me tell you about the obvious ones, easy ones.

At least, the first revision we're not going to do anything fancy with them, later on we will. Sorry, is even suffix. We're just going to store this as a bit vector. This is 1 if SA k is even, 0 if it's odd. So if we just store that is a bit vector, this is n sub k bits that we can afford. Because this is a geometric series, it's going to be order n.

Next is even rank. This is just the rank one structure that we covered last class, on this thing. So this is going to be nk-- I think we did log log nk over log nk. And this can be improved to nk over log to the k-- or log to the something of nk.

But that's an OK bound. It's little o of N. Again, this is geometric, so this overall will be little o of n. So those are easy, the remaining part is doing even successor.

A little optimization. If the i's where Sk of i is even, we don't really need to store anything. Because then, even successor is the identity function. So let's forget about those guys. I'll say, it's trivial for even successors-- for even suffixes.

So what I'd like to do, is store the answers for odd suffixes. That's what we're going to do. We're going to store them in a weird way, as we will see. So that's the odd suffixes.

There are nk over 2 evens, and there are nk over 2 odds. So we've just saved a factor of 2. This wasn't a very deep observation. But it turns out, if you focus in on the odd ones, has a nice little structure to them.

This step isn't really necessary, but it saves a factor of 2. Now the kind of interesting observation.

What I'd like to do is store these answers in order by i. That's the obvious thing to do. I want to store basically an array. Just store it in order by i, so I'm skipping the even suffixes, just storing the answers for the odd suffixes.

So if I was given a number i, how would I look it up? Well, given an index i into the suffix array, what I need to know is-- this is basically the inverse of what we did with SA k plus 1. SA k plus 1 is extracting the even entries, here we're extracting the odd entries.

So all I need to know is the odd rank of i, and then I look up into this array at position odd rank of i. That will give me the answer I want. Well, first I check is is it an even suffix, which I have stored as a bit vector. If it's an even suffix, I do nothing, I just return i.

But if it's an odd suffix, then I compute the odd rank. How do I compute the odd rank? I take the even rank and take i minus that. Odd rank, we don't need to store anything for it. I mean, you could if you wanted to, but odd rank is just i minus even rank.

Because every index is either odd or even. OK, great. So I can look up odd rank and then look at this array. That'll give me the answer I need. But I'm not going to actually store this as an array. I lied.

But in any case, let's worry about how I'm going to store it in a moment. Let's think about i-- if I I'm storing these answers-- the even successor answers, these j values, in order by i.

I claim that order is a very special order, because what does it mean to order by i? Ordering by i, that means the suffixes are sorted, right? So this is the same thing as ordering by an odd suffix in Tk from SA of i onwards.

That's the suffix that we're-- sorting by that suffix, is sorting by i. Now we can unpack an odd suffix-- it has the first character-- and then an even suffix. So this is the same thing as ordering by-- this should look familiar because we did the same kinds of tricks when we were building suffix trees.

This is even. In fact, it's the even successor. There's a typo here, [? see ?] If we follow SA k, and then we add 1. If we follow SA k backwards, that was the definition of even successor.

So I can rewrite this thing. This part is the same thing as Tk SA k even successor k of i, closed bracket, colon, closed bracket. Get that right? Yes.

That was the definition of even successors. Even successor is the value j, for which if I do SA k of j, I get SA k of i plus 1. That's the definition. OK, now Tk of SA of k. Sorry, the suffix-- that's not Tk of. There's a colon here.

The suffix of Tk starting at SA k. If I sort by those suffixes-- they're sorted, right? I mean, that was the point of the suffix array, is to sort the suffixes. So if I say I'm ordering by the suffixes given in order by SA k, they're already sorted. There's no reason to do this Tk of SA k part.

This is going to be the same thing as the order by this first letter, Tk SA k of i comma, even successor. The suffix array is defined to have this property, that these orders are the same thing. And sorting by the suffixes is the same thing as sorting by the indices into the suffix array.

Interesting, because this is what I want to store, right? Those are the answers that I'm trying to store. I'm trying to store even successor for every i that has an odd-- that starts in an odd suffix.

So really, all I need to do is order by this thing. And then once I've ordered by this thing, I'll store these guys in order by their value. Cool. So these are the pairs I'm going to store. I'm not going to-- I'm going to store this comma this, for all i, in order by this value. That is my goal.

If I can store these in order by this value, then by computing odd rank, I know where in this list of pairs to go. And I just look at the second value of the pair, that is my answer. Why am I storing this? We'll see.

I don't know if you really need to, but you can. OK. So what we're going to-- I feel like it's cheating. I say, actually store these pairs. We're not really going to actually store them. We still have another trick up our sleeve. But more or less, we're going to store these pairs-- I'll cross out, actually. Store these pairs in order by value.

Storing them in order by value is the same thing as order by i. That's what we just proved. And at this point, is when I'm going to assume a binary alphabet. OK. Maybe, I'll go through here. Need lots of stuff.

Think we don't need this giant recursion up here. Just remember, it's enough to know how to

compute even successor, the rest is easy. So here we go. We're trying to store these pairs, so we're trying to store a sorted array of nk over 2 values.

That's how many odd suffixes there are. And they're each 2 to the k plus log nk bits, I claim. Why? Because this was a single character in Tk. But a single character in Tk was actually 2 to the k bits, in the original string for binary alphabet, and general sigma to the k.

So that's that part of this 2 to the k bits. The even successor, well, that's just an index into something of size nk. So it's log nk bits. OK, fine. If I store that explicitly, I would be in trouble, because 2 to the k times nk is n. And so I would be storing n bits at every level-- well, so I guess they get n log log n space.

That part's actually OK. I can afford that much if I'm just going for an n log log n bound. This part, not so much. Because in particular, when k equals 0, that's going to be n times log n. I don't want to spend n log n space.

And the whole point, is we're trying to avoid storing these explicitly. Because if I did, I'd get n log n space. So we're not going to store them explicitly. As follows, we are going to store so there are these big bit vectors. We're going to look at the leading log nk bits.

This is kind of weird, because the log nk bits we care about are at the end. But we're going to look at the leading log nk bits especially, because this is a sorted list of bit vectors. So if you look at the leading bits, most of the time, they're going to be the same. They don't change very much. Leading bits are going to be all 0's for a while, and then occasionally they'll increment. How many times will it increment? nk times, at most, if we look at the leading log nk bits.

Here's the crazy idea, we're going to use unary encoding, unary differential encoding. Differential encoding means, instead of storing a list of values, you store the first value. Then the next value, minus the first value, and then the next value minus that value, and so on.

And unary means we're going to represent those differences in unary. Seems like a bad idea, but it turns out it's a good idea. So here's what it looks like, you look at-- I'm going to write down 0. I'm going to write down a bunch of 0's, however big v1 is. Then I'm going to write a 1. Then I'm going to write a bunch of 0's, however big v2 minus v1 is.

Then I'll write a 1, and so on. 0 to the lead, the leading bits of v-- sorry. It's the leading bits of v2 minus the leading bits of v1. That's what I meant. And then leading bits of v3 minus the leading bits of v2. And then 1, and so on.

OK, that is unary differential encoding. I claim this is small, looks kind of crazy. But it's small, because how many 0's are there total? Well, at most, nk 0's. Because I start at the value 0. With log nk bits, at most I get up to n k minus 1.

So the number of times I increment is, at most, nk. How many 1's are there? Well, there's one 1, per value. So there's nk over 2 1's. So total size of this bit factor is 3/2 nk. So storing those leading bits in this weird way is cheap.

Linear-- again, this geometric series is going to add up to 3/2. All right, it's going to add up to 3 times n. Cool. But that's just the leading bits-- I need to store this thing. I need to store the leading bits, and I need to store the remaining bits.

Now the remaining bits, there's only 2 to the k remaining bits. We switched the order. We looked at the high log nk bits, but then the low end bits, there's going to be 2 to the k of them. That I already said was OK. We could afford that-- kind of, we'd lose a log log factor.

So we store the trailing 2 of the k bits. This we actually store explicitly. So this is going to be 2 to the k times nk over 2, which is n/2 bits. nk is n over 2 to the k. Cancel, n over 2.

OK, so total number of bits-- we add these up-- is going to be 1/2 n plus 3/2 nk plus-- we'll get to this later. And then the total, this we have to do for log log n levels. We're summing k equals 0 to log log n. This thing.

And this comes out to 1/2 n log log n. This is bad, we want to get rid of that. But that was our first aim, then we have 5n-- did I miss a term? OK. Where did I miss the nk?

This was the cost for even successor. OK, but there was also, is even suffix, which was nk bits, and there was even rank, which was little o of that. So there's an extra nk here for is even suffix.

OK, so we have nk plus 3/2 nk. That's 5/2 nk. And then the 1/2 disappears because it's a geometric series. So we end up with 5n, for what it's worth. Plus big O of something.

OK, I left out something, because there's one data structure we haven't yet described. There's one more thing we need. And that comes up if you want to do a query in the structure. How do I do a query?

I already did odd rank, so I'm just trying to look up into the sorted array, at a given index. Well,

first thing is to compute the leading bits. Actually, computing leading bits is really easy if I have rank and select. What I want, if I'm trying to index into index i, I want the i-th one bit.

To look at the i-th one bit, which is select sub 1 of i, which we already know how to do, then that corresponds to the i-th value. And in particular, if I look at how many 0's are there up to that point, it's going to be the sum of this. Plus this, plus this, it's a telescoping sum.

It's just going to give me the leading bits. Because this plus this is just lead of v2. This plus that is lead of v3. So they all cancel. I just count the number of 0 bits. That's exactly the value I want to know. So I want to do rank sub 0 of that position. That will tell me the leading bits. In a query, it's not really lead of i, I guess. Lead of vi is what we're trying to compute.

Now, we also need the trailing bits. The trailing bits, they're just in an array, so you just look that up. You get the trailing bits. You concatenate those two words, the leading bits of the trailing bits-- boom, you have your answer.

That gives you the even successor. So the only thing is we need to store rank and a select structure. And for rank, we used nk over log log nk space. Again, that can be improved to nk over polylog nk. But let's not worry about that.

Item 1 completes. We now have a T log log T bit suffix array. Next, we need to make it order T, then we need to make it into suffix tree. We're going to move a little faster. Where to go now?

Now I want a compact suffix array. I'm going to use the same definition. Everything's going to be more or less the same. I just can't afford to store all these levels. There were log log n levels. Log log n levels is too expensive. Each one costs linear space. So I'm only going to store a constant number of levels.

Only store 1 over epsilon plus 1 levels. And not just any levels, but the first level, the epsilon l-th level, the 2 epsilon l-th level, up to the l-th level. So it's still log log n levels. I'm just going to skip a lot of them.

Now, it's going to be different. I can't use even successor anymore. Instead, even is going to be replaced with the notion of divisible by 2 to the epsilon l, instead of divisible by 2. So I do all this, but replace the notion of even with divisible by epsilon l.

Because this is when you are in SA sub k plus 1 epsilon l. The whole name of the game is,

you're trying to do a query in SA k epsilon l, and now you want to reduce it to SA k plus 1 epsilon l.

And these are the suffixes that are explicitly represented. Everything else needs to be rounded to that value, then rounded back, like we had with our giant formula before. It's not so easy to write a single formula anymore, it's now really an algorithm.

So to compute SA k epsilon l of i, what you do is follow a new thing, which I'm going to call just successor of i, repeatedly to get a new index j. Or I guess call it i prime, make it a little clearer-- until it's even.

So before, we just had to make one step, and then we were even. Now, we're going to have to make potentially epsilon l steps. So this could cost log log n. Log log n, that's not much. Actually-- sorry, not log log n.

This is going to cost 2 to the epsilon l, because it's divisible by 2 to the epsilon l. 2 to the epsilon l is log to the epsilon. So this now may take log to the epsilon T steps. This is where we're going to get the log to the epsilon penalty, in time.

OK, but it's simple linear search, nothing clever here. Now, what is successor? Well, successor is just the same thing. If you're even in this strong sense, then nothing happens. Otherwise, you just-- same definition. This part is exactly the same. Just go to the next position, the next suffix.

But now we have to follow it several times, until we get to an even one. OK. Then we recurse, just like before on SA k plus 1, epsilon l. The next level down of the-- I think we can still call it even rank. And then we multiply by 2 to the epsilon l. And then subtract the number of steps we did, in 1.

We made several steps here, we need to undo those steps at the end. That's it. So it's just the same as before, except before there was one step here, and at most, one step here. Now you just count them, subtract at the end. So exactly the same template, just skipping a lot of the levels.

And now the space is going to be 1 over epsilon, plus 1 times n. That's it. OK, so let me analyze a little bit. So you have to check that all of this works. Is is even suffix, that's easy. It's still nk bits. Even rank, still nk bits. Even successor, we did all this fancy encoding.

The one thing you can't do, is this part. I mean, there aren't very many even suffixes anymore. So it really doesn't help you, it buys you a very tiny factor. But 1 over 2 to the epsilon l are going to be even. So that's very few. So you still have to store all the answers, basically.

But you can do all this ordering trick, it still works. We weren't really exploiting the fact that it was odd. And now you have to-- this is not a single character, it's a bunch of characters. But still-- and so now instead of 2 to the k, it's probably 2 to the k epsilon l.

But it all works out. It's just a renaming of everything. It's still going to be linear number of bits, I claim. I don't want to go through a formal proof for that, we don't have time. But all the same tricks work.

So the claim is space going to be sum k equals 0 to 1 over epsilon. nk epsilon l, plus n, plus 2 nk epsilon l, plus the select bound, n over log log n. Why? Because this is storing the is even structure. That was just nk bits. And then, this is the successor. This is, is even.

Same as we had over here, except there's no 1/2 anymore. It's just n plus-- claim is 2 nk epsilon l. That's the right answer. Yeah, that 3 was because of this, plus this. So we still have the 3, just don't divide it by 2 anymore.

So this equals some constant times n, 6n plus 1 over epsilon n. Plus order n over log log n bits. OK, not bad. Not quite as good as this bound for binary alphabet, so ignore the log sigma. Before we had 1 plus 1 over epsilon, now we have 6 plus 1 over epsilon.

Kind of running out of time. I'll just tell you, you can tune this to 1 over epsilon n, plus the little o, with two very simple tricks. Two simple observations. The first one is, the successor structure. At level 0, there's nothing to do.

Why? Because level 0-- a single step just corresponds to walking in the string. I've got to think about this a little bit. Successor-- Actually not quite clear to me why that's true, but it turns out to be true. It's an exercise, I guess.

At level 0, you don't need to [? the ?] successor structure. So that actually saves you a big factor, because if you can skip the very-- k equals 0, then you get to skip-- you get to divide by 2 to the epsilon l, the space. So that gets rid of this term.

Then there's this other term, which you can skip, or you can store is even more efficiently. So before is even, should be a big factor. Because half of them are even, half of them are odd,

that's the optimal thing to do. But in this structure, most of them are not even. So you can save a little bit using succinct dictionaries.

Because there are very few ones-- you can achieve log, the total number of things, choose the number of ones. [? Bog ?] of that binomial coefficient is the number of 0's plus 1's. Not going to work it out, it's worked out in the notes.

But if you store that more efficient dictionary, which we claimed could be done last time, then this turns out to get a nice sort of cascading thing. And it's little of of n, in the end. So that gets rid of this term. And so you're left with just n times 1 over epsilon. Plus 1, because you have to store the text also.

Or maybe because of this plus 1, anyway. Boom. That's all I want to say about this structure. So I wanted to focus on the ideas, which got us the T log log T. Just apply the same ideas, but much more sparsely.

You lose in running time, instead of paying log log T. Now we pay-- we pay log to the epsilon times log log T, but that's just log to some other epsilon. So that gives us better space. Now linear space, instead of n log log space. Any questions about that? All right.

Now, I get to hurry through transforming suffix arrays, into suffix trees. This is actually a much older paper. It's by [? Monroe, ?] [? Roman, ?] and [? Row. ?]

There's two versions of it in the same paper. First version is going to be compact, second version is succinct. Probably won't have much time to cover the succinct version, but here's what we do. Start with compact. Store compressed-- we're going to assume binary alphabet again, as this paper does, I believe.

Store the suffix tree, but only store the trie part of it. Suffix tree really consists of trie-- binary trie, if it's a binary alphabet. Plus, lengths on the edges. Don't store the links. Or, as Ian likes to call it, skip the skips. The lengths of an edge is how many bits you're supposed to skip, so skip those.

Just store the trie structure. So the trie structure is on 2n plus 1 nodes, because there is n leaves, and minus 1. Telling me it's plus 1, I don't know. 2n plus a constant nodes. So this is 4n bits. We know how to do binary tries, finally we're using last lecture. We use rank and select a lot, but now are using the binary trie.

We're going to store this using the balanced paren structure. OK, so you have to double that-- this linear number of bits, so if we're just looking for compact, that's fine. Now the hard part is in a search, where we go from one node, to the next node. We need to know the length of this edge, we've got to figure that out.

We need to know whether the pattern jumped off, or something. We need to know at position y, which letter of the pattern should we branch on. So we need to measure this length. Not too hard.

What you do, you look at this subtree. You look at the leftmost leaf and the rightmost leaf. You look at their longest common prefix, starting from the character you care about.

And you look at the longest common prefix with the pattern P. All sounds easy-- how do you actually do it? So you need to be able to find the leftmost leaf in a subtree. Leaves in the balanced paren expression-- I think last class, I mistakenly thought they were that. In fact, they are this. Think about it long enough.

This was leaves in the rooted order tree, but what we care about are leaves in the binary tree. And they always look like open paren, closed paren, and closed paren. So this is a leaf, and so what we're asking for is in a subtree, we'll find the first leaf.

That's actually just going to be right after this open paren. Or, I guess, you do a select, select sub this, to jump to the next leaf. Then also, you can jump to the end of the subtree and then go back to the previous leaf, using rank and select.

So I won't go into details, but that's easy to do. So you can identify the two leaves using rank sub, this thing. I can identify the leaf number, so I can identify where these leaves are. Now, I have a suffix array. If I look up the suffix array of these two leaf numbers-- remember leaves are ordered by suffix in sorted order by suffix array.

These are really indices into the suffix array. They're giving me-- oh, this is the i-th suffix, this is the j-th suffix. So I look at those two positions of the suffix array, I teleport over to the string T. Now I have the actual suffixes corresponding to this and this. And I just look at where they match.

I know that if I've already gone down to depth d, letter depth d. I already know that they match the first d characters. I don't compare those. They're guaranteed to match. So I start at

position d plus 1. I know they should match, but one more letter. How many more letters do they match? That is the length of this thing.

OK. How can I afford to pay for that? I'm just going to pen linear cost, the total number of characters I compare, is going to be equal to the length of the pattern. So we're going to end up getting length of the pattern, times the cost to do a suffix array access.

Because I have to do this at every single step, in the worst case. So not perfect, but pretty good. Roughly P, suffix array access is like log to the epsilon. So we're getting a P log to the epsilon. Not quite as good as this bound, but because here the P is not multiplied by log to the epsilon. But, it's just log to the epsilon.

If you want to see a better way to do it, you can read the Grossi-Vitter paper. But this is a decent way to do it. Now briefly, this is the compact version, and let me tell you how to make it succinct.

I'm not going to touch the suffix array. Suffix array, to make that succinct is harder. But if I just want to make the suffix tree parts succinct, I can use this same idea, but I can't afford to store the whole trie. So just going to use a little bit of indirection. You can use as little as you want, this is the log log log log log log log n factor.

Use the suffix tree above every b-th suffix. So throw away all but a 1/b fraction of the leaves. And then, take the tree that remains. So once you do a search, you won't find exactly the leaf you want, but you'll be within an additive b of the leaf you want. b here can be arbitrarily small. This can be log log log log log n. But something super constant.

Then if I use this structure, instead of being n-- order n over b space-- instead of being order n space, it's going to be order n over b bits. So, we win. The only issue is now, how do you find the correct leaf, as opposed to the incorrect leaf?

I don't really have time to talk about it. You can look at the notes. Rough idea is, well, you can have a look-up table that lets you do whatever you want on b bits. As long as b is less than, like, 1/2 log n. Then you can encompass the whole trie, more or less.

And just hit it with a big look-up table and do everything in constant time. It's not quite so simple, because-- easy summary, here. Essentially, what you're doing is-- these are the blocks. So this is length b. You're finding this suffix, and you want to know, which of these is the correct one. In some sense, you have to do the search simultaneously for all b of these

guys.

And so you run down the search again, but instead of searching for one pattern, you search for all b of these patterns at the same time. Now they're mostly the same, and so you can prove it doesn't hurt you much. Maybe it hurts you an additive b.

I believe the correct answer is, in time, you end up paying quarter p plus b time. Sorry, times the cost of a suffix array access. OK, so we're still paying the log to the epsilon, because of the suffix array. If that was constant, it would be free.

P plus b time is fine, if b is log log log log n. Or you can make it log log n. Then you save a log log n factor in the bits. You pay an additive log log n. That's going to be absorbed by the log to the epsilon anyway. So it's pretty efficient.

I guess you can make this log to the epsilon, if you felt like it, to balance out here. Still would be P times log to the epsilon. And so this stuff is really quite cheap, see the notes for details. That ends our succinct coverage. Sorry, it was a little more succinct than intended. Get the idea.