MITOCW | watch?v=Fs4-E4Nj1Ks

ERIK DEMAINE: All right, today we're going to do some crossover between two kinds of data structures, memory hierarchy data structures and geometric data structures. And this will be the final lecture in the memory hierarchy series, so the end of cache oblivious.

So we're going to look at two-dimensional geometric data structure problems, both offline and online. So our good friend, orthogonal 2D range searching, which we spent a lot of time in a few years ago, we will come back to, and try to get our bounds good, even cache obliviously.

So instead of log n, we want log base b of n to make things interesting. And the batch version is where you're given a whole bunch of rectangles, and a whole bunch of points up front, and you want to find all the points that live in all the rectangles. So that's an easier version of the problem.

We'll start with that and then we'll go to the usual online version, where you have queries coming one at a time, rectangles coming one at a time. The points are pre-processed, it will be static. And to do the batched, we're going to introduce a new technique called distribution sweep, which is a combination of the sweep line technique we saw back as we used persistence to make sweep line thing into a data structure thing.

But we're just going to use the algorithmic version of that plus a cache oblivious sorting algorithm. So we'll finally do cache oblivious sorting and optimal N/B log base M/B / of N/B using a particular algorithm called lazy funnel sort, which you can actually also use to make another kind of cache oblivious priority queue, but we won't get into that.

And so by combining those two things, we'll get a divide and conquer technique for geometric problems that lets us solve the batched thing, and then we'll use completely different techniques for the online thing. So for starters, let's finally do cache oblivious optimal sorting.

I'm not going to analyze this algorithm because it's just an algorithm, not a data structure, and also because the analysis is pretty close to the analysis for priority queues we did last class. So funnel sort is basically a merge sort.

I mentioned last time that in external memory, the right way to do, or a right way to do optimal external memory sorting is an m over B-way merge sort. In cache obliviously, you don't know

what m and b are, so it's hard to do m over B-way merge. So instead, you basically do a N-way merge. Not quite N-way, I can't afford that, but it's going to be n to the 1/3 way merge sort.

And the big question then becomes, how do you do emerge? And the answer is with a funnel. And so the heart of the algorithm is a funnel. So if you have K-sorted lists that are big, sized K cubed, then you can merge them in, basically, the optimal bound.

So K-funnel, K-sorted lists, total size K cubed. Number of memory transfers to merge them is K cubed over B times log base M/B of K cubed over B. There's a plus K term and when you plug this into an actual sorting algorithm, you need to think about that, but that's not a big deal. Usually this term will dominate.

OK, so let me show you how funnel works. We're just going to go through the algorithmic part and I won't analyze the number of memory transfers. Maybe I'll draw this here. So we're going to have the inputs down at the bottom of this funnel. It's going to have some data in them.

Those k inputs down here, total size, all these is theta K cubed. And then at the top here, we have our output buffer. This is where we're going to put the results and this will have size K cubed. Maybe we've already done some work and we've filled some of it.

OK, the question is what do you put in this triangle to do the merge? And the obvious thing is recursive triangles. Recursion is like the one technique we know in cache oblivious data structures. So we're going to take square root of K-funnels and just join them together in the obvious way.

So just like [INAUDIBLE] layout, except-- I didn't quite leave enough room here-- in between the levels are buffers. There's a buffer here two between the nodes of this funnel and the nodes of this funnel. OK, these buffers may have some stuff in them at any moment.

OK, and the big question is how do you set the buffer size? This is the key step. And the claim is each buffer, we set to a size of K to the 3/2 because the number of buffers is about square root of K because there's one per leaf of this funnel. And a K-funnel has K inputs, so a root K funnel is going to have root K inputs here.

And so the total size of all the buffers is K squared, which is not too big. I'm not going to go through the recurrence, but if you add up the total size of this thing, it is linear size in the output, K cubed. I think also if you don't count the output buffer, it's linear and K squared. If I

recall correctly.

We're not too concerned with that here, just overall. Once we have K-funnels, funnel sort is just going to be N to the 1/3 way merge sort with an N to the 1/3 funnel as the merger.

We can only up to n the 1/3 because of this cubic thing. We can only merge-- if we want the sorting bound N/B log base M/B of N/B we can only afford K being up to n to the 1/3. So that's the biggest we can do. So it's a recursive algorithm where each of the merging steps is this recursive data structure.

Now, this is really just about layout. I haven't told you what the actual algorithm is yet, but it's a recursive layout. You store the entire upper triangle, then each of the triangles, somewhere you put the buffers. It doesn't really matter where the buffers are as long as each triangle is stored. As a consecutive array of memory, we'll be OK.

And now let me tell you about the actual algorithm to do this. It's a very simple lazy algorithm. So there's a whole bunch of buffers. If you want to do this merge, really what you'd like to do is fill this output buffer. So you call this subroutine called fill on the output buffer and say, I would like to fill this entire buffer with elements.

Precondition, if you're going to do a fill, right now the buffer is empty, and then at the end of the fill you'd like this to be completely full. And how do you do it? Well, if you look at any buffer-- partially filled, whatever-- and you look right below it, there's a node in this tree. You recurse all the way down. In the end, this is just a binary tree with buffers in it.

So it's going to be there's a buffer, then there's a node, then there's two children, each of which is a buffer, and then there's a node below that. OK, so how do I fill this thing? I just read the first item, the beginning, the smallest item for each of these, compare them. Whichever smaller, I stick at here. It's just a regular binary merge which is kind of cool.

You've got two arrays. You want to merge them. Stick the results here. So that's how we do fill. Binary merge of the two children buffers until we're full. But there's one thing that can happen, which is that one of the child buffers might empty. What do we do then? Recursively fill it.

That's the algorithm. Very simple. The obvious lazy thing to do. Do a binary merge. This is going to be nice because it's like two scans, until one of these guys empties, and then you pause this merge, and then say OK, I'm going to fill this entire buffer, which will recursively do

stuff until it's completely full or I run out of input elements, whichever comes first, and then resume this merge. Question?

AUDIENCE: Aren't there more than two child buffers?

ERIK DEMAINE: Should only be two children buffers. The question is, are there more than two? This recursion of the root k and root k child triangles of size root k is exactly the recursion we did on a binary tree.

I didn't say, but underlying this is a binary tree. The only difference between this and a [INAUDIBLE] layout is we're adding these buffers. I intended to draw this as binary. It's a little hard to tell because I didn't draw the base case, but it is indeed a binary tree in the end.

OK, other questions? So that's the algorithm and as I said, I'm not going to analyze it, but it's the same kind of analysis. You look at the threshold where things fit in cache or don't and argue accordingly. It's pretty hand-wavy. What I want to get to is how we use this to solve more interesting problems than sorting.

Sorting is a little bit boring. So let's go to batched orthogonal range searching. And in general, this technique called distribution sweep. The idea with distribution sweep is that not only can we use this cool funnel sort algorithm to sort, but we can think of it as doing a divide and conquer on the key value.

And in this case, we have two coordinates. We're going to use the divide and conquer on one of the coordinates. And where we have some flexibility is in this binary merge step. We're doing this binary merge, and normally it's just you take the min, you spit it out here, you take the min, you spit it out here.

That's the min of one particular coordinate. Now you've got to deal with some auxiliary information about the other coordinates. So in general, you're merging two sorted things. If there's other geometric information, you can try to preserve it during the merge. As long as you can do that, this is the conqueror part or that combine step of divide and conquer.

You can do a lot. There's a powerful technique, it turns out. It's by Brodal and Fagerberg. It's in their early days of cache oblivious. It was the first geometric paper. Fine, so replace or say augment the binary merge, which is, in the end, the only part of the algorithm other than the recursion. So it's the only thing you need to do to maintain auxiliary information.

That's the generic idea of distribution sweep. And distribution sweep has been applied to solve lots of different problems. Batched orthogonal range queries is one of them. Generally, you've got a bunch of orthogonal segments, rectangles, points, and you want to compute how they intersect. Those sorts of problems that can be solved here.

Also weird things like I give you a bunch of points and I want to know for every point what's its nearest neighbor. In Euclidean sense, that can be solved. But I like orthogonal range searching because it's the closest to our data structure problem and that's a problem we've seen.

So the actual batched orthogonal range searching is your given N points, and N rectangles, and you want to know which points are in which rectangles. That's the general problem. So normally, we're given the points first, and then we're given the rectangles one at a time. That's what we've solved in the past. That's what we will solve later. That's the online version.

The batched version is I give you a whole bunch of queries I want to simultaneously and we're going to achieve the sorting bound N/B log base M/B of N/B plus the size of the output over B. And this is generally the optimal bound you could hope for. It's not obvious you need the log, but I think for most problems in external memory you need this log.

It's hard to beat the sorting bound, and then once you pay the sorting bound, this is the optimal linear time to just write down the output. Now, this problem can be solved. Give me all the point rectangle pairs that result. I'm not going to solve it here exactly. We're going to solve a slightly different version, or in general-- whatever.

Let me tell you about another version of this problem, which is a little bit easier. Then I'll sketch how you solve that problem. So remember, we've talked about range reporting and also range counting, which is you just want to know the number of answers.

Here's something in between. You want to know for every point, how many rectangles contain it? And particularly, this will tell you for each point, does it appear in any of the rectangles in the set? It will tell you how many and this is actually necessary as a first step because one of the hard parts in solving these kinds of problems or reporting problems, is that the output could be big.

We know that's always an issue, but with cache oblivious, it's a big issue, literally, because space is important. You can't afford to put space anywhere. If these buffers have to get much

bigger in order to store those answers, then life is kind of tough because then this data structure gets too big, and then my analysis goes out the window because things that used to fit in cache, no longer fit in cache. The analysis I didn't show you.

So it's an issue. So the first step of this algorithm is to first figure out how big those buffers have to be so that we don't have to allocate them too large. And to do that, we need to basically count how many answers there are, and this is what we'll do. To compute these values, the answers aren't very big. These answers are just single numbers per point, so it's no big deal.

OK, so here's what we do. Sort the points and the corners of the rectangles by x-coordinate using lazy final sort. Nothing fancy here. No augmentation, regular old sort. Then-- this will be useful later-- then we're going to divide and conquer on y via a distribution sweep.

And here, our binary merger is going to be an upward sweep line algorithm. So let's talk about that sweep line algorithm. We presorted our points by x. If you think about the merging step, what this means-- it's confusing.

We're trying to sort by y, we were in a certain sense, but we're always going to be sorted by x because we did that up front. So the picture is going to be something like this. We're in a slab. There's going to be the left slab. So here's the binary merger. Here's the L points and the R points.

The L points are going to be in a particular x interval. The R points are going to be in an adjacent x interval corresponding to this tree picture. And then we have these points, which they overlap and why? Because the whole point is we're trying to merge by y.

OK, we also have some rectangles, and their corners are what we have represented. I probably should have used colors here. Something like this. So we're given, essentially-- we have whatever we want on the points and corners in here. We have whatever we want in the points and corners in this slab.

Let me add a little bit of color. These lines. And now we want to merge these two things and merging here is all about counting how many rectangles contain each point. Now, we already know how many points over here are contained in rectangles that are over here. So we've presumably already found that this point lies in this rectangle.

We've already found-- I guess there's no points here. We've already found that this point is contained in this rectangle. OK, because these corners were in this slab, and so let's say every corner knows the entire rectangle. So when you were processing R, you saw these corners, you saw this point. Somehow you figured that out.

What we're missing are things like this rectangle, where none of the corners are inside R. So R knew nothing about this rectangle, and yet it has points that are contained in it. Similarly, there are these rectangles that completely span L, and so therefore none of the corners are inside L. But we need to know that these points are in there. Those are the only things that will be missing at this level.

There might be other rectangles that completely span L and R. Those will be discovered at higher levels, now here. It's a little bit awkward to check if this will actually find everything, but it will. So to figure this out, when we're merging L and R-- see, L knows about this rectangle because it sees these points. We want to keep track as we sweep upwards.

We want to realize that these points are in a big rectangle here, whereas they weren't discovered in L, and they weren't discovered in R. To do that, we maintain a number as-- we have a horizontal line, we're sweeping up. We want to maintain the number of active rectangles.

Active means that it's currently being sliced by the sweep line. That have left corners in L and completely span R. So that's these guys. So that's easy to do. We're merging these points. So that each of them has been sorted by y. Now we're doing a merge, so we're considering all the corners, and all the points, and increasing the y-coordinate as we do that binary merge.

So whenever we visit a left corner of a rectangle-- a lower left corner-- we say oh, does this rectangle go all the way across? This one does not. By the time we get to here, this one goes all the way cross R, and so we increment CL. And when we get to the upper left corner, we decrement CL. Say oh, that rectangle's over.

So it's very easy to do constant time, but it's only going to be 1/B memory transfers per one of these because it's a nice, cheap merge. And then symmetrically, we do CR. It's the number of active rectangles with the right corners in R that span L. So that's this guy, CR, I guess, this guy is CL.

In general, there might be a lot of them, so you count them. And then the only thing we need

to do is whenever we encounter a point as opposed to a corner, because we're storing them all together, we add-- I got this right-- CR to it's counter. We want to know how many rectangles contain that point.

And so for example, when we see this point, and CR is currently one, then we know that this point appeared in some rectangle that spanned L. So we increment this points counter. Similarly, when we see these points, CL is positive, so we increment these guys counters by whatever CL is. So this is a symmetric version in R when we add CL.

Probably should have called them the other names, but anyway, CL, CR, doesn't matter. CLRS. Question?

AUDIENCE: The bottom is the x-axis, right?

ERIK DEMAINE: This is the x-axis, yeah.

AUDIENCE: So are we dividing and conquering on x?

ERIK DEMAINE: It does look like we're dividing and conquering on x, I think you're right. Sorry. For some reason I thought it was y. You're right. So it's a funny thing. We're pre-sorting by x, which is what's getting us-- thank you. That's much clearer now. In my mind I was like there's something weird here.

We're presorting on x and then we're just sticking these guys down here. So evenly dividing them into lists. Or, I guess actually, we're doing our funnel sort, the merge sort. Things have already been sorted by x, but now we're merge sorting again, and this time when we merge, we carry along this information. So they're both in terms of x, which is kind of funny. Is there another question?

AUDIENCE: Sorry, is it important that we do the upward sweep [INAUDIBLE]?

ERIK DEMAINE: The upward sweep. Yeah, we have to do the points in order by y.

AUDIENCE: So do we want to just sort by y, and then [INAUDIBLE].

ERIK DEMAINE: Ah, so confused now.

AUDIENCE: Because in the notes, it said x and then y.

ERIK DEMAINE: Yeah, I know in the notes it says y. It used to say x. I believe, we're dividing and conquering on

x, but we're sorting by y, and that's the confusion. I'll double check this, but in order for this sweep to work-- so it's like you first sort by x. You

We are in some sense doing divide and conquer by x because we did this sort by x. But the merge short is on y. It makes more sense. If you're already in x order, sorting isn't going to learn you much. It isn't going to teach you much. So first you sort by x. Things are nicely ordered by x. So we get these nice horizontal slabs in the decomposition, but now when we merge--

Now we're going to sort by y. So we're going to reorder the points and that's what lets us do the sweep. And we are, in the end, merging all these points together in y order. And as we do it, then we get the information we want about rectangles and points. OK, this is why I wanted this to be both x and y.

But really, the divide and conquer is happening on x, but we are doing a merge sort on y. Finally clear. Thanks for helping me. This is a new lecturers, as you may have guessed, so still working out some kinks.

I really wanted to introduce this lecture because the next thing we're going to cover, which is a way to do orthogonal 2D range search and cache obviously, is super cool. It's like one of the craziest things there is. At least in the cache oblivious world. Any other questions before-- Oh, I should say a little bit more about this.

We've now solved this first step, which is figuring out the output size. Counting for each point how many rectangles contain it, which is an interesting problem by itself. That's the range counting problem. You can also use it to figure out, at this level, at this merging step, how many things will be output here? How many new outputs are there? How many points in rectangles are there? It's essentially just the sum of all those things.

So you can count the number of outputs per merge and so then there's a natural strategy, which is you build a new funnel structure where these buffers have the right size. You've pre-computed what all sizes need to be. At every merge you know how many things are going to get spit out here.

So you could allocate that much space and that will be a kind of decent merge sort. Because I haven't done the analysis, it's hard to get into detail about this. But it will not be optimal, unfortunately. To actually make it work, you end up having to take this tree, carving it into

subtrees of linear size.

So normally, the whole thing is linear size. Everything's fine. And where the analysis breaks, essentially, is if you have a giant buffer because one of the outputs-- potentially, the output size here is quadratic. And so the overall thing might be super linear.

And so when you have a super linear buffer or a bunch of very large buffers that sum to linear size, you essentially need to carve that tree, which you do by recursive carving of the tree. So that each of the trees has linear size. Then you apply the analysis to each of the trees separately.

You store them consecutively, separately. Each of them has good optimal running time and then the combination does. That's the hand-wavy version of how to do actual range reporting with end points and end rectangles. If you're interested in the details, read the paper. It's just a little bit messy and especially when you don't know the analysis.

I want to move on to online orthogonal 2D range searching because it's the hardest and coolest of them all. Unless there are more questions. All right.

AUDIENCE: So you do the range counting [INAUDIBLE] in detail, and [INAUDIBLE] to the [INAUDIBLE].

ERIK DEMAINE: Exactly. At this point, if you believe in funnel sort, you should believe that range counting is easy to do, and I've just hand waved the range reporting part. Are you scribing? Is that why you ask? That's where we stand.

The next thing we're going to do is regular range reporting, regular online stuff. So this is orthogonal 2D range search. And we spent a couple of lectures on 2D and 3D range search. All this crazy stuff with fractional cascading, and so on, and the layered range trees.

We're going to use some of those techniques that we built there, and in particular, you may recall there was this idea that if we have a bunch of points, regular 2D range searching is I give you a rectangle, give me all the points in the rectangle. Fine. Our goal is to achieve log base B of N plus output size over B. That's the new optimal bound.

This is how long it takes to do a regular search in one dimension. So if you have output size whatever-- and we'll probably be able to do range counting, but I won't worry about it here. We'll just think about range reporting. If there's this many points, we'll output them all in that much over B.

This is what we call a regular range search, but I'm going to distinguish it and call it a four sided range search because a rectangle has four sides. But you could think of the other versions and we actually did this when we were doing the 3-D problem. So if these are two rays and an edge, this you might call a three sided rectangle, and you can go all the way down to two sides.

Hard to go down to one side. Here's a two sided rectangle, it just has two rays. OK, as you might expect, this is easier than that. And if I recall, in 3-D we ended up doing this thing in linear space with this fancy-- first you do a search on the left coordinate and then you just walk.

We'd subdivided with fractional cascading so that every face had constant size, and so you could just walk, and each step you'd report a new point. If you may recall for this kind of two sided thing. First, you would search for this, and then you would basically just follow this line until you found this point, this corner.

This we could achieve in a linear space, logarithmic time. This one we needed N log N space. Actually, the best known is N log N divided by log log N. But we could N log N using range trees. And we got down to log N time using-- log N query time and log N space using layered range trees. That was the internal memory regular algorithms.

AUDIENCE: Aren't you missing an M/B though?

ERIK DEMAINE: Am I missing an M/B? No, this is log base B of N, not log base M/B of N. Yeah, it's good to ask. When we're sorting this kind of thing, we get log base M/B, but when you're searching, the best you can do is log base B. We actually proved a lower bound about this in the first memory hierarchy lecture.

Because this is online, you read it in a block. You can only learn where you fit among B items. And so the best you can hope to achieve is log base B of N for search in one dimension. So this is a lower bound for search. When you're doing batch operations, then you can hope to achieve this stuff, which is a lot faster. Then it's like 1/B times log base M/B of M/B.

OK, so in a certain sense, this is slower than the batched operations, but it's more online. So it's a trade-off. So for all these problems we can achieve log base B of N plus [? out ?] over B. The issue is with space. Maybe I'll do sort of regular RAM algorithms versus cache oblivious.

So we've got two sided, three sided, four sided. And for two sided, I believe these are the right answers. Log N over log log N. But we haven't actually seen this one. And cache oblivious, here's what we can do. This is with optimal query times and this is all static.

OK, and if there's time, I'll cover all of these. So they're not perfect. These two were off by a log factor, but not bad. Pretty good orthogonal 2D range queries. And really, the coolest one is this one. This one blows my mind every time I see it. So let's do it.

We'll start with two sided and then we have existing techniques once you have two sided to add on more sides, you may recall from the 3D range searching lecture. So we're going to use those techniques and refine them a little bit to get that log log factor.

But you may recall way back when, at lecture six or so, that we had a technique. Once it was two sided, every time we added a log factor in space, we could add another side. The hard part was getting up the number of dimensions. Then the easy part was turning half infinite intervals into regular intervals.

So once we have this, it's easy to add a log, add another log. With a bit of sophistication, we can save a log log factor. OK, but let's do two sided. This will be the bulk of the lecture. This is a paper by [? Harga ?] and [? Zey ?] in 2006. All right, so we want to do-- I'm going to assume that they are this kind of quarter plain query.

So less than or equal to x, less than or equal to some y-coordinate. We want to know all the points in that quarter plane. So here's what we're going to do. It's all static. We're going to have a Van Emde Boas layout. So a binary tree on the y-coordinate.

So this just stores all the points sorted by y. So if you want to do this query, use search for that value of y, then each of these positions in between two keys in here has a pointer to an array. The array is not sorted by x or y, it's a very weird thing. And then here's the algorithm you follow.

You follow this pointer, you go here, you walk to the right until you find a point whose x-coordinate is too big. It's bigger than x. I should probably call this x2, y2. So first you search for a y2 here, in this thing keyed by y. Follow the pointer. You look at all the points that have x-coordinate less than or equal to x2. Those are the ones you want.

Once you find a point whose x-coordinate is bigger than x2, you stop, and then you report these points. It's not quite so simple because some of these points might be duplicates. You

have to remove duplicates. That is your answer. To me, this is an insane idea. I would never imagine this to work.

But the claim is you can make this array have linear size. That's the hard part. Make this, the amount of stuff that you have to traverse here, be linear in out in the number of points that are actually in this range. You are going to do a little bit more work because there are duplicates in here, but only a constant factor of more work.

And yet somehow, you've taken this two dimensional problem and squashed it onto a line. You did one search at the beginning, which costs you log base B of N, then you do this linear scan, and you get the right answer, magically. I don't know how they thought this would be possible, but magically, it turns out it is possible. It was kind of a breakthrough in cache oblivious range searching.

It was known how to do this for external memory a lot easier. For example, you can do it with persistence, but this is a much cooler way to do two sided range queries. All right, so I've explained the query algorithm. The big thing I haven't explained is how to build this array.

Maybe I'll write down the things we need to prove as well before we get there, so you can think about them as we're writing down the algorithm. First claim is that this algorithm, which just decides to stop whenever it gets an x-coordinate that is too big, actually finds the right answer. It Finds all points in the range that we care about.

The second thing is that the number of scanned points, the length of that step here, is order the size of the output. The number of actual output points. We don't waste time doing the scan. And the other thing is that the array has size order N. That's the biggest surprise to me.

So those are the three things we need to prove about the algorithm, which I will now tell you. OK, before I can define how this array works, I need to define a concept called density. If we look at a query, there's two things that could happen.

The good thing for us would be if-- get this right. The number of points in lesser or equal to x star is at most, alpha times the number of points in the answer. OK, star means no restriction on y. Minus infinity to infinity.

This would be good for us because it says-- ultimately what we're trying to do here is do a scan in x. It's the right thing to do here. Then for this particular y-coordinate, we could just

basically start at the beginning of the array, start scanning, and just report all the points that are actually in our range.

Sorry, I need to also potentially throw away points that are not low enough. So the answer is contained in here. I should say to throw away duplicates, you have to throw away points that are not in the range lesser or equal to x, comma lesser or equal to y. Still, we claim the number of scan points is proportional to the output size. That's what we need.

So if this held for every query, we'd be happy. Just start at the beginning, scan, and as long as this alpha is some constant-- it's going to be a constant bigger than 1, then the number of points in the answer is proportional-- sorry, the number of points we had to scan through is proportional to the number of points in the answer, and so we're done.

So this is the easy case. We need to distinguish it, otherwise we call this range query sparse, and those are the interesting cases. So nothing deep here, but we're going to use this concept a lot. OK, so we're going to actually try to solve this problem twice.

The first try isn't going to be quite successful, but it gets a lot of the right ideas. So I'm going to let S0 be all the points sorted by x. It's going to be sorted by x. I put things down here. And just to give you an idea of where we're going, the array we're imagining here is first we write down all the points, then we'll write down some subset of the points, S1, then some subset of that subset, and so on until we get down to a constant size structure.

OK, first we write down all the points. Why? Because for dense queries, that's what we want. We want all the points just sitting there. So then you can just read through all the points and dense queries will be happy.

So if we detect a y-coordinate where the queries going to be dense-- I don't know how we detect that. Let's not worry about it right now-- then you could just look through S0. That's fine. But some queries are going to be sparse, and for that we're going to use S1, S2, and so on.

The intuition is the following. If in your query, the y-coordinate is very large, like say infinity, then your query is guaranteed to be dense. It doesn't matter what x is. And in general, if y is near the top, like it's at the top most point, or maybe the next of top most point, or maybe a little bit farther down, it depends on the point set, then a lot of queries are going to be dense.

So that's good news. Let's consider the first time when there's a sparse query. So we're going to let yi be the largest y-coordinate where some query, some x-coordinate-- that y-coordinate.

This is going to be less than or equal to x, comma less than or equal to yi-- is sparse in Si minus 1.

OK, so initially we have S0, all points. y1 is the largest y co-ordinate where there's-- so we work our way down until there's some sparse query in S0. That's yi. So then we just filter, based on that. So throw away all the points above yi.

So we're going to say take Si minus 1, intersect it with the range query, star less than or equal to yi. OK, so the picture is we have some point set. Up here, every possible query along this line is going to be dense because everything to the left of the x-coordinate will be in the output.

At some point, we're going to decide this is too scary. There's a query here, maybe this one, or maybe it's this query that's sparse. And so we say OK, throw away these points. Redo the data structure from here down, ignoring all these points, repeat, and write down these things.

So the idea is that if you look at a particular query, it will be dense in one of these Si's. And you can tell that just according to your y-coordinate. Because you said oh, well, if you're up here in y-coordinate, you're guaranteed safe. So just do that search and you're OK.

In general, we continue this process until we get to some Si that has constant size. At that point, we're done, and then we can afford to look through all the points. Unfortunately, this is not a very good strategy, but it's the first cut, and it's close to what works.

Here's a problem with it. Suppose you have this point set. OK, what happens is you start at the top, everything looks fine. At some point you decide there's a query here, namely this one, which has an empty answer, and yet there are points to the left of this x-coordinate. So that's bad because it's very hard to get within a constant factor of zero.

So pretty much immediately you've got to draw a line here and say OK, S0 is all points, S1 is these points, S2 is going to be these points. In general, there's suffixes of the points, and so the total space will be quadratic. So the first two properties will be correct because you're just looking in S0, or S1, or whatever. Everything looks fine, but your right does not have linear size.

So no good. First try, failed. Second time's the charm. You need a little more sophistication in how we do this partitioning, how we build our array, and we'll get it. I didn't read this before. This one line that says maximize common suffix. I have no idea what this means, but maybe it

will mean something by the end. Let's see.

OK, this is the part I read. So xi is going to be-- so we had a yi That's going to be the same as before. This is why I did the first attempt. This definition remains the same. So largest y where we have some sparse query in Si minus 1. I want to look at what that x-coordinate is. It's just that here it says there's some x. What is that x?

Let's just look at the maximum possible x that it could be. This will turn out to be really useful. The maximum x-coordinate where less than or equal to xi, comma less than or equal to yi is sparse-- and Si minus 1. OK, we know there's something we can put in here that makes yi sparse. So look at the largest possible such x.

So that means any query-- so we have this new point. It's not an actual point in our problem, but it's a query, xi, yi. And it's dense, oh sorry, it's sparse. It's bad. We know that any query up here is dense. That was the definition of yi. And now we also know that any query over here, I guess, that's saying a lot. But these queries are also dense.

Because again, if you're far enough to the right, that's going to be basically everything. So let's get rid of that as well. And this is a problem, queries over here are also potentially a problem. We don't know. It doesn't seem like much, but it will be enough. We're going to redefine Si as well.

So here's the fun part. If we have some Si minus 1, we're going to define a new thing, which is Pi minus 1, which is this. This is a funny thing, but it is this part of the point set. This is Pi minus 1. So the points we care about are kind of here, but let's just take everything to the left of this x-coordinate. Why not? It's a thing. That is Pi minus 1.

So Si minus 1 is everything in this picture. First, let's restrict to x, then the next step is we're going to restrict to y. But it's in a funny way. This is the Si, the next s set. Take the previous set and we intersect it with a funny thing.

It's harder to write algebraically than it is to draw the picture. So it's intersected with a union, which is basically-- dare I draw it on the same picture? Where's my red? It's going to be less than or equal to y. This thing is going to be Si. We'll see why, eventually, this works.

I still don't know what maximize common suffix means, but we'll get there. So we're looking at the points below the line. That's what we did before. We used to say Si is just the intersection with less than or equal to yi. But things are just a little bit messier because of this restriction.

Do I really not have a P here? OK, here's the difference. The reason we have to go through this business.

The array that we're going to store is not the Si's. Si's are still too big, potentially. What we're going to store are the Pi's. Pi minus 1. And then in the end, we're in a store Si. Si, again, has constant size. The final Si has constants size. I probably should have used a different letter, Sk or whatever.

We keep doing this until we get down to something constant sized, then we store it. That's the easy case. Until then, we just store the Pi's, because really, we know that all the queries up here and over here are OK. They're nice and dense. We sort of only care about the points to the left of the line.

OK, but essentially, the Si has to pick up the slack and we have to include these points in the next Si. Whereas, before, we did not. Before we just took things below the line. Now we have to take things that are below the line or to the right of the vertical line. This is essentially necessary for correctness.

So we kind of win some, we lose some. But it turns out all is well. So I know this is weird, but let's jump to the analysis. These claims, in particular, that the array has linear size. Let's think about that and it will become clear why the heck we've made these choices. Unless you have a question first.

AUDIENCE: Is there any relationship between the Si here and the Si on the first try?

ERIK DEMAINE: No, this definition of Si is no longer in effect. S0 is correct, and all the Si's are still sorted by x. We're no longer doing this. Instead of this rule, we're doing this rule. This part is the same, but we have this extra union, which contradicts the previous rule. So the yi definition is the same. Sorry, it's a little weird. xi is new, Pi is new, and Si is new.

At this point, it's this algebraic weird thing. Here's the cool thing. For the space bound, the claim is Pi minus 1 intersect Si is less than or equal to 1 over alpha times Pi minus 1. This is hard to even interpret what it means, but it's good news.

So remember, alpha is a number bigger than 1. It's what we use in the definition of density, and you could set this parameter to whatever you want, say 2. So then we're going to get that this thing, whatever it is, is at most half the size of the previous one.

I claim this is good news. I claim it means that these Pi's essentially are geometrically decreasing in size, which is how we get-- that's not quite right, but this will give us a charging scheme. which will prove that the whole thing has linear size.

First, why is this true? It could really only be true for sparsity from the alpha. Right, so we said oh, density is good. If we have dense, there's nothing to do. Just put the points in x order, we're done. Sparse is bad. But actually, sparse tells us something. It tells us there are a lot of points that are not in the answer.

So we're looking at this query, xi yi. And we'd like to just say oh, start at negative infinity, and just take all the points up to here. If we're dense, that is within a constant factor of the number of points that are actually in the answer, which is down here. If we're sparse, that means there are a lot of points up here. Most of the points have to be up here in order to be sparse. And that's actually what this is saying if you expand the definitions.

So Pi minus 1, that was all the stuff to the left. So that's this thing. This is what we would get if we just did a linear scan from left to right. Versus we're considering the points in Pi minus 1, which just restricts to x, and then we're looking at Si.

Si does this business. But if we restrict to the Si points that are to the left of the line-- so we're looking at, basically, this left portion, which was this white rectangle, intersected with this funny red rectangle, which was kind of awkward-- the intersection is just this. That's the answer for this query, xi yi. OK, so this is the size of the answer for xi yi. And this was the number of points in less than or equal to xi star.

We wanted to just do a linear scan like this. But this is the correct answer and because we know that this point is sparse-- that was the definition of xi and yi, it was the maximum sparse point. So it's a sparse point, therefore we know that this does not hold.

So the number of points less than or equal to x comma star is greater than alpha times the number of points in the correct range. And if I got it right, that should be this. You could put alpha over here without the one over and I guess this is strictly greater. No big deal. So that's the definition of sparsity. So this is the cool thing we know. Now, we're going to use-- this is now a numbered less than 1. Question?

AUDIENCE: So for Pi minus 1, we add them as the number of points less than xi star. But for example--

ERIK DEMAINE: Yes, that's the definition here.

AUDIENCE: [INAUDIBLE] like Pi, you don't have that block in the top left corner, right?

ERIK DEMAINE: Right. After we restrict to Si, yeah, we've thrown away all of these points.

AUDIENCE: Right. So if you take the next Pi, it's not necessarily going to be the points less than or equal to xi--

ERIK DEMAINE: It's true. When I say points, I don't mean all points. I mean points in Si minus 1. I'm dropping that because it gets awkward to keep talking about. So that's a correctness issue, essentially. You have to argue that we can throw away these points and it's safe. Once we do, then you could just ignore their existence.

You can ignore their existence because you already solved all the dense queries, which are over here, or over here, which involve those points. And so we now know that we're only going to be doing queries from here down. Otherwise, you look at P0.

So forget about those. Forget about those points. Now you're going to be searching in one of these structures. So you can forget about all the points over here. So that's that argument. Once you've restricted to Si minus 1 and you don't have to look at any other points, among those points, this is going to be all the points less than or equal to xi.

But that's how we were defining bar sparse. We said sparse in Si minus 1. So it's among those points we have sparsity. So this is the definition of what we have. OK, the claim is it's a good thing. Here's the charging scheme. So this is by sparsity.

So I'm going to charge storing Pi minus 1 to Pi minus 1 minus Si. This algebra, I have to interpret every single time, but that's fine. Let's look at the picture. OK, Pi minus 1 remember, was this white rectangle over here. Everything to the left of the line.

We have to store Pi. We want that the sum of the sizes of the Pi's is good. And so here's my charging scheme. We have to store Pi minus 1. I'm going to charge it to these points. What are those points? Those are the points that are inside the white rectangle, but outside the red L-shape.

So that's these points. This is Pi minus 1 minus Si. Those are the points that I'm throwing away. That's good. So if I charge them now, I will never charge them in the future because I

just threw them away. They are not in the next Si. Each point overall in the point set only gets charged once.

OK, how much does it get charged? How do these things relate to each other in size? That's where we use this thing. It gets confusing to think about intersection versus difference, but the point is if we look at the Pi minus ones that are in Si, that's a small fraction.

Think of alpha as 100. So then the Pi minus 1-- so this part down here that's in Si, this is only 1/100 of the whole white rectangle. So that means this part is 99/100 of the Pi. So if we charged the storing of the entire rectangle to these guys, we're only losing a very small factor like 100/99 or something. It isn't actually exactly 100/99, I believe.

I worked it out and the factor of charging, assuming I did it correctly, is 1 over 1 minus 1 over alpha, which works out to alpha over alpha minus 1. It doesn't really matter, but the point is it's constant. I think that's easy to believe. Maybe it's actually easiest to think about when alpha is 2.

At most, half the points are here. At least, half the points are here. And so we're charging storing the entire point set to these points, which will never get charged again. So we're only charging with a factor of two. That's all we need, a constant factor.

OK, therefore, this thing has linear size. That's the cool thing. We get more though. We also get the query bound we want. Let's think about the query bound. This is fun. Think about where the query is. It used to be over here. We do a search in S0, or we do a search in S1, or we do a search in S2.

We'd never look at multiple Si's because there'd be no point. Either S0 was dense, and we're fine, just do it. Or you have to jump to S1, skip some guys up top, do the search in there. Fine. We no longer have that luxury over here because we're using Pi's instead of Si's.

So it actually may be the search starts in P1, but then has to go through P2, and has to go through P3. But it's OK because the farther we go right, we have this sparsity condition that tells us basically the points we're looking at are-- the number of points we're looking at are getting smaller and smaller.

So I'll wave my hands a little bit here, but the claim is it's a geometric series. This needs a formal proof, but we won't go through it here. Decreasing-- so this is the query bound. The number of scanned points is order output size.

So you have to check that no matter where you start in Pi-- that's the little bit tricky part. We're not looking at all of Pi. We're looking at some of Pi and then we're going to the right from there. Actually, is that true? Maybe we always look at all of Pi. Let me think about this. I think we do, actually. Sorry. That's what we did before.

We basically figure out where we are in y-coordinate. That was the overall structure. We had a Van Emde Boas search tree on y. So all we know at this point is the y-coordinate of our search. And so we use that to determine which of the Pi's we go to, based on where the yi becomes no longer dense.

And then we're going to have to search through that entire Pi and potentially more of them because this is no longer an Si. It's just doing the things to the left. And so if we're lucky, the Pi we're looking at, or the query we're doing, is not to the right of this point. OK, maybe it's right here. That would be great. Then all our answers are done.

If our query is here, that would have been dense, so we would have done it at an earlier stage. Our query might be down here though. When the query's down here, we need to report on these points. Then we're going to have to do more and that's going to be Pi plus 1. So we'll do more and more Pi's until we get to our actual query here.

But in any case, the claim is that this is geometrically decreasing by the same charging scheme. OK, that's two out of the three claims. There's one more, which is closely related. It's still about the query problem. What we haven't shown is that we actually find all the points. This is what you might call correctness.

To prove this, what we need to say-- what we claim is that after you do the P1's-- and now you do the P2's. Well, I'll tell you. The claim is that you visited some x-coordinates here. The Pi's were all the things up to some x-coordinate. Claim that the very next point in here, in P2, has a smaller x-coordinate than what you just did.

I think that should be clear because presumably there are some points in here, and so the very next Pi, it's restricted within this red thing, but it's going to be up to some x-coordinate. So you're basically starting over. Every time you go to the Pi's, you're starting over in x. Go back to minus infinity in x.

So the idea is the picture will look something like this. You start at minus infinity, you read

some points. At some point, you run out of the Pi's. Then you start over again, you read some smaller set of the points. Maybe you get a little farther. You start over again, read a little farther. At some point, you're going to reach your threshold x. That's when you stop. So that's correctness. I feel like I need another sentence there.

Once your Pi encompasses your x range, that's going to have your answer. Then you're done. So that's this moment. And so the only worry is that an early Pi, basically, or maybe the next Pi does this, and then we do this or something like this. That never happens basically because you're always resetting x range. And so your x will always start over to something less than what you had.

And so the termination condition, which I probably didn't write down here, but which is stop when your x-coordinate is bigger than what you want. Never terminates early. Therefore we get all the points we care about.

OK, a little bit hand-wavy, but that is why this structure works. It's a very weird set up, but linear sized, and you just jump into the right point in the array, start reading, throw away the points that aren't in your range because they just happen to be there. Those would be these points up here.

Throw away duplicates. Just output the points in your range and it gives you, magically, all the points in here by a linear scan. I still find this so weird, but it's true. Truth is stranger than fiction, I guess. They're fun facts. You can actually compute this thing in the sorting bound. So pre-processing is just sort. I won't prove that here.

So this was two sided. Let me briefly tell you how to solve three sided and four sided. We basically already did this one, which was-- I'll remind you what it looks like. So you have a binary tree, and in each node you store two augmented structures.

One which can do ranged queries like this, and one which can do inverted range queries like this. This should look familiar. And so you do a search on-- let's say we want to do this thing. So we have x1, x2, y2. You search for x1, you search for x2. You find the LCA and then in this subtree, you do a search.

In this subtree, you already know that you're less than x2, and so you do the x1, y2 search in this node. And then in the right subtree, you do the x2, y2 search. You take the union of those two results and that is this query. That's how we did it before. No difficulty here.

And the point is, you can build this, put it in a Van Emde Boas layout. You do this search, you do this search, you find the LCA in log base B of N-- to check that everything works, cache obviously. Then these structures are just structures which we already built, and so yes, we lose a lag factor because every point appears in log data structures, but that's it. Everything else works the same.

So we get N log N space log base B of N plus output over B query. Because now we just have to do two queries instead of one. We don't there's a log factor. That's the trick we did before OK, that was easy. One more. So that was three sided. Next is four sided.

Four sided, of course, we could do exactly the same thing. Lose another log factor in space. Maintain log base B of N plus output over B query time. But I want to do slightly better and this is a trick we could have done in internal memory as well. But I have two minutes to show it to you. So here's a bonus.

Didn't have to do this in external memory context, but we can. Four sided. So we're going to do the same thing, but not on a binary tree. Take this binary tree, this is sorted by x, I suppose. This is key on x. Instead of making it binary, make it root log [? N ary. ?]

So imagine taking the binary tree, taking little chunks, which have size square root log N. Its capital N. And imagine contracting those chunks into single nodes. So we have a single note which has square root log N. Children [INAUDIBLE] has square root log N children. This is all static. And so on.

Otherwise, the same. The augmentation is going to be a little bit different. If we look at a node, we're going to store the same things we had before, which was this kind of query, and this kind of query. We're going to store a little bit more.

Namely, for any interval of children, like here you have some start child and some end child. I want to store for all the points that are down there. For this thing, store a regular binary search tree on y for those points. Why? Because if we do a search-- OK, same deal-- we find the LCA of x1, x1? I don't know. Let's say it's on x.

We'll have to do it again on y whatever. So here's the LCA. Let's say there's a lot of children. OK, maybe here is x1 and here is x2. So in this subtree, we do this-- sorry, we do this range query because we want to go from x1 to infinity.

Over in this subtree, we want to do this range query because we want to go from negative infinity to x2. But then there's all this stuff in the middle. I don't want to have to do a query for every single tree. Instead, I have this augmentation that says for this interval, here are all the points sorted by x-coordinate. I guess we're doing it this way.

Fine, so then it is a range query. I want to know what are all the points. Whoa, this is confusing. I feel like I've missed something here. No, this on y. Sorry. These points I've got sorted by y. So I should draw it the other way. These points we already know are in-between x1 and x2 in x. We've already solved the x problem here.

So now I just need to restrict to the y range from y1 to y2. In these trees, these already match in x. I just need to make sure they match in y. So I do a regular 1D range tree. I search for y1, I search for y2, take all the points in between. This is cheap if I just have a regular old binary search tree.

Now, this thing has linear size. This thing has-- sorry, I think I actually need-- I should have a three sided range query. Thanks. These should be three sided because here I know that I've got the right side covered already in this tree, I've got the left side covered already in this tree, but I still need the remaining three sides.

In here, I only need these two sides because I've already got x1 and x2 covered. OK, so this is cheap. I only need a linear space data structure. This thing is not so cheap. I'm using the previous data structure. This thing, which has N log N size, these are three sided range queries. Sorry for drawing it wrong.

So I need two three sided structures. Then I need actually a whole bunch of these structures because this was for every interval. But conveniently, they're only log N intervals because there's root log N children. So root log N squared is log N. So there's root N, but then we need log N of them. And so that's why these things balance out. See?

So normally, this would be N log squared N because every point would appear in log N trees. But now the height of my tree is merely log N over log log N with a factor 2 out here because I have a square root here.

OK, so the tree has height log N over log log N. So each point only appears in log N over log log N structures. Each of them needs a structure size N log N. So we end up with N log squared N over log log N space.

Kind of crazy, but this is how you get that last little bit of log log N space improvement by contracting nodes, doing a simpler data structure for these middle children, and just focusing on-- The left child and the right child you have to do one three sided call, but then the middle is a very simple two sided call. It's just a 1D structure and so it's really cheap. That's it.